Koç University
COMP438/538
Introduction to Reinforcement Learning
Assignment 1

Instructor: Barış Akgün
Due Date: April 25 2023, 23:59
Submission Through: Blackboard
Make sure you read and understand every part of this document

This programming assignment will test your knowledge and your implementation abilities of the dynamic programming and tabular model free reinforcement learning parts of the course. You will to submit your implementation through blackboard. **Important:** Download your submission to make sure it is not corrupted and it has your latest code. You are only going to be graded by your blackboard submission.

This homework must be completed individually. Discussion about algorithms, algorithm properties, code structure, and Python is allowed but group work is not. In addition, you can use Large Language Models (LLMs) in helping you code. You are required to cite your usage and accept any of the mistakes made by them. **Note that we strongly encourage you to only use them to help you when you get stuck instead of solving the entire homework.** Academic dishonesty (copying code, not citing work etc.) will not be tolerated. **By submitting your assignment, you agree to abide by the Koç University codes of conduct.**

Introduction

In this homework, you are going to implement several control algorithms, along with an exploration approach, presented below on grid worlds. The environment is handled for you. Furtermore, we are providing the implementation of several prediction algorithms to help you. The algorithms are:

- ϵ-greedy action selection
- Dynamic Programming
  - Policy Evaluation: Implemented for you
  - Q-Value Iteration
  - Policy Iteration
- Monte Carlo Methods
  - Monte Carlo Evaluation: Implemented for you
  - Monte Carlo Control
- Bootstrapping Methods
  - Temporal Difference Prediction - TD(0): Implemented for you
  - Sarsa
  - Q-Learning

Some of these algorithms are very similar so you can re-use a lot of your code. We also provide a lot of common code so you only need to implement the fundamentals. The amount of code you need to write is relatively low. However, there is some amount of code you need to read and play around with.

Implementation

There are three types of files; (1) files you can ignore, (2) files you should look at but not modify and (3) files you should modify. Most of these files have documentation. We briefly describe them below.

Files You Can Ignore

File Name	Description
environment.py	Base class that describes an RL environment
GraphicsGridworldDisplay.py	Includes code to draw the grid world
graphicsUtils.py	The main code to create and interact with the graphic grid world
gridWorld.py	The grid world MDP
parameterSchedulers.py[1]	Removed from the homework so you can ignore this
mdp.py	Base class that describes a MDP
textGridworldDisplay.py[2]	The text counterpart to graphics grid world.

Notes:

1. An earlier version of the homework required you to implement learning rate and epsilon scheduling. You are encouraged to play around with this. (Note from instructor: This may lead to obsession with converging to the exact values at which point you should stop)
2. The text grid world is not thoroughly tested.

Files You May Need to Go Over but not Modify

File Name	Description
baseAgents.py[1]	File that includes the base classes for the RL agents. You should familiarize yourself with the existing functionality.
main.py[2]	Contains code to run the agents. We go over how to call it in the document.
util.py[3]	Contains utilities used by the rest of the codebase. The values and q-values are represented with the `Counter` object defined here.

Notes:

1. You should especially go over the *baseAgents.py*.
2. We are going to give you commands on how to run *main.py*. You do not have to look at it but it is recommended.
3. The `Counter` object is very similar to Python's own counter (something like a dictionary that by default returns 0 if the key does not exist). Look at *util.py* if you want to see the details.

Files You Need to Modify

File Name	Description
policies.py	The file that contains various policies. Code: `epsilonGreedyAction` method of the `BasePolicy` class.
dpAgents.py	The file that includes the Dynamic Programming agents. Code: `run` method of the `QValueIterationAgent` and `PolicyIterationAgent` classes
mcAgents.py	The file that includes the Monte Carlo agents. Code: `run` method of the `MonteCarloControlAgent` class
tdAgents.py	The file that includes the Temporal Difference agents. Code: `run` method of the `SarsaAgent` and `QLearningAgent` classes

The files have detailed documentation to guide you. You are asked the implement the `run` function of each agent. You need to implement a total of five agents. We provide a prediction algorithm from each family to help you with the process. **You need to select the correct policy representation for some of the agents in the constructor.**
You can add your own fields and functions to the classes given in these files.

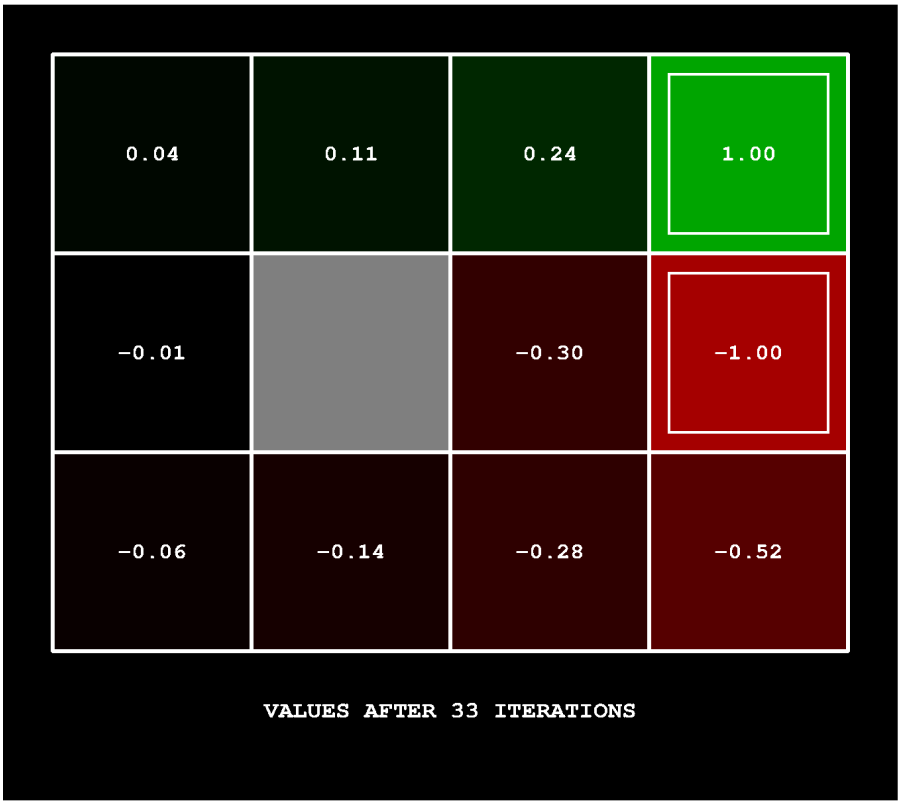

Figure 1: Values for the random policy, obtained using the DP policy evaluation algorithm.

Policy Evaluation

An agent that implements the policy evaluation algorithm is given in the *dpAgents.py* file, described by the `PolicyEvaluationAgent` class. This is implemented for you. Run the following code from the command line to call the agent:

```
python main.py -a pe
```

This runs the policy evaluation algorithm for the uniformly random (all legal actions are equally likely) policy. There should be a *latest_pe_BookGrid.png* file under the same folder as your main file. It should look like Figure 1.

Side Note: Run `python main.py -h` to see the available parameters.

Q-Value Iteration

An agent that should implement the Q-Value iteration algorithm is given in the *dpAgents.py* file, described by the `QValueIterationAgent` class. You need to implement it. Run the following code from the command line to call the agent:

```
python main.py -a qi
```

This runs the Q-Value iteration to obtain the optimal values. There should be a *latest_qi_BookGrid.png* file under the same folder as your main file. It should look like Figure 2.

Policy Iteration

An agent that should implement the policy iteration algorithm is given in the *dpAgents.py* file, described by the `PolicyIterationAgent` class. You need to implement it. Run the following code from the command line to call the agent:

```
python main.py -a pi
```

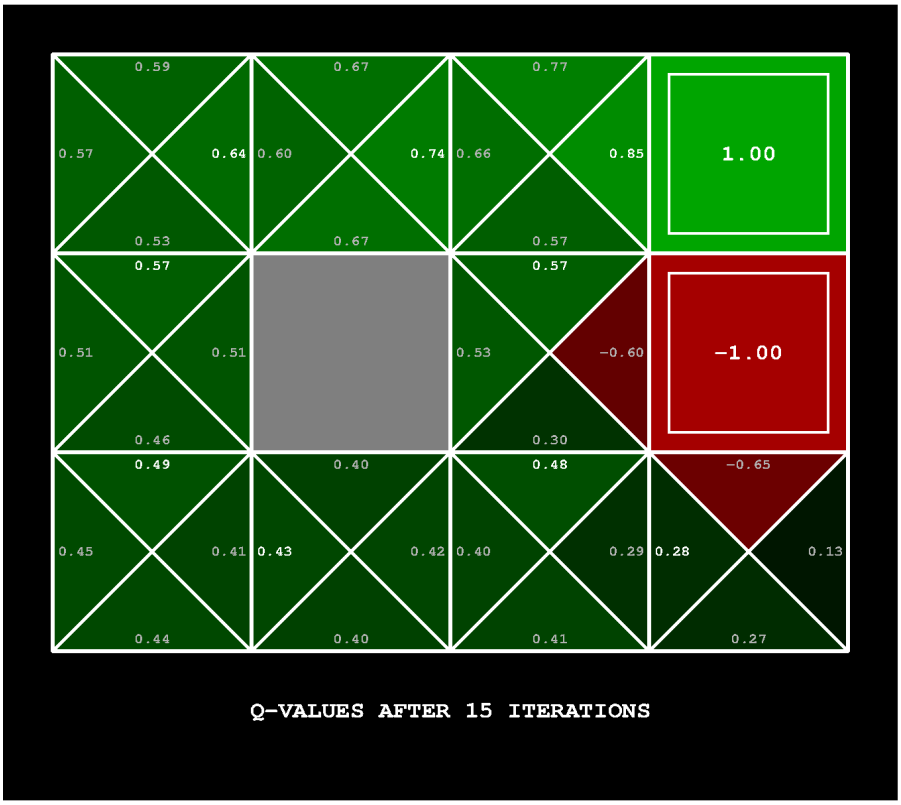

Figure 2: The optimal q-values (within a threshold), obtained using the Q-Value iteration algorithm.

This runs the policy iteration to obtain the optimal values. There should be a *latest_pi_BookGrid.png* file under the same folder as your main file. It should look like Figure 3.

Monte Carlo Prediction

An agent that implements the every-visit MC prediction algorithm is given in the *mcAgents.py* file, described by the `MonteCarloPredictionAgent` class. This is implemented for you. Run the following code from the command line to call the agent:

```
python main.py -a mcp -k 1000 -q
```

This runs the every-visit MC prediction algorithm for the uniformly random (all legal actions are equally likely) policy for 1000 episodes. There should be a *latest_mcp_BookGrid.png* file under the same folder as your main file. It should look like Figure 4. Note that the obtained values are close the values obtained with the policy evaluation algorithm (Figure 1). Note that we are not decaying the learning rate.

`-q parameter`: This means "quiet". If you do not pass this, you can see the actions that the agent takes. It may be useful for debugging but it makes the learning very slow.

Monte Carlo Control

An agent that implements the every-visit MC control algorithm is given in the *mcAgents.py* file, described by the `MonteCarloControlAgent` class. You need to implement it. In addition to filling the `run` function, you need to pick the correct policy representation (look at *policies.py*) and call it correctly while the agent is running. Run the following code from the command line to call the agent:

```
python main.py -a mcc -k 3000 -q
```

This runs the every-visit MC control algorithm for 3000 episodes. There should be a *latest_mcc_BookGrid.png* file under the same folder as your main file. It should look like Figure 5. Since we are not decaying the learning rate and the ϵ, we do not converge to the optimal q-values (as in Figure 2).

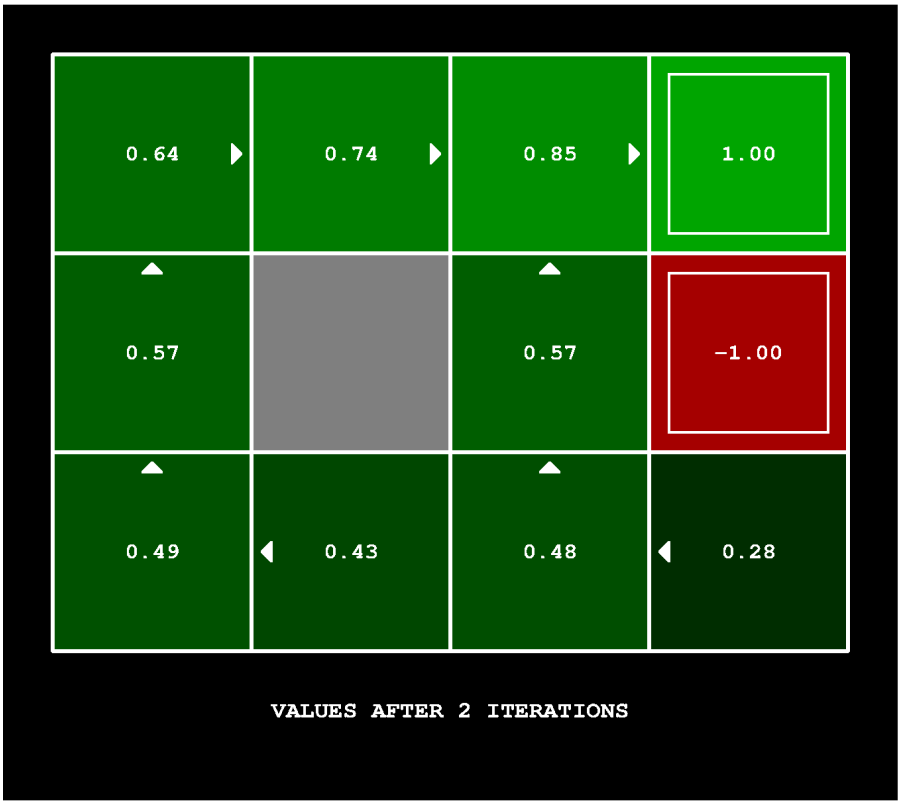

Figure 3: The optimal values (within a threshold) and the optimal policy, obtained using the policy iteration algorithm.

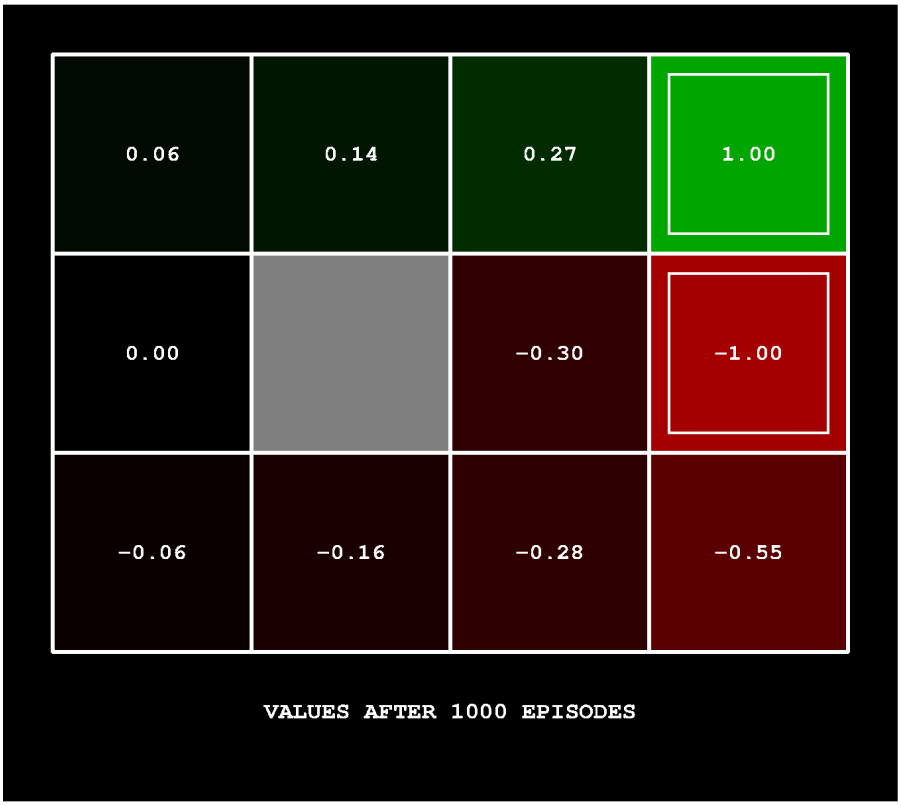

Figure 4: Values for the random policy, estimated by the every-visit MC prediction algorithm with 1000 episodes.

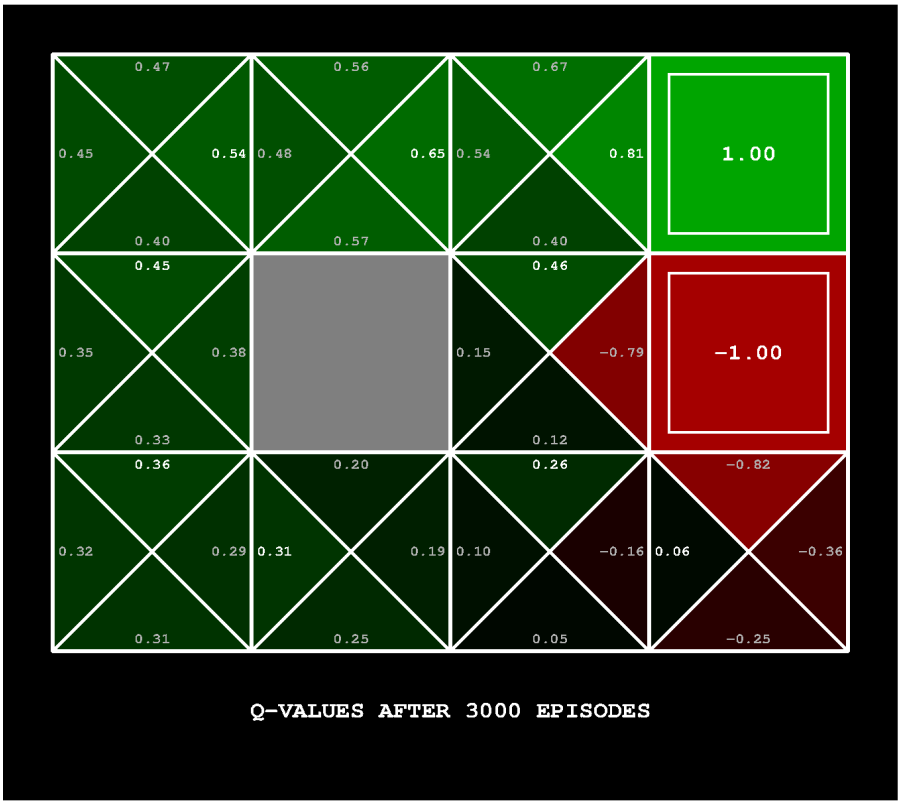

Figure 5: The q-values, obtained using the every-visit MC control algorithm with 3000 episdes.

TD(0)

An agent that implements the vanilla temporal difference prediciton algorithm is given in the *tdAgents.py* file, described by the `TemporalDifferencePredictionAgent` class. This is implemented for you. Run the following code from the command line to call the agent:

```
python main.py -a td -k 1000 -q
```

This runs the TD(0) for the uniformly random (all legal actions are equally likely) policy for 1000 episodes. There should be a *latest_td_BookGrid.png* file under the same folder as your main file. It should look like Figure 6. Note that we are not decaying the learning rate but it should look like the outcome of the other prediction algorithms.

SARSA(0)

An agent that implements the SARSA algorithm is given in the *tdAgents.py* file, described by the `SarsaAgent` class. You need to implement it. In addition to filling the `run` function, you need to pick the correct policy representation (look at *policies.py*) and call it correctly while the agent is running. Run the following code from the command line to call the agent:

```
python main.py -a sr -k 3000 -q
```

This runs the SARSA algorithm for 3000 episodes. There should be a *latest_sr_BookGrid.png* file under the same folder as your main file. It should look like Figure 7. Since we are not decaying the learning rate and the ϵ, we do not converge to the optimal q-values.

Side Note: An earlier version of the algorithm asked you to do SARSA with "optimistic initialization". We recommend that you try it by initializing all the q-values to 1 to see this effect.

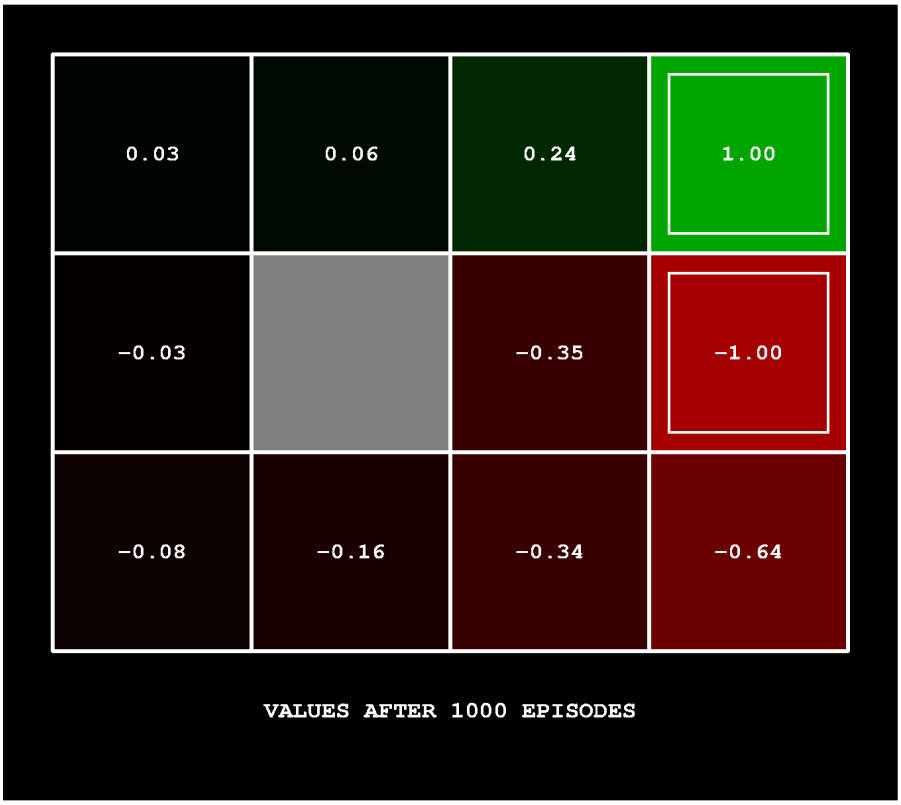

Figure 6: Values for the random policy, estimated by the TD(0) algorithm with 1000 episodes.

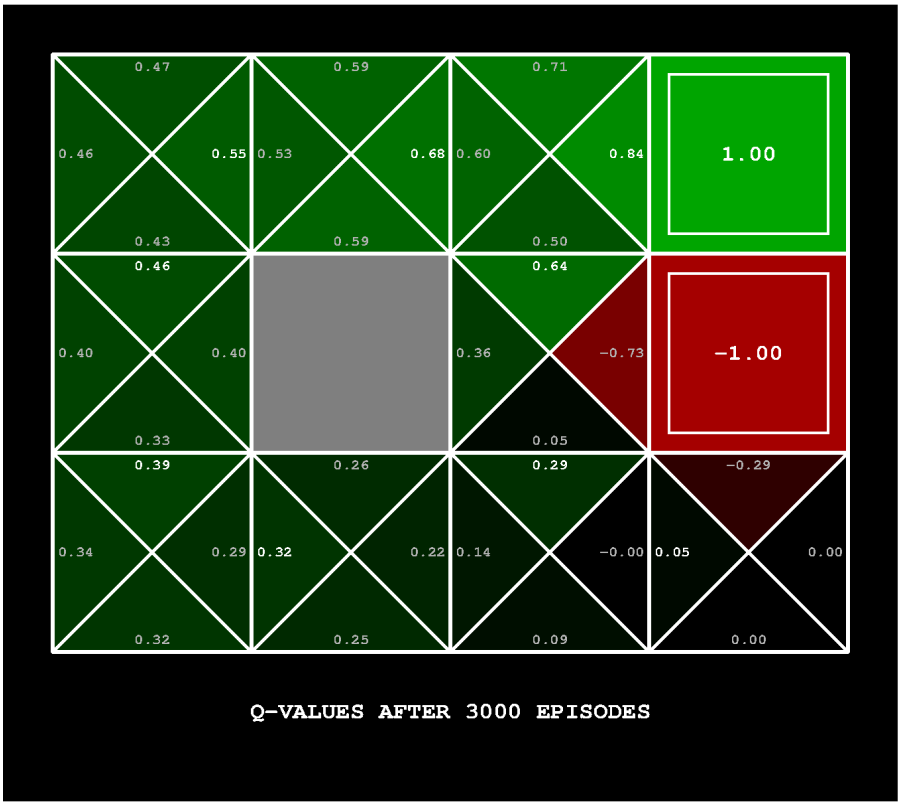

Figure 7: The q-values, obtained using the SARSA algorithm with 3000 episdes.

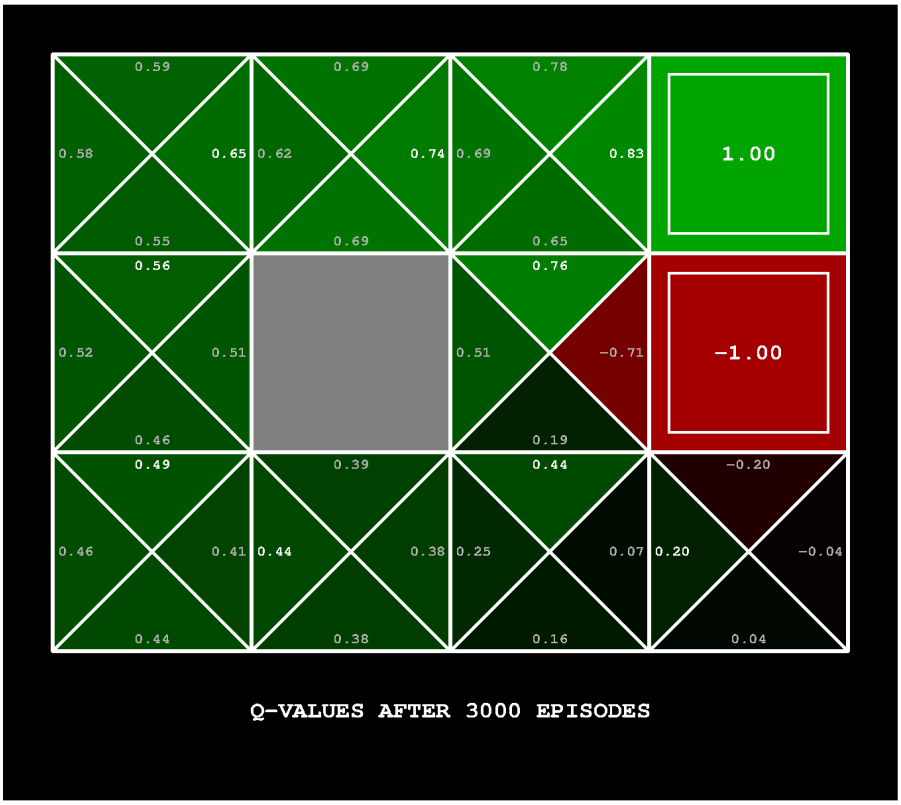

Figure 8: The q-values, obtained using the Q-Learning algorithm with 3000 episdes.

Q-Learning

An agent that implements the Q-Learning algorithm is given in the *tdAgents.py* file, described by the `QLearningAgent` class. You need to implement it. In addition to filling the `run` function, you need to pick the correct policy representation (look at *policies.py*) and call it correctly while the agent is running. Run the following code from the command line to call the agent:

```
python main.py -a ql -k 3000 -q
```

This runs the Q-Learning algorithm for 3000 episodes. There should be a *latest_ql_BookGrid.png* file under the same folder as your main file. It should look like Figure 8. Since we are not decaying the learning rate and the ϵ, we do not converge to the optimal q-values.

External Libraries

We are using the Python Imaging Library to save the grid at the end of runs as a PNG file. You can either install PIL or Pillow (through pip or conda) for this. If you can't, comment out the last line of the *main.py* and uncomment the previous one which saves the grid as an EPS file (which requires you to have an EPS viewer in Windows).

More Grids

There are other grids available in the code. You are encouraged to play around with them as well to see the behaviors of the algorithms or for debugging. You can call these grids with the `-g` argument and case sensitive options as `BookGrid, BridgeGrid, CliffGrid, MazeGrid`. A few examples:

- Monte Carlo Control with the BridgeGrid: `python main.py -a mcc -k 3000 -q -g BridgeGrid`
- SARSA with CliffGrid: `python main.py -a sr -k 3000 -q -g CliffGrid`
- Q-Value Iteration with MazeGrid: `python main -a qi -g MazeGrid`

The output file will have the corresponding maze in its name (e.g. *latest_mcc_BridgeGrid.png*).

Submission Instructions

- You are going to submit a **single** compressed archive through the blackboard site. The file can have *zip*, *rar*, *tar*, *tar.gz* or *7z* format.
- This compressed archive should include the following files; *policies.py*, *dpAgents.py*, *mcAgents.py* and *tdAgents.py*.
- You are fine as long as the compressed archive has the required files anywhere inside it within 4 folder levels. Other files will be deleted and/or overwritten.
- Code that does not run (e.g. due to syntax errors), terminate (e.g. due to infinite loops) or blows up the memory will not receive any credits.
- **Important:** Download your submission to make sure it is not corrupted and it has your latest report/code. You are only going to be graded by your blackboard submission. We are not going to accept a version on your harddrive or cloud storage.

Best of luck and happy coding!