Box World 2 (Class 5 -1A)

Gonçalo Pereira (201705971)
Faculdade de Engenharia
da Universidade do Porto)
Porto, Portugal
up201705971@fe.up.pt

Eduardo Macedo (201703658)
Faculdade de Engenharia
da Universidade do Porto)
Porto, Portugal
up201703658@fe.up.pt

António Dantas (201703878)
Faculdade de Engenharia
da Universidade do Porto)
Porto, Portugal
up201703878@fe.up.pt

***Abstract*—This article contains the description of a project whose objective is to solve an optimization problem using two different reinforcement learning algorithms (Q-learning and SARSA)**

***Index Terms*—Artificial Intelligence, Q-Learning, SARSA, Reinforcement Learning**

I. Introduction

Reinforcement learning (RL) is an area of machine learning concerned with how software agents ought to take actions in an environment in order to maximize the notion of cumulative reward. Reinforcement learning is one of three basic machine learning paradigms, alongside supervised learning and unsupervised learning.

II. Description of the problem

Our project consists in replicating the famous Box Word 2 game, as well as implementing an AI that can beat it. The game consists in a series of puzzles, where the player has to reach the exit of the stage in order to advance to the next level. Various obstacles are placed around the arena, making the goal harder. At some stages, there are holes, which can be filled with the various boxes placed around.

Our AI consists in an agent solving the puzzles using SARSA and Q-Learning.

III. Approach

We started by implementing the Q-Learning algorithm by adding 4 values to each tile, associating each floor block of the arena to the corresponding four possible actions (left, right , up or down).

```
self.floor = [
  [pygame.Rect(100, 100, 25, 25),0,0,0,0],
  [pygame.Rect(100, 125, 25, 25),0,0,0,0],
  [pygame.Rect(100, 150, 25, 25),0,0,0,0],
  [pygame.Rect(100, 175, 25, 25),0,0,0,0],
  ...
]
```

Then we came to the conclusion that this approach would only make sense for a board that was static. In our game this is not the case because of the moving blocks scattered around the arena and, therefore, this implementation would not be the ideal.

We then thought that the most effective approach would be to create a python list with several lists that would contain the state as the first element, followed by Q-values corresponding to the four possible actions.

State 1	2.231	-0.172	4.845	4.530
State 2	6.217	4.505	5.122	5.495
State 3	2.231	-0.171	4.845	4.531

Table 1: Q-table representation

This was crucial for our implementation since that, when we move a block, the path to the solution may have been altered and therefore the q-values of the previous tile(s) should not be the same in this new display of the arena. So, when we change position, news "paths" can appear or disappear, adding new states to the q-table as we move along the map. With this implementation truly different states have unique values. A state differs not only by the player's position but also by the boxes and holes layout.

To update the values on our q-table we use the following formula. Alpha is the learning rate, gamma is the discount factor and the reward is a pre-calculated value based on the rules of the game.

```
q_instance[n] = q_instance[n] + alpha *
                (reward + gamma*q_instance[np] -
                    q_instance[n])
```

Code 1: Update rule

The value of q_instance[np] is what differentiates the SARSA and Q-Learning approach. Being an on-policy method, SARSA learns action values relative to the policy it follows, while off-policy Q-Learning does it relative to the greedy policy. Under some common conditions, they both converge to the real value function, but at different rates.

```
q_instance[np] = epsilon *
                    mean_value(q_instance[np])
                + (1 - epsilon) *
                    max_value(q_instance[np])
```

Code 2: SARSA

```
q_instance[np] = max_value(q_instance[np])
```

Code 3: Q-Learning

Regarding our reward policy, we started off the project with the following values

Reach the exit	10
Block the exit	-20
Take a step	-0.1

Table 2: Reward policy

This policy gave us positive results across various levels. The player receives a slight negative score for taking a single step, resulting in a constant exploration of the level's surface by prioritizing states that have not yet been visited. The positive reward for reaching the exit tile gives it something to accomplish and makes that its prime objective. The harsh negative reward for blocking the exit is a good example of what it should never do, avoiding this kind of behaviour. Besides this, we also implemented an optional reward instance that can be introduced in the program's command line.

Approached the exit	-0.08

Table 3: Additional reward instance

What this does basically is give the A.I. a slightly higher reward if the step it takes is in the direction of the exit. This comes in handy in levels where that is the main objective, lessening the number of episodes needed to stabilize the Q-Table.

Finally, we also implemented an Exploration vs. Exploitation mechanism. The agent needs to take actions that prove to be effective according to the reward system, but to discover such actions it has to try ones never selected before. The variable epsilon serves as a ratio to cover that. It serves as the probability of the agent making an action based on the Q-table entry. This variable can be introduced on the program's command line by the user, but it is at 0.3 by default. This means it has a 30% chance of exploring in a given action and a 70% chance of exploiting.

```
if random.uniform(0, 1) < epsilon:
    # explores
else:
    # exploits
```

Code 4: Exploration vs. Exploitation

IV. Experimental evaluation

With our final implementation we managed to achieve good results in various arenas using both reward policies implemented. In this section we will show two different types of graphs: one that shows the rewards earned as steps increased (each iteration); and another showing and many steps the agent did in each progressing episode. We made several runs and created various graphs on all levels that will all be included in the annex, but for this report we will be focusing on the difference of the algorithms on level 2 and the different use in reward policies in level 3. The picture below shows us the arena of level 2.

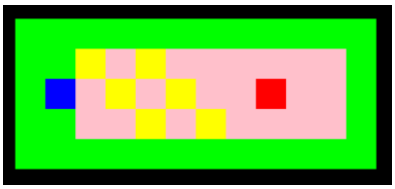
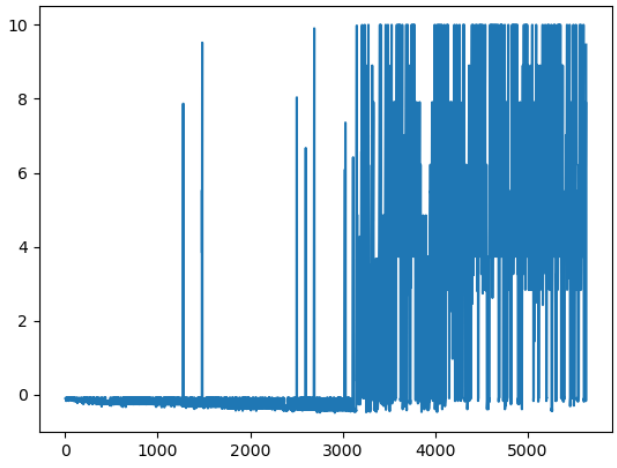
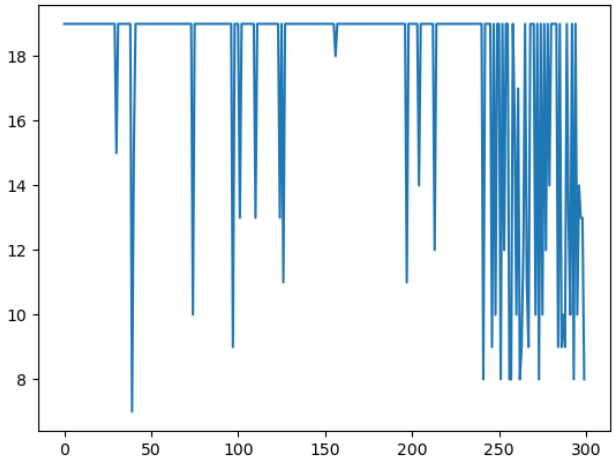


Figure 1: Level 2 ingame



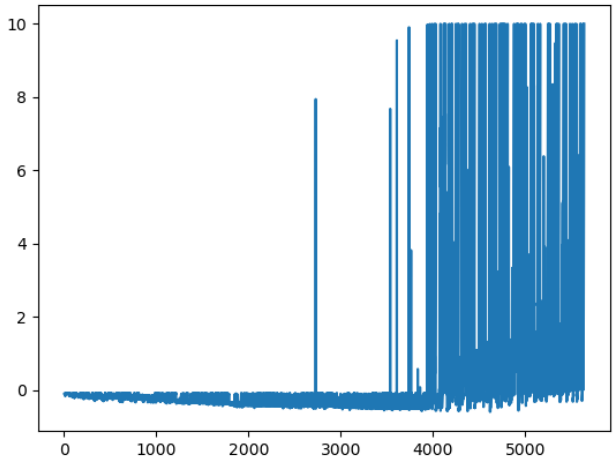
Graph 1: Rewards along iterations in level 2 using Q-Learning with 400 episodes

In this graph we can see the rewards the agent receives when it takes a step (reward is +10 when it reaches the exit). In the early episodes the agent is not able to find the exit and therefore no positive reward was given. A long time can pass before the agent reaches the exit for the first time. However, each time the solution is found, the agent becomes increasingly more likely to achieve success in the next iteration. This can be seen by the white spaces between the graph spikes (when y = 10 and the exit is found): they become increasingly shorter in each iteration/episode. Given the nature of reinforcement learning, this is a behaviour which will be noticed through all graphs of this kind, throughout both algorithms in all levels.

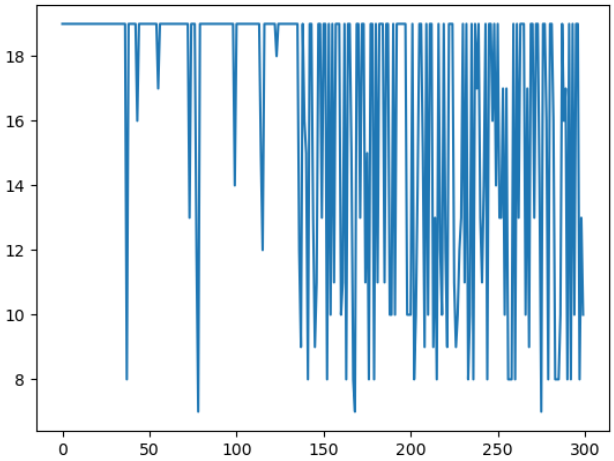


Graph 2: Final Reward in each episode in level 2 using Q-Learning with 400 episodes

Here we can observe that as episodes go by, the number of steps the agent takes in each episode tends to decrease as the agent becomes more efficient at finding a solution. In the two graphs below we can see the same analysis, this time using the SARSA algorithm.



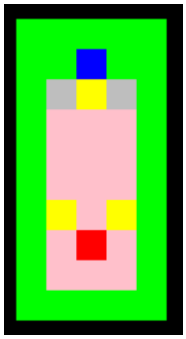
Graph 3: Rewards along iterations in level 2 using SARSA with 400 episodes



Graph 4: Final Reward in each episode in level 2 using SARSA with 400 episodes

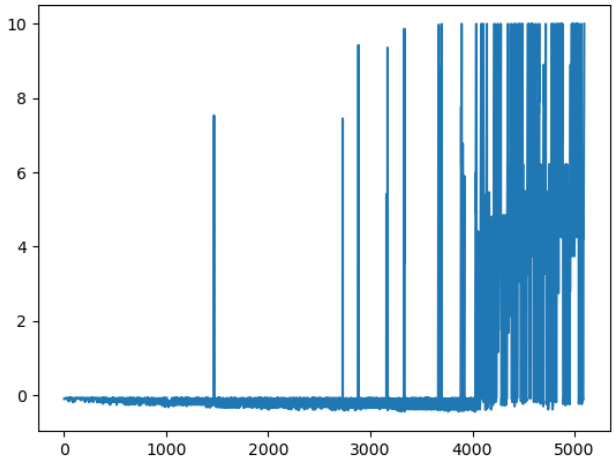
Comparing the two algorithms (not only using these two graphs but and also using several runs of the program in this specific level), we can see, although when SARSA finds several solutions it tends to be more consistent in finding the exit in later stages, Q-Learning seems to find a solution much faster than SARSA. The same can be seen in the number of steps taken by the agent in each algorithm, as these tend be more frequently lower in Q-Learning throughout the iterations.

The image below represents the level 3 ingame.

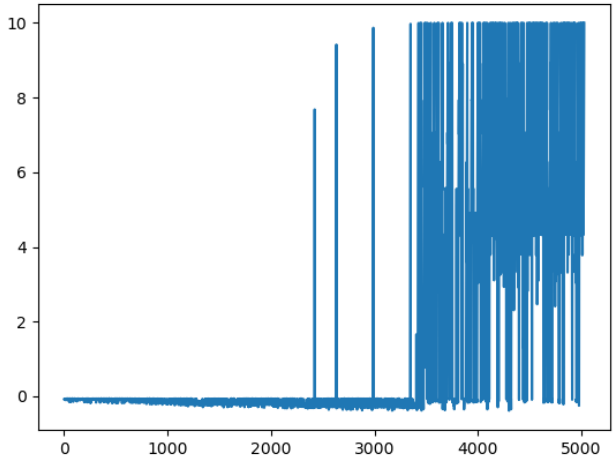


Picture 2: Level 3 ingame

Here we are looking to find how different reward policies effect the agent's results. Two graphs will be displayed below of rewards the agent received throughout the iterations. The second graph uses a different reward policy then the previous examples.



Graph 5: Level 3 graph rewards per iteration with "-rw false" using QLearning



Graph 5: Level 3 graph rewards per iteration with "-rw true" using QLearning

In the second graph, besides the reward policies previously referred, there is also another: whenever the agent moves closer to the exit, instead of the reward being -0.1 it "only" receives -0.08. This may seem like a small change but depending on the level, encouraging this "aggressive" behaviour can lead to good results which is the case in this level. We can observe in the second graph that not only the solution was found earlier compared to the first graph but it also needed less number of successes to consistently find the solution throughout the iterations.

V. Conclusions

From the resolution of this project we can conclude that Reinforcement Learning is a very powerful tool. If we give the agent enough time to train, in theory he should be able to find an optimal solution to every level we put him in. The amount of time the training takes however is dependent on the size of the arena and the number of boxes it has, since this will exponentially increase the number of states. It is quick and effective on small to medium sized maps but not recommended on the later levels of the game. With our testing along the development of the project, we could see that SARSA was more conservative when trying to find the optimal path. Q-Learning tends to try solutions that might not look optimal at first glance. This makes it so that the two algorithms shine depending on the level. The more linear the level is, SARSA tends to do better and be more consistent while Q-Learning has a better performance in levels where some deviation might be required.

References

[1] Q-Learning demo
https://github.com/llSourcell/q$_l earning_d emo$
[2] StackOverflow SARSA AND Q-Learning
https://stackoverflow.com/questions/6848828/what-is-the-difference-between-q-learning-and-sarsa
[3] BoxWorld 2
http://hirudov.com/others/BoxWorld2.php
[4] Wikipedia Q-Learning
https://en.wikipedia.org/wiki/Q-learning