AQUARIUM

REINFORCEMENT LEARNING

Nuno Oliveira up201806525

Luís Miguel Pinto up201806206

Marcelo Reis up201809566

SPECIFICATION OF THE WORK

The rules of Aquarium are simple:

- The puzzle is played on a rectangular grid divided into blocks called "aquariums".
- The objective is to "fill" the aquariums up to a certain level or leave it empty.
- The water level in each aquarium is one and the same across its full width.
- The numbers outside the grid show the number of cells that must be filled horizontally and vertically.
- We must use reinforcement learning algorithms to solve this game.

RELATED WORK AND REFERENCES

- Repository with Software to a Reinforcement Learning approach to solve sudoku
  https://github.com/kiruthihan10/Creating-Sudoku-with-reinforcement-learning

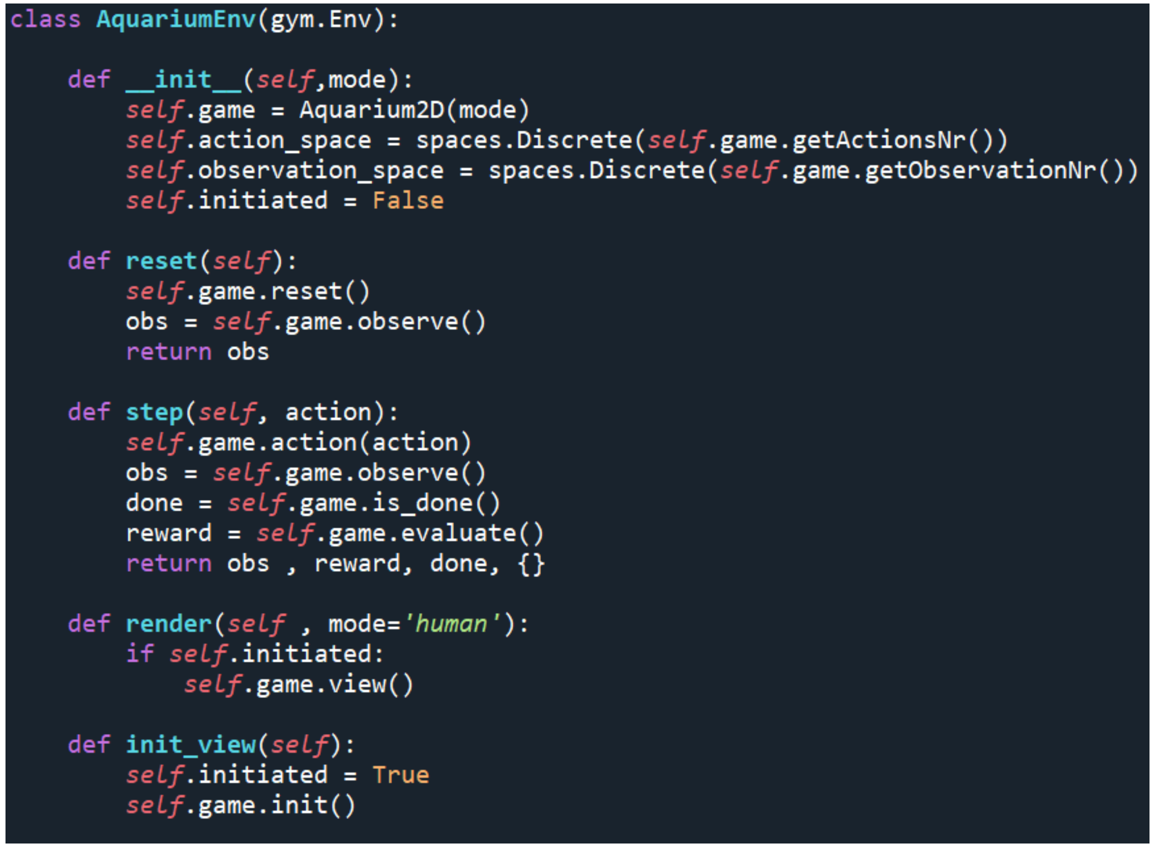
- Paper on "Reinforcement Learning For Constraint Satisfaction Game Agents (15-Puzzle, Minesweeper, 2048, and Sudoku)"
  https://arxiv.org/abs/2102.06019

TOOLS AND ALGORITHMS

- The tools that we are going to use in the assignment are:
  - Python
  - OpenAI Gym
  - Pygame
- The algorithms that we implemented:
  - Q-Learning
  - SARSA
- Actions:
  - There is N*2 actions, N being the number of aquariums in the puzzle
  - The actions will be to paint and unpaint the next level on one of the aquariums
- Rewards:
  - Each action taken: **Reward –1**
  - Solving the puzzle: **Reward +100**

IMPLEMENTED WORK

- Class AquariumEnv implements gym.env interface that allows us to build our own custom enviroment.
- This class contains the main methods that will be called in the reinforcement learning algorithms.
- Reset and Step methods are used inside the algorithms to change the current state of the game.
- On the other hand, render and init_view are used to display the game board together with algorithm choices.

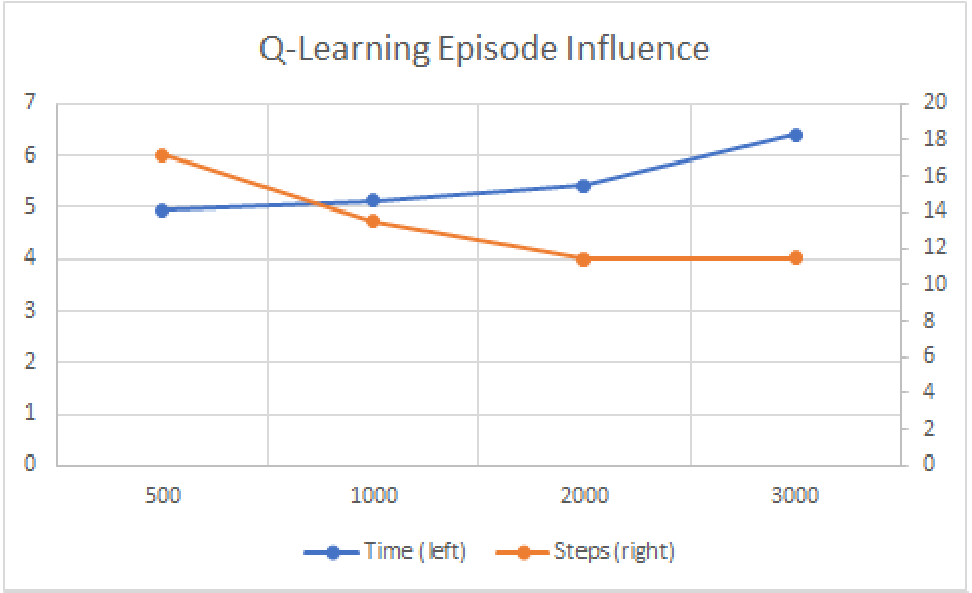


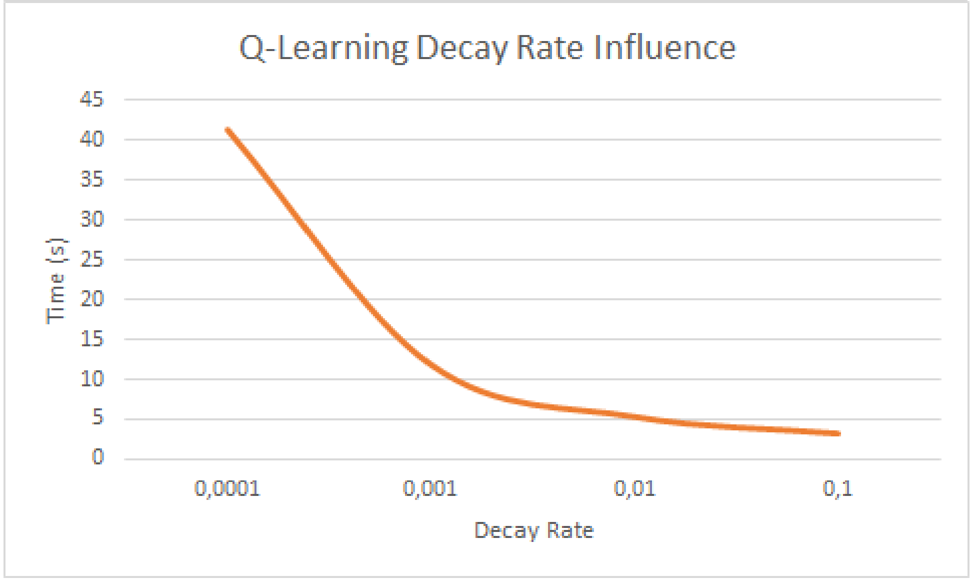
```python
class AquariumEnv(gym.Env):

    def __init__(self,mode):
        self.game = Aquarium2D(mode)
        self.action_space = spaces.Discrete(self.game.getActionsNr())
        self.observation_space = spaces.Discrete(self.game.getObservationNr())
        self.initiated = False

    def reset(self):
        self.game.reset()
        obs = self.game.observe()
        return obs

    def step(self, action):
        self.game.action(action)
        obs = self.game.observe()
        done = self.game.is_done()
        reward = self.game.evaluate()
        return obs , reward, done, {}

    def render(self , mode='human'):
        if self.initiated:
            self.game.view()

    def init_view(self):
        self.initiated = True
        self.game.init()
```

IMPLEMENTED WORK

- The game state is represented as an array with the number of levels painted for each aquarium. The maximum number of states can be caculated by multipling the number of levels of each aquarium.
- To avoid the prior calculation of the states each time the program starts we chose to create a new state when an action is made, i.e, the game states are generated in runtime.
- As previously stated, the rewards for each action are –1 this way the algorithm will learn by himself what are the best actions to make to achieve the puzzle solution.

Q-LEARNING PARAMETERIZATIONS





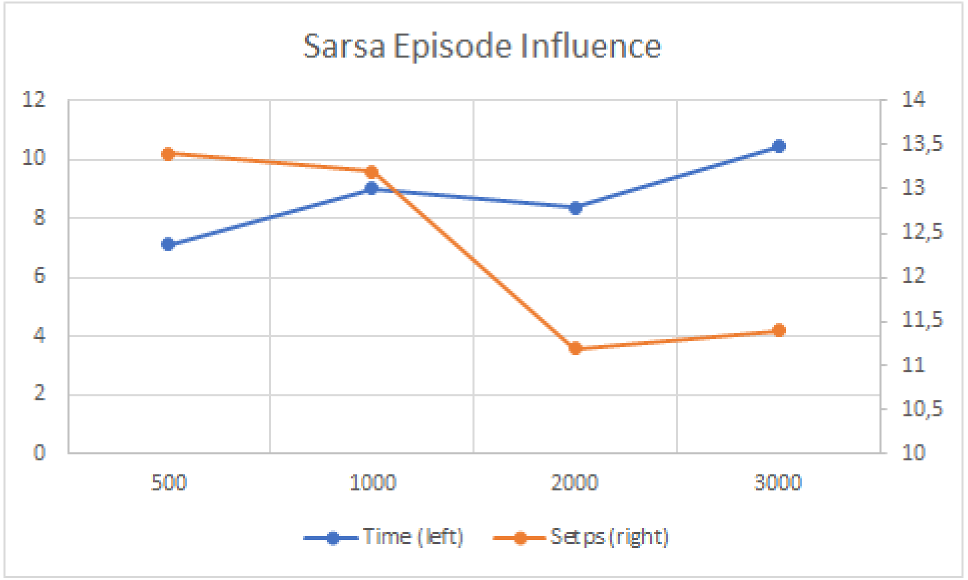
The quality of the solution increases the higher the number of episodes until a certain point where it reaches the best solution possible.

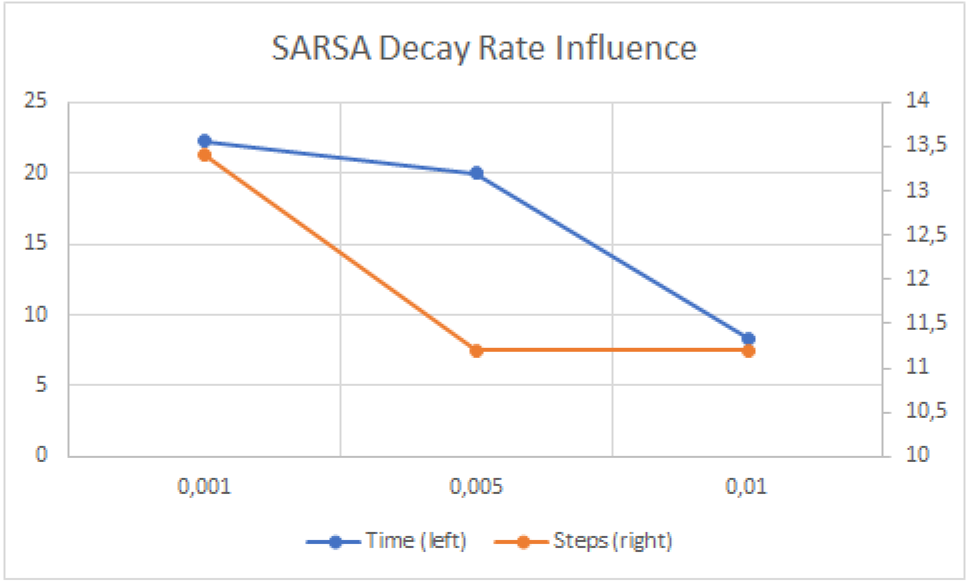
More episodes lead to more iterations and a better(hopefully) solution. However the diference is not big because once the alg. converges the episodes take less time.

It's easy to understand that the time spent training the algorithm increases with the decrease of the decay rate.

That can be explained because the smaller the decay rate the more random the choice will be at the end of the process and so it will be harder to achieve a right solution.

SARSA PARAMETERIZATIONS





The quality of the solution increases the higher the number of episodes until it converges.

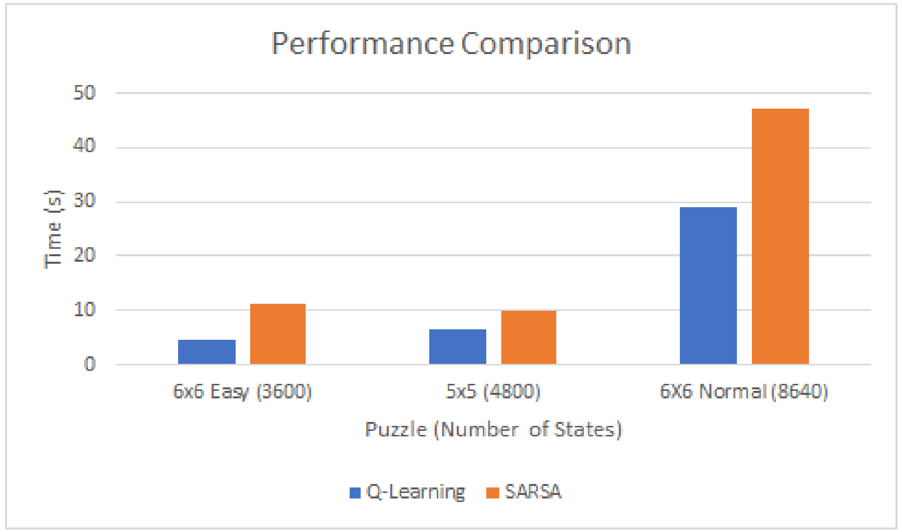
More episodes lead to more iterations and a better(hopefully) solution but also more time.

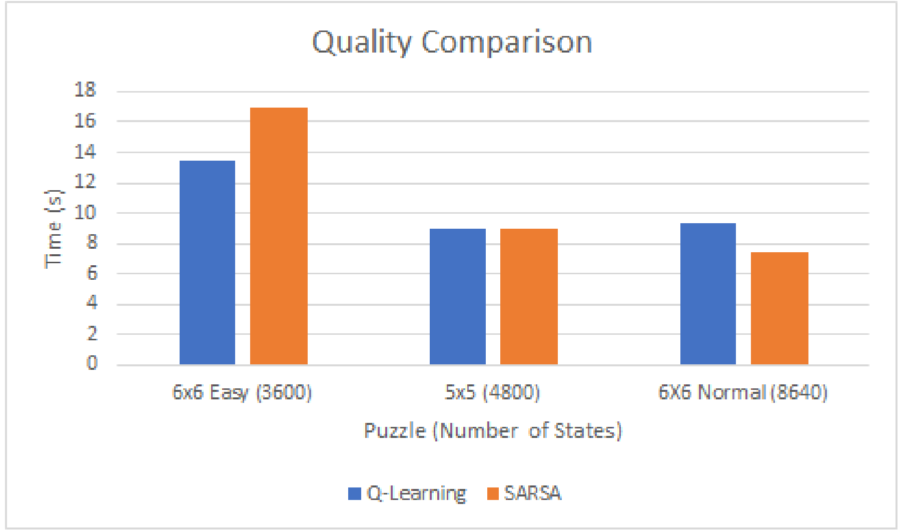
It's easy to understand that the time spent training the algorithm increases with the decrease of the decay rate.

For this algoritm, a decay rate of 0.005 leads to a 20% incorrect solutions and 0.001 to a 30% of incorrect solutions.

EFFICIENCY COMPARISON

- Both algorithms could only generate a valid game with more than 50 steps per episode.
- Changing the number of steps for each episode does not affect in a significant way the quality or speed of the solutions of both algorithms and the measurements were not conclusive enough.
- Through our testing we concluded that SARSA normally takes more time than Q-Learning to reach a solution, often taking a longer path to reach said solution.





CONCLUSIONS

- From the results obtained during the project it is possible to conclude that Q-Learning and Sarsa appear to be quite similar, despite Sarsa showing worse results than Q-Learning. This is probably caused by the lower episode and step values, because Sarsa tends to work better the more it explores a state.
- Due to the greedy action selection, both algorithms converge to the optimal policy, as expected, reaching optimal solutions most of the times.
- Through our tests we also concluded that Q-learning is a bit more inconsistent in the quality of the results even though it has better result more often.
- Q-learning showed a better overall performance, being able to solve the problem almost every time, in the other hand, SARSA couldn't always achieve a valid solution.