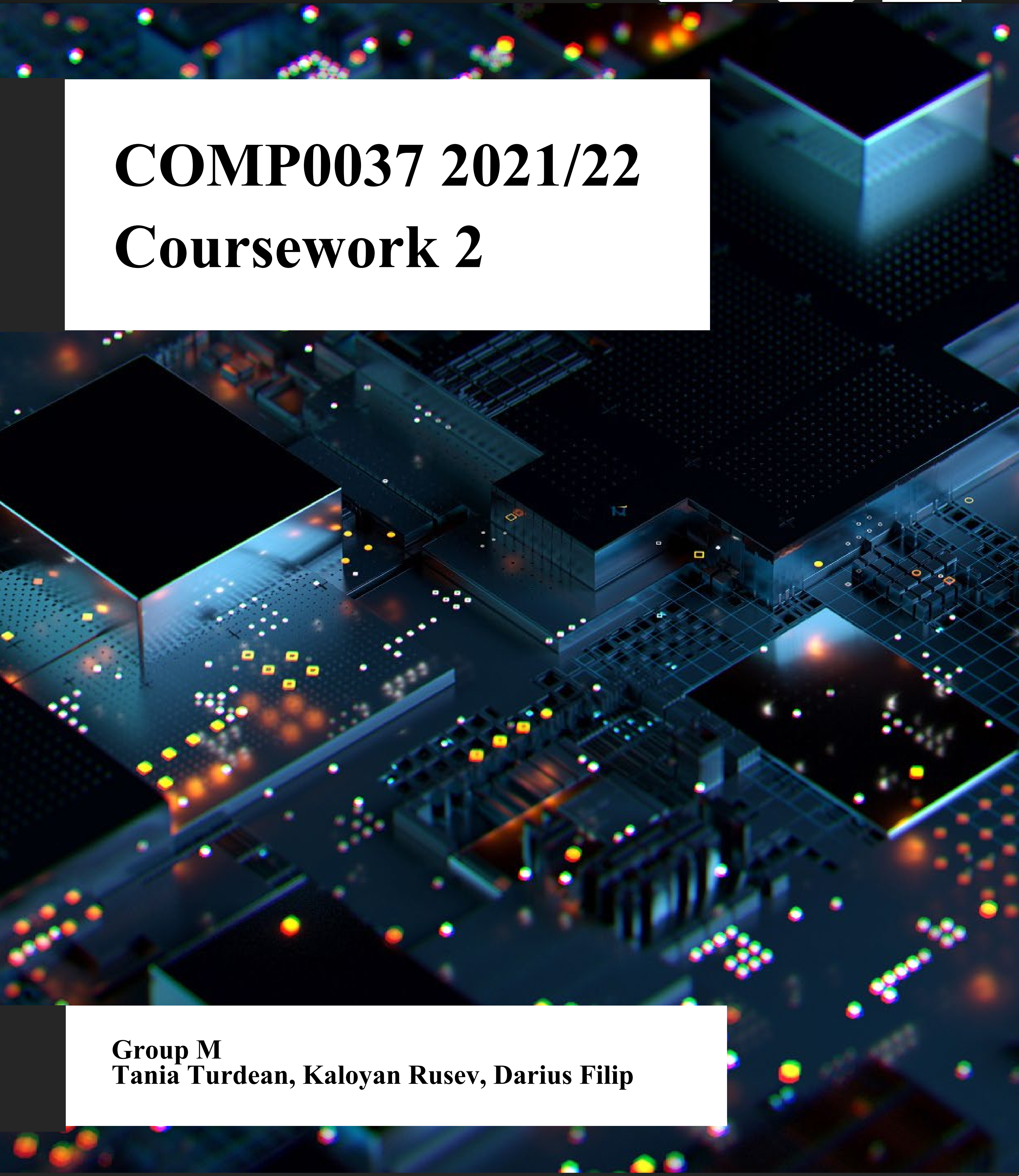
DEPARTMENT OF COMPUTER SCIENCE

UCL



COMP0037 2021/22 Coursework 2

Group M
Tania Turdean, Kaloyan Rusev, Darius Filip

APRIL 19th, 2022

Question 1

a. Monte Carlo (MC) approaches are computational algorithms used in reinfrocement learning that rely on a system being run multiple times with quasi-random inputs to obtain output results, which are then measured and analysed (i.e., averaged) (1). These approaches provide a way to extract information about the behaviour of a system directly from the data.

   Advantages over policy and value iteration:

   - MC approaches are potentially better because they work on the principle of learning from experience. In essence, the state-transition values can be approximated by sample means, by running many scenarios and analysing outputs.
   - Unlike dynamic programming algorithms, they do not need a model of the environment (i.e., of the rewards and state transitions) because they use an empirical distribution.
   - They are not required to sample across the entire state space, which can become computationally expensive if the state space becomes very large.
   - Can be on or off policy.
   - Enable real-time learning.
   - Provide probabilistic results, as well as the opportunity to perform scenario analysis.

b. MC techniques are used to empiricaly quantify how well a policy is performing. First, as it can be seen in Figure 1, there is an initial state (S_0) and a terminal state that delimitate the episodes. As the policy π is run, new actions (A_i), rewards (R_i) and states (S_i) are generated, until the goal is reached. To evaluate the policy, multiple episodes must be experienced, with the action-value function approaching q_* asymptotically (1). To determine the value of the policy V(s_t), the expected value of the return G_t over π must be calculated, following *equations 1* and *2*.

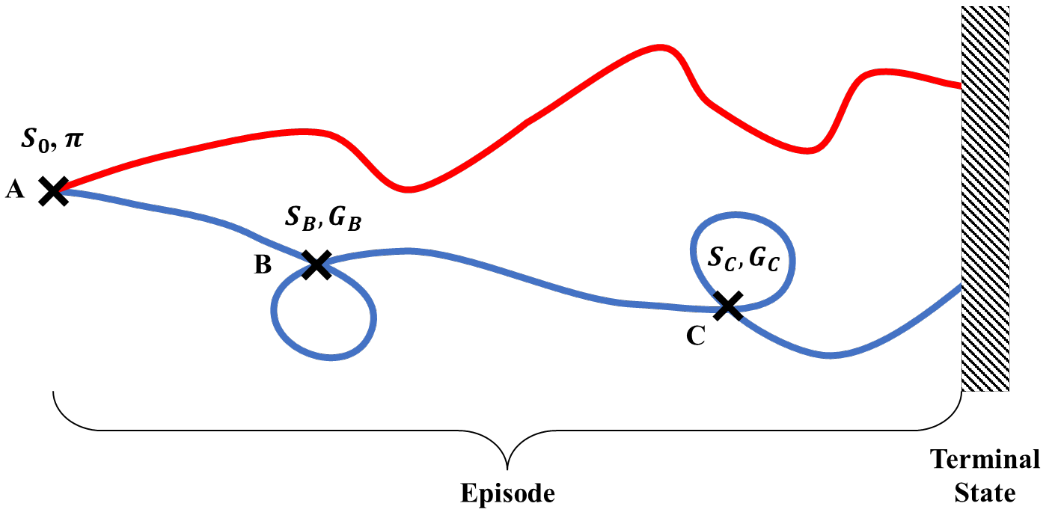


Figure 1 – Trajectories of different Monte Carlo Methods Episodes

$$V(S_t) = V(S_t) + \alpha\,[G_t - V(S_t)] \tag{1}$$

$$G_t = R_{t+1} + \underbrace{\gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots}_{\text{Episodic terms}} \tag{2}$$

The step size $\alpha \in [0,1)$ handles the non stationary environments. As $\alpha \to 1$, changes in the environment are handled better, but the errors are higher.

Sometimes the trajectory generated by the policy might revisit a state (points B and C from Figure 1) and a new reward value will be generated. However, if a first visit algorithm is employed, the value of the policy $V(S_t)$ will not be updated, and the value recorded during the first visit will be kept. If an every visit algorithm is used, the value of the policy will be updated every time the trajectory revisits that state.

The pseudocode that gives the first visit algorithm can be seen below. By removing line 2, the algorithm turns into every visit. What can be observed is that the reward value is always updated, regardless the algorithm.

```
$G \leftarrow \gamma G + R_{t+1}$
$Unless\ S_t\ a[[ears\ in\ S_0, S_1, \dots, S_{t-1}$:
$Append\ G\ to\ Return(S_t)$
$V(S_t) \leftarrow average(Return(S_t))$
```

c. Once the MC evaluation has been performed, the policy can be extracted using value iteration. Model-free policy iteration must be used to estimate the policy π_* itself, since the state-transition model is unknown. It is known that the optimal value function is given by *equation 3*, and the corresponding policy that can achieve this can be seen in *equation 4*. For this approach, once the choice of policy has been fixed, it cannot be changed again.

$$v_*(x) = max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s') \tag{3}$$

$$\pi_*(s) = argmax_a q_x(s, a) \tag{4}$$

Therefore, only the expected value of the state-action return pairs, $q_*(s, a)$, needs to be estimated. This is how the need to learn the state transition model $p(s', r|s, a)$ is removed.

As such, all the returns for the state and action pairs are averaged regardless of the policy enforced when these entries were registered. This MC algorithm is unable to converge to any suboptimal policy, because this would require the value function to converge to the value function given by that policy. Subsequently, this would make the policy change, which is not possible. This leads to policy freezing, which is the reason why this approach is not used in practice.

d. In order to overcome the effects of policy freezing, exploration can be adopted. There are two ways this approach can be implemented:
   - Exploring starts in order to start from different locations within the state space.
   - Using soft policies to explore random actions.

   Exploring starts simply involves picking starting locations at random, as seen in Figure 2. By starting different episodes in different places, more space can be explored. Assuming that one of the starting points leads to a “stuck” trajectory, by changing the initial location the chance of escaping policy freezing is increased. The downside of this method, however, is that it only creates a change in one spot (the starting point), and therefore might be inefficient if the state space is very large.

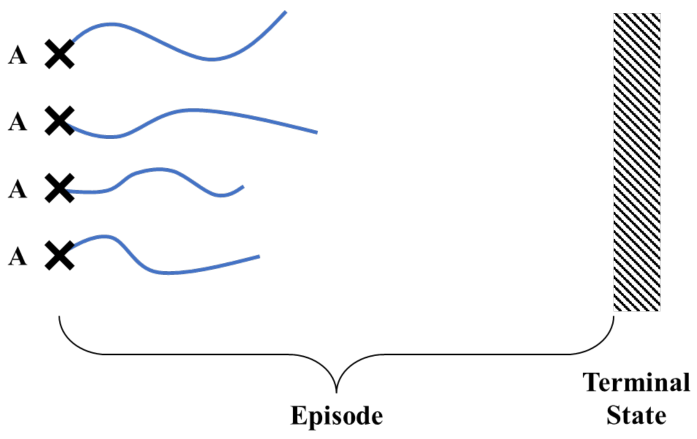


Figure 2 – Exploring Starts

   Soft policies remain random throughout exploration and do not converge to deterministic functions, which makes them adequate to avoid freezing. One common type of soft policies employ the ϵ-greedy strategy used traditionally in bandit problems, which can be seen in *equation 5*. This means that the agent can deviate from the trajectory at any given point, enabling continuous exploration within the space.

$$\pi(a|S_t) \leftarrow \begin{cases} 1-\epsilon+\dfrac{\epsilon}{|A(S_t)|} & if\ a = A^* \\ \dfrac{\epsilon}{|A(S_t)|} & if a \neq A^* \end{cases} \tag{5}$$

   Nonetheless, the optimal policy created by this approach will embed the ϵ-greedy algorithm, which selects non-optimal actions, therefore making the policy overly-cautious and non-optimal. To address this issue, the algorithm should be able to swith off the ϵ-greedy technique, resulting in two subsequent policies:
   - Off-Policy (behaviour policy, for learning): $b(A|S)$

- On-Policy (target policy, after learning is complete): $\pi(A|S)$

e. The main disadvantage of MC algorithms is being computationally inefficient. In order to increase accuracy, sometimes a high number of variables and inputs are necessary, and it can require a significant amount of time to compute the probabilistic solution. It can be observed that the convergence (O) of the mean (*equations 6* and *7*) is faster than that of the covariance (*equations 8* and *9*). This is also illustrated in Table 1, which displays the required sample size required to reach a certain accuracy for the mean and covariance. However, because N (the sample size) is finite, MC estimates will not be exact ($O \xrightarrow{N\to\infty} 0$).

$$\bar{x} \approx \frac{1}{N}\sum_{i=1}^{N} x_i \qquad (6) \qquad\qquad O\left(\frac{1}{N}\right) \qquad (7)$$

$$\sigma^2 \approx \frac{1}{N-1}\sum_{i=1}^{N} (x_i - \bar{x})^2 \qquad (8) \qquad\qquad O\left(\frac{1}{\sqrt{N}}\right) \qquad (9)$$

Table 1 – Comparison of sample size vs desired accuracy, for mean and covariance

Desired Accuracy	Sample Size (Mean)	Sample Size (Covariance)
0.1	10	100
0.01	100	10,000

Additionally, the solutions are heavily dependent on input data, so poor input parameters will give poor output results. Lastly, MC algorithms are episodic, so an entire episode needs to be run in order to obtain the expected return. This can be very time-consuming if episodes are long.

Question 2

a. Temporal Difference methods builds off two techniques: Monte Carlo (described in Question 1) and Bootstrapping. Bootstrapping computes estimates using successor estimates of state and action values. Both policy and value iteration use bootstrapping. By combining the methods, TD learning are an attempt to overcome the limitations of each: like Monte Carlo, it does not require a model; like bootstrapping, it doesn't require waiting until the episode has completed.

<u>Advantages over Monte Carlo methods:</u>

- It does not require the system to be episodic. By eliminating the need to run an entire episode, TD methods have an advatage over extremely long episodes. Moreover, some tasks aren't episodic. For example, a data centre task scheduler needs to run continuisly.
- It does not require the environment to be stationary. Some environments might change from episode-to-episode. For example, if we look at the Rover driving on Mars example, the properties of the field might variate from rocky to soft sand.
- Much simpler to write down.
- Converges much faster.

<u>Advantages over policy and value iteration:</u>

- Just like Monte Carlo methods, it does not require pre-specified models of the reward and the environment.
- Much simpler to write down.

b. The TD(0) method for policy prediction is given by the formula:

$$V(S_t) = V(S_t) + \alpha\,[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{10}$$

The step size $\alpha \in (0,1]$ handles the non stationary environments. As $\alpha \rightarrow 1$, changes in the environment are handled better, but the errors are higher.

The *equation 10* is derived from *equation 1*, the Monte Carlo estimate of the value function. G_t (the estimate of return given by the future returns multiplied by a weight factor) is replaced with *equation 11*, the bootstrap estimate of return, given by the return and the value function at the next timestep. Because of this modification, TD(0) method is able to handle non-episodic systems.

$$G_t = R_{t+1} + \gamma V(S_{t+1}) \tag{11}$$

Different solution explanation

Monte Carlo and TD(0) compute a different solution in general because they learn different things. Monte Carlo methods find the estimate that minimizes the mean squared error on the dataset,as it computes the average of the returns for the states, while TD(0) finds the estimate which would be correct for the maximum-likelihood model of the Markov process.

c. On-policy alghorithms use the target policy π(A|S) to learn. The target policy is what the robot will run after the training. In contrast, off-policy alghorithms consider two policies, as the policy used to learn is different from the the policy used by the robot after learning. The policy used to learn the target policy in this case is called the behaviour policy b(A|S).

   An important difference to consider is how the risk is handled. One disadvantage for off-policy alghoritms is that they do not take into account the possible negative costs in exploration. On the other hand, on-policy approaches do. Moreover, if a good starting policy is available, on-policy would be good to use, but may not explore other policies well. If more exploration is neccesary, then off-policy would be more advisable, but might be slower (2).

d. Sarsa
   The Sarsa algorithm learns the state-action pair using the expression:

$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\,[R_{t+1} + \gamma Q(S_{t+1},A_{t+1}) - Q(S_t,A_t)\,] \tag{12}$$

   Sarsa updates the estimated costs using the state and the action of the following timestep in the target policy. The policy derived from Q follows an ϵ -greedy policy, so the algorithm has 2 parameters: α and ϵ . The policy used for updating the estimated costs is the same as the policy the robot takes for its actions, so Sarsa is an on-policy approach.

   Q-learning
   The Q-learning algorithm learns the state-action pair using the expression:

$$Q(S_t\,,\,A_t) \leftarrow Q(S_t\,,\,A_t) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1},a) - Q(S_t\,,\,A_t)\right] \tag{13}$$

   Q-learning is one of the best-known TD algorithms. Rather than use the action taken, the estimate of the action value is updated with the value of the best possible action at S_{t+1}, making the algorithm an off-policy approach. The policy derived from Q follows an -greedy policy, so the algorithm has 2 parameters: α and ϵ.

Question 3

a. In order to implement Sarsa in the given code, we had to add the code to choose the action A' from S' and to implement the equation for updating $Q(S, A)$:

$$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)] \tag{14}$$

The former was done, using the *sample_action* function from the already implemented *PolicyGrid* class, while the latter was done using the given α, R and γ values and the *self._q.value* function. We also need a check if the robot has reached a terminal state (*done* is true), because then S' would be None. Running the q3_a_sarsa script, the algorithm converged to the following results:

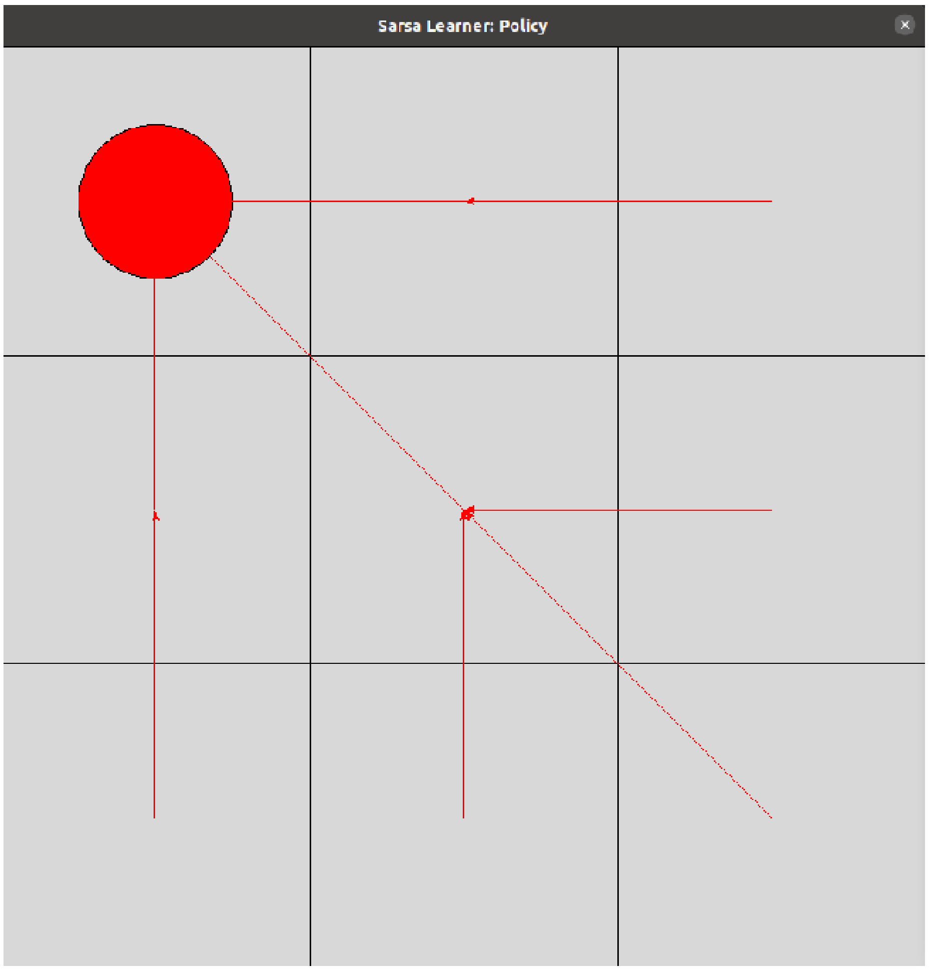
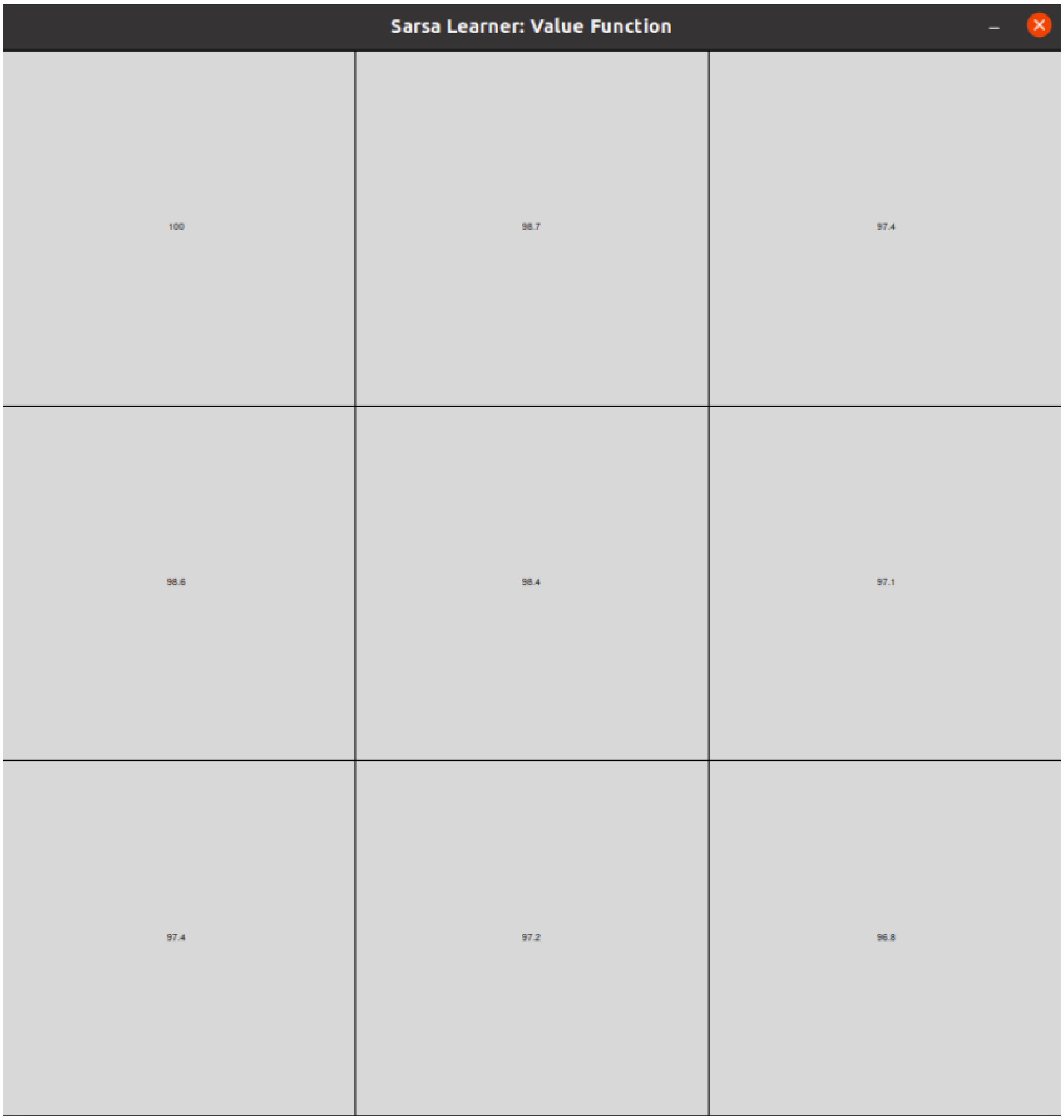


Figure 3 - Policy learned by Sarsa



100	98.7	97.4
98.6	98.4	97.1
97.4	97.2	96.8

Figure 4 - Value function of Sarsa

Results from the second picture as it is not very clear:

100	98.7	97.4
98.6	98.4	97.1
97.4	97.2	96.8

The policy iteration converged to:

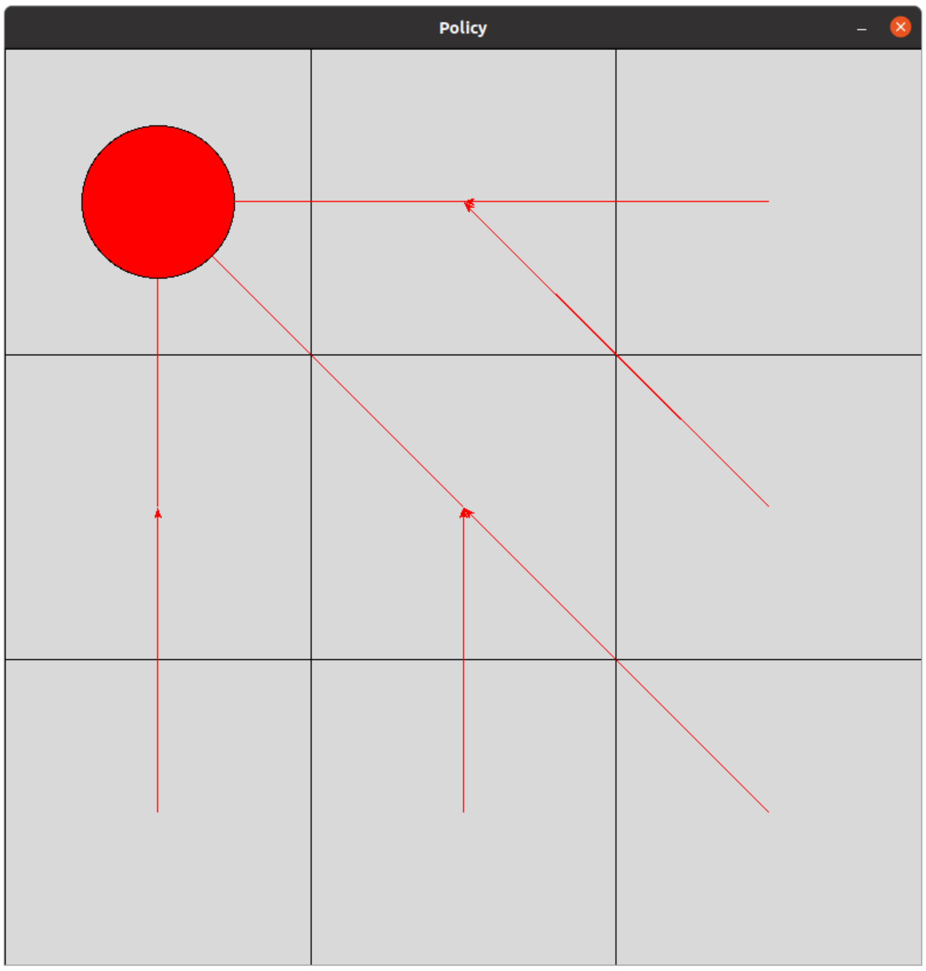


Figure 5 - Policy learned by Policy Iteration

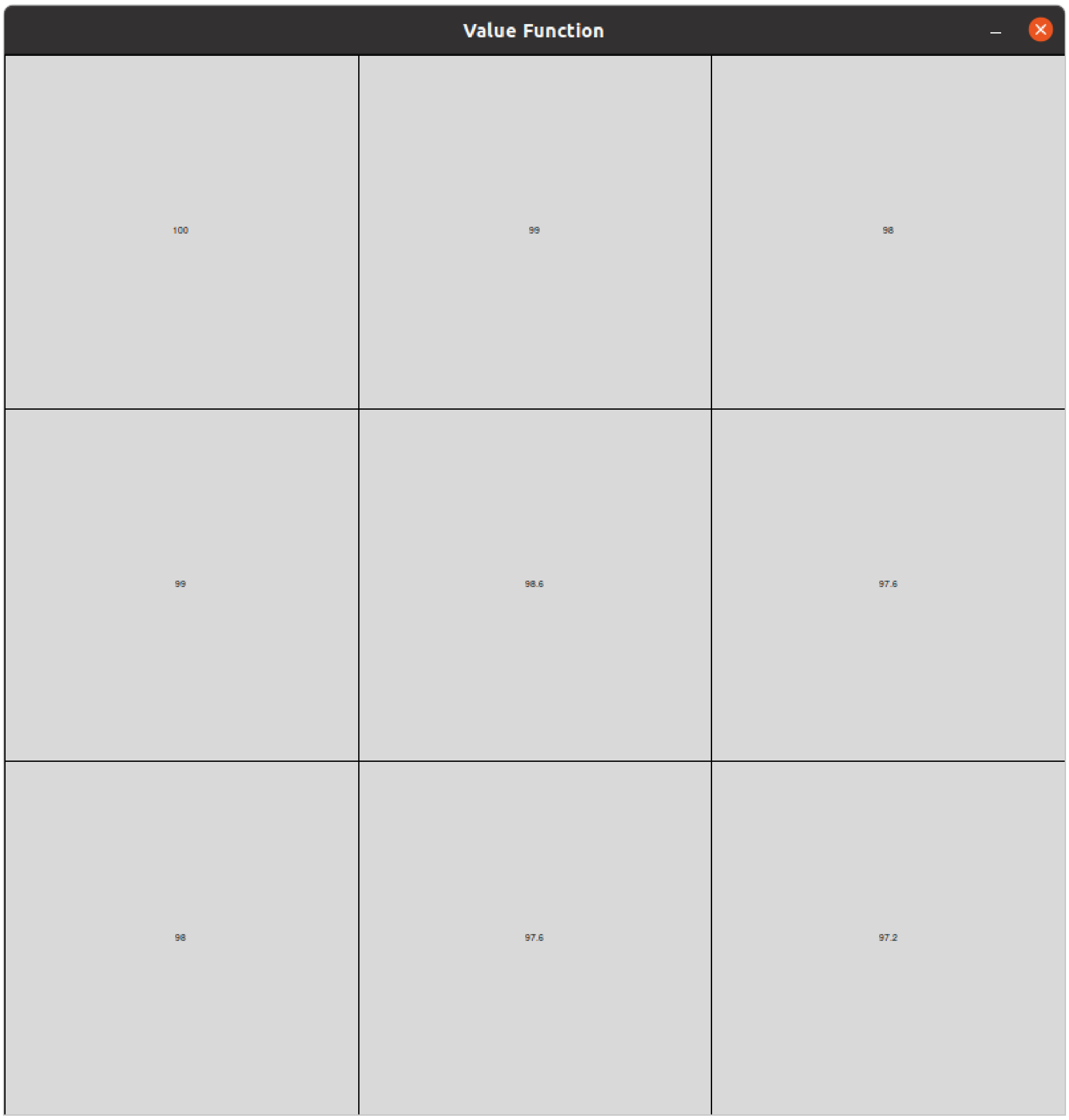


Figure 6 - Value function of Policy Iteration

Results from the second picture as it is not very clear:

100	99	98
99	98.6	97.6
98	97.6	97.2

It can be seen that the policy iteration converges to a higher value function and the only difference in the learned policy is for cell (2,1). Although the polices are different for this cell, they both result in a $1 + \sqrt{2}$ units travelled distance to the terminal cell. However, the key difference is that the policy iteration converged in less than a second while Sarsa took over 18 minutes to reach the max number of iterations – 100,000, and some values of the value function grid were being updated.

b. In order to implement the Q-learning algorithm in the given code, we had to add the code to find the action that maximises the value of $Q(S', A)$ and record this maximal value. Then, we had to use it to implement the equation to update $Q(S, A)$:

$$Q(S,A) \leftarrow Q(S,A) + \alpha \left[R + \gamma \max_{a} Q(S',A) - Q(S,A)\right] \tag{14}$$

The former was done, using a loop that iterates through all possible actions, checks the value of $Q(S', A)$ and records the best one, while the latter was done using the given α, R and γ values and the *self._q.value* function. We also need a check if the robot has reached a terminal state (*done* is true), because then S' would be None. Running the q3_b_ql script, the algorithm converged to the following results:

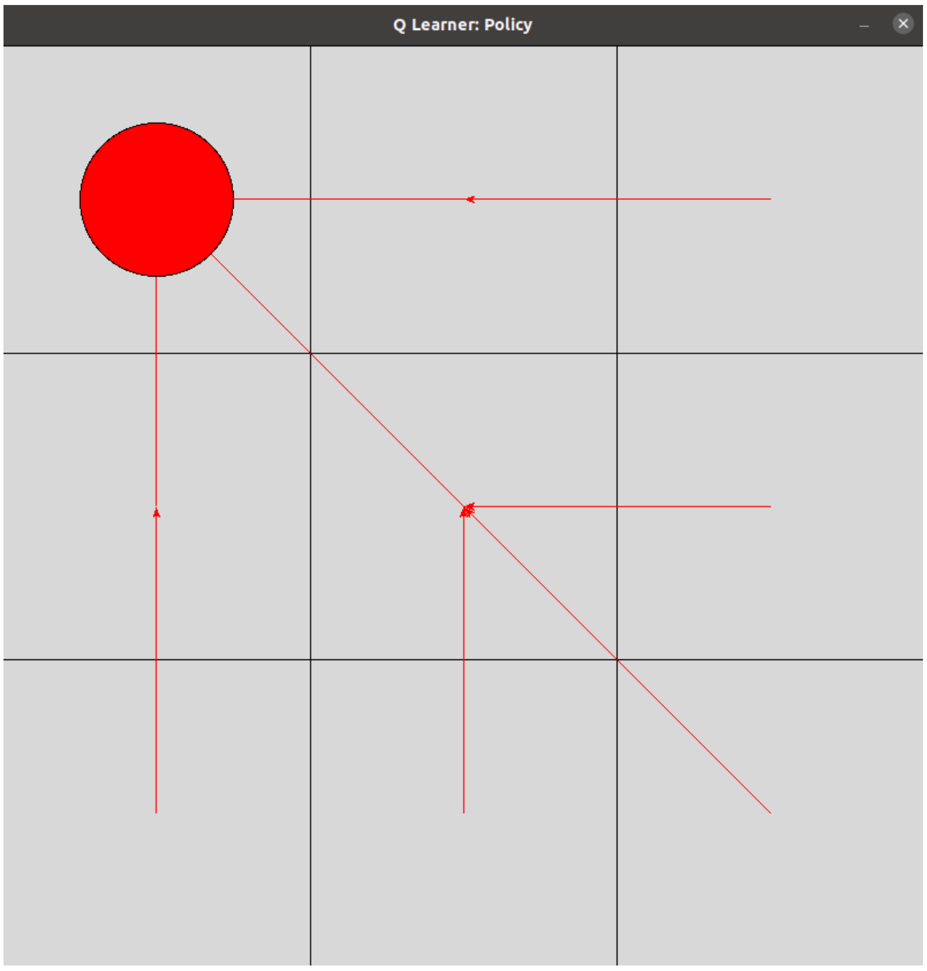
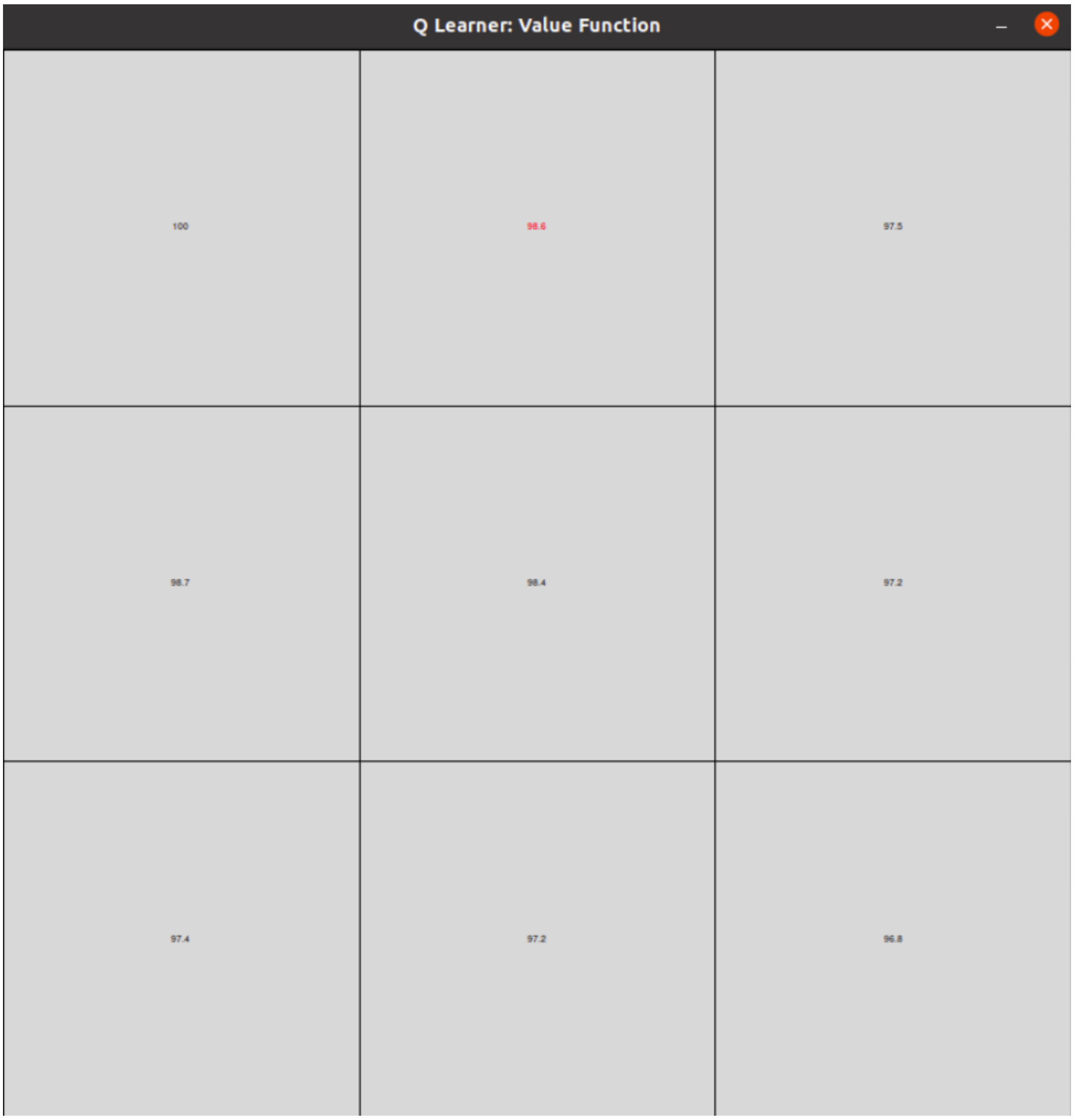


Figure 7 - Policy learned by Q-learning



100	98.6	97.5
98.7	98.4	97.2
97.4	97.2	96.8

Figure 8 - Value function of Q-learning

Results from the second picture as it is not very clear:

100	98.6	97.5
98.7	98.4	97.2
97.4	97.2	96.8

We can see that Q-learning ends up learning the same policy as Sarsa, but has a bit higher value function. The policy again seems to be as optimal as that from the Policy Iteration (PI), however, the value function is still lower than that of PI. This again might be due to the long time that the algorithm takes to converge – after running for 23 minutes, and reaching the maximum number of iterations – 100,000, it still hasn't fully converged, as can be seen by the value at cell (1,2) that is red, meaning that it is still being updated.

c. Each algorithm is run for 100,000 iterations and there are 100 episodes per iterations, so 10,000,000 episodes in total. In each episode we sum the rewards we get after each action until the robot reaches a terminal state and store the final result in a list. We plotted the values from these lists as well as the averaged cumulative reward per episode:

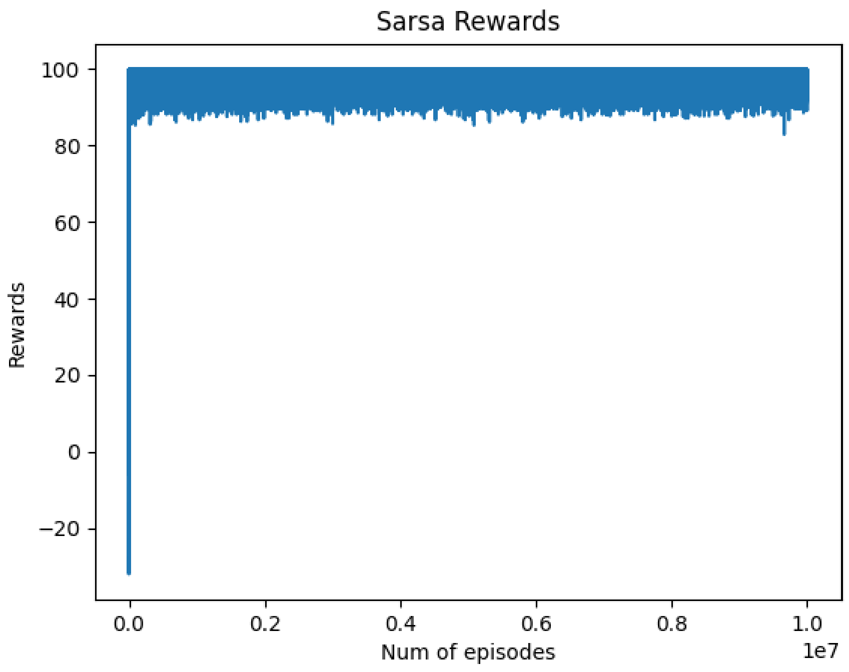


Figure 9 - Total rewards per episode for Sarsa

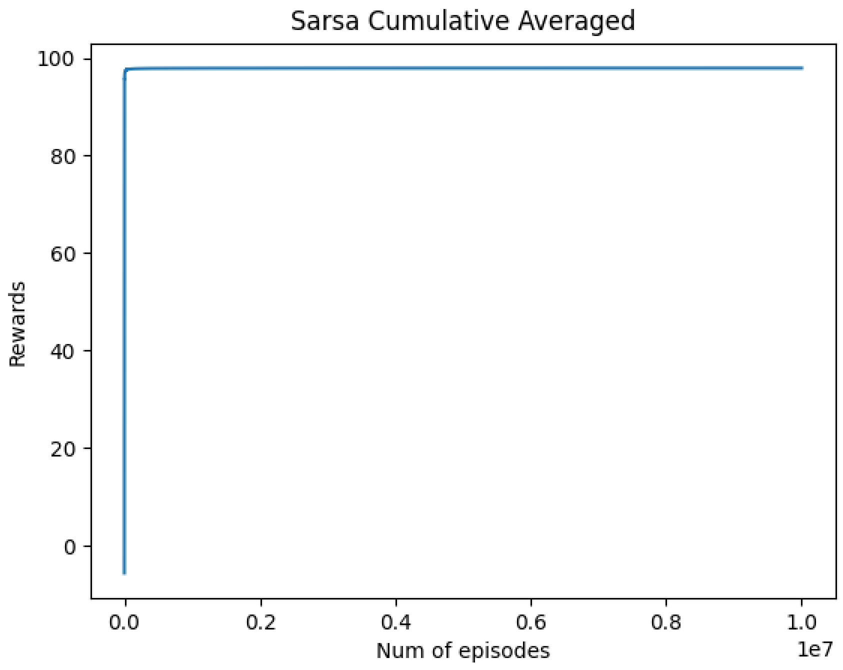


Figure 10 - Averaged Cumulative reward per episode for Sarsa

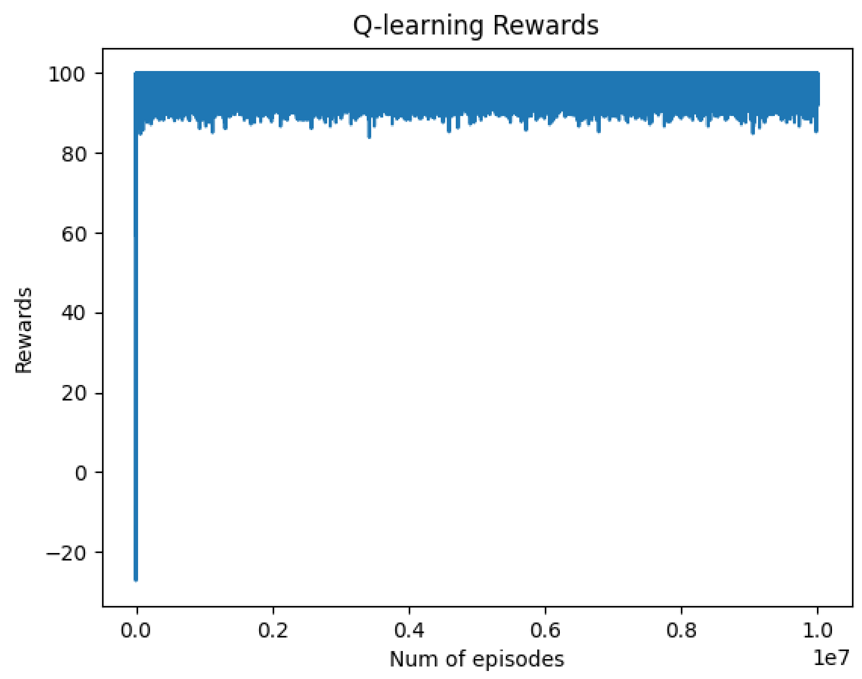


Figure 11 - Total rewards per episode for Q-learning

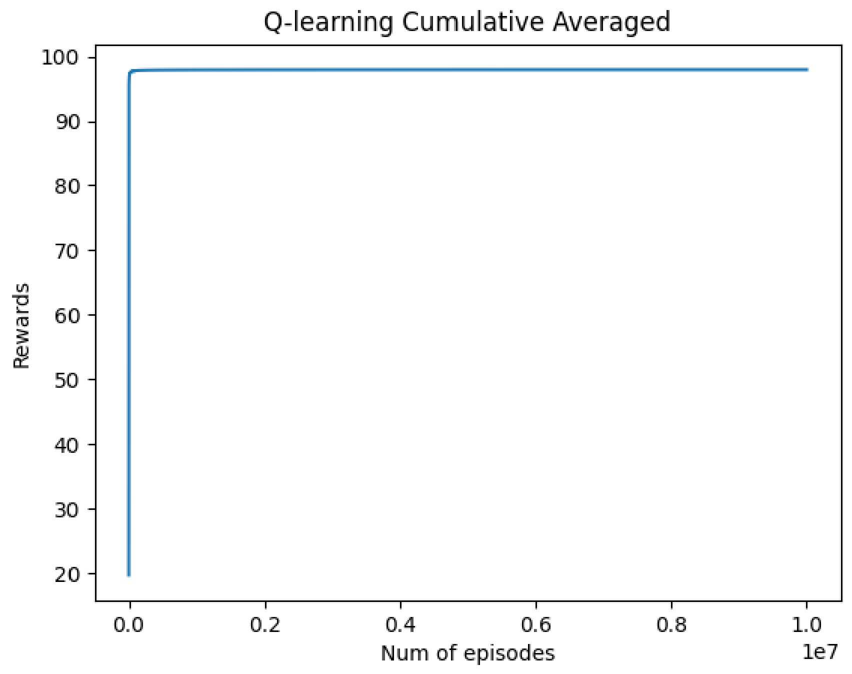


Figure 12 - Averaged Cumulative reward per episode for Q-learning

We can see that they both start with a pretty low reward, but quckly learn a policy. Then they jump around, because of the use of ε -greedy policy, but this jumping is around a pretty high average of about 98. This proves that the algorithms support the proposed explanation. The usual diffence in the rewards of the two algorithms (Sarsa finding a safer, but less optimal policy) is not observed here, because of the simplicity of the test model and the fact that there are no states with negative rewards that Sarsa will try to avoid.

d. We believe the most relevant techniques for improving the performance of the Q-learning algorithm are double Q-learning and deep Q-learning with prioritized experience replay. We believe so, because using double Q-learning will help us overcome the problem of overestimation of single Q-learning, by maintaining two Q-value functions, while deep Q-learning will allow us to have continuous state space and so model the robot's movement better (e.g. in the 3x3 test case, this will allow the robot to go from cell (2,1) to cell (0,2) in a straight line instead of going through cells (1,2) or (1,1), which is much more efficient). Using experience replay will also allow the robot to learn from previous experience (e.g. if the surface in the airport is different in different places, the robot will not redo all of what it has already learned, but finetune it for the current task).

Double Q-learning:
Double Q-learning overcomes the overestimation if single Q-learning's max $Q\ (S',\ A)$ term by concurrently maintaining two Q-value functions and updating them with a probability of $\frac{1}{2}$.

Deep Q-learning with Prioritized Experience Replay:
Deep Q-learning uses function approximation to allow the model to have a continuos state space rather than having specific discrete states, while still having a discrete action space. This allows for a much better representation of real-world situations, is much more scalable for higher number of dimensions and can handle missing data. It also uses experience replay (learning from previous experiences), which provides a natural way to reuse existing data. This also provides greater data efficiency. Basic experience replay, however, uses random sampling over a sliding window, meaning that some transitions may never be picked. To overcome this, prioritized experience replay uses a probability function to pick the most "interesting" ones more often.

Bibliography

1. Sutton & Barto summary chap 05 - Monte Carlo methods [Internet]. lcalem. 2018 [cited 2022 Apr 19]. Available from: https://lcalem.github.io/blog/2018/10/22/sutton-chap05-montecarlo

2. Herman M. On policy and off policy algorithms [Internet]. University of Edinburgh, School of Informatics; 2015 [cited 16 April 2022]. Available from: https://www.inf.ed.ac.uk/teaching/courses/rl/slides15/rl05.pdf

All other information is sourced from COMP0037 2021/22 lecture notes and the code provided.