Multi-Agent Systems, Individual Assignment

Andrea Rossolini

January 2021

Contents

1 Monte Carlo Tree Search (MCTS)

This section explains first, the implementation and the execution of the algorithm and then argue the results. Moreover, it discusses a variant of the algorithm. This variant consists of the classical algorithm, but, the root of the three and the node from where the roll-out is computed change during the algorithm computation. Finally, it makes a comparison of the two algorithms.
This report follows the attached Python notebook's structure (there is no need to check the code to understand this report content).

1.1 Implementation

The implementation has been realized from scratch. Each node of the tree is represented through a Python object. This object contains the essential information and methods to make the node fully independent from the tree itself

(useful during the debugging phase). Each node links to its parent and the left and right child. In this way, it is possible to build the binary tree recursively. Besides, just the leaf nodes have a value; this is randomly generated within an uniform distribution between 0 and 100, as indicated in the assignment text. The object (*MCTS*) contains the logic of the algorithm. It calls node methods following the steps of the algorithm: *selection, expansion, roll-out, and backup*. This code factorization allowed me to perform several experiments with trees of different depth, and try different values for the algorithm's parameters.

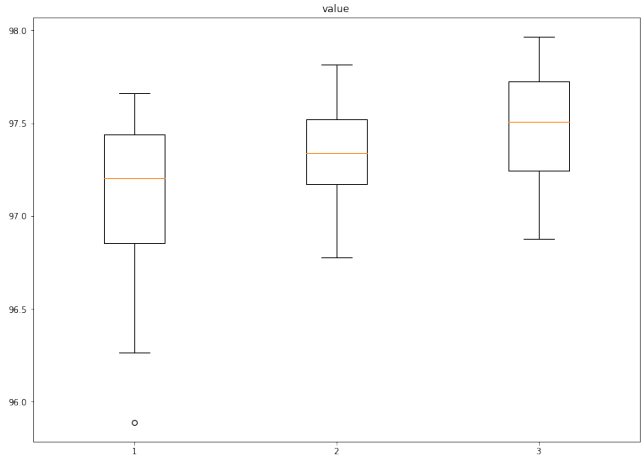
1.2 Execution

The experiments have been performed on a binary tree of a depth $d = 14$ (just for time reason, because the code works perfectly using higher depths; but with a depth $d > 20$, Colab notebook's RAM tends to get full) and a snow-cap of 50 and 100. The algorithm run with 100 different values for the hyper-parameter c, used to compute UCB-score. To collect precise results, each of the 100 c parameters is used 1000 times, storing the results and the computation times. For the study of the algorithm behaviour, I decided to generate c value randomly; but between three different intervals. Thus, the first 34 c values are randomly generated within $0 \leq c \leq 3$, other 34 c values within $3 < c \leq 6$, and finally the last 32 c values are between $6 < c \leq 10$ (I also tried values higher than 10, but the results were not significantly relevant).

1.3 Results

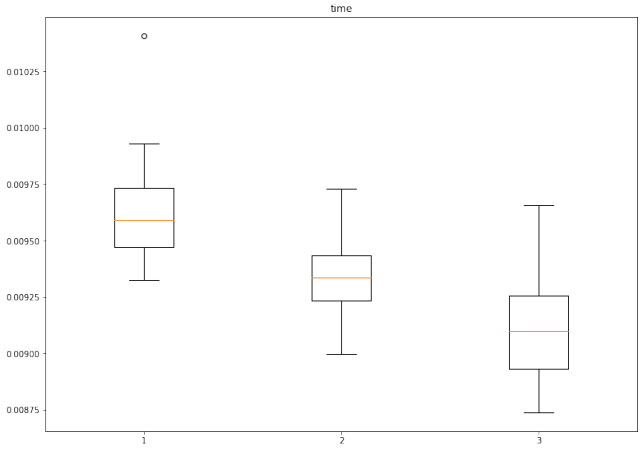
Let's start analyzing the general results obtained computing the algorithm with different values for c.

Figure 1: Mean results achieved by the algorithm for different intervals of c. The left plot represent the results with $0 \leq c \leq 3$, the central one $3 < c \leq 6$, the right one $6 < c \leq 10$.



From figure 1 it can be seen that with a higher value for c the algorithm computes, on average, slightly better result. Anyhow, this difference is not significant enough to jump to a conclusion. Indeed, the results show the c values most likeable to yield to the best leaf node (with a value of ~ 99.999) are: 0.265, 1.302, 1.565, 2.165, 2.246, 3.28, 4.152, 4.771, 4.792, 4.879, 5.833, 7.763, 8.054, 8.391, 9.053. Hence, we can say to find the best results we must not relay just on the parameter c; but there are other variables to take into account.

Figure 2: Average time taken by the algorithm to reach the best leaf node, according to the value of c. As before the left plot represent the results with $0 \leq c \leq 3$, the central one $3 < c \leq 6$, the right one $6 < c \leq 10$.



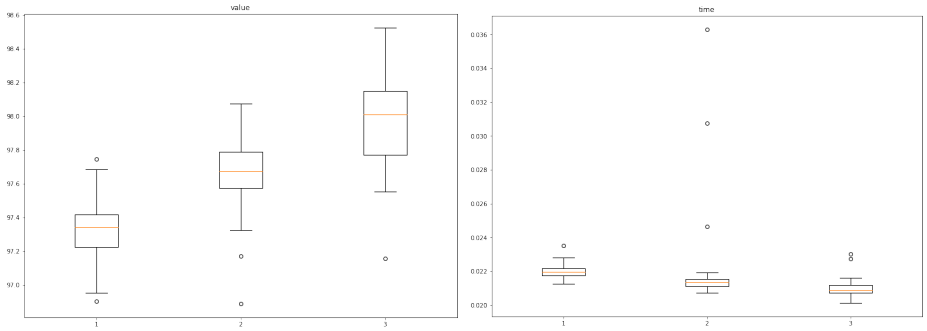
Studying figure 2, we can infer similar consideration as before. The bigger is parameter c the quickest is the algorithm computation. Again, the result obtained is not significant enough to say that parameter c is heavily influencing the algorithm performance. Moreover, the quickest computation has been carried out in 0.00759 seconds, using $c = 5.9628$.

I repeated the experiment, but this time with a snow-cap of 100. My expectations were better average results as the expense of time performance. The results are shown in figure 3.

Here, it is easy to notice that the average results are slightly better than before. While the execution time is way higher (more than 100% higher). For this reason, a snow-cap of 100 is not worthwhile. Moreover, even with a bigger snow-cap, the effects of the hyper-parameter c seem to be very light.
In this case the fastest result takes ~ 0.01627 with $c \approx 7.2034$.

These results are honestly unexpected. The influence of the c parameter seems to a have random behaviour. Even trying with very high values for c (e.g. 500), the results do not differ too much. Probably, we could obtain more

Figure 3: Mean results (right) and time (left) achieved by the algorithm for different intervals of c and a snow-cap of 100. In both figures, the left plot represent the results with $0 \leq c \leq 3$, the central one $3 < c \leq 6$, the right one $6 < c \leq 10$.



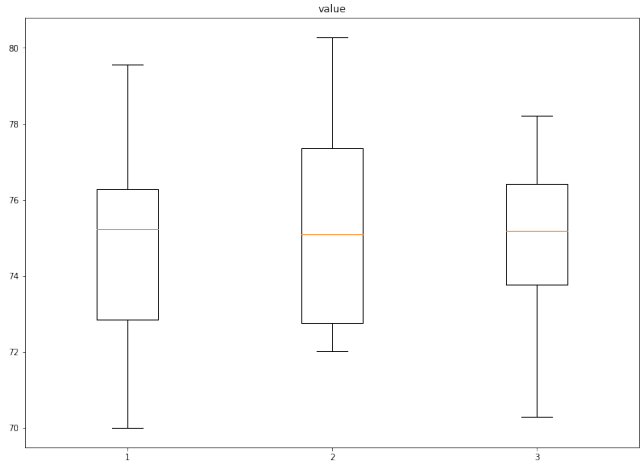
relevant results using trees with higher branching factor.

1.4 MTCS Variant

In this variant, the MCTS-iterations starting from a specific node are limited to 10. After those 10 iterations, the root changes and one of the best former root's children became the new root. Moreover, a similar assumption has been done for the number of roll-outs starting from a particular snow-cap node. Indeed, the roll-out starts from the same node two times in a row then, it changes to the most promising node in the snow-cap. When this change occurs, even if the new "roll-out node" collects worse results, the old "roll-out node" can not be used as such anymore.
Using the same c parameters of the MTCS implementation from above and the same tree, we can study the newly collected data.

The two box plots (figure 4 and 5) show that the results obtained by this variant of the algorithm are visibly lower than the original MCTS algorithm. Moreover, the standard deviation is far higher. On the other hand, computational times are strongly decreased compared to the original algorithm. Apparently, with this algorithm, the execution time is almost reduced by 1/3, and now the standard deviation is moderate. This is due the fact that the more the root goes down, the faster are the rollouts and, besides, the tree gets heavily pruned. Finally, considering that the algorithm complexity is exponential, the deeper is the tree, the more this algorithm will save time compared to the original one.
With this version, the quickest run has been computed in ~ 0.002852 seconds with the parameter $c \approx 1.5649$ (almost 50% faster than the original algorithm fastest solution).

Figure 4: Mean results achieved by the variant algorithm for different intervals of c. The left plot represent the results with $0 \leq c \leq 3$, the central one $3 < c \leq 6$, the right one $6 < c \leq 10$.



As for MCTS, a second experiment has been carried out. It consists of the same algorithm, but now the snow-cap is 100, and the maximum number of roll-outs from the same node increased to 10. The figure 6 shows the results are even worse than the MCTS variant algorithm. Not only it finds worse outcomes, but it also takes more time to do so.

2 Reinforcement Learning: SARSA and Q-Learning for Gridworld

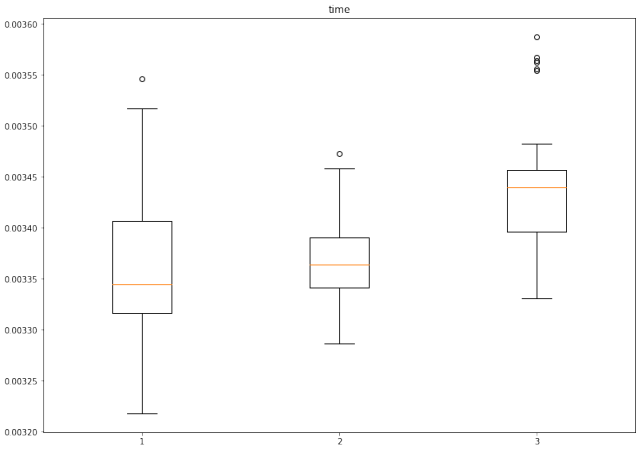
2.0.1 Implementation of the General Environment

As for the first question, the code has been written from scratch. Similarly, as I did for the binary tree, I decided on an Object-Oriented approach. Each gridworld cell is independent and contains all the information that concerns the cell itself (type, coordinates on the grid, immediate reward, state-value and action-value functions, and so forth). Another object manages the logic of the world. Cell values can be modified through this last object. Besides, it returns the states according to the agent actions. It also allows to print the cells (for debugging) and plot the gridworld heat-map.

2.1 Monte-Carlo for Policy Evaluation

This section explains the implementation of the Monte-Carlo algorithm for policy evaluation and then discusses the results.
The text of the assignment does not explicit if we should implement the *first-visit* Monte-Carlo evaluation or the *every-visit*. Because so, I decided to implement

Figure 5: Average time taken by the variant algorithm to reach the best leaf node, according to the value of c. As before the left plot represent the results with $0 \leq c \leq 3$, the central one $3 < c \leq 6$, the right one $6 < c \leq 10$.



both of them and compare the results.

2.1.1 Implementation

An object encapsulates the logic of Monte-Carlo algorithm. This object takes in input the policy with which the agent takes decisions. This feature allowed me to play with policies during development. Furthermore, the number of episodes per simulations is editable. The results, represented as heatmaps, show the state-value functions of the environment after all the episodes taken into account.

2.1.2 Execution and Results

I decided to run the algorithm(s) with an increasing number of episodes. In this way, it is possible to see how the gridworld changes. The value of γ is set to 0.99 to exploit the foresight of the algorithm. Thanks to this, I could run more experiments using fewer episodes and save time. As always, the showed data are the result of 1000 experiments with the same number of episodes and parameters.

Comparing figure 7 and 8, it is possible to notice there is no big difference, in terms of state-value function, between the *first-visit* and *every-visit* algorithms. Nonetheless, the last one, as foreseeable, takes more time to compute the result.

Figure 6: Mean results (right) and time (left) achieved by the algorithm for different intervals of c and a snow-cap of 100. In both figures, the left plot represent the results with $0 \leq c \leq 3$, the central one $3 < c \leq 6$, the right one $6 < c \leq 10$.

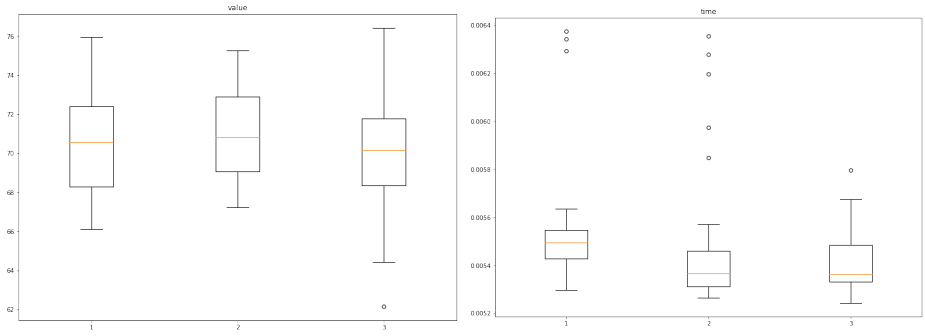
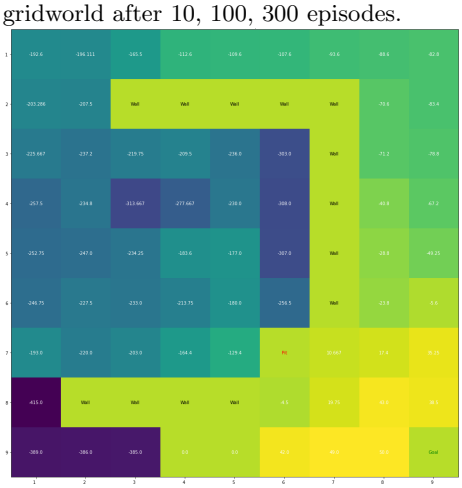
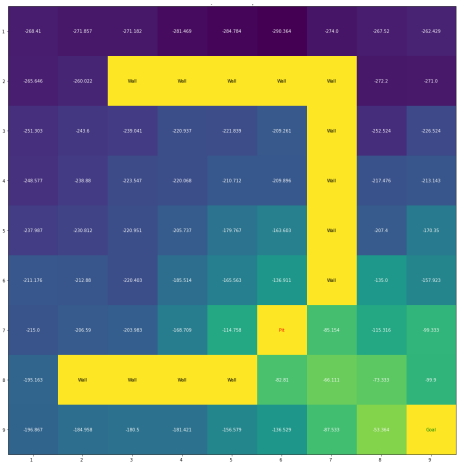


Figure 7: The figure shows the effects of the Monte-Carlo *first-visit* algorithm on gridworld after 10, 100, 300 episodes.



Time: 0.0685 secs

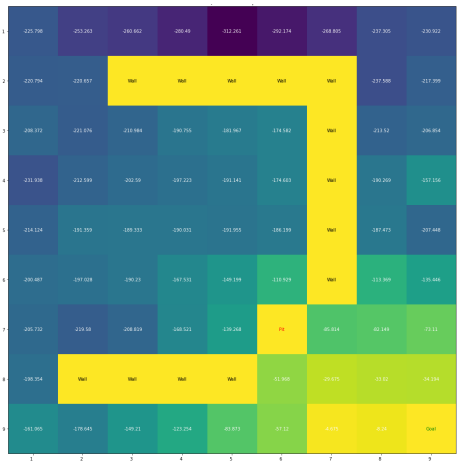


Time: 0.6703 secs

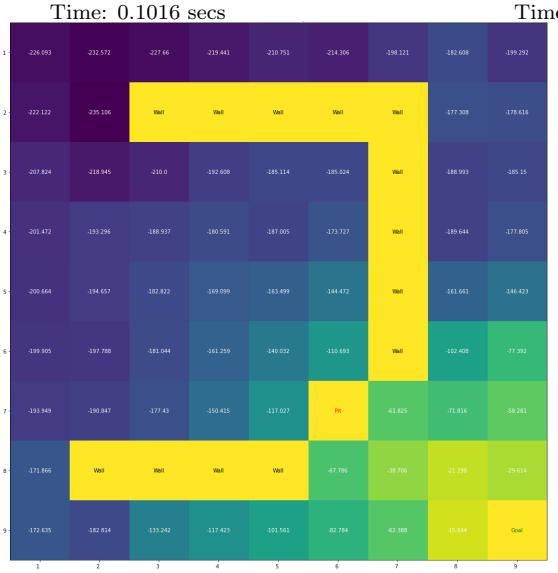
Time: 1.998 secs

Figure 8: The figure shows the effects of the Monte-Carlo *every-visit* algorithm on gridworld after 10, 100, 300 episodes.

Time: 0.1016 secs



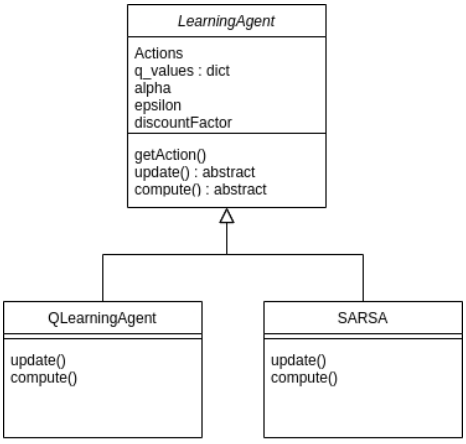
Time: 1.0003 secs



Time: 3.0024 secs

2.2 Learning Agents

The implementation of the SARSA and Q-Learning agents is summarized in the diagram in figure 2.2. An abstract class contains all the methods and parameters common to both algorithms. The two implementations are completely sticking with the pseudo-code seen during the lectures. As did for Monte-Carlo policy evaluation, both, Q-Learning and SARSA, are computed used increasing number of episodes. To obtain more precise results, for each number of episodes the algorithm is computed 1000 times. That is to say, if we want to run SARSA with, let's say, 100 episodes, we do so, but, after the 100 episodes, the algorithm is computed other 999 times. At each iteration, the results are stored. All the heat-maps related to Q-Learning and SARSA show in each cell (except walls, pit, and goal) the expected action-value function.



2.2.1 SARSA Results

Starting with 10 episodes, this experiment aims to show how the quality of the results grows with the increasing of the number of episodes. All the experiments have been carried out using a learning rate $\alpha = 0.5$, an exploration probability $\epsilon = 0.25$, a discount factor $\gamma = 1$. I chose these value for the following reasons: $\alpha = 0.5$ is a value very common on literature and I decide to take this value for granted, $\gamma = 1$ simply because the math used to check if the algorithms were working correctly was easier, and $\epsilon = 0.25$ is explained at the end of this section.

To obtain balanced data, each episode consists on the agent starting from the top-left cell (1,1) until it reaches a terminal cell (Pit or Goal). The figures 9-13 show the expected action-value functions of the environment computed by the algorithm after a certain number of episodes (excluding walls, pit, and goals). The figures show the average results after all the episodes and the 1000 repetitions of each experiment.
As expected, the average total reward and the execution time grow with the increment of the episodes. That is to say, the algorithm is effectively reaching the best policy. Indeed, from the visualization, it is possible to see the path the agent tends to follow. After 1000 episodes, the agent still explores the environment. This behaviour can be corrected tuning the exploration parameter ϵ.

2.2.2 Q-Learning Results

The results are computed as for SARSA, using the same parameters and number of episodes.
Discussing the results from figure 14-18, We can easily notice that this algorithm is way faster at reaching an acceptable policy. Indeed, with 100 episodes, Q-Learning agent is already able to draw a pattern on the gridworld. Nonetheless, the average total reward is still negative. Besides, in 300 episodes, this agent reaches the total reward that SARSA agent obtains in 500 episodes. As expected, the Q-Learning algorithm converges faster than SARSA. The last figure of Q-Learning results shows a more "convinced" path from the starting node to the Goal.

2.2.3 Conclusion

The two algorithms are very similar in terms of execution time. The key difference between these two algorithms is shown on figures 19-20. These gridworlds represent the result of an "exaggerated" experiment run with $\epsilon = 0.99$ and with 500 episodes (again, both experiments has been run 1000 times). This key difference between SARSA and Q-learning is that the first one is an *on-policy* algorithm, so it follows the policy that is learning, and Q-learning is *off-policy*. This means that Q-Learning agent updates its Q-values using the Q-value of the next state s' and the greedy action a'. In other words, Q-Learning estimates the return (total discounted future reward) for state-action values following a greedy policy, so the learner learns the value of the optimal policy independently of the agent's actions. This behaviour can clearly seen on figure 20, which shows that Q-Learning agent can find the (almost) perfect policy (i.e. the cell at position 1,1 has a $Q^*(s, a) = 34.5$ which is a very good value, considering the minimum number of cells that the agent has to cross is 15). Despite this SARSA implementation uses a greedification to find an optimal policy, the difference between the two algorithms is still deep. it updates its Q-values using the Q-value of the next state s' and the current policy's action a''. It estimates the return for state-action pairs assuming the current policy continues to be followed. Indeed, with a high ϵ SARSA, after 500 episodes, is still struggling to find an optimal policy.

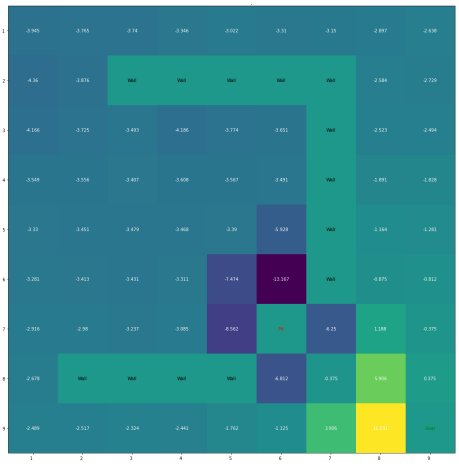


Figure 9: SARSA 10 episodes.
Avg total reward ~ -156.585
Avg execution time ~ 0.0323 secs

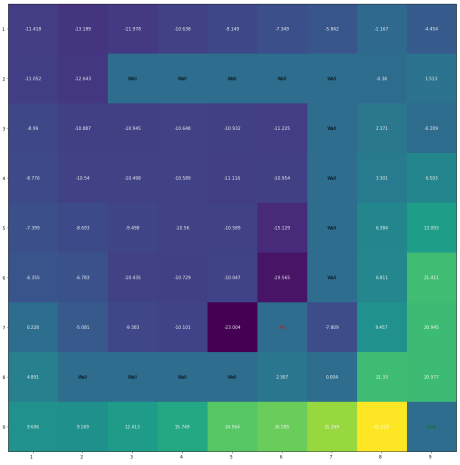


Figure 10: SARSA 100 episodes.
Avg total reward ~ -17.472
Avg execution time ~ 0.1268 secs

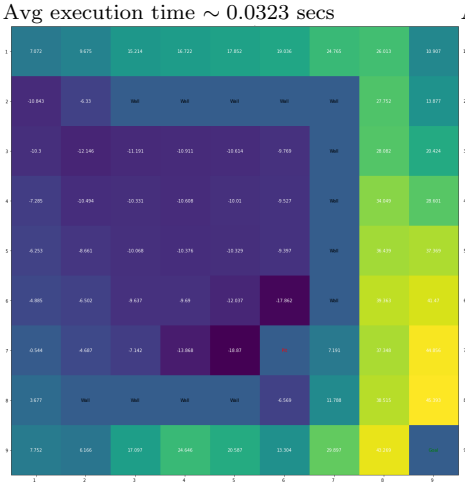


Figure 11: SARSA 300 episodes.
Avg total reward ~ 9.147
Avg execution time ~ 0.2473 secs

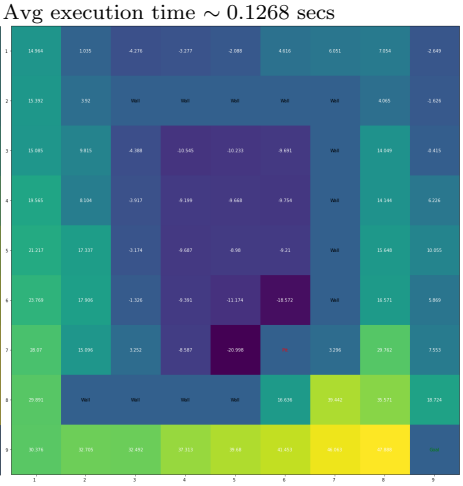


Figure 12: SARSA 500 episodes.
Avg total reward ~ 14.788
Avg execution time ~ 0.3667 secs

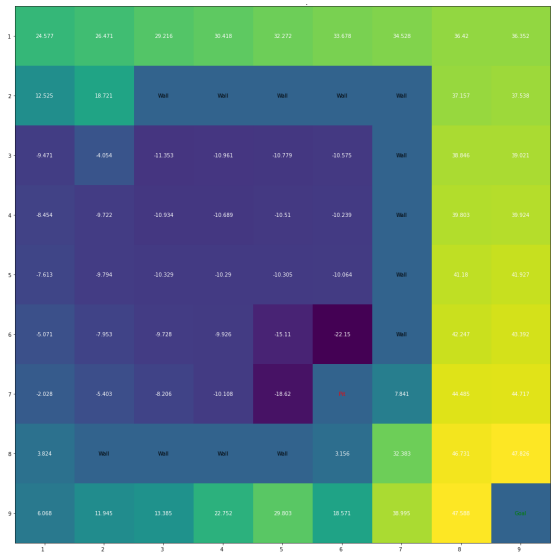


Figure 13: SARSA 1000 episodes.
Avg total reward ~ 19.653
Avg execution time ~ 0.6587 secs

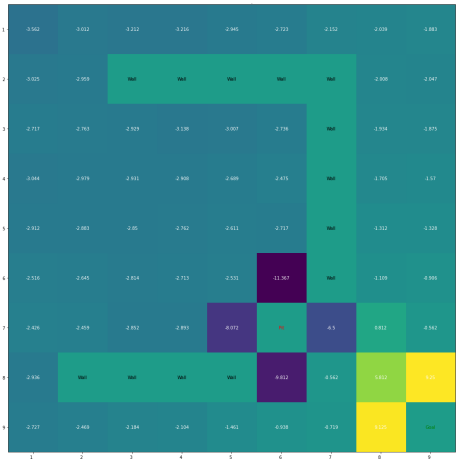


Figure 14: Q-Learning 10 episodes.
Avg total reward ~ -145.2289
Avg execution time ~ 0.0349 secs

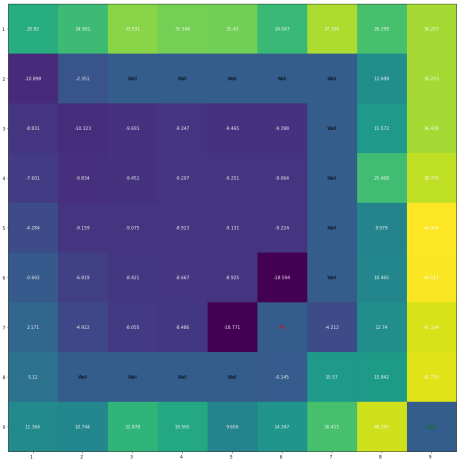


Figure 15: Q-Learning 100 episodes.
Avg total reward ~ -11.578
Avg execution time ~ 0.1304 secs

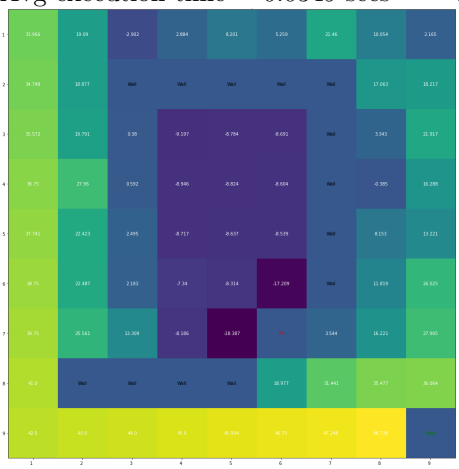


Figure 16: Q-Learning 300 episodes.
Avg total reward ~ 14.992
Avg execution time ~ 0.2425 secs

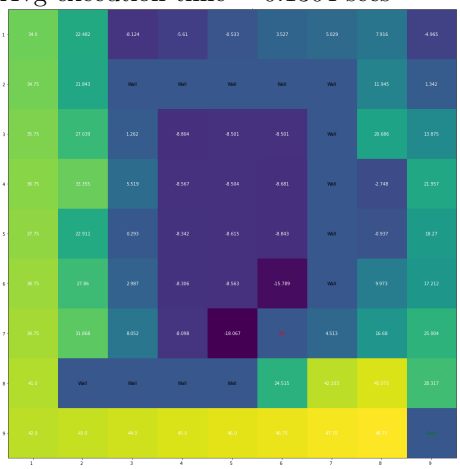


Figure 17: Q-Learning 500 episodes.
Avg total reward ~ 20.375
Avg execution time ~ 0.354 secs

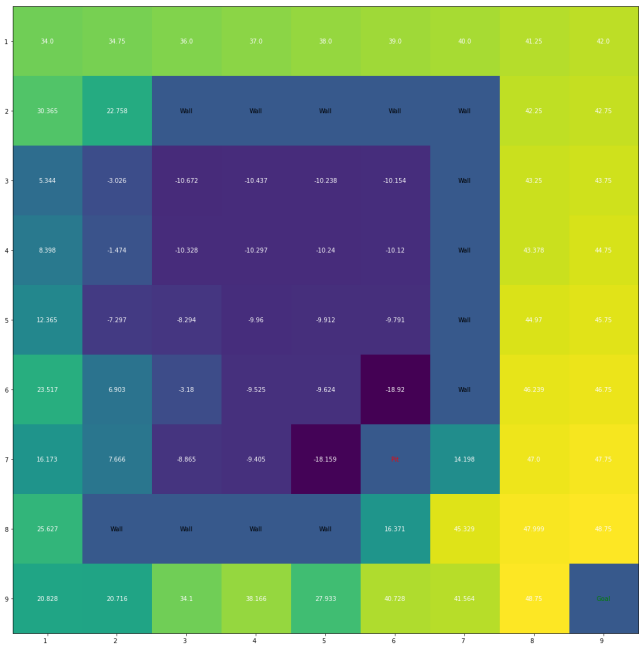


Figure 18: Q-Learning 1000 episodes.
Avg total reward ~ 24.601
Avg execution time ~ 0.6241 secs

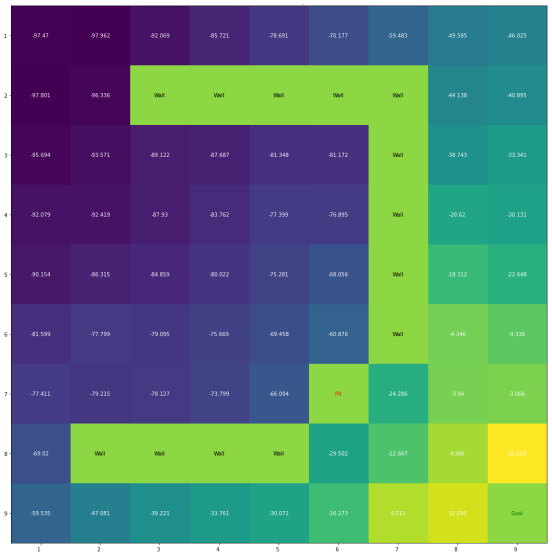


Figure 19: SARSA 500 episodes and $\epsilon = 0.99$.
Avg total reward ~ -131.78
Avg execution time ~ 2.1038 secs

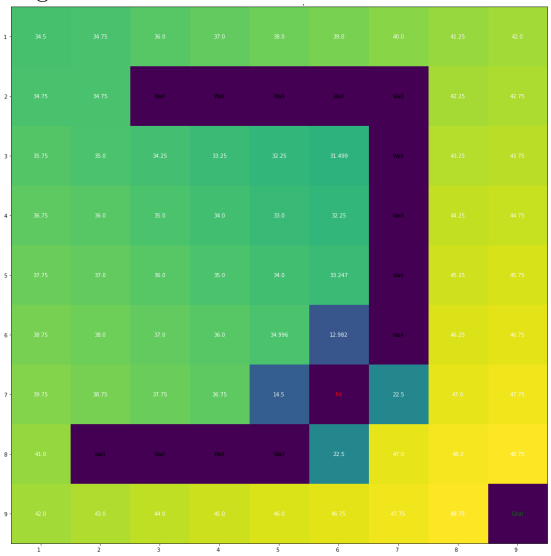


Figure 20: Q-Learning 500 episodes and $\epsilon = 0.99$.
Avg total reward ~ -137.84
Avg execution time ~ 2.9095 secs