Final Homework Assignment - Multi-Agent Systems

Enrico Calleris (2692023)

Vrije Universiteit Amsterdam
De Boelelaan 1105 1081 HV
Amsterdam, Netherlands
`http://www.vu.nl`

1 Monte Carlo Tree Search (MCTS)

The results presented in this section are based on the Python code *MCTS.py*.

1.1 Introduction

Monte Carlo Tree Search (MCTS) is a method for finding the highest-rewarding branches in a tree of which we know nothing if not the leaf nodes rewards. A key feature in such algorithm is the computation of Upper Confidence Bounds (UCBs), used to estimate the highest expected end-reward of a node. By computing such UCBs, one can decide which node to choose in order to get the maximum possible final reward [5]. The formula for UCB is:

$$UCB(node_i) = \overline{x_i} + c\sqrt{\frac{logN}{n_i}}$$

with x_i = mean expected reward of node i ($x_i = \frac{value_i}{n_i}$), N = number of visits to parent node, n_i = number of visits to node i. The hyper-parameter c is used to adjust the estimation, determining the impact of the number of visits on the mean value x_i.

In the following section, a MCTS algorithm will be analysed, gaining insights into its ability to find the highest-rewarding node while also investigating the influence of the hyper-parameter c. For such evaluations, a binary tree of depth $d = 12$ will be used. Literature suggests $c = \sqrt{2}$ as the optimal value [2], so it is expected that, when using a similar one, the performance of the algorithm will show improvements compared to other values.

1.2 Method

Code A class $Tree$ was created, with several methods for the generation of the tree and the MCTS computation. More specifically:

1. The class takes as inputs the depth d and the c-value.

2. The method *buildTree*() is used to recursively create the binary tree, represented through the use of nested dictionaries containing the needed informations (e.g value of a node, type of node, number of visits). Leaf nodes are created trough the method *createTerminal*() which picks a random number from the uniform distribution $U(0, 100)$ that represents the end-reward of that specific leaf node. The maximum generated reward is stored in the class and can be obtained by calling the *getMax*() method.
3. The method *mcts*() performs the Monte Carlo Tree Search, encompassing the 4 basic steps of Selection, Expansion, Rollout and Backup. The method takes as input a node (which in the first iteration will be the root node, and then will become a child node as the function is recursively called until a leaf node is reached) and the number of iterations n as an optional parameter. The snowcap is always defined by the current root node plus the two child nodes. At every iteration, after selecting a child node, the *rollout*() method is called, which returns the highest reward encountered after 5 rollouts starting from the chosen child node.

Experimental Design The aim of the experiment was to evaluate the performance of the created MCTS algorithm under different values of c. Given that the optimum is found at $c = \sqrt{2}$, values in the closed interval [0, 2] were investigated, with a stepsize of 0.2. For every c-value, a new tree of depth $d = 12$ was generated, and then the MCTS algorithm was run for 1000 iterations. For every iteration, the difference between the actual maximum value (retrieved through the *getMax*() method) and the maximal value found by the MCTS was registered. After the 1000 iterations, the mean value for the difference (Δ) and the standard deviation (SD) were computed, together with the number of hits of the algorithm (i.e. number of times in which the actual maximum was found).

1.3 Results

The results of the analysis are presented in Table 1. By looking at the table it appears clear, without the need to recur to statistical analysis, that there are no significant differences in the parameters studied when the c-value changes. The values of Δ are always comprised in the open interval (15, 18), while the SDs are all very big, comparable with the delta values themselves. It is reasonable to assume that the maximum value of the leaf nodes is almost always bigger than 99.9, given the fact that with a depth of 12 the number of leaf nodes is equal to $2^d = 4096$. While looking at the number of hits, it also appears clear that the percentage of correct findings of the maximal value is negligible (5 in the best case, accounting for 0.5% of the total number of iterations) and probably due to chance.

c-value	Δ	SD	$Hits$
0	15.79	11.82	2
0.2	16.13	13.11	5
0.4	16.26	12.98	0
0.6	14.74	13.93	0
0.8	16.03	13.31	2
1	16.33	12.60	0
1.2	16.18	13.58	0
1.4	15.49	13.34	1
1.6	17.18	13.84	0
1.8	15.73	13.53	4
2	15.61	13.19	3

Table 1: Collected parameters for each $c \in [0, 2]$.

1.4 Discussion

The results of the experiment did not support the expectations, showing no effect of the hyper-parameter c on the performance of the algorithm. The effect of such parameter has indeed proved to exist in the literature [1], with the optimal value being $c = \sqrt{2}$, as previously mentioned [2]. The experiment performed and described in the current report might have been impacted by different factors, the most important one being the number of iterations performed. Such a number was set to $n = 1000$, but with a higher n (e.g. $n = 100'000$) results would have likely been different. Increasing the number of iterations, however, brings about a trade-off with computational complexity, increasing the runtime of the algorithm. This is why n has been kept relatively low, while also ensuring a fair amount of iterations to appropriately analyze results. Increasing the number of rollouts and iterations in every new root node is also likely to have an impact on the algorithm's performance.

In spite of these problems, the algorithm still performed in a satisfactory way, given that in average it was able to reach leaf nodes with values higher than 80. Such values are still distant from the maximum possible score, but represent a good starting point to further improve the current code.

2 Reinforcement Learning: SARSA and Q-Learning for Gridworld

The results presented in this section are based on the Python code *RL.py*.

2.1 Introduction

Reinforcement Learning (RL) is a technique used to train agents to discover how to get the maximum possible rewards by interacting with the environment. In this section, the focus will be on model-free learning, in which the agents does not possess information on the rewards (i.e. the consequences of its moves) and therefore cannot predict in advance what would be the best move to perform. The focus of this section will be a 9x9 gridworld problem focused on three aspects:

1. Monte Carlo Policy Evaluation (MCPE) which allows to estimate the value of the Action-Value Function v_π by randomly generating an episode (given a starting point) and using the total reward received before reaching a terminal state as an estimate of v_π. By averaging the results over multiple iterations, a somewhat accurate estimation of the Action-Value Function can be obtained [5].
2. SARSA, a RL method that allows to find the optimal policy π^* by iterating between two steps: (1) Policy Evaluation, which estimates the State-Action Value Function q_π in the current state, and (2) Policy Improvement, which focuses on the selection of the best action (”greedy” action) resulting from the previous estimation of q_π. This method will eventually converge to the optimal State-Action Value function q^* and the optimal policy π^*. Bellman Equations are used for the calculation of q_π in every step of the Policy Evaluation [5].
3. Q-Learning, a method for finding q^* and π^* that employs Bellman Equations for Optimality in order to directly bootstrap with the best action and converge to the optimal policy more quickly [5].

2.2 Method

Code The code was developed by referring to a version by Gerard Martínez [4] and adapting it for the purposes of our problem. First, all the important parameters for the gridworld (i.e. size, reward, position of walls, position of terminal states, permitted moves and number of iterations) were set, together with other parameters to store and evaluate the State-Action Value Function q_π and the policy π. The following part of the code can be divided into three chunks:

1. For Monte Carlo Policy Evaluation, a function *mcpolicyeval*() was created, devoted to the evaluation of an episode and consequent update of the Action Value Function v_π. At each iteration, an episode is generated through the *generateEpisode*() function, that deals with controlling whether the next

picked state is a legal one (i.e. no walls or exceedance of grid boundaries). After $n = 1000$ iterations, the updated v_π can be plotted in an heatmap, through the use of the function *heatmap*(), which takes as an input a 2-dimensional array. The type of MCPE performed is a first-visit MCPE.

2. For SARSA, a function *sarsa*() was created, taking as optional parameters the discounting factor γ and the learning rate α. These two parameters were set respectively at 0.9 and 0.4, in order to ensure the best possible convergence [3]. The function iterates for $n = 1000$ times and generates an episode at each iteration, evaluating and updating the State-Action Value Function q_π, as well as keeping track of the changes in the policy π (*deltapi*) and in q_π (*deltaqsa*). At each iteration, the episode is generated through the use of the *generateEpisode*() function, which takes as optional parameters the learning type, together with γ and α. If such function is called by specifying a *learning* parameter, then while generating an episode it can also improve the policy and keep track of the deltas.
3. For Q-Learning, the function *qLearning*() was created. This function mirrors in all respects the one for SARSA (*sarsa*()), but works by bootstrapping with the best action when updating the policy π in the *generateEpisode*() function.

In addition to the above, also a *graph*() function is present, taking as input a dictionary. This function is used to plot the deltas of the State-Action Value Function (*deltaqsa*) over time, to analyze the convergence of the algorithm.

Experimental Design For the purpose of this assignment, two different experiments were run, by setting all parameters in order to emulate the 9x9 gridworld shown in Fig. 1. The four permitted actions are 'up', 'down', 'right', and 'left'.

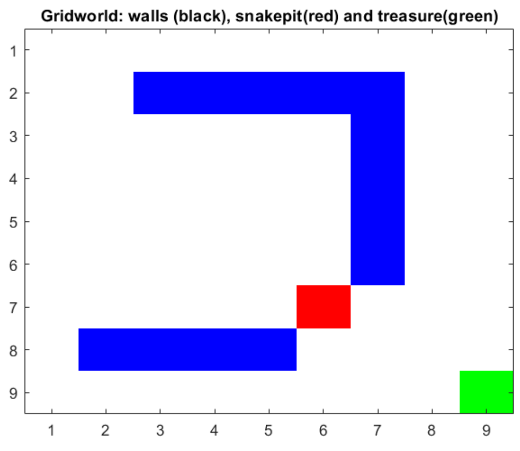

Fig. 1: Depiction of gridworld used for the experiment

1. First, Monte Carlo Policy Evaluation was performed on the equiprobable policy (i.e. all four actions have the same probability of being chosen). An heatmap was then created in order to visualize results and discuss the given policy π. The expectation is that such a policy is not optimal, since it can easily lead to the snakepit when starting in (1, 1).
2. Second, SARSA and Q-Learning were both performed for the gridworld in Figure 1 and their performance was compared. More specifically, the convergence of the *deltaqsa* parameter in the two functions was studied, with the expectation of seeing a more efficient performance for Q-Learning than SARSA [5].

2.3 Results

Monte Carlo Policy Evaluation First, the results for the Monte Carlo Policy Evaluation are presented. The heatmap for v_π for the equiprobable policy is shown in Fig. 2. Darker colours indicate lower values, while brighter ones denote higher scores.

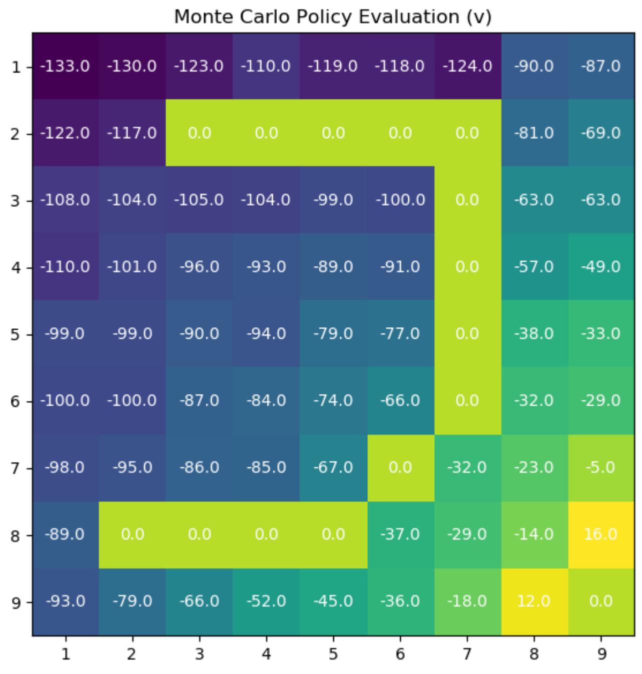

Fig. 2: Heatmap for Monte Carlo Policy Evaluation

The heatmap shows the estimated value of v_π for each of the states represented by a cell. It can be seen that states that are the most further away from the treasure cell (9, 9) have the lowest values (with state (1, 1) having the lowest), while cells in close proximity to (9, 9) have higher values, with the two adjacent ones having positive values. The absorbing states and the walls have v_π = 0, since it is not possible to transition on them and continue with movements.

SARSA vs. Q-Learning The results for SARSA are shown in Fig. 3, while the results for Q-Learning are presented in Fig. 4.

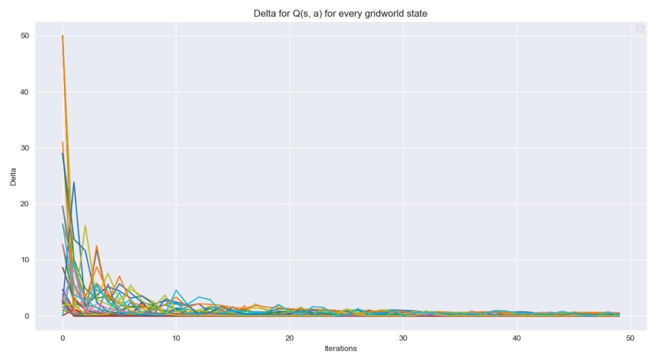

Fig. 3: Convergence to optimal policy for SARSA

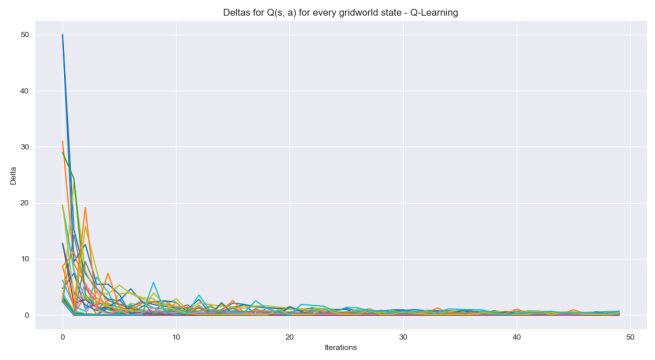

Fig. 4: Convergence to optimal policy for Q-Learning

By observing the two graphs, it can be seen that there are no major differences between the performance of the two algorithms, but the Q-Learning (Fig. 4) presents a slightly steeer curve, suggesting a better performance when compared to SARSA (Fig. 3). It can therefore be assumed that the performance of the Q-Learning algorithm was better compared to SARSA.

2.4 Discussion

In this second section, a gridworld Reinforcement Learning problem was analysed, by performing Monte Carlo Policy Evaluation, SARSA, and Q-Learning. For what concerns MCPE, the results showed that the equiprobable policy is not the ideal ones, since all v_π values are very low, and the only two positive ones (the adjacent cells) do not show the ideal value of 50 (which would entail just one move to the treasure absorbing state). For the comparison between SARSA and Q-Learning, the hypothesis was supported, showing that the second algorithm reaches convergence in less time than the first one, although both are able to find the optimal policy.

References

[1] Peter Auer. "Using Confidence Bounds for Exploitation-Exploration Trade-offs". en. In: (), p. 26.

[2] Levente Kocsis and Csaba Szepesvári. "Bandit Based Monte-Carlo Planning". en. In: *Machine Learning: ECML 2006*. Ed. by David Hutchison et al. Vol. 4212. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 282–293. ISBN: 978-3-540-45375-8 978-3-540-46056-5. DOI: `10.1007/11871842_29`. URL: `http://link.springer.com/10.1007/11871842_29`.

[3] Vaibhav Kumar. *Reinforcement learning: Temporal-Difference, SARSA, Q-Learning & Expected SARSA on python*. en. Oct. 2019. URL: `https://towardsdatascience.com/reinforcement-learning-temporal-difference-sarsa-q-learning-expected-sarsa-on-python-9fecfda7467e`.

[4] Gerard Martínez. *Reinforcement learning (RL) 101 with Python*. en. May 2019. URL: `https://towardsdatascience.com/reinforcement-learning-rl-101-with-python-e1aa0d37d43b`.

[5] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning, second edition: An Introduction*. English. 2nd. Cambridge, Massachusetts: Bradford Books. ISBN: 978-0-262-03924-6.