Final Assignment MAS

Tiddo Loos, 2574974

1. Monte Carlo Tree Search

The binary tree constructed for this assignment has an initial depth of 12. This means the tree consist of 8190 nodes. At the leaf nodes of the tree, random real numbers are assigned. The basic flow of the implemented Monte Carlo Tree Search (MCTS) algorithm is schematically shown in *Figure 1.*

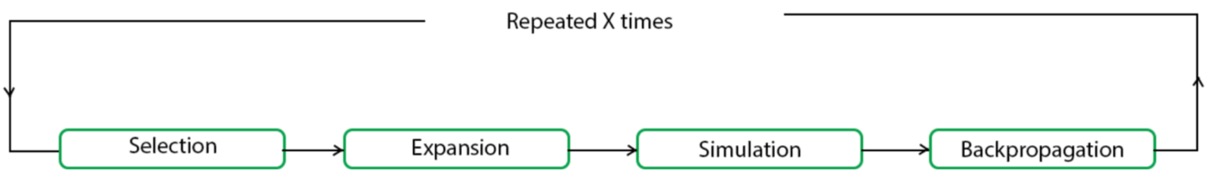
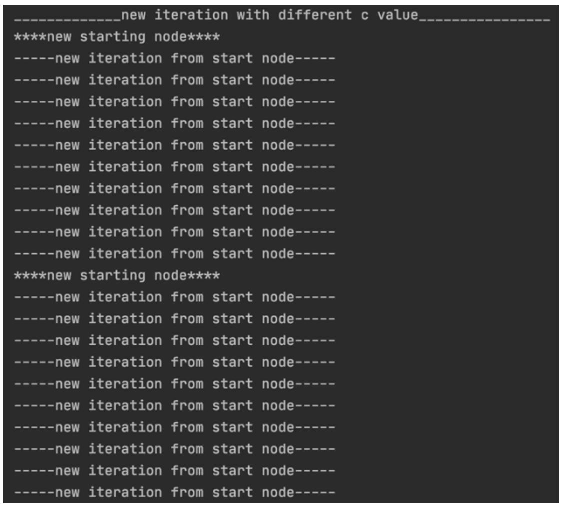


Figure 1: schematic overview of the MCTS flow.

The implementation of this MCTS has the aim the give the node with the highest average reward of the explored nodes in the search tree. The algorithm will never go all the way down to a leaf node of the constructed binary tree, it aims to give a direction (node) with the highest reward earlier in search tree.

The MCTS algorithm will repeat the flow of *Figure 1* five times and starts in the root node. Simulations are executed from the new explored node (creating a 'snowcap') with five rollouts per simulation. The rewards coming from the leaf nodes are summed and divided by the number of rollouts. The nodes in the path are updated with the rewards. Then, after ten iterations, a new starting node is selected from the explored nodes, maximizing the UCB-value. This is again iterated 10 times. In *Figure 2* the terminal output is shown during the iteration of the described process. The MCTS algorithm will give the explored leaf node with the highest average reward (total reward divided by the number of visits) at that moment as a result. For each value of *c* the results are collected 10 times and averaged out. This whole process is iterated a hundred times, increasing the c-value with 0.02 after each iteration. The initial value of *c=0.0*.



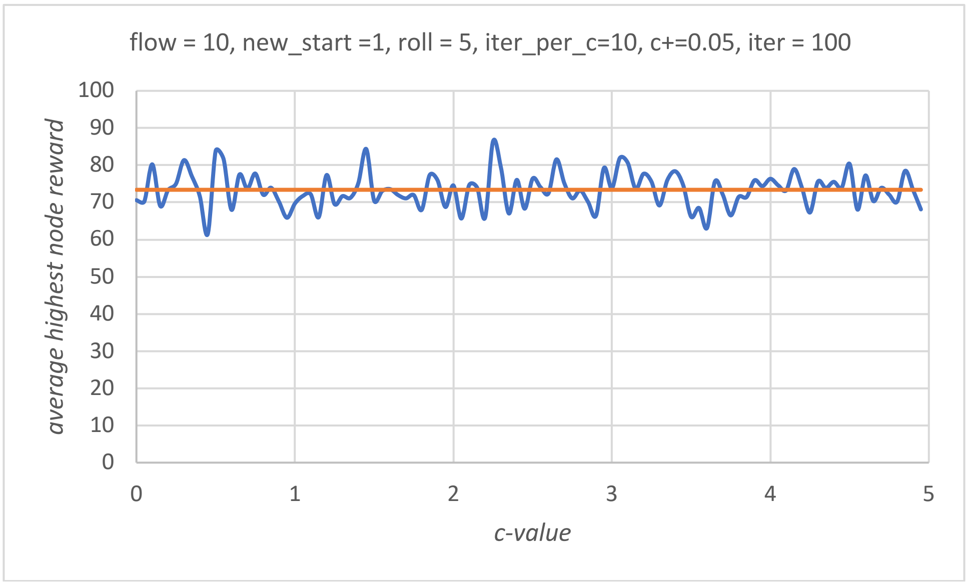
```
_____________new iteration with different c value_______________
****new starting node****
-----new iteration from start node-----
-----new iteration from start node-----
-----new iteration from start node-----
-----new iteration from start node-----
-----new iteration from start node-----
-----new iteration from start node-----
-----new iteration from start node-----
-----new iteration from start node-----
-----new iteration from start node-----
-----new iteration from start node-----
****new starting node****
-----new iteration from start node-----
-----new iteration from start node-----
-----new iteration from start node-----
-----new iteration from start node-----
-----new iteration from start node-----
-----new iteration from start node-----
-----new iteration from start node-----
-----new iteration from start node-----
-----new iteration from start node-----
-----new iteration from start node-----
```

Figure 2: Terminal output for 1 iteration (out of 10, out of 100)

To sum up the initial conditions starting the research on the MCTS implementation:

- The flow (*Figure 1*) is iterated 10 times starting in the root node of the tree *(flow = 10).*
- After the 10 iterations, a new starting node is selected based on the UCB-value of the explored nodes *(new_start = 1)*.
- When a new starting node is selected the other explored nodes, which are not ancestors of the selected new starting node, will not be taking into account (because this will take up memory when considering bigger search trees). The new starting node can be seen as a new root node.
- The simulations are performed with five rollouts per simulation *(roll = 5)*
- The process of *Figure 2* is iterated ten times for each value of *c (iter_per_c=10).*
- At the first iteration the value of *c=0.0*.
- The whole process is iterated a hundred times, increasing c after each iteration (*c+=0.05*)

The results of the setup as described above are presented in *Graph 1.*



Graph 1: Standard deviation = 5.2

After examining *Graph 1* the outlying higher rewards with the corresponding *c-values* were explored. For example: for *c=2.25* was ran 100 times to find out if this was an optimal value for *c.* It turned out it wasn't and running the experiment for c=2.25 resulted in a same average reward for the optimal nodes. What is striking is that the value for *c* doesn't seem to have an impact on the resulting average node reward. When looking at the formula for the UCB-value, where *c* plays a role, it decreases *(c<1)* or increases *(c>1)* the impact of the upper bound element in the UCB-value formula:

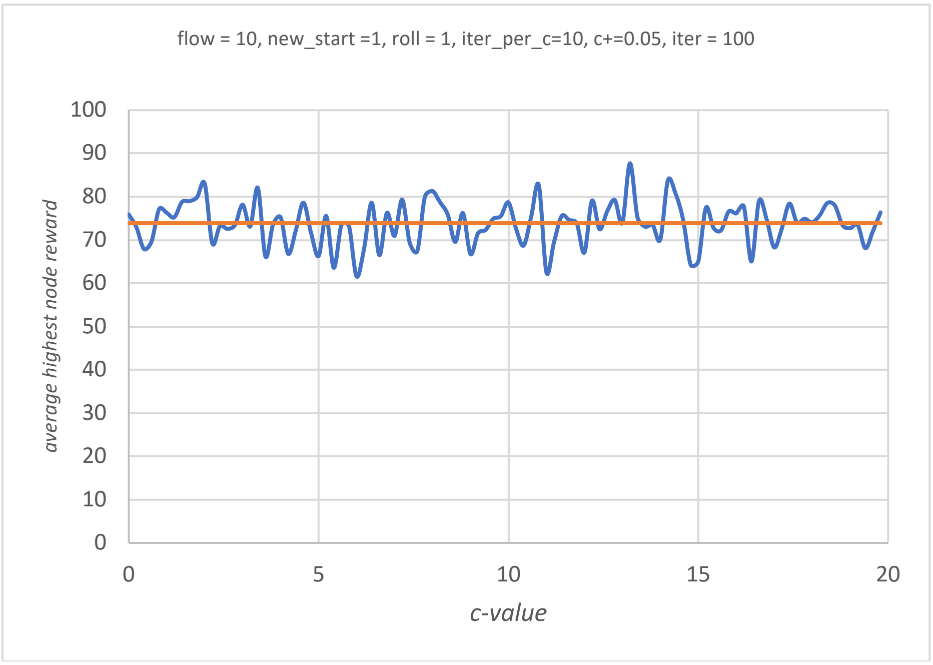
$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$$

Formula 1: formula for calculating the UCB-value.

The second term of the formula however, is a small number when considering rewards between (0 and 100) like in this assignment. The UCB-value relies mostly on the first term $\frac{w_i}{n_i}$.

Therefore, the possible cause for the average highest leaf nodes reward being around 73 for the best leaf nodes (red line in *Graph 1*), may be due to the fact that the rollout also averages the rewards of the leaf nodes down the search tree since the rollouts were set to 5. To give the simulated reward and the *c* value a greater influence, the next experimental conditions are as follows and the result is presented in *Graph 2*:

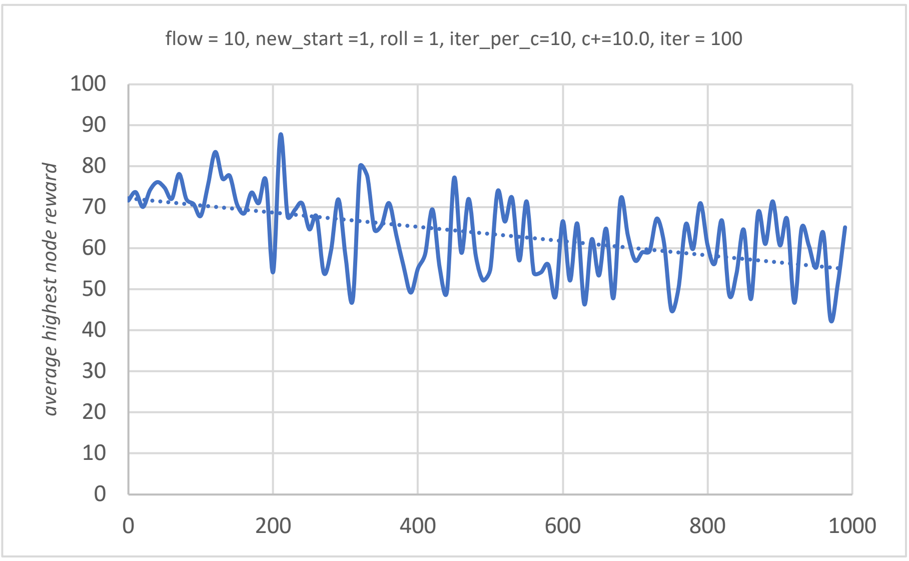
- *C = 0.0 and C+=0.2 (to give C a greater influence on the UCB-value)*
- *flow = 10, new_start = 1, iter_per_c = 10 and iter=100, roll = 1*



Graph 2: Standard deviation = 4.9.

In *Graph 2* the same average of around 73 is the result of the setup as explained above. The *c*-value increases to 20. However, this seems to have no impact on the resulted nodes and their average reward.

Changing the *c* value, with rewards between 0 and 100 has a greater impact when *c* changes with bigger steps. To show this, the experiment is conducted with *c+=10*, and the last node is chosen based on the highest UCB value of the explored nodes. The average reward of that node still is the resulting value and indicated in *Graph 3.* The trend line suggests a decrease in the average reward of the highest node.



Graph 3: Standard deviation = 9.7.

A last experiment is conducted. An extra new starting point is selected. This could result in a higher average reward of the selected node. The iterations are reduced as in a binary tree of depth 12 and with this it could be that MCTS ends up in a leaf node of the binary tree. This experiment was designed to generate a node as result that has the highest averaged reward, which is not a leaf node of the constructed binary tree. In this last setup the resulting node is selected, maximizing the average value of the node. The setup is as follows:

- *C = 20, C+=0 (arbitrarily chosen as the UCB influences now more the selecting procedure of the MCTS algorithm.)*
- *flow = 10, new_start = =2, iter_per_c = 10 and iter=100, roll = 1*
- *The new starting nodes are selected, maximizing the average node value.*

Result:

Average reward	Standard deviation
76,10	4,8

2. SARSA and Q-learning for grid world

2.1 Heatmap of state values

For this assignment the state values in a certain grid world are calculated. The state values are calculated with the following equation which is derived from the Bellman functions:

$$v_\pi(S_t) = \sum R_{t+1} + \gamma v_\pi(S_{t+1})$$

Where $v_\pi(S_t)$ is the current state, R_{t+1} is the reward, yield by going to the next state and $\gamma v_\pi(S_{t+1})$ is the state value of the next state. The discount factor is set to γ = 1.0 and all actions have probability 1/4. Thus, there is no discount when updating the state values. *Figure 3.* shows the heatmap indicating the values for each state. The map is constructed with 10000 iterations, updating the state values to the equilibrium presented in the figure below.

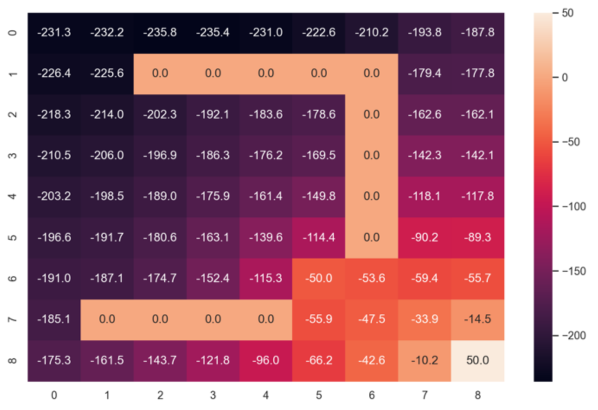


Figure 3: Heatmap of the state values.

2.2 SARSA and Q-learning: searching for optimal policies

For this assignment the learning factor (α) and the discount factor (γ) are fixed and set on 0.5 and 1.0 respectively.

For this experiment SARSA was combined with greedification updating the Q-values in the *north*, *south, east* and *west* direction for every state. An agent was initialized to walk through the grid world starting at a random position. It updates the Q-values based on the total reward when a terminal state (snakepit, or treasure) was reached. The *e* values for greedyfication are set as follows:

- SARSA *(e=0.1)*
- Q-learning *(e=0.0)*

The formulas that constantly update the q-values for both SARSA and Q-learning are presented below in *Figure 4.*

Q Learning:

$$Q(s_t, at_{)} \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)]$$

SARSA:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Figure 4: SARSA and Q-learning update formulas

To explain the difference between the two: The Q-value in the second part is sometimes randomly chosen for SARSA. When *e=0.1*, on average it chooses a random q-value (from *north, south, east, west*) 1/10 times to update the Q-value. Q-values always uses the maximum Q-value of the next state (the agent will take the action with maximum Q-value and ends up in that particular state. *Figure 4* shows the optimal policy maps, constructed with SARSA and Q-learning.

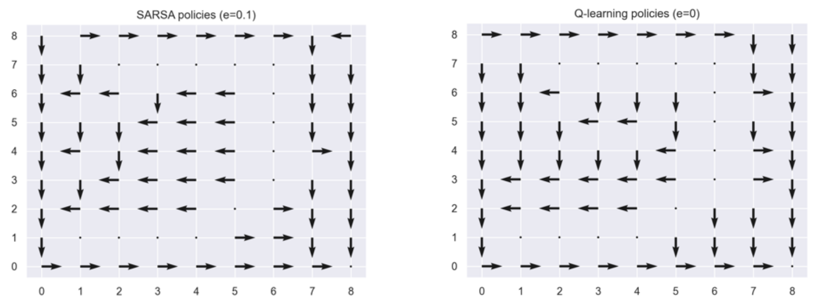


Figure 5: policy maps constructed with agent (10000 iterations)

The policy maps do not differ a lot form each other. The SARSA police map indicates a not optimal policy arrow in top right corner, while the Q-learning policy map shows a perfect optimal policy.

Both of these maps were formed by an agent that was initialized 10000 times. It did 10000 walks ending in a terminal state and updating the Q-values. The next Figures show the same 'quiver' maps but with the agent initialized fewer times to investigate which algorithm constructs best the optimal policies circumstances of less iterations.

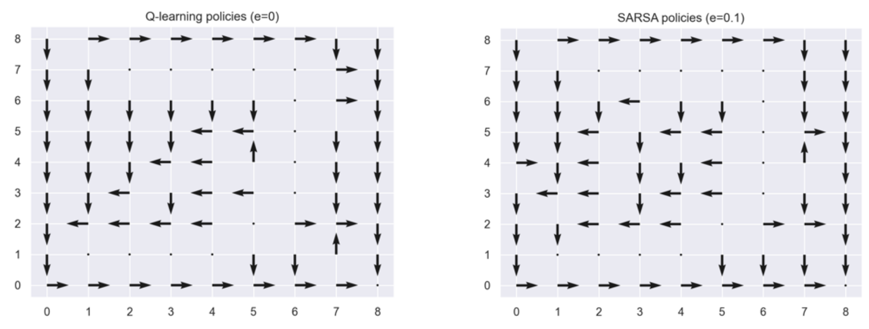


Figure 6: policy maps constructed with agent (1000 iterations)

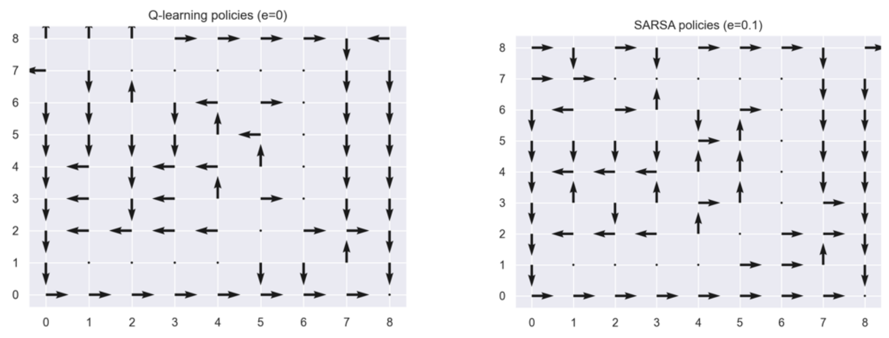


Figure 7: policy maps constructed with agent (500 iterations)

Considering *Figures 5, 6 & 7* it cannot be concluded that one of the two algorithms outperforms the other. Both maps show many not optimal policies.