Multi-Agent Systems Assignment 6

Alex Hoorn

December 30, 2021

1 Monte Carlo Sampling

In this task we consider choosing a house to rent as a sequential decision problem. We visit n houses. But after every visit we must make the choice to either rent the home or leave it. We cannot come back to a previously visited house.

To get a distribution of house rating we use Monte Carlo sampling. Here X is sampled from a standard normal distribution. Then we run every $x \in X$ through a function $f(x)$ to get the actual distribution of the house ratings. I have defined $f(x) = \sin^2(x)$

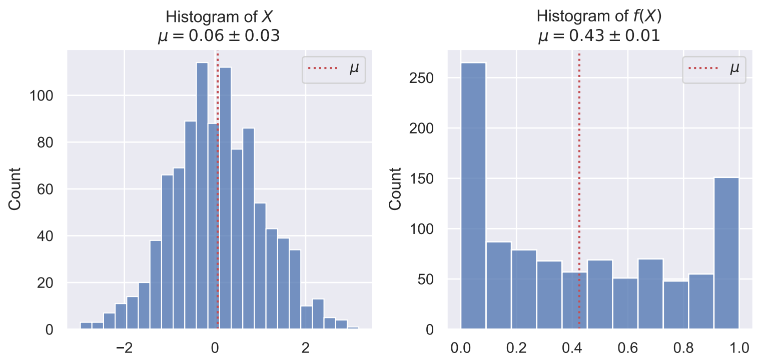

Figure 1: Monte Carlo sampling.

The strategy I use here is based on the statistical Law of Large Numbers[1]—LLN in short. I determine an expected utility after every n observations. And then simply pick the first observation which is better than the expectation.

[1]https://en.wikipedia.org/wiki/Law_of_large_numbers

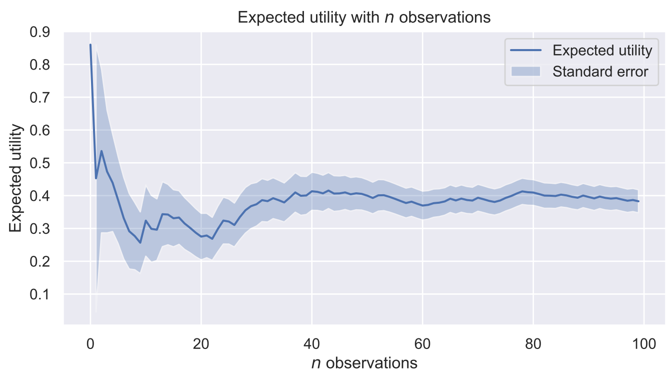

Figure 2: The Law of Large Number visualized for $f(X)$.

If simply taking the mean of the results into account then this strategy performs very well. Starting from a mean of about 0.75. However, I have noticed that the results of this strategy varies wildly in practice and thus probably shouldn't be relied on. In Figure 3 we can see that the range of the expected rewards is always rather large for any n.

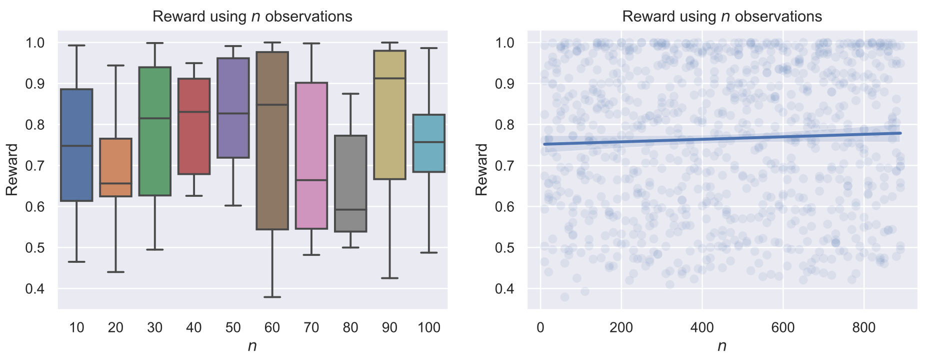

Figure 3: Results using the strategy "pick first $> e$".

2 Monte Carlo Tree Search

To develop and try out my Tree Search algorithm I have implemented a very basic class to supply the logic behind the binary tree. This class doesn't actually generate a whole tree on creation. Instead it generates a target path and lazily calculates a reward when asked. The benefit of this approach is that the tree can be (pretty much) of infinite size. However, in my testing I used a tree with depth 25.

Rewards are calculated using $x_i = Be^{-d_i/r}$. d is the edit distance between the path and the target set in the tree. B and r have been set to 2 and 20 respectively to make differences between different paths non-negligible.

One big difference between the default MCTS implementation and mine is that when a new node is reached its direct children are immediately rolled out. Instead of waiting for a new iteration that would select the child node (very often) anyway. This is much easier to do programmatically instead of dealing with unexplored children and divisions by zero.

MCTS traverses a tree by calculating the UCB of nodes. UCB is calculated by $UCB(\text{node}_i) = \bar{x}_i + c\sqrt{\frac{\log N}{n_i}}$. I have noticed that the tunable parameter c in this plays a huge role in finding the optimal reward in a tree. With a high c the algorithm is more likely to reach unexplored areas of the tree. This makes it more likely to find the optimal solution at the cost of additional iterations necessary to do so (and therefore runtime). The results are shown in Figure 4.

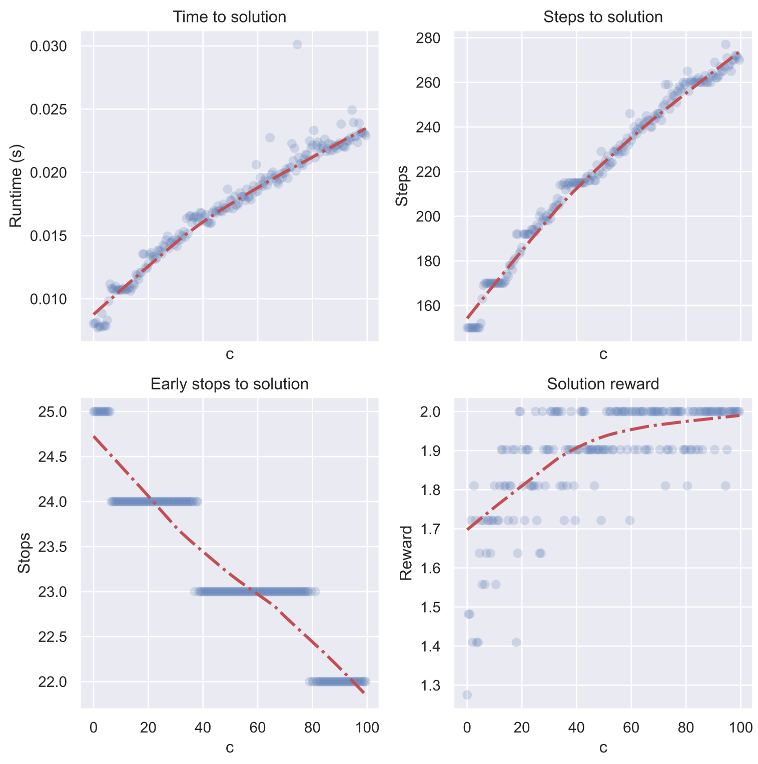

Figure 4: Statistics for different settings of c with 5 roll-outs and 50 iterations.

3 Reinforcement Learning

In this task we consider a 9×9 grid world as depicted in Figure 5. The blue cells depict walls to which we cannot walk. The green cell is a terminal cell which rewards 50 and the red cell is a terminal cell that yields -50. Any transition to a different state yields a reward of -1.

The grid world is evaluated with three different reinforcement learning algorithms:

- Monte Carlo sweep policy evaluation. *Computes the state value function for the equiprobable policy.*
- Greedy SARSA.
- Q-Learning.

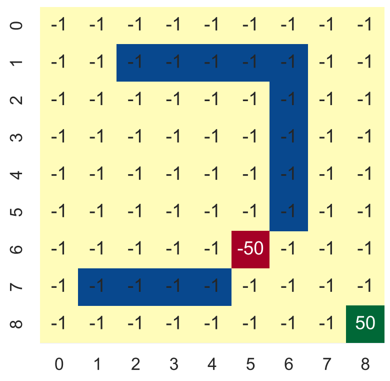

Figure 5: The environment, blue denotes walls, red and green are terminal states.

3.1 Monte Carlo Sweep

The Monte Carlo policy evaluation in combination with the equiprobable policy means that the agent simply *wanders around* the grid world until it reaches a terminal state. There is no clever selection of the action to take. The agent could even attempt to walk off the grid of against walls. This results in very low returned rewards, because for every transition the total reward is lowered by -1. The results for this evaluation is denoted in Figure 6.

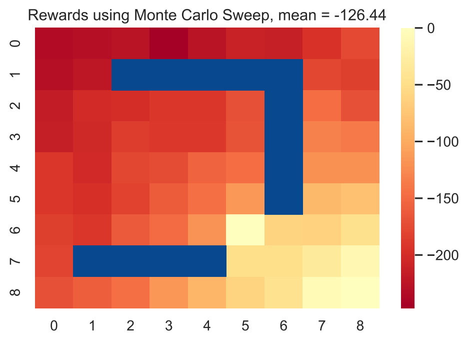

Figure 6: Heatmap of Monte Carlo Sweep.

3.2 Greedy SARSA and Q-Learning

Compared to the Monte Carlo Sweep the SARSA and Q-Learning algorithms are much smarter. These will not wander around aimlessly but actively follow the path to the best reward. Only with a chance defined by parameter ϵ will a random action be taken rather than an optimal action (ϵ-greedy).

The only difference between SARSA and Q-Learning is how they update the optimal values along the way. For SARSA the values are updated with:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma Q\left(s', a'\right) - Q(s,a)\right]$$

For Q-Learning we use:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a} Q\left(s', a\right) - Q(s,a)\right]$$

In the actual implementation Q-Learning inherits the SARSA class. It only changes the update method.

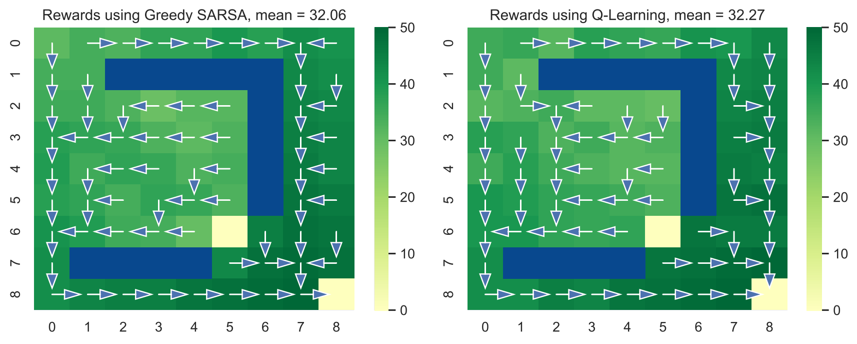

Figure 7: Heatmaps with path of Greedy SARSA (left) and Q-Learning (right)

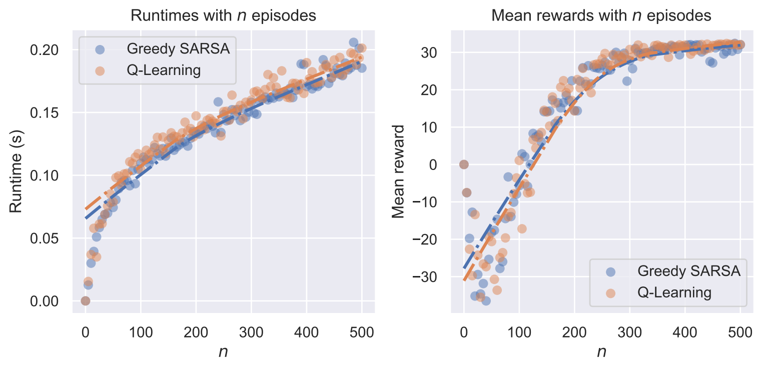

Figure 8: