About the code:

This code creates a game world in which the player can move around. The goal of the game is to find the treasure on the quickest possible way.

To run the game we created multiple grids. The size of the grids is 10x10. This includes the 8x8 game and a wall around it.

The grid for the q-values is initialized with zeros. During the game the grid is updates with every action in the game. The rewards for moving to each state are stored in a reward grid. The snakepit has a reward of -20 and the treasure a reward of +10, all other states reward -1.

The start position of the player is (1,1) and the game is terminated when player reaches the snakepit or treasure.

To move around the game world the player has four actions: Go up, go down, go left, go right.

If the player attempts to move into a wall, it gets rewarded -1 but remains in the same position.

With each action we can update the new position of the player and access his new state and q-value

We used Phython 3.7 and the Scipy library ‘numpy’ for this code

Output:

To visualize the game we output the q-grid and a vector with the visited positions. When the position list contains two following equal positions, it indicates that the player tried to run into a wall.

Q-learning:

We implemented the Q-learning on the grid game. The player explores the game world and updates the q-values of each position using this function:

GRAFIC: Function q

The constants set the priority of the player between exploration and exploitation.

The alpha value tells how much of the learned policy the player will apply. The gamma value is a discount factor on the reward. Epsilon defines how greedy the player acts.

We chose the following values:

Epsilon = 1

Alpha value 0.7

Gamma value 0.9

The games was iterated 10 000 times. This is the final q-grid:

GRAFIC: q-learn q-grid

The optimal solution is clearly defined and the path of the player lead straight to the treasure. This is the position list of the final iteration:

GRAFIC: q-learn pos

the area in the middle and lower left contains a lot of noise and not always the optimal policy. For example when the player is at position (4,6) it will move upwards instead of taking the shortest path and move left. This area lacks exploration.

SARSA:

For the implementation of SARSA we used the same code as above but with a new update policy. To update the q-values we uses this function:

GRAFIC: Sarsa function

The Sarsa implementation explores more the game world and finds alternative optimal policies.

Since the implemented code seems to use more computational resources than given, we could only iterate 13 times. The las position list and the resulting q-grid already shows the mentioned behavior:

GRAFIC: Sarsa pos

GRAFIC:Sarsa q-grid

The las path takes the upper way to the treasure. The updated grip indicates to go take the lower path in the next round.

With SARSA the player finds both shortest path, the upper and the lower, whereas Q-learning only choses the upper path.

Source:

<https://en.wikipedia.org/wiki/Q-learning>

<https://en.wikipedia.org/wiki/State%E2%80%93action%E2%80%93reward%E2%80%93state%E2%80%93action>