Multi-Agent Systems: Final Homework Assignment

Caroline Hallmann, 2640914

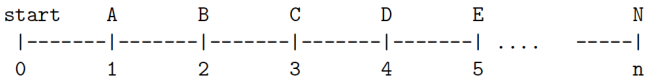
Vrije Universiteit Amsterdam, MSc AI

Abstract. This assignment consisted of three topical questions that require programming. The report examined the following three topics: 1) Monte Carlo estimation of Shapley value, 2) Monte Carlo Tree Search (MCTS) and 3) Reinforcement Learning: SARSA and Q-Learning for Gridworld.

Keywords: Shapley value · MCTS · SARSA · Q-learning.

1 Monte Carlo estimation of Shapley value

Consider n agents $(A, B, C \dots N)$ boarding a (sufficiently large) taxi at location 0 on the real line. They share the taxi to return home. Agent A lives at distance 1, B at distance 2, etc. In general, for agent a_i the distance to home equals i. The taxi driver agrees that they only need to pay the fare to the most remote destination (which is at location n).|



1.1 Question

Compute (using the theory) the Shapley values for this problem when n is small, e.g. n = 4 or 5. This will allow you to generalise to arbitrary values of n.

Solution

Method To compute the shapley values for when n = 4 and n = 5, the marginal contribution of an agent needs to be defined. It is the change in the total value of the coalition when the agent is added to it. The implementation of the Monte Carlo share taxi shapley value calculation method is given below:

```
from itertools import combinations

def shapley_value(n, i):
    total_value = 0
    for S in combinations(range(1, n+1), i):
        total_value += marginal_contribution(S, i) * probability(S, n)
    return total_value

def marginal_contribution(S, i):
    return i if i in S else 0

def probability(S, n):
    return 1 / 2**n

# Set n = 4
n = 4

# Calculate Shapley values for each agent
for i in range(1, n+1):
    print(f"Shapley value for agent {i}: {shapley_value(n, i)}")

# Set n = 5
n = 5

# Calculate Shapley values for each agent
for i in range(1, n+1):
    print(f"Shapley value for agent {i}: {shapley_value(n, i)}")
```

Answer After running the code in the notebook, 4 Shapley values were generated at n = 4 (Fig.1), and 5 values at n = 5 (Fig.2).

Shapley value for agent 1: 0.0625	Shapley value for agent 2: 0.375
Shapley value for agent 3: 0.5625	Shapley value for agent 4: 0.25

Fig. 1. Shapley values at n=4

Shapley value for agent 1: 0.03125	Shapley value for agent 2: 0.25	Shapley value for agent 3: 0.5625
Shapley value for agent 4: 0.5	Shapley value for agent 5: 0.15625	

Fig. 2. Shapley values at n=5

The results show that the shapley values for the agents are higher at n = 4 compared to n = 5.

1.2 Question

Now set n to a large value, e.g n = 50 or n = 100. From the above, you are able to guess what the Shapley values will be. However, use Monte Carlo sampling to find an approximate value for the Shapley values in this case. Discuss how effective Monte Carlo sampling is for this problem.

Solution

Method To calculate the approximate shapley values for large n_sample values, a large number of random permutations of the agents needs to be generated. The permutations are then used to estimate the shapley values. As the problem gives two large n_sample values, n = 50 and n = 100, it is necessary to use a more structured distribution of coalitions and Monte Carlo sampling. The applied method is illustrated in the code below:

```
import random

def approximate_shapley_values(n, num_samples):
    # Initialize a dictionary to store the sum of marginal contributions for each agent
    total_contributions = {i: 0 for i in range(1, n+1)}

    # Generate num_samples random subsets of the agents
    for _ in range(num_samples):
        S = random.sample(range(1, n+1), random.randint(1, n))
        for i in range(1, n+1):
            if i not in S:
                p = 1 / i # probability of adding agent i to the coalition
                if random.random() < p:
                    S.append(i)
            total_contributions[i] += marginal_contribution(S, i)

    # Compute the approximate Shapley values by averaging the marginal contributions
    approximate_shapley_values = {i: total_contributions[i] / num_samples for i in range(1, n+1)}

    return approximate_shapley_values

def marginal_contribution(S, i):
    return i if i in S else 0

# Show Approximate Shapley value computation scores at n = 50
print("Approximate_Shapley_values_for_given_problem_and_50_agents\n")
print(approximate_shapley_values(50, 10000))
# Show Approximate Shapley value computation scores at n = 100
print("\n\nApproximate_Shapley_values_for_given_problem_and_100_agents\n")
print(approximate_shapley_values(100, 10000))
```

Answer The approximate shapley values at n=50 and n=100 represent the importance of each agent in determining the final cost of the taxi fare. They are calculated by considering the marginal contribution of the agent to the total cost for a large number of randomly generated coalitions of agents, and averaging these contributions. The resulted approximate shapley values at n=50 and n=100 with a large number sample of 10000 are shown via the blue and red lines on the given line graph (Fig.3).

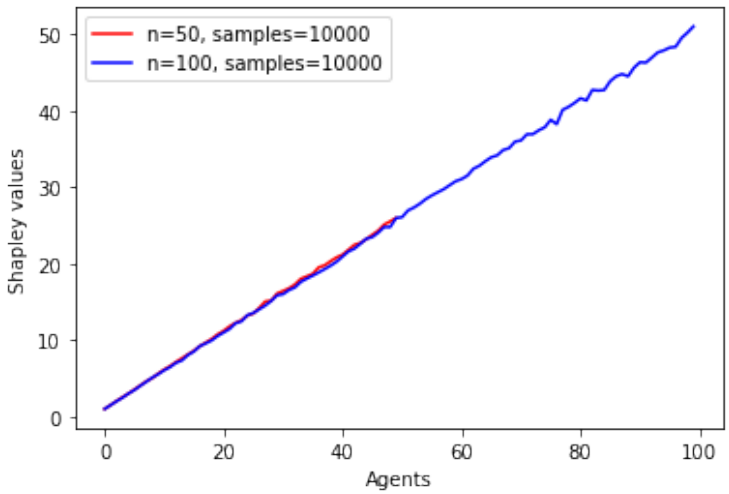


Fig. 3. Approximate shapley values at n=50 and n=100

From the graph, it can be seen the estimated taxi fare cost, represented as 'shapley values' on the y-axis, increases with the number of agents on the x-axis. When n = 100, the estimated shapley value is higher than when n = 50.

Discussion In general, the Monte Carlo method is a good choice for approximating the shapley values when the number of agents is large and the number of coalitions is too large to enumerate. The accuracy of the approximate shapley values depend on the value of num_samples. The larger the 'num_samples', the more accurate the approximation will be. However, when increasing the value of 'num_samples', the computational cost of the function rises. As stated previously, Monte Carlo sampling uses random generated coalitions of agents which influences the models accuracy. As a solution to this issue, a more structured distribution of coalitions represented via the probability of adding agent i to the coalition needs to be integrated (see code above).

Construct binary tree Construct a *binary tree* of depth $d = 20$ (or more – if you're feeling lucky). Since the tree is binary, there are two *branches* (aka. edges, directions, decisions, etc) emanating from each node, each branch (call them L(eft) and R(ight)) pointing to a unique child node (except, of course, for the leaf nodes – see Fig 1). We can therefore assign to each node a unique "address" (A) that captures the route down the tree to reach that node (e.g. $A = LLRL$ – for an example, again see Fig 1). Finally, pick a random leaf-node as your target node and let's denote its address as A_t.

Assign values to leaf-nodes Next, assign to each of the 2^d leaf-nodes a value as follows:

- For every leaf-node i compute the **edit-distance** between its address A_i and the address A_t of the target leaf, i.e. $d_i := d(A_i, A_t)$.
- Recall that the **edit-distance** counts the number of positions at which two strings differ, e.g.:

$$d(LRLR, LRRR) = 1, \qquad d(LRRL, LLLL) = 2, \qquad d(RRLL, RLRR) = 3$$

- Finally, define the value x_i of leaf-node i to be a decreasing function of the distance $d_i = d(A_i, A_t)$ to the target node, and sprinkle a bit of noise for good measure: e.g.

$$x_i = Be^{-d_i/\tau} + \varepsilon_i$$

where B and τ are chosen such that most nodes have a non-negligible value (e.g. $B = 10$ and $\tau = d_{max}/5$ and $\varepsilon_i \sim N(0, 1)$).

2 Monte Carlo Tree Search (MCTS)

2.1 Question

Implement the MCTS algorithm and apply it to the above tree to search for the optimal (i.e. highest) value.

Assume that the number MCTS-iterations starting in a specific root node is limited (e.g. to 10 or 50). Make a similar assumption for the number of roll-outs starting in a particular ("snowcap") leaf node (e.g. 1 or 5).

Solution The Monte Carlo Tree Search (MCTS) algorithm is a heuristic search algorithm for decision-making problems. The method includes the creation of search trees and running them through the play-outs process. The tree consists of nodes and edges. The nodes represent a particular state in the decision-making problem and the edges between the nodes show possible action for that certain state.

The MCTS algorithm consists of four main steps: selection, expansion, simulation, and back-propagation (Fig.4) [1].

1. Selection Selection of the child node with the highest UCT value for the current node.

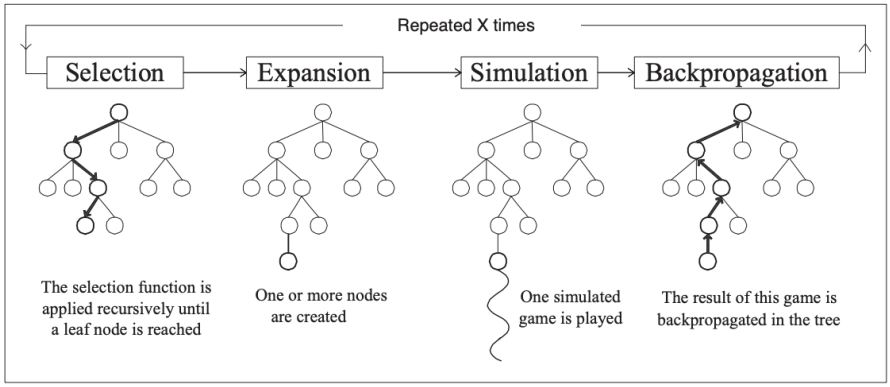


Fig. 4. MCTS algorithm

2. Expansion Expansion of nodes in the current tree, resulting in sub-nodes (children). Randomly select one node.

3. Simulation Choose a random direction (L or R) until the terminal node is reached. Perform random roll-out to leaf nodes starting at specific 'snowcap'.

4. Back-propagation After reaching a leaf node, update all the values and visits obtained in the selection and expansion step to total value and increase visits by 1 unit.

UCB To measure the performance of MCTS algorithms, the Upper Confidence Bounds (UCBs) are calculated. UCB scores are used to estimate the highest expected end reward of a node [2]. UCBs can thus help in deciding which node to choose in order to receive the maximum possible final reward.

$$UCB(node_i) = \frac{w_i}{n_i} + c \cdot \sqrt{\frac{ln(N)}{n_i}}$$

where:

- w_i : cumulated node's value (reward)
- n_i : number of time the node has been visited
- c : confidence value that controls the level of exploration
- N : total number of times that the node's parent has been visited

Method To apply the MCTS algorithm to the binary tree and search for the optimal value, we can instantiate an MCTS object with the desired parameters, and call the search method with the desired number of iterations. The search method can then use the Node class to implement the four steps of the MCTS algorithm and return the optimal value. To solve the given problem, different coding methods were used to implement a binary tree of depth = 20.

Code description The first step was to code the binary tree structure which included the following parts:

1. Define 'edit_ distance' function
2. Define 'Node_class'
3. Define 'MCTS_class' under 'Node_class'
4. Implement 'build_Tree' method to run binary tree
6. Define 'search' method in 'MCTS_class'
7. Create an instance of the 'MCTS_class' with desired values
8. Call the 'search' method to perform search and return optimal node

Answer For the implementation process, a tree_search file was used that contained all the MCTS methods and the structure of a binary tree with the snowcap rule. To run the code, the parameters were set to depth = 20, c = 2 and B = 25, as stated in the given problem. The function used to build the binary tree, creates the nodes at each depth level and then links the child nodes to its parents with values assigned to its leaf nodes. Ultimately, it returns the root node of the whole tree, enabling nodes to travel across the entire tree.

To run the 4 MCTS steps given above, the 'tra = mc.run()' function was applied. By running this command, the 4 steps were implemented with the set parameters.

As MCTS is used to search for the optimal value, it evidently focuses on nodes holding higher accumulated value estimates. It uses these values to guide towards highly rewarding trajectories in the search tree. The trajectory values at d = 50 were calculated and resulted in a set of optimal and returned trajectory values. The value of the terminal node in trajectory = 6.23 with distance = 5.

2.2 Question

Collect statistics on the performance and discuss the role of the hyperparameter c in the UCB-score.

Solution As stated, in section 2.1, the c value controls the level of exploration and thus affect the UCB scores.

Method To explore the impact of the c hyperparamater in UCB scores, an experiment was conducted with different c values to collect results based on performance.

The first step involved changing the parameters to depth = 5 and sample = 10. To explore the impact of changing C values on MCTS performance and UCB

scores, c values ranging from 0 to 10 were evaluated and the payoff received was plotted on a graph.

Answer The graph below show the generated payoff results from c values between 0 and 10 (Fig.5)

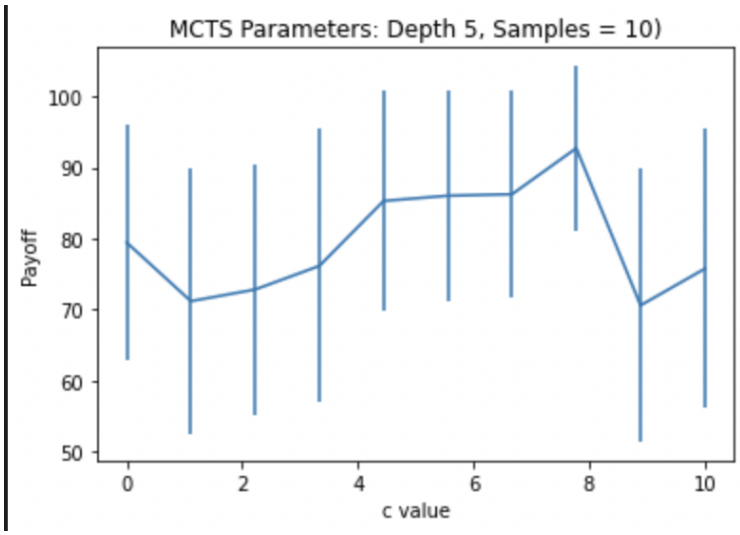


Fig. 5. Effect of parmeter c values on MCTS performance

The above figure shows the standard deviation as the vertical lines at each c interval. Based on the above analysis, low c values between 0 and 2 and high c values between 8 and 10 do not have a significant effect on performance and can be eliminated. At c = 2, the slope starts to increase until reaching a c value of 8. Thus, c values ranging between 2 and 8 at the set parameters (depth = 5 and sample = 10,), show a positive effect on the value of payoffs. This shows that changing the c parameter does indeed affect the MCTS performance. For the future, more experiments can be conducted that take into account different depth values, c values and sample sizes.

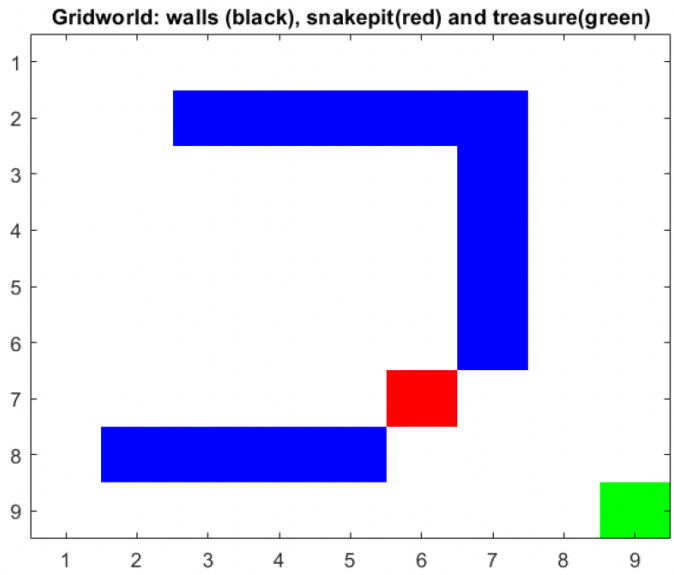
3 Reinforcement Learning: SARSA and Q-Learning for Gridworld

3.1 Question

Perform, compare and comment on, the following experiments: Use SARSA in combination with greedification to search for an optimal policy.

Consider the 9×9 gridworld example depicted in the figure 2 below. The blue gridcells represent walls that cannot be traversed. The green cell represent a treasure and transition to this cell yields a reward of $+50$ whereupon the episode is terminated (i.e. absorbing state). The red cell represents the snakepit: this state is also absorbing and entering it yields a negative reward of -50. All other cells represent regular states that are accessible to the agent. In each cell, the agent can take four actions: move north, east, south or wes (not moving is NOT a valid action). These actions result in a *deterministic transition* to the corresponding neighbouring cell. An action that makes the agent bump into a wall or the grid-borders, leaves its state unchanged. All non-terminal transitions (including running into walls or grid borders) incur a negative reward ("cost") of -1.

For the questions below, we assume that the agent is not aware of all the above information and needs to discover it by interacting with the environment (i.e. model-free setting).



Solution The SARSA algorithm is used to find an optimal policy in the given 9 x 9 Gridworld. SARSA is an RL method that finds the optimal policy π^* by running the following two steps: 1) Policy Evaluation and 2) Policy Improvement[4]. Step 1, estimates the State_Action function q_π in the current state. Step 2, uses greedification to select the best action from the previous estimation of q_π.

Method To implement SARSA, the Gridwold and actions taken by agents are first defined. The Gridworld is a 2D array, where the value of each cell represents the cell type of the cell through colors. In each cell, agents can take four actions which are defined as "North", "East", "South", and "West".

The SARSA algorithm takes into account the following inputs: 1) Gridwold, 2) List of actions, and 3) Starting position of the agent[4]. The function iterates over a number of episodes, in which the agent explores the Gridworld and learns the optimal policy. In each episode, the agent selects an action using an epsilon-greedy policy (greedification). The action with the highest expected reward with a probability of 1-epsilon and a random action with a probability of epsilon is then chosen.

After selecting an action, the agent executes the action and receives a reward from the environment. The Q value of the state action is updated by the error and adjusted by the learning rate α. The Q value represents the possible reward for taking action a in state s in the next time step. It also takes into account the discounted future reward received from the next state-action observation[4]. The agent then updates its action-value function using the SARSA update rule, which is given by:

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

After updating the action-value function, the agent selects the next action using the same greedification policy. This is repeated until the episode is terminated by reaching the treasure, snakepit, or maximum number of steps[4].

Answer The SARSA algorithm using greedification with a random initial was run to search for an optimal policy and optimal state values. SARSA was performed on the given 9 x 9 Gridworld. A total of 405 number of episodes were run using the greedification method. Graphs showing the the optimal policy and state values were generated at episode = 405 (Fig.6, Fig.7).

Figure 6 represents the Q tables generated with SARSA using the e-greedy method and figure 7 shows the total rewards across all 405 learning episodes. Figure 7 shows that the convergence of rewards start at roughly 75 number of episodes.

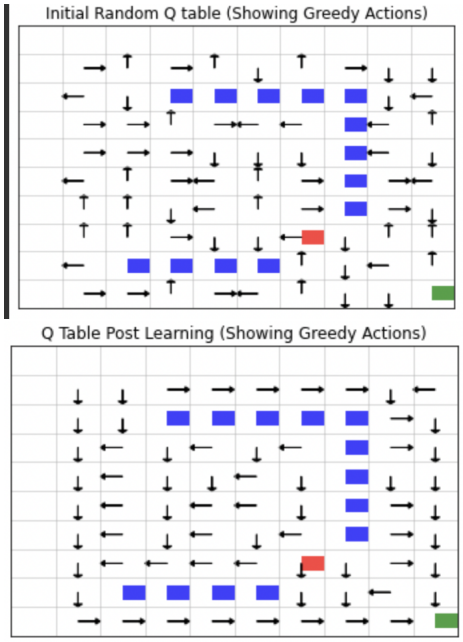


Fig. 6. Initial random and post learning Q Table, Episodes: 405

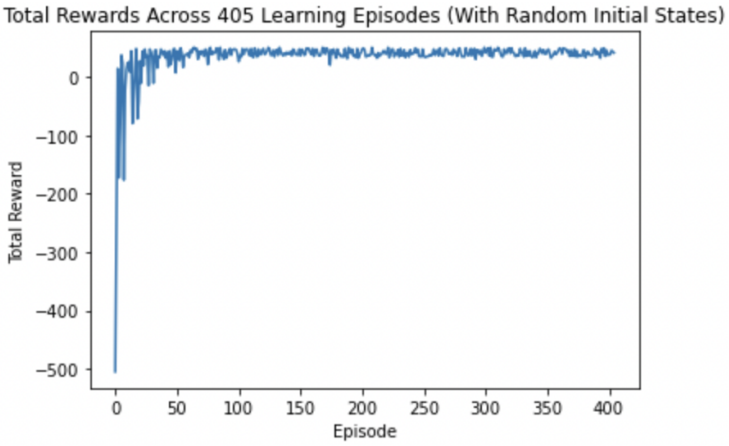


Fig. 7. Total Rewards with SARSA using e-greedy, Episodes: 405

3.2 Question

Use Q-learning to search for an optimal policy. Implement two different update strategies:

(a) Direct updates: Update the Q-table while rolling out each sample path;

(b) Replay buffer: Collect the experiences (s, a, r, s') in a replay buffer and sample from this buffer.

Compare the solutions and the corresponding computational effort for the three solution strategies.

Solution Q-learning, similar to the SARSA algorithm, is used to evaluate optimal policy. Q-learning is applied to the FMDP and starts at the current state. To find the optimal policy, it maximizes the expected value of the total reward for all successive steps[3]. Q-learning follows a partial random strategy and has an infinite exploration time from which it derives the best action selection strategy for any given FMDP. Q-learning is structured as an off-policy learning algorithm, enabling the implementation of two different update strategies[3]. It also follows a deterministic action selection approach based on the estimated state action-value function and the highest action value[3]. In Q-learning, the update of Q of state action also follows the e-greedy method and is updated using the following equation:

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

When executing the off-policy algorithm and applying the e-greedy method, Q-tables are generated. To implement two different update strategies into Q-learning, the Q-tables are updated first, using the direct update rule, and second, using the replay buffer rule.

Method The implementation process is similar to SARSA, only with the addition of the direct updates and replay buffer, along with the necessary helper functions. First, the direct updates were implemented with the same number of episodes, e = 405 and at the same method used before in SARSA. The 405 episodes were simulates, starting at an initial random position every time. Q-learning was then executed with the e-greedy and the Q-table was directly updated while rolling out each sample path.

Second Q tables for q-learning using the replay buffer method were generated. The method included a non-learning approach of simulating the 405 episodes and then executing Q-learning based on the sample from the stored experiences.

Answer The generated graphs for Q-learning with the direct update strategy are shown below (Fig.8, Fig 9).

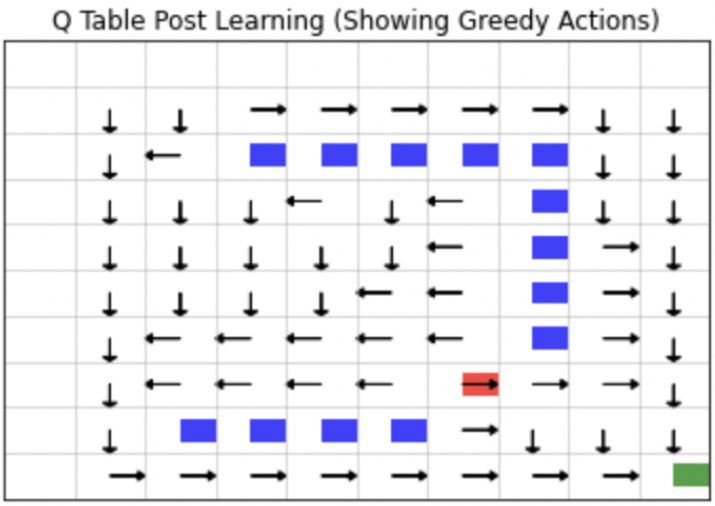


Fig. 8. Post learning Q table for Q-learning with the direct Episodes: 405

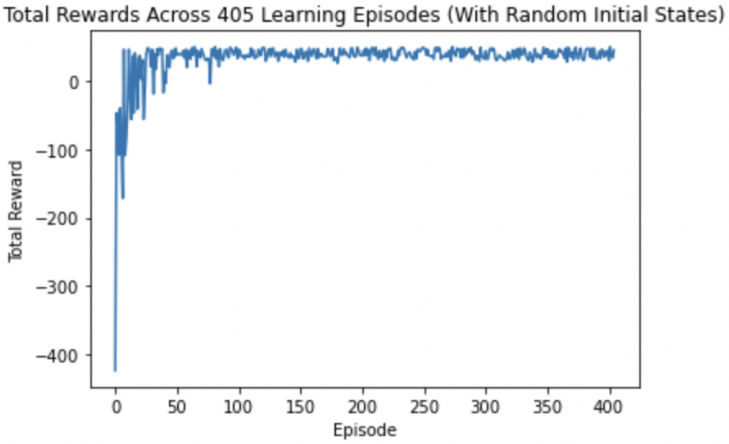


Fig. 9. Total Rewards with Q-learning using e-greedy and direct update, Episodes: 405

The results shown in fig.8 are similar to the trends displayed in fig.7 of the total rewards generated with SARSA and the greedification method. Both methods start to converge at roughly episode value 75.

After running Q-learning with the replay buffer method, the following graphs were produced (Fig.10, Fig.11).

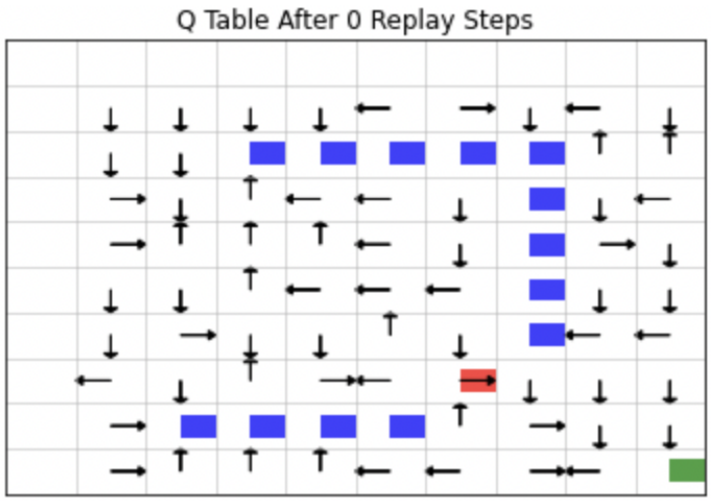


Fig. 10. Q-learning Q table with buffer method after 0 replay steps

The results shown in figure 10 and 11 show the most diversity comapred to the SARSA and Q-learing direct update method results. The main difference of this method that it does not the 405 sequential episodes and the other two methods. Overall, SARSA and q learning shown as the two classical examples of on policy and o policy learning approaches in RL.

References

1. Boris Iolis and Gianluca Bontempi. Comparison of selection strategies in monte-carlo tree search for computer poker. 01 2010.
2. Thomas M. Moerland, Joost Broekens, Aske Plaat, and Catholijn M. Jonker. Monte carlo tree search for asymmetric trees. *ArXiv*, abs/1805.09218, 2018.
3. Alexey Skrynnik, Aleksey Staroverov, Ermek Aitygulov, Kirill Aksenov, Vasilii Davydov, and Aleksandr I. Panov. Forgetful experience replay in hierarchical reinforcement learning from demonstrations, 2020.
4. Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018.

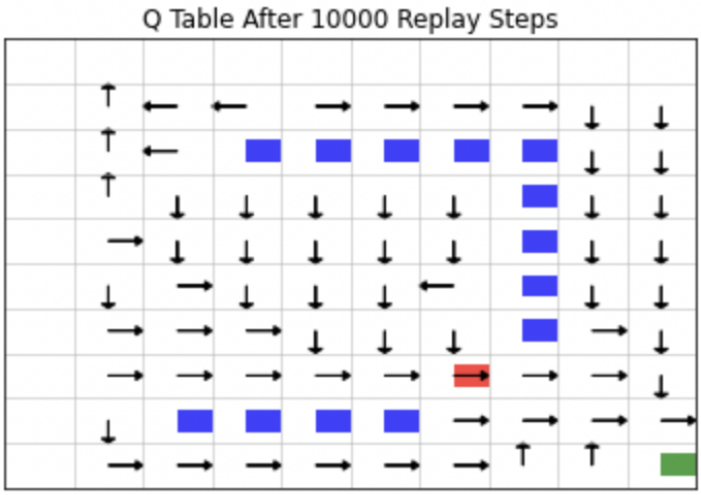


Fig. 11. Q-learning Q table with buffer method after 10000 replay steps