MAS Individual Assignment

Haohui Zhang
2722930
Vrije University Amsterdam
h17.zhang@student.vu.nl

December 31, 2021

1 Introduction

In this assignment, we learn in depth about Monte Carlo sampling, Monte Carlo Tree Search, SARSA and Q-Learning. Practical applications of these principles are examined through experimenting with applications in renting problem, binary tree and learning optimal policy for grid world. Throughout this report questions are answered in such depth to form a bridge between the theory and its application in the reinforcement learning domain. By reading this report my understandence of the fundamental mechanisms on these algorithms and how they are related hopefully becomes apparent to the reader.

2 Monte Carlo Sampling

2.1 Basics

Monte Carlo sampling is a method of estimating the value of an unknown quantity with the help of inferential statistics. The basic concept of Monte Carlo is to use randomness to solve problems that may be deterministic in principle. Monte Carlo has 3 main usages: estimate parameters or statistical measures, examine the properties of the estimates, approximate integrals. In our question, we use Monte Carlo to estimate statistical measures.

Let $X_1, X_2, X_3, \ldots, X_n$ be a sample from given discrete or continuous distribution, we can get:

$$EX = \begin{cases} \sum_{k=0}^{\infty} x_k P(X = x_k) = \sum_{k=0}^{\infty} x_k P_k & \text{if discrete} \\ \int_a^b x f(x)\, dx & \text{if continuous} \end{cases} \approx \frac{1}{n} \sum_{i=1}^{\infty} X_i \tag{1}$$

Besides, it is important to know that MC estimate depends upon two things:

1. Size: the size of the sample
2. Variance: variance of the sample

2.2 Renting problem

This question asks us to use MC to explore possible approaches and to maximise the probability that we will pick the best house in the list. The precondition is that the sample size n is sufficiently large, which means $n \to \infty$, as well as the score of each house is $0 \leq x \leq 1$. According to 1 and $n \to \infty$, $EX = 0.5$, thus we randomly assign score x_i to all candidate houses between 0 and 1 (contains 0 and 1) and the mean of the score can be regarded as 0.5.
This question essentially belongs to optimal Stopping, which is concerned with the problem of choosing a time to take a particular action, in order to maximize an expected reward or minimize an expected cost.
If the decision to select an candidate house was to be taken in the end of all appointments n, a simple

n	n/e	max score	min score	avg score
10	4	1.0	0.0	0.501
100	37	1.0	0.0	0.627
1000	368	1.0	0.0	0.749
10000	3679	1.0	0.0	0.999
100000	36788	1.0	1.0	1.0

Table 1: Simulation results using MC sampling with different n

solution is to use maximum selection algorithm, track the running maximum and choose the overall maximum at the end. However, the difficult part of this problem is that the decision must be made immediately after interviewing a candidate.
If we think about it carefully, we obviously cannot choose the first candidate house because we cannot compare the first candidate with others. **A better strategy is to select a few candidates as samples and set a benchmark for the remaining candidates**. Therefore, we use the 1/e law of Optimal Strategy.
According to this strategy, the optimal probability of selecting the best is always at least 1/e. The optimal stopping rule stipulates that the first n/e candidate house is always rejected, and then stops at the first one more than every candidate house visited so far. In this way, the probability of selecting the best candidate is 1/e. In other words, this strategy selects the best candidate approximately 37 percent of the time.
Therefore, optimal sample size k is n/e and probability of success for different values of n is given by:

$$P(x) = x \int_1^x \frac{1}{t}\, dt = -xln(x)$$

, where x = k/n.
Because we cannot simulate the infinite n, we did some comparison experiments to analyze and infer the conclusion. Each n is iterated for 1000 times. According to the Table 1, we find that if the sample size n is sufficiently large, using the strategy which select a few candidates as samples and set a benchmark will always get the best house in the list.

2.3 Suggestion

After I listened carefully to all the lessons of the teacher, including all the lessons, and will watch the recording and broadcasting again during the review, I found that the Kullback-Leibler Divergence is a difficult knowledge point. Although, in the homework 4, there is some proof and experiments, I still hope our teacher can provide a real life problems to practice and taht will make us understand the KL Divergence more sufficiently. As far as I know, KL divergence is a way of measuring the matching between two distributions. Thus, for example, combine the renting problem with two different score distribution. Then compute the KL divergence for each of the approximate distributions we came up with, and quantitatively decide which ones the best, after that use the better distribution to assign the score for the renting problem and use MC to explore.

3 Monte Carlo Tree Search (MCTS)

MCTS can be used along with different heuristics and is most-easily applied to games with an inherent tree structure. Thus, we try to develop an MCTS algorithm to solve this model-free reinforcement leaning problem.

3.1 Construct the environment

In order to solve the problem, we firstly need to construct the environment, a binary tree, which depth is 12, each branch pointing to a unique child node, each node have an address and using edit-distance algorithm to assign values to leaf-nodes.

1. For assigning value, we need firstly to pick a random leaf-node as the target node whose address is A_t.

2. For every leaf-node i compute the edit-distance between its address A_i and A_t based on Levenshtein distance, the equation is as follows:

$$levenshtein(a,b) = \begin{cases} |a| & \text{if —b—=0} \\ |b| & \text{if —a—=0} \\ levenshtein(tail(a), tail(b)) & \text{if a[0]=b[0]} \\ 1 + min \begin{cases} levenshtein(tail(a), b) \\ levenshtein(a, tail(b)) \\ levenshtein(tail(a), tail(b)) \end{cases} & \text{otherwise} \end{cases} \tag{2}$$

3. Finally, define the value x_i of leaf-node i to be a decreasing function of the distance levenshtein(A_i,A_t) to the target node:

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

In our simulation, we assign the hyper-parameter $B = 23$ and $\tau = 5$.

3.2 MCTS

The only precondition of MCTS is a generative model of the environment show below:

$$x', r \sim G(x, a). \tag{3}$$

In our environment, it meets this prerequisite, for taking each action, it will go to another node, however, the rewards are only given after reaching the terminal node which is the leaf node.
The overall idea of MCTS is to estimate the highest expected return.

$$V^*(x) = Q^*(x, a = \pi^*) = E_{\pi^*}[r_o + \gamma r_1 + \gamma^2 r_2 + \cdots] \tag{4}$$

From the above equation, we calculate the highest expected return and use it to find the best policy.
Except for the epsilon-greedy, we determined to use Upper Confidence Bound 1 applied to trees(UCT) to explore because UCT's efficient exploration of the tree enables to return rapidly a good value, and improve precision if more time is provided.
UCB1-Algorithm formula is as follows:

$$a^*_{t+1} = \underset{a}{argmax}(Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}}) \tag{5}$$

The main difficulty in selecting child nodes for MCTS is to maintain a certain balance between the development of post-movement depth variants with a high average win rate and the exploration of moves with few simulations. To solve this, we generalize the UCB1-Algorithm into MCTS which becomes to UCT. Now, the formula is transformed to:

$$a^*_{i+1} = \underset{a_i}{argmax}(\frac{r_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}) \tag{6}$$

Where a is all selectable nodes; r_i is the sum of rewards of the node considered after the ith move; n_i represents the number of simulations for the child node; N_i defines total number of simulations for the parent node.
The first part of the above formula corresponds to exploitation; it is high for nodes with a high average winning percentage. The second part corresponds to exploration; high for nodes that are rarely simulated.
Based on the Figure 1, we refine and explain each steps:

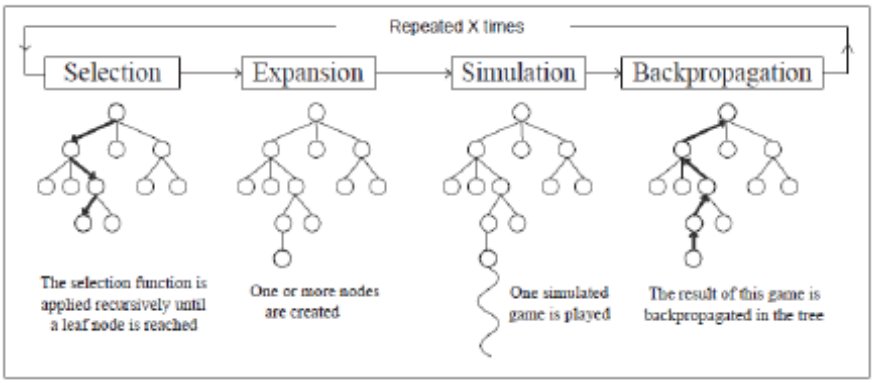


Figure 1: Illustration of MCTS algorithm (Chaslot et al. 2008)

c	max reward	min reward	avg reward
0	23.0	10.33	16.82
0.05	23.0	10.33	16.39
0.1	23.0	10.33	16.50
0.5	23.0	10.33	16.39
1	23.0	12.62	17.44
1.4	23.0	12.62	18.37
2	23.0	12.62	19.18
5	23.0	12.62	21.46
10	23.0	5.67	17.63

Table 2: Simulation results using MCTS with different hyperparameter c

1. Selection : Used for nodes we have selected before - pick the action according to UCT
2. Expansion : Used when we reach a sub node in the current tree - randomly select one node
3. Simulation : Used for the new sub node - perform a random rollout to the leaf nodes
4. Backpropagation : After reaching a leaf node - update value and visits for nodes expanded in selection and expansion

After repeating a fix times which is limited between 10 to 50, we obtain the numerical results of the current node, find the policy which nearly converges to the optimal one and then move to next node. The algorithm is shown below 1.

3.3 Simulations and analysis of hyperparameter c

For all the simulations, the number MCTS-iterations starting in a specific root node is **23** the number of roll-outs starting in a particular ("snowcap") leaf node is **1**. The iteration times for each c is 100. Because, in the previous section, we fixed the hyperparameter $B = 23$, $\tau = 5$, so the leaf reward distribution is [23.0, 15.42, 12.62, 10.33, 8.46, 6.93, 5.67, 4.64]
The simulation result is shown in the below Table 1.

Based on the simulation results in Table 1, we find that whether or not finding the best policy is related to the value of hyperparameter c.
First of all, we can see from the max reward column that no matter how c is set, as long as the number of traversals is sufficient, the optimal strategy will always be found, but the number and probability of finding the optimal strategy are different. From the min reward column, we can conclude that the situation where c is too large is basically undesirable. This is because if c is too large, the explore term

Algorithm 1 UCT

$r \leftarrow 0$
$n \leftarrow 0$
$N \leftarrow 0$
for $d = 0, 1, \ldots, 12$ **do**
  for $t = 0, 1, \ldots, 23$ **do**
    for each $a \in \mathcal{A} = (L, R)$ **do**
      Compute the UCB bounds according to $B_i \leftarrow \frac{r_i}{n_i} + c\sqrt{\frac{\ln N_i}{n_i}}$
    end for
    if $B_i > B_{i'}$ **then**
      Select the node i_{t+1} as a child of node i_t
      $i^*_{t+1} \in argmax_{i_t}(B_i)$
    else
      Randomly select a sub-node i_{t+1} as a child of node i_t
    end if
    $n_i \leftarrow n_i + 1$
    Roll-out to the terminal node using pure random selection
    $r_i \leftarrow r_i + new\_r$
    $N \leftarrow N + 1$
  end for
  for each $a \in \mathcal{A} = (L, R)$ **do**
    Compute the UCB bounds B_a
  end for
  $a^*_d \in argmax_a(B_a)$
end for
return optimal policy a^* and reward r^*

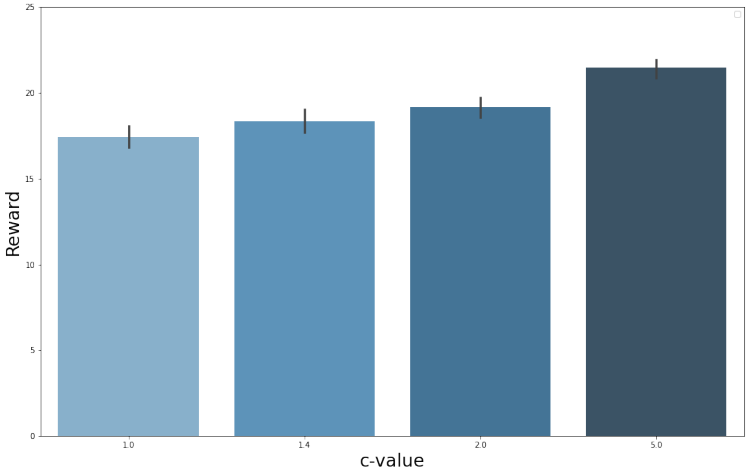
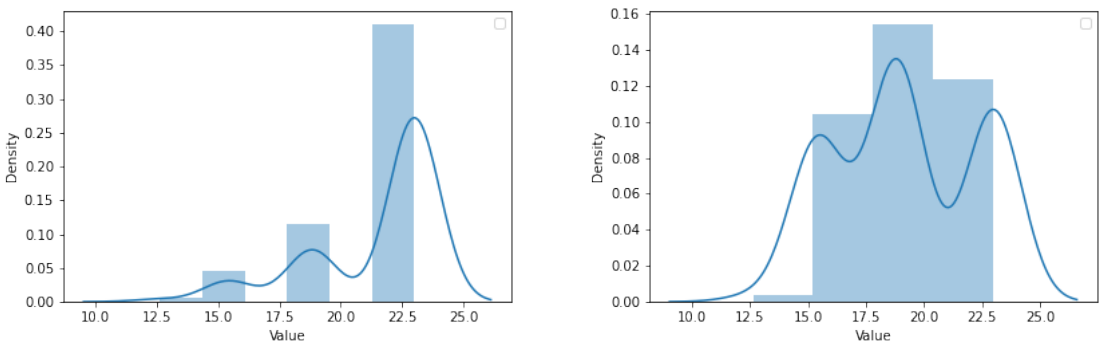


Figure 2: The central tendency plot of the simulations when c=1, 1.4, 2, 5



(a) Hist Plot of the simulations when c=5

(b) Hist Plot of the simulations when c=2

Figure 3: Histgram comparison when c=2 and c=5

will be too large according to equation 6, which may cause the system to always explore instead of adopting the best policy at a specific node. In this way, there is a high probability of causing serious deviations, which leads to the result reward as 5.67. Besides, when c is too small, this will result to continuous exploitation, leading to failure to converge to the optimal policy.
Based on the above analysis, we eliminated the cases where c is too large and too small. Now we turn our attention to the average reward column, we can find that when c is 2 or 5, we have a larger average reward. Next, we combine the histogram for analysis.
The Figure 3 shows the reward distribution when c=2 and c=5. We can find out that

- when c=5, we find that the frequency of the optimal strategy is higher, but there is also a probability close to 0.6 that it does not converge to the optimal policy and there is a certain probability that it converges to a poor policy like r=12.62.
- when c=2, although the probability of converging to the optimal policy is not as great as when c=5, it converges to a better policy in most cases.

3.4 conclusion

Through the above analysis and overall comparison, when c=2 and c=5, the search results have their own advantages and disadvantages. Therefore, the conclusion is that when c is between 2 and 5, our MCTS algorithm will eventually converge to a better policy.

4 Reinforcement Learning: SARSA and Q-Learning for Grid-world

4.1 Construct the environment

In order to implement the monte carlo policy evaluation, sarsa and q-learning, we firstly need to construct the environment, the grid world, which has walls, treasure and snakepit. The transition is deterministic with a reward(cost) of -1. The transition to the treasure will yield a reward of 50, con the contrary, the transition to the treasure will yield a reward of -50. There is also a assumption, we can start with anywhere we want except for the wall, treasure and snakepit grid.
Because this model is a model-free model, we need to sample the iterative process as $S_t \rightarrow A_t \rightarrow R_{t+1} \rightarrow S_{t+1} \rightarrow A_{t+1} \rightarrow R_{t+2} \rightarrow S_{t+2} \ldots$ and using the trajectories to train our model. The agent-environment interaction cycle is shown in Figure 4

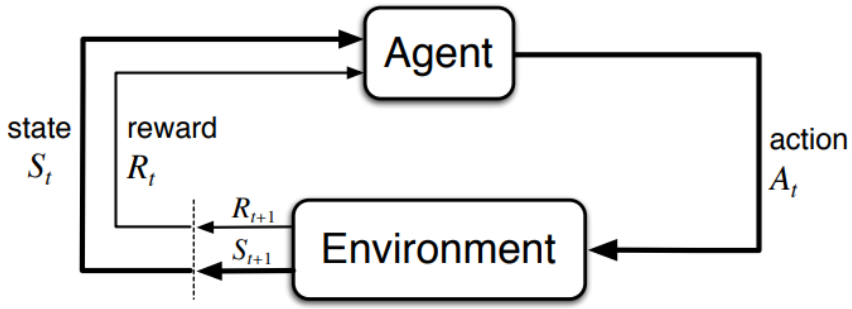


Figure 4: Agent-environment interaction cycle. Source: Reinforcement Learning: An Introduction (Sutton, R., Barto A.)[1].

First-visit MC prediction, for estimating $V \approx v_\pi$

Input: a policy π to be evaluated
Initialize:
  $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
  $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
  Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
    $G \leftarrow \gamma G + R_{t+1}$
    Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:
      Append G to $Returns(S_t)$
      $V(S_t) \leftarrow$ average$(Returns(S_t))$

Figure 5: MC algorithm. Source: Reinforcement Learning: An Introduction (Sutton, R., Barto A.)[1].

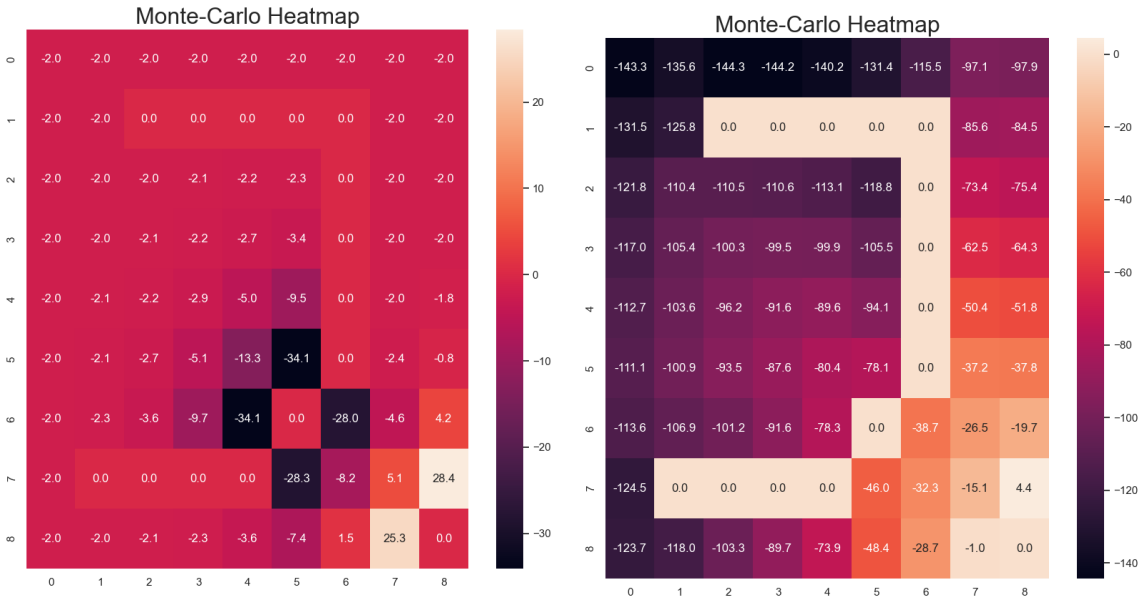
4.2 Monte Carlo Method

Monte Carlo (MC) methods can learn directly from experience or events instead of relying on prior knowledge of environmental dynamics. The Monte Carlo update rule is based on:

$$V_\pi(S_t) \leftarrow V_\pi(S_t) + \alpha[G_t(S_t) - V_\phi(S_t)] \tag{7}$$

$$G_t(S_t) = R_{t+1} + \gamma G_{t+1}(S_{t+1}) \tag{8}$$

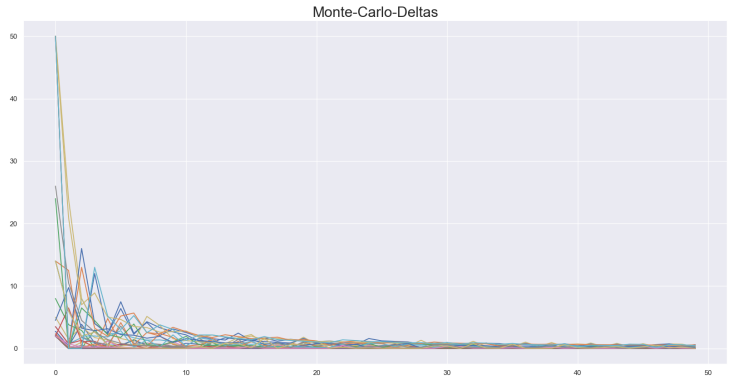
The algorithm is shown in Figure 5: Basically, we can generate n simulations from random points in the grid world, and let the agent move randomly in four directions until it reaches the end state. For each iteration, we save 4 values: (1) current state, (2) action taken, (3) reward received and (4) next state. In the end, the simulation is just an array containing x arrays of these values, where x is the number of steps the agent must take before reaching the final state.
Now, after each simulation, we iterate from the end of the "experience" array and calculate G based on equation 8. Then we store G in a $Returns(S_t)$ array. Eventually, for each state, we calculate the average of $Returns(S_t)$ and set the new value to the state's value at a specific iteration. In this situation, the learning rate $\alpha = 1/n$.
The iteration number is 10000. The result is shown in Figure 6. Based on the heat map, we can conclude that the value output by the monte carlo method has some deviations. This is because the actions in each state are randomly selected, which leads to the possibility of sampling for a long time before reaching the terminal state. If the γ is not big enough, the initial state G_t of the value is infinitely close to R_{t+1} when the value is updated, which leads to the situation in the figure 6a. Based on the delta variance graph in Figure ($\Delta \leftarrow |V_t^{old} - V_t^{new}|$), we can observed that in the first 50 iterations, the policy has already converged, but it may not converge to the optimal policy.



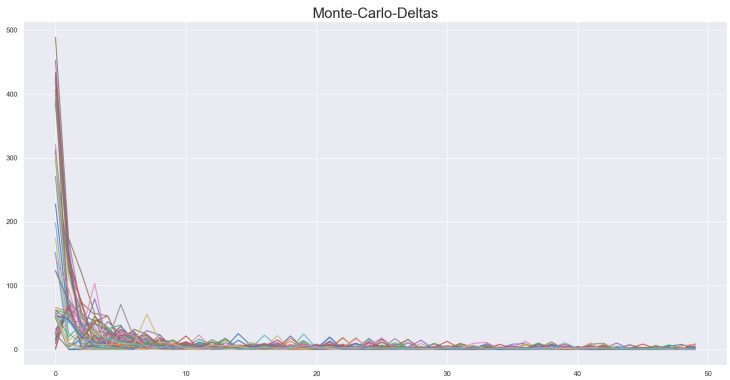
(a) Heatmap when $\gamma = 0.5$

(b) Heatmap when $\gamma = 1$

Figure 6: HeatMap of Monte Carlo Method when $\gamma = 0.5$ and $\gamma = 1$



(a) Delta variance when $\gamma = 0.5$



(b) Delta variance when $\gamma = 1$

Figure 7: Delta variance of Monte Carlo Method when $\gamma = 0.5$ and $\gamma = 1$

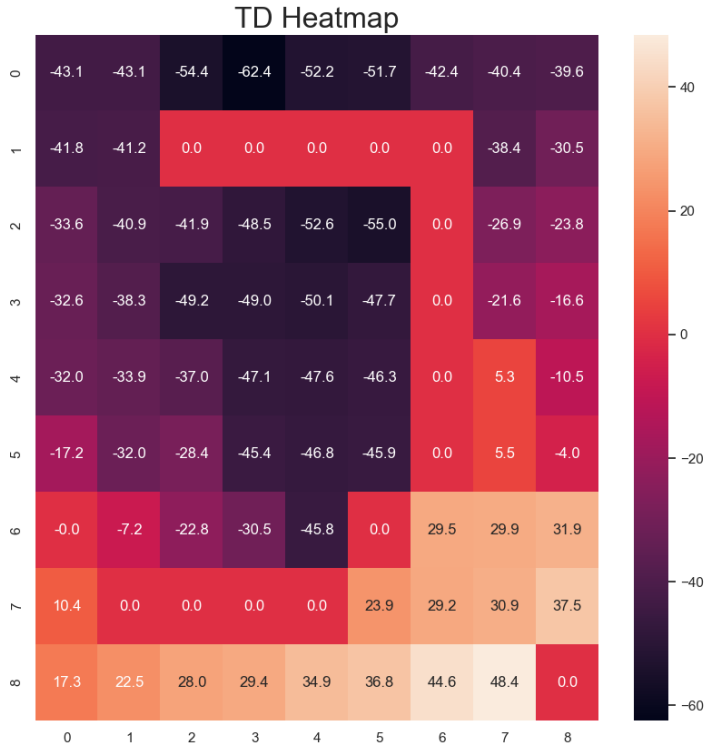


Figure 8: HeatMap of TD Method when $\alpha = 0.5, \epsilon = 0.7, \gamma = 0.9$

4.3 Temporal-difference Learning

We then try the Temporal-difference Learning method, which is said to merge the best of dynamic programming and the best of Monte Carlo approaches. The Temporal-difference update rule is based on:

$$V_\pi(S_t) \leftarrow V_\pi(S_t) + \alpha[R_{t+1} + \gamma V_\pi(S_{t+1}) - V_\phi(S_t)] \tag{9}$$

The results is shown in Figure 8

4.4 SARSA

The SARSA agent interacts with the environment and updates the policy based on the actions taken, so it belongs to on-policy learning algorithm. 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, plus the discounted future reward received from the next state-action observation. The SARSA update rule is as follows:

$$Q_\pi(s_t, a_t) \leftarrow Q_\pi(s_t, a_t) + \alpha[r_{t+1} + \gamma Q_\pi(s_{t+1}, a) - Q_\pi(s_t, a_t)] \tag{10}$$

In this situation, we combine the SARSA with greedification. The update step is as below:

1. Select the initial state s randomly.
2. Perform an action a using epsilon-greedy in a state s to end up in s' with a reward r, then using the epsilon-greedy policy to select an action a' for state s'.
3. Now, we get a tuple (s, a, r, s', a') at each step.

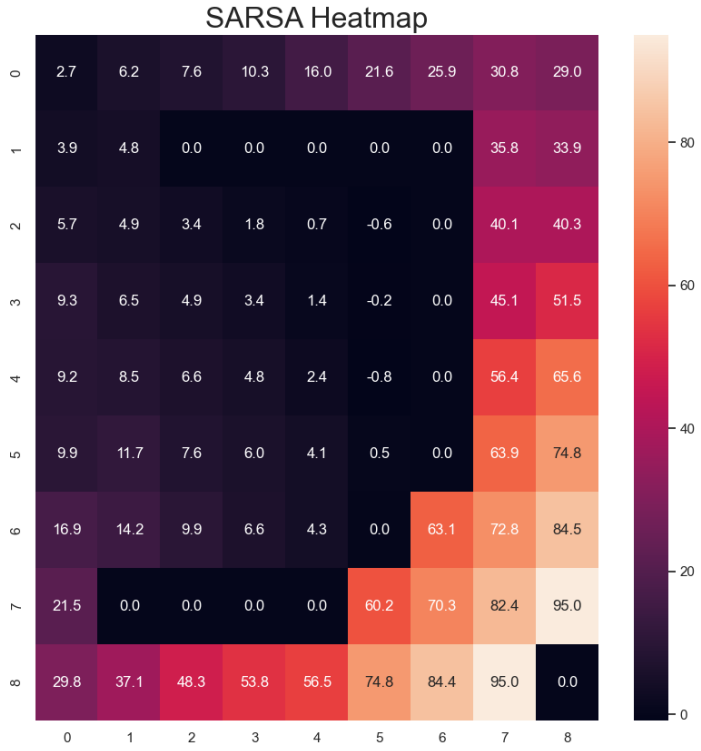


Figure 9: HeatMap of SARSA Method when $\alpha = 0.5, \epsilon = 0.3, \gamma = 0.9$

4. Loop until the terminal state.
5. Update the Q-value based on the equation 10.

In the simulation, we set the iteration times is 1000, $\alpha = 0.5, \epsilon = 0.3, \gamma = 0.9$, which means for each action selection step, there is 0.3 probability to select randomly. The result is shown in Figure 9 and 10.

4.5 Q-learning

For any finite Markov decision process (FMDP), Q-learning starts from the current state and finds the best strategy in the sense of maximizing the expected value of the total reward for any and all successive steps. Given an infinite exploration time and a partially random strategy, Q-learning can determine the best action selection strategy for any given FMDP[2].
Different from SARSA, Q-learning is an off-policy learning algorithm. Q-learning estimates the state-action value function of the target strategy, which deterministically selects the action with the highest value. Similar to SARSA, the Q value of the state action is updated by the error and adjusted by the learning rate α. The Q-learning update rule is as follows:

$$Q^{new}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t)] \tag{11}$$

1. Select the initial state s randomly.
2. Perform an action a using epsilon-greedy in a state s to end up in s' with a reward r.
3. Now, we get a tuple (s, a, r, s') at each step.

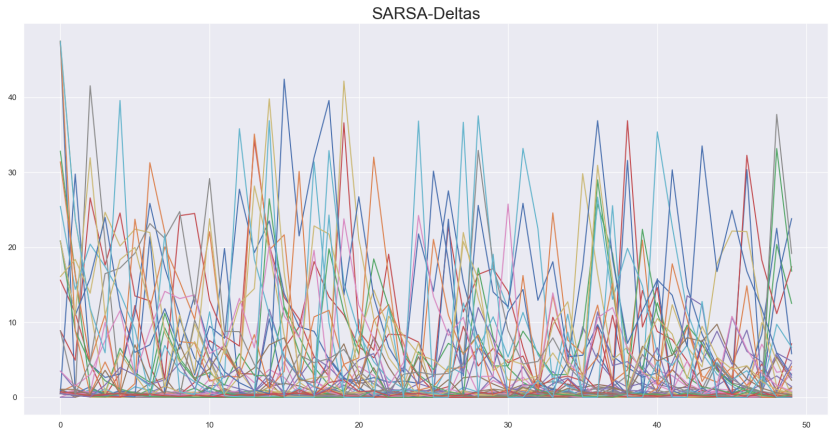


Figure 10: Delta variance of SARSA Method of the first 50 iterations when $\alpha = 0.5, \epsilon = 0.3, \gamma = 0.9$

Q-learning: An off-policy TD control algorithm

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
    $S \leftarrow S'$
  until S is terminal

Figure 11: Q-Learning Algorithm Source: Reinforcement Learning: An Introduction (Sutton, R., Barto A.)[1].

4. Loop until the terminal state.

5. Give an estimate of the current Q value, which is equal to the current reward plus the maximum Q value of the next state multiplied by the decay rate γ.

6. Update the estimation of the current Q value by adding α times a temporal difference.

The algorithm is as follows in 11.
In the simulation, we set the iteration times is 1000, $\alpha = 0.5, \epsilon = 0.3, \gamma = 0.9$. The result is shown in Figure 12 and 13.

4.6 Comparison

The theory of the SARSA and Q-learning difference is shown in Figure 14 like what we previous discussed. As for our experiments, we can conclude that both Q-value and policy for SARSA and Q-learning converge in a right direction on the account of Figure 9, 12 and 15. Although the optimal policies are slightly different based on Figure 15, using both the optimal policies will always result in the treasure grid, which means both policy is a real sense of optimal. What's more, by comparing the two figures 10 and 13, we can observe that the average delta of SARSA is obviously bigger than delta of Q-learning. This is because Q-learning is always select the optimal Q, however, the SARSA have ϵ probability to explore. Although in my results graph, both SARSA and Q-learning converge to the optimal policy, through multiple experiments after modifying the hyperparameters like α, iteration step etc., it is found that Q-learning converges to the optimal policy faster than SARSA.

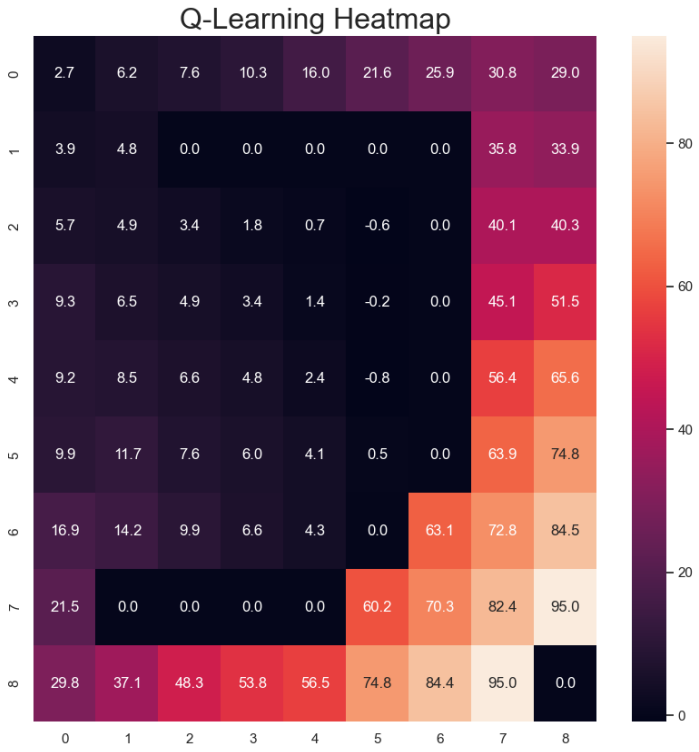


Figure 12: HeatMap of Q-learning Method when $\alpha = 0.5, \epsilon = 0.3, \gamma = 0.9$

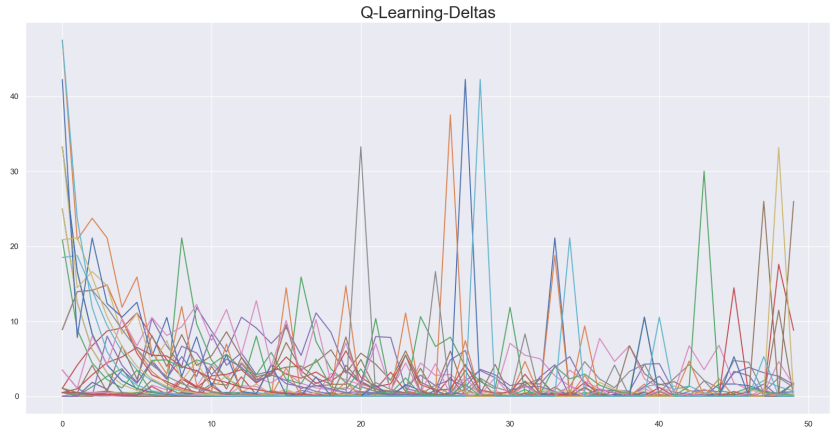
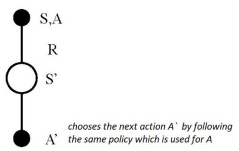
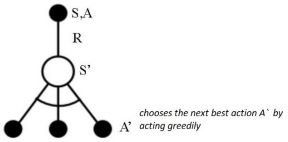


Figure 13: Delta variance of Q-learning Method of the first 50 iterations when $\alpha = 0.5, \epsilon = 0.3, \gamma = 0.9$





$$Q(S,A) \leftarrow Q(S,A) + \alpha\left(R + \gamma Q(S',A') - Q(S,A)\right) \qquad Q(S,A) \leftarrow Q(S,A) + \alpha\left(R + \gamma \max_{a'} Q(S',a') - Q(S,A)\right)$$

Figure 14: SARSA and Q-learning difference

east	east	south	east	east	east	east	south	south
south	north	WALL	WALL	WALL	WALL	WALL	south	south
south	south	south	south	south	south	WALL	east	south
south	south	west	south	west	west	WALL	east	south
south	west	south	west	west	west	WALL	south	south
south	west	south	west	west	west	WALL	south	south
south	west	west	west	west	LOSS	south	east	south
south	WALL	WALL	WALL	WALL	south	south	south	south
east	east	east	east	east	east	east	east	WIN

(a) Optimal policy using SARSA

east	east	east	east	east	east	east	east	south
south	west	WALL	WALL	WALL	WALL	WALL	east	south
south	west	west	west	west	west	WALL	east	south
south	west	west	west	west	west	WALL	east	south
south	west	west	west	west	west	WALL	east	south
south	west	west	west	west	west	WALL	east	south
south	west	west	west	west	LOSS	east	east	south
south	WALL	WALL	WALL	WALL	east	east	east	south
east	east	east	east	east	east	east	east	WIN

(b) Optimal policy using Q-learning

Figure 15: Comparison of the optimal policy using SARSA and Q-learning

5 Conclusion

In this assignment, we implemented the Monte Carlo sampling to solve the renting problem, the Monte Carlo Tree Search to solve the binary tree model, as well as the Monte Carlo policy evaluation, Temporal-difference Learning algorithm, SARSA algorithm and Q-learnig algorithm to search for the optimal policy for the grid world.
Through a lot of comparative experiments, I found that the hyperparameter c in the MCTS will have a direct impact on the search results. In the part of the model-free model, hyperparameters like α, ϵ, γ will also have a more or less impact on the convergence time and convergence results. Therefore, in order to obtain a better model, reasonable hyperparameter settings are very essential.
The goal of this assignment was developing an in depth understandance of fundamental reinforcement learning algorithm. Through this assignment, I have a very comprehensive understanding of the basic algorithms of reinforcement learning, and my coding skills have also improved. I will use my spare time to research and reproduce more state-of-the-art algorithms such as Actor-Critic and DQN.

6 Acknowledgement

I would like to express my special thanks of gratitude to my teacher Eric who answered questions very patiently and exhaustively for me during each class. It helped me in doing a lot of research and implementation and I came to know about a lot of things related to this direction.
Besides, I would also like to thank TAs and my friends who helped me a lot in doing the assignments.

References

[1] Sutton, Richard; Barto, Andrew (1998). Reinforcement Learning: An Introduction. MIT Press.
[2] Melo, Francisco S. "Convergence of Q-learning: simple proof".