Monte Carlo Tree Search and Reinforcement Learning

Longxiang Wang, 2686176
wlongxiang1119@gmail.com

January 3, 2021

1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is an important algorithm behind the major successes of recent AI applications such as AlphaGo. In this report, we will first briefly describe how MCTS algorithm works, then apply it to a problem of finding the leaf node of a binary tree with the highest reward.

1.1 MCTS Algorithm

At it base, MCTS uses Monte Carlo simulation to accumulate value estimates to guide towards highly rewarding trajectories in the search tree.

In more detail, starting at a given root node or initial state (this state will change when a action is taken after the repeated iterations of MCTS), a basic version of MCTS iteration is performed repeated until certain resource limitation is exhausted. Each iteration consists of four steps:

- **Selection**:
  - use *tree policy* to construct path from root to most promising leaf node.
  - tree policy is an informed policy used for action selection in the snowcap (explored part of the tree) where child node with the best UCB score is selected. UCB in this context is defined as:

$$UCB(node_i) = \bar{x}_i + c\sqrt{\frac{logN}{n_i}}$$

    where $\bar{x}_i$ is mean node value, c is a hyperparameter, N is number of visits to parent node and n_i is number of visits to node i.
  - traverse the snowcap or explored part of the tree until a leaf node has been reached. A leaf node is a node which has unexplored child node(s).
- **Expansion**: randomly pick an unexplored node of a leaf node from last step.
- **Simulation**: roll out one or multiple simulations with reward accumulated for each simulation. Roll-out policy is normally simply or even pure random such that it is fast to execute.
- **Backup**: update values along the tree traversal path within the snowcap (not including the roll-out states!).

As some might prefer, a more visual representation of the core four steps of one MCTS iteration is shown in Figure 1:

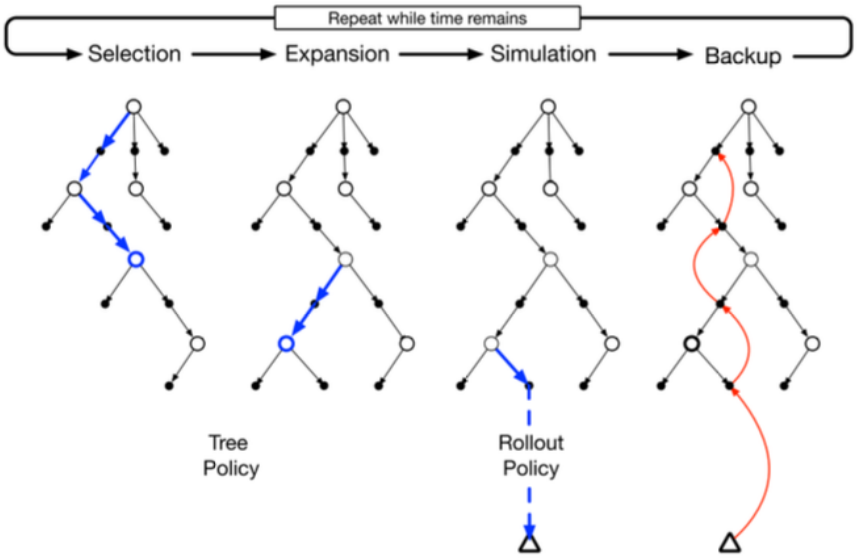

Figure 1: MCTS: repeated loops over four steps. Credits: Sutton & Barto.

1.2 Implementation

In the implementation part, we first need to have some utility code to construct a binary tree with depth $d = 12$, and assign different values (rewards) to each of the 2^d leaf nodes (note: the leaf node here means the final level of the binary tree, whereas in MCTS algorithim description it means the node of tree which contains unexplored child node). The values assigned are drawn from uniform distribution $U(0, 100)$.

1.2.1 Make a binary tree

It wasn't that straight forward to create a binary tree of arbitrary depth. Here we only illustrate the construct of a node and a general idea of the function. The details can be found at `binary_tree.py`.

A node in the binary tree is defined as a class below, which contains a left and right child, a node id serving as a unique identifier and the value attached to it.

```
class Node:
    def __init__(self, node_id=None):
        self.left = None
        self.right = None
        self.node_id = node_id
        self.value = None
```

The function to generate a binary tree of arbitrary depth is defined as:

```
def make_binary_tree(depth=12):
```

```
    all_nodes = []
    for i in range(depth + 1):
        nodes_at_depth = []
        num_of_nodes = pow(2, i)
        for j in range(num_of_nodes):
            nodes_at_depth.append(Node(str(i) + "_" + str(j)))
        all_nodes.append(nodes_at_depth)

    leaf_nodes_dict = dict()
    for level, nodes in enumerate(all_nodes):
        for loc, n in enumerate(nodes):
            if level >= len(all_nodes) - 1:
                # we assign reward value to leaf nodes of the tree
                n.value = random.uniform(0, 100)
                leaf_nodes_dict[n] = n.value
            else:
                left = all_nodes[level + 1][2 * loc]
                right = all_nodes[level + 1][2 * loc + 1]
                n.left = left
                n.right = right
    root = all_nodes[0][0]
    return root, leaf_nodes_dict
```

As we can see, this function first creates the nodes at each depth level, then link the child nodes to its parents with values assigned to its leaf nodes. In the end, it returns the root node, with which one can traverse the whole tree.

1.2.2 MCTS implementation

A class `MCTS` is implemented in `monte_carlo_tree_search.py`, which contains 4 main public methods: `select`, `expand` , `simulate` , `backup` corresponding to the 4 core steps described earlier, and a `choose` method to choose a next action to go to the next best child node after MCTS iteration(s) is completed.

The main entry point `run` of class `MCTS` is defined as follows:

```
def run(self, node, num_rollout):
    # Run one iteration of select -> expand -> simulation(rollout) -> backup
    path = self.select(node)
    leaf = path[-1]
    self.expand(leaf)
    reward = 0
    for i in range(num_rollout):
        reward += self.simulate(leaf)
    self.backup(path, reward)
```

1.3 Experiments

An experiment to check the effect of exploration weight on how close the value found by MCTS is to the real optimal value 100 (because of the leaf node reward is drawn from $U(0, 100)$ as shown earlier) is conducted in controlled environment as shown in Figure 2. A boxplot zoomed in in the peaked area is provided in Figure 3.

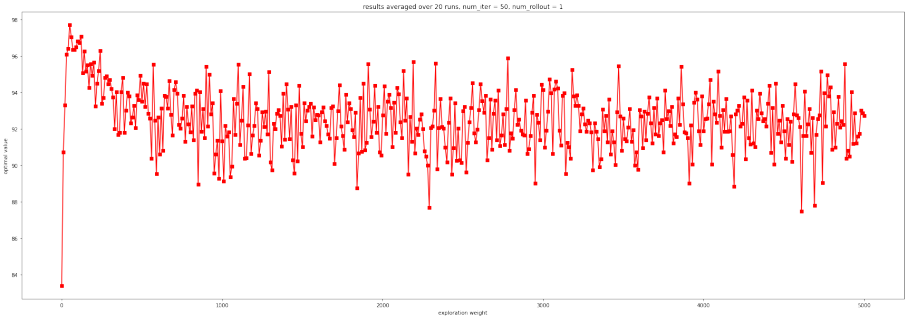

Figure 2: Effect of hyperparameter c or exploration weight: optimal value averaged under 20 runs of different random seeds. Number of MCTS iterations is fixed at 50 and number of roll out is fixed at 1.

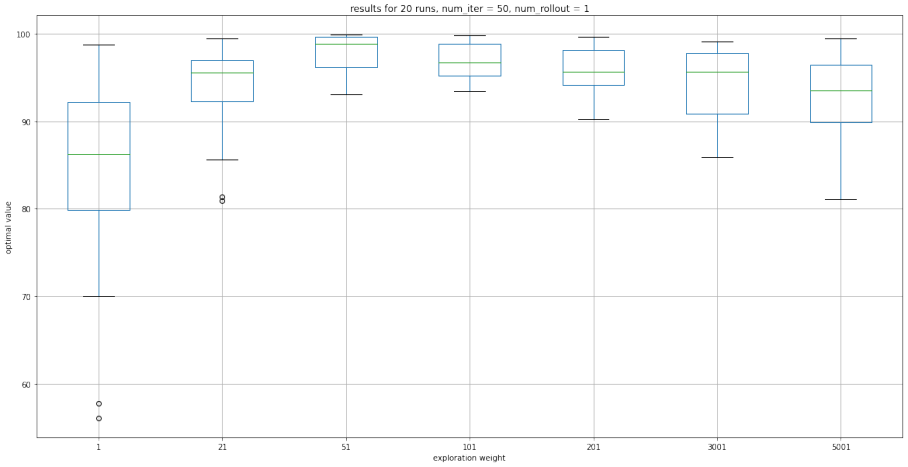

Figure 3: Zoomed in boxplot of 20 runs of different random seeds. Number of MCTS iterations is fixed at 50 and number of roll out is fixed at 1.

As we can see, there is clearly an optimal setting for exploration weight, which is approximately at 51 in this specific problem settings. At this sweet spot, not only the average reward is higher, also the variance is lower. This experiment results shows the balance of exploration and exploitation is of vital important in MCTS algorithm where the actual value of best exploration weight depends on the problem at hand.

1.4 Influence of UCB Hyperparameter c

As we have seen in the experiement results already, the value of c should not be too large nor too small. Generally, it is also true that when c is larger, during the action selection step, the UCB higher bound tends to be close to each other for different actions, which leads to more randomness in action selection, i.e., more exploration and less exploitation.

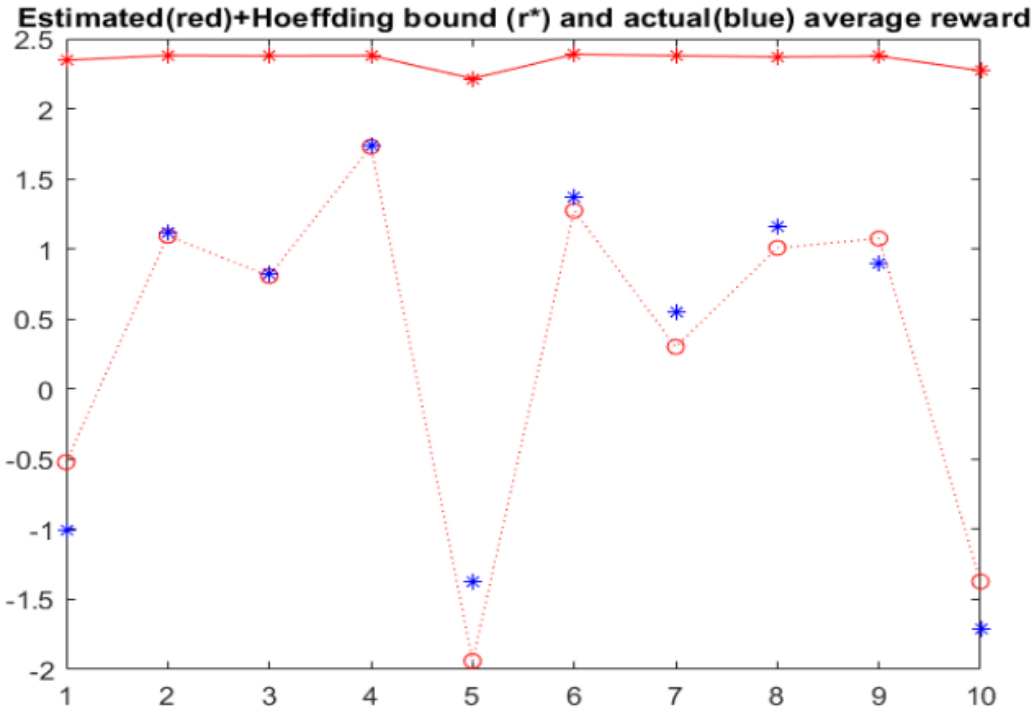

Figure 4: Influence of UCB in 10-arm bandit problem with Gaussian means under large $c = 5.0$. Note: Hoeffding bound can be interpreted as UCB bound. Credits: Course MAS lecture notes.

Whereas if c is smaller, the UCB bounds have more variations and the action selection becomes greedier. In other words, it leads to more exploitation than exploration.

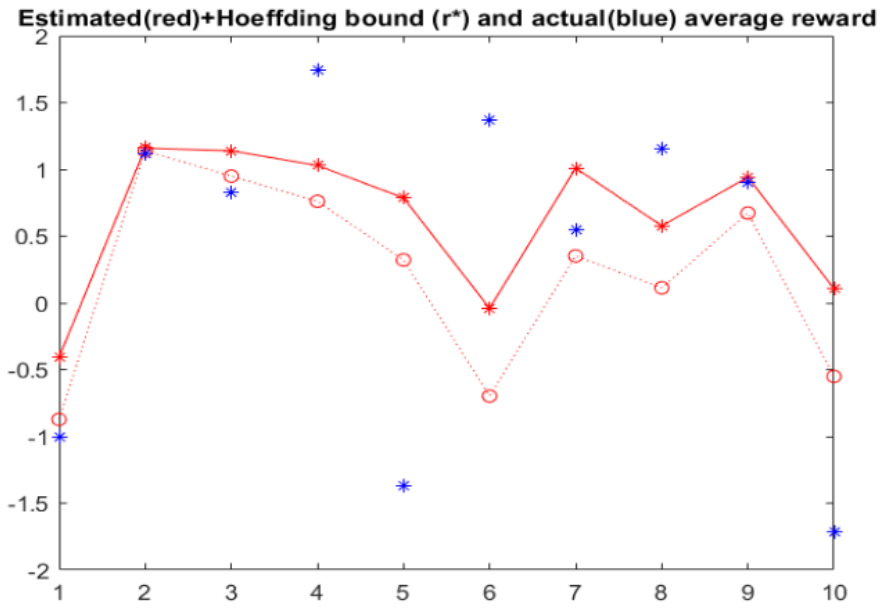

Figure 5: Influence of UCB in 10-arm bandit problem with Gaussian means under large $c = 0.5$. Note: Hoeffding bound can be interpreted as UCB bound. Credits: Course MAS lecture notes.

2 Reinforcement Learning: SARSA and Q-Learning for Gridworld

In this report, we use the gridworld case as example illustrated in Figure. 6 to look into Monte Carlo policy evaluation and SARSA and Q-learning approaches.

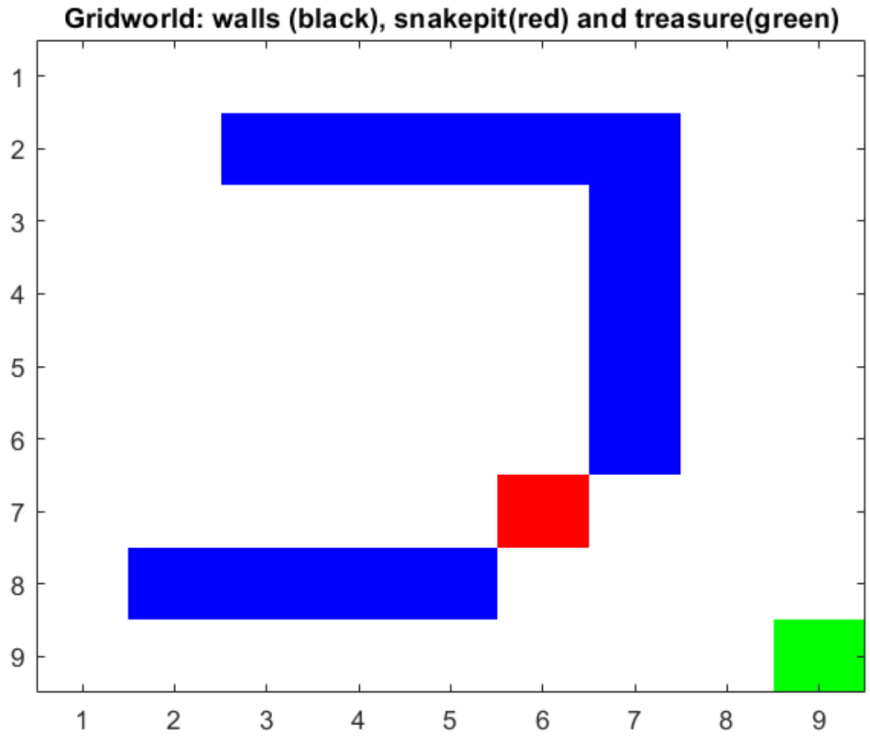

Figure 6: Gridworld example. Entering the treasure and snakepit yield reward of $+50$ and -50 repspectively. The walls in blue can not be traversed. Other transistions yield reward of -1.

To model the gridworld, OpenAI Gym package is used. The details can be seen in code `gridworld.py`.

2.1 Monte Carlo Policy Evaluation

First visit MC method is used to evaluation state value of a random policy where each action of a certain state is taken with equal probability.

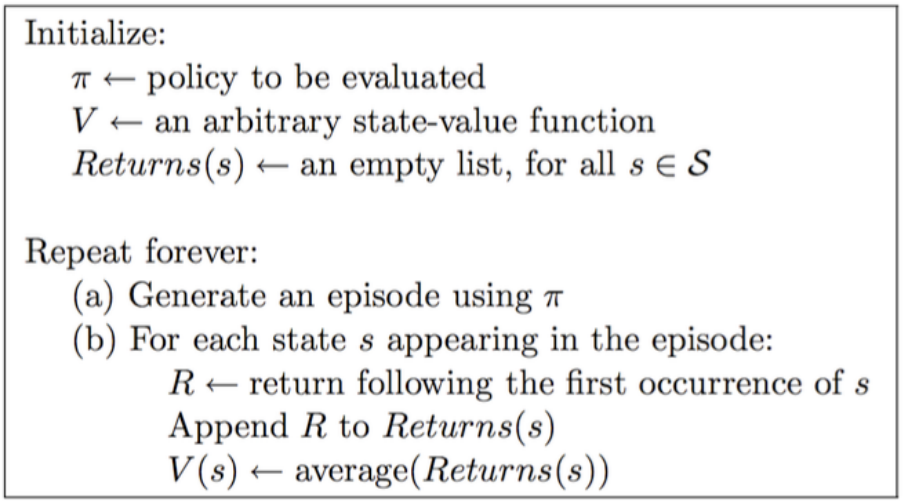

Figure 7: Algorithm: first-visit MC policy evaluation. Credits: MAS course lecture notes.

The algorithm of first visit MC is described in Figure 7. The implementation is shown below:

```
def mc_policy_evaluation_random_policy(env, num_episodes=1000):
    # Start with an all 0 value function
    V = defaultdict(float)
    for _s in env.P:
        V[_s] = 0.0
    returns = defaultdict(list)  # an empty list for each state
    for i in range(num_episodes):
        episodes = []
        init_state = choice(list(set(env.P.keys())))  # draw random init state
        while not env.is_terminal(init_state):
            action = choice(list(env.P[init_state].keys()))  # random policy
            next_state = env.P[init_state][action][0][1]
            reward = env.P[init_state][action][0][2]
            episodes.append([init_state, action, reward])
            init_state = next_state
        G = 0
        states_seen = set()
        for S, A, R in reversed(episodes):
            G = 1.0 * G + R  # assuming discount factor is 1.0
            if S not in states_seen:
                states_seen.add(S)
                returns[S].append(G)
                V[S] = np.mean(returns[S])
    V_sorted = sorted(V.items(), key=lambda x: x[0])  # sort by state
    return V_sorted
```

The result is visualized in Figure 8. As we can see, Both terminal states have state value 0. The states closer to treasure point have relatively high state values than those farther away from it. Whereas those close to the snakepit (highly negative reward) have very low state values. Also those at wall have some of the lowest values because it takes a long way for agent starting there

to terminate hence resulting in long route and accumulated high negative rewards.

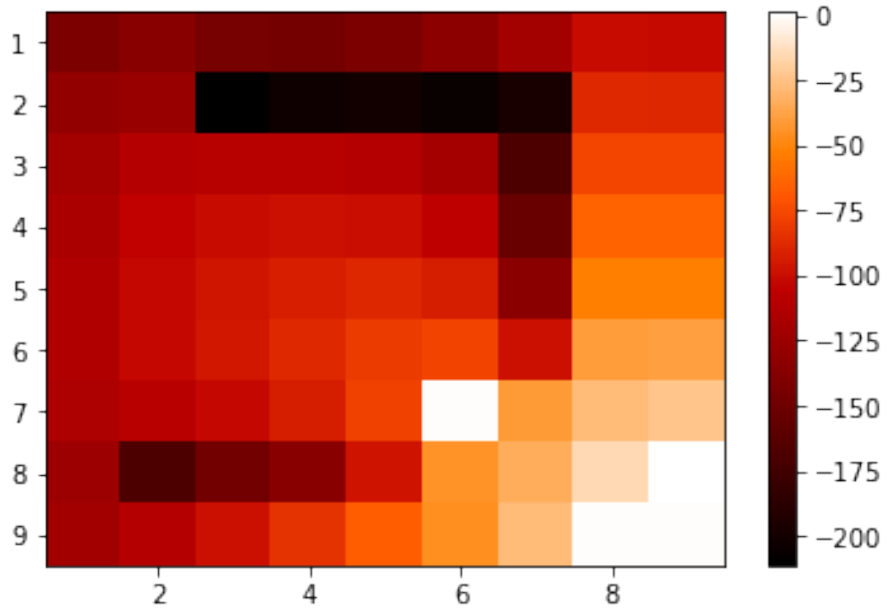

Figure 8: State value heatmap.

2.2 On-policy vs Off-policy

Le's first take a detour and look into the different between on-policy and off-policy learning, because SARSA is good example of on-policy learning and Q-Learning is of latter.

In SARSA the target policy (the optimal policy that agent tries to learn eventually) and behavior policy (the policy used for action selection during learning) are the same. The action selection during learning is following the same policy which is eventually learned as target policy.

Whereas in Q-learning, the behavior policy during learning is greedy (always choose the best Q value) despite the taget policy can be anything else such as ϵ-greedy.

2.3 SARSA

The SARSA algorithm is brifly described in Figure. 9. The actual implemetation can be seen in `sarsa.py`.

Initialize $Q(s, a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Choose a from s using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action a, observe r, s'
    Choose a' from s' using policy derived from Q (e.g., ε-greedy)
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$; $a \leftarrow a'$;
  until s is terminal

Figure 9: SARSA algorithm.

The optimal policy learned by SARSA after 50k episodes is shown in Figure 10.

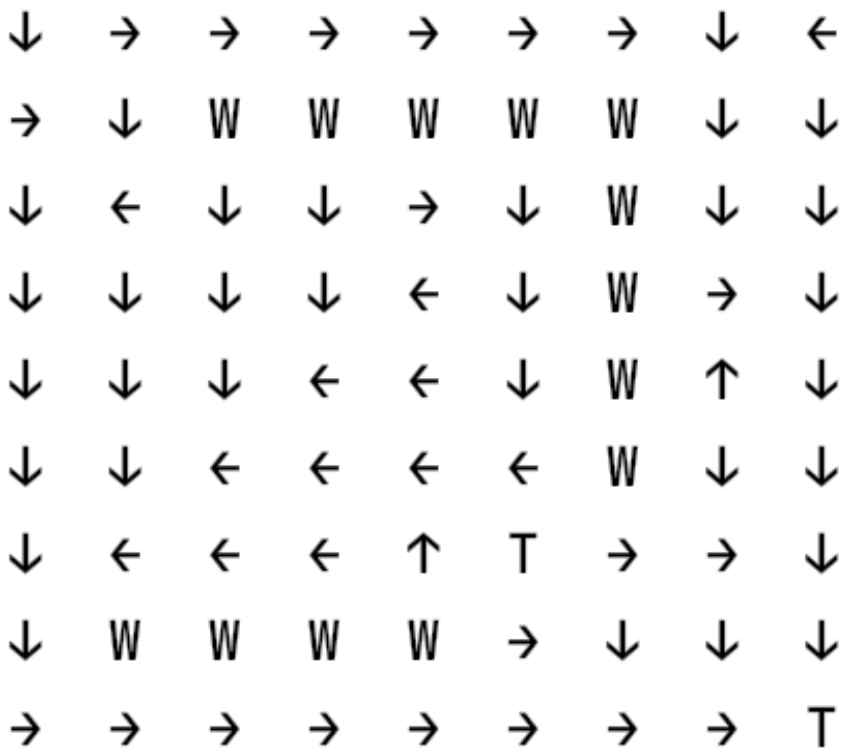

Figure 10: SARSA optimal policy.

As we can see, the optimal policy tends to go as fast as possible to treasure point and avoid the snakepit point as much as possible. It also knows when to circumvent the wall along the route.

2.4 Q-Learning

The Q-learning algorithm is similar to SARSA, but differ in that it is a off-policy approach. The algorithm is brifly described in Figure. 11. The actual implemetation can be seen in `q_learning.py`.

Initialize $Q(s, a)$ arbitrarily
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'
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
    $s \leftarrow s'$;
  until s is terminal

Figure 11: Q-Learning algorithm.

The optimal policy learned by SARSA after 50k episodes is shown in Figure 12.

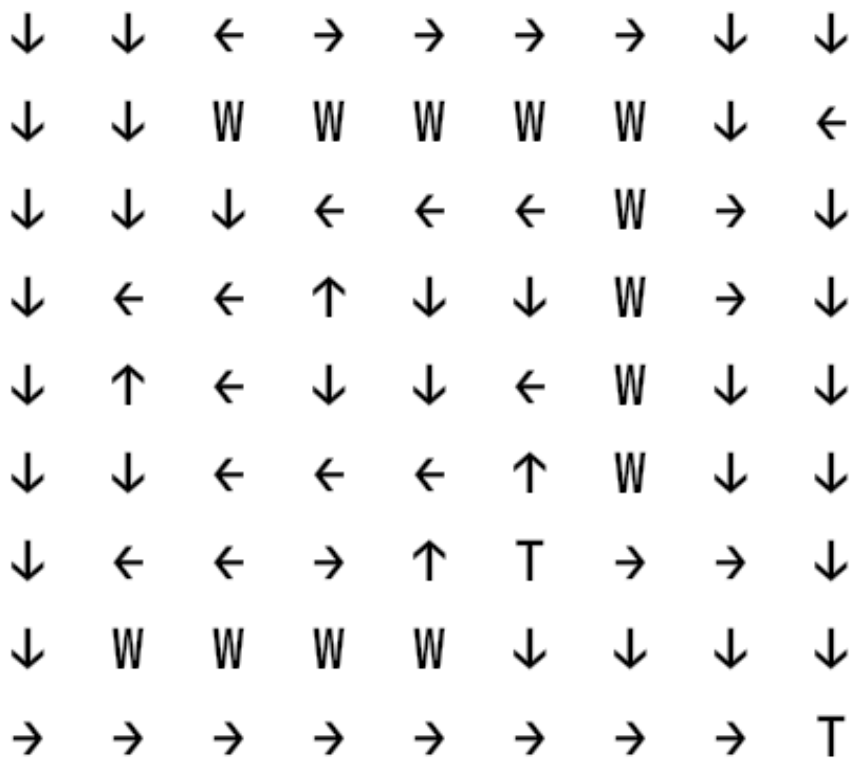

Figure 12: Q-Learning optimal policy.

As we can see, the optimal policy tends to go as fast as possible to treasure point and avoid

the snakepit point as much as possible. It also knows when to circumvent the wall along the route.

Interestingly, observing the states above the snakepit where the major difference lies, we see the the optimal policy learned by Q-learning tends to avoid walking towards snakepit as if it is penalized more, while for SARSA it also avoids it in the end, but it does so less aggressively.

3 Summary

In this report, we looked at the both algorithmic aspects as well as case studies in toy examples. First, we explained the MCTS algorithm and applied it to a binary tree max leaf value search problem. Then, we turned to reinforcement learning. Monte Carlo policy evaluation is studied and implemented, the results are visualized with a heatmap which match our intuitions. In the end, we looked at both SARSA and Q-Learning, the two classical examples of on-policy and off-policy learning approaches in RL. The results obtained from gridworld example help us gain more insights in the similarities and differences between these two algorithms.