MLSALT7 Reinforcement Learning Coursework

Author: Yeziwei (Maggie) Wang

Contents

1 Introduction

This coursework explores four different basic algorithms in reinforcement learning by applying them on grid-world models. There are three different grid model used: smallworld, gridworld and cliffworld. Value iteration and policy iteration are applied on grid world to find the optimal policy. Both algorithms are examples of dynamic programming, which is a model-based reinforcement learning method. On the other hand, SARSA and Q-learning are both model-free reinforcement learning algorithms. In the experiment, they both are applied on smallworld individually and then compared against each other on cliffworld. The following concepts from reinforcement learning are also discussed in the report: differences and similarities between on-policy and off-policy learning, Dynamic Programming, Monte Carlo methods and Temporal Different Learning.

2 Experiments

2.1 Value Iteration

```
Algorithm 1: Value Iteration
Initialise $V_0$ arbitrarily;
repeat
    $V_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)(r + \gamma V_k(s'))$;
until $\max_s |V_{k+1}(s) - V_k(s)| < \theta$;
```

As a type of a dynamic programming method, value iteration is a model-based algorithm. It evaluates the value functions and at the same time update the policy. Its update rule is the same as the policy evaluation except that it calculates the maximum value over all actions at a specific state. The action which gives the maximum value function is used to update the policy. This results in simple backup and fast convergence. The algorithm stops after the value function has converged like shown in pseudocode of Algorithm 1. Since the backup takes the maximum reward, the algorithm should converge to an optimal policy.

Figure 1 shows the value function and policy the algorithm converges to in gridworld. The algorithm successfully learns the correct policy from the start state to goal state avoiding the obstacle given the model of the environment.

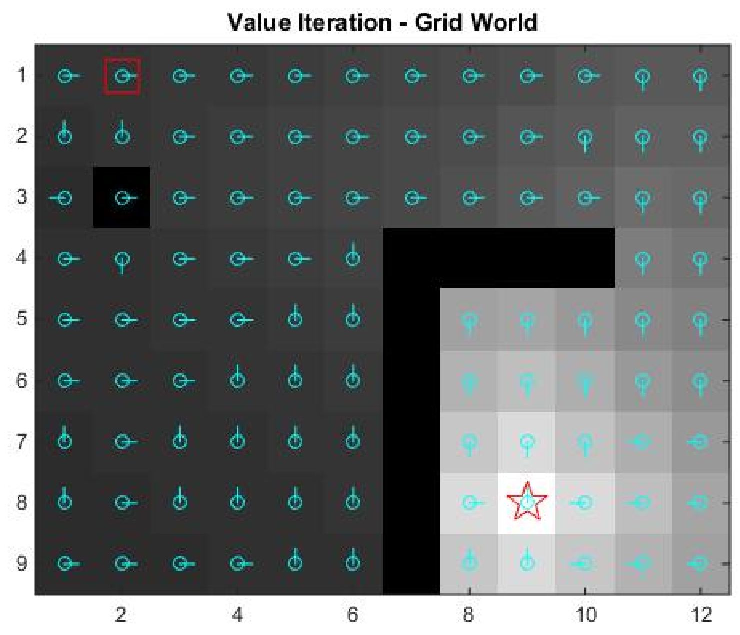


Figure 1: Gridworld: Value Iteration

In the algorithm implementation shown below, the policy is recorded every iteration while calculating the update for value function. The new policy over-write the previous one resulting in optimal policy to be saved at the end of the algorithm.

```
function [v, pi] = valueIteration(model, maxit)
% initialize the value function
v = zeros(model.stateCount, 1);

for i = 1:maxit,
    % initialize the policy and the new value function
    pi = ones(model.stateCount, 1);
    v_ = zeros(model.stateCount, 1);
    % perform the Bellman update for each state
    for s = 1:model.stateCount,
        [v_(s),pi(s)] = max(sum(squeeze(model.P(s,:,:))...
                            .*(repmat(model.R(s,:),model.stateCount,1) ...
                            + repmat(model.gamma*v,1,4)),1));
    end
    % exit early
    if max(abs(v-v_)) < 0.00001,
        fprintf('value function converge after %d iterations', i);
        break;
    end
    v = v_; %update
end
```

2.2 Policy Iteration

Algorithm 2: Policy Iteration

```
Initialise $V$ and $\pi$ arbitrarily;
repeat
    Evaluate $V$ using $\pi$: $V_{k+1}(s) = \sum_a \pi(a,s) \sum_{s',r} p(s',r|s,a)(r + \gamma V_k(s'))$;
    Improve $\pi$ using $V$: $\pi'(s) = \arg\max_a \sum_{s',r} p(s',r|s,a)(r + V_\pi(s'))$
until $\max_s |V_{k+1}(s) - V_k(s)| < \theta$;
```

Policy iteration is also a model-based algorithm. It contains two steps in each iteration: 1) policy evaluation: updating the value function using current policy, 2) policy improvement: find the action that gives the maximum reward in the next step using updated value function. Alternating between these two steps can guarantee improvement over iterations and therefore the value function and policy will converge to optimal within finite iterations. This also provides fast convergence as each iteration is an improvement from previous one.

The final result of value function and optimal policy are shown in Figure 2. The optimal policy finds the path from start state to the goal state while avoiding obstacles in the world, where the black blocks are. The optimal policy is the same as what value iteration algorithm produced. So is the value function. The results from both value iteration and policy iteration shows great agreement between value function and policy, where the lighter the colour the closer it is to the target and all actions try to lead to where the target is.

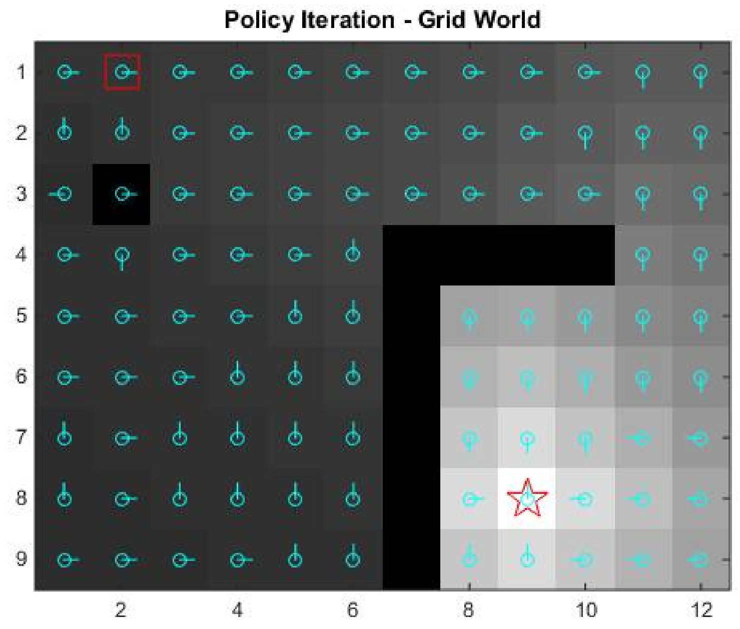


Figure 2: Gridworld: Policy Iteration

In the implementation of the algorithm, policy evaluation is conducted over all states before policy improvement is applied. This is to avoid updating policy before the evaluation is finished, so the policy improvement produces guaranteed improvement in value function and vice versa. The convergence condition is the same as value iteration.

```
function [v, pi] = policyIteration(model, maxit)

% initialize the value function
v = zeros(model.stateCount, 1);
v_ = zeros(model.stateCount, 1);
pi = ones(model.stateCount, 1);

for i = 1:maxit,
    v = v_;
    for s = 1:model.stateCount   % policy evaluation
         temp = sum(squeeze(model.P(s,:,:))...
                .*(repmat(model.R(s,:),model.stateCount,1)...
                + repmat(model.gamma*v,1,4)),1);
         v_(s) = temp(pi(s));
    end
    for s = 1:model.stateCount   % policy imporovement
        [¬,pi(s)] = max(sum(squeeze(model.P(s,:,:))...
                    .*(repmat(model.R(s,:),model.stateCount,1)...
                    + repmat(v_,1,4)),1));
    end

    % exit early
    if max(abs(v_ - v))<0.0001
```

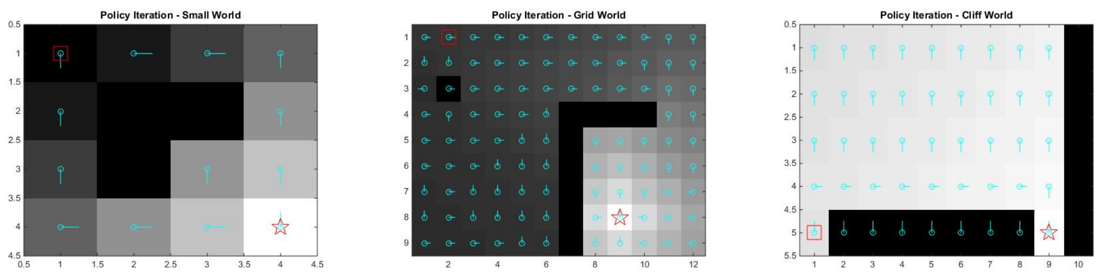
```
            fprintf('Value function converged after %d itarations!', i)
            break;
        end

end
```

Value Iteration and Policy Iteration

	Small world	Grid world	Cliff world
Value Iteration	23	59	14
Policy Iteration	21	53	14

Table 1: Number of iterations till convergence

Both value iteration and policy iteration converge fast in three different world and all find optimal policy(Table 1). In both algorithm we have prior knowledge of the environment and at each update step the two algorithms update the policy with action that gives the maximum reward. Due to these reasons, the algorithm can quickly computes the optimal value function and policy. Both algorithms computes the same value function and optimal policy shown in Figure 3.



(a) Smallworld (b) Gridworld (c) Cliffworld

Figure 3: Optimal value function and policy from value iteration and policy iteration

2.3 SARSA

Algorithm 3: SARSA algorithm

```
Initialise $Q$ arbitrarily, $Q(terminal, .) = 0$ ;
repeat
    Initialise s as the start state;
    Choose a $\epsilon$-greedily ;
    repeat
        Take action $a$, observe $r, s'$ ;
        Choose $a'$ $\epsilon$-greedily ;
        $Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma Q(s', a') - Q(s, a))$ ;
        $s \leftarrow s', a \leftarrow a'$ ;
    until $s$ reaches the goal state;
until $\max_s |V_{k+1}(s) - V_k(s)| < \theta$;
```

SARSA is an on-policy temporal-difference (TD) learning algorithm which has the advantages from both dynamic programming and Monte Carlo methods. Like dynamic programming, TD learning updates estimates based on other learned estimates, bootstrapping. Like Monte Carlo methods, it doesn't require knowledge of the environment and learns directly from raw experiences except that the update takes place at each time step:

$$V(s_t) \leftarrow V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t))$$

where $\alpha > 0$ is the learning rate. SARSA learns the action-value function in the same manner. As an on-policy algorithm, it takes the action it evaluates. This is shown in the update step in Algorithm 3, where state and policy are both updated after the backup. The terminal state has zero value for Q-function. As SARSA learns from experience, for each episode the algorithm starts from the starting state and explores the world until it reaches the goal state. The Q-functon is updated after choosing next action ϵ-greedily and it is updated using temporal difference in action-value function.

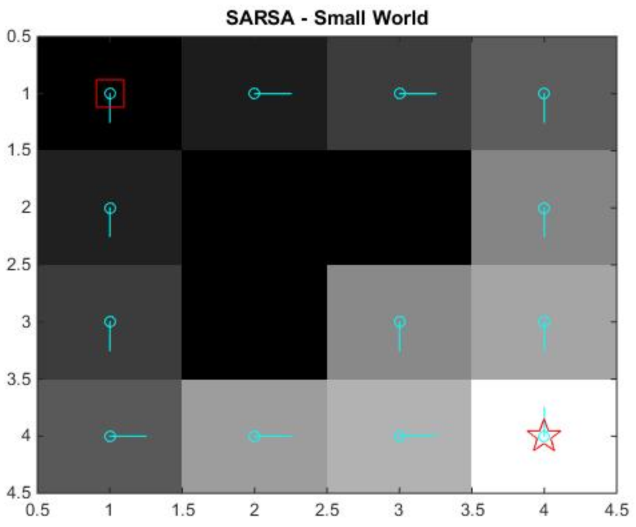


Figure 4: Smallworld: SARSA

The learning rate α is chosen as 0.3 and ϵ 0.1, so that the algorithm has a modest learning rate and keeps exploring throughout iterations. In smallworld the optimal policy is obtained. The same convergence criteria is used as in policy iteration. Although SARSA never reached the convergence criteria, it still produced optimal policy. This may be due to the exploration of the algorithm and the simplicity of the model itself. As the algorithm takes a random action every 10 steps, the value function is not able to converge to the one obtained from value iteration and policy iteration. The maximum number of steps for each episode is 30 and the maximum number of episode explored is 400. There are 16 states in smallworld, so a maximum number of 30 steps should be sufficient for the algorithm to reach the goal state. Once the goal stated is reached, current episode is terminated and the algorithm starts a new episode. Once the value function or when the maximum number of iteration is reached, the learning stops.

```
function [v, pi, acc_R] = sarsa(model, maxit, maxeps, alpha, epsilon)
rand('seed',287)
% initialize the value function
Q = zeros(model.stateCount, 4);
v = zeros(model.stateCount, 1);
v_ = zeros(model.stateCount, 1);
pi = ones(model.stateCount, 1);
acc_R = zeros(maxeps,1);

for i = 1:maxeps,
    % every time we reset the episode, start at the given startState
    s = model.startState;
    prob1 = rand;
    if prob1 < epsilon  % epsilon greedily
        a = datasample([1 2 3 4], 1); % radom action
    else
        [¬,a] = max(Q(s,:));  % optimal action
    end

    for j = 1:maxit,
        % PICK AN ACTION
        p = 0;
        r = rand;
        for s_ = 1:model.stateCount,
            p = p + model.P(s, s_, a);
            if r ≤ p,
                break;
            end
        end
        acc_R(i) = acc_R(i) + model.R(s,a); % accumulated reward
        % pick next action epsilon greedily
        prob2 = rand;
        if prob2 < epsilon
            a_ = datasample([1 2 3 4], 1);
        else
            [¬,a_] = max(Q(s_,:));
        end

        Q(s,a) = Q(s,a) + alpha*(model.R(s,a)+model.gamma*Q(s_,a_)-Q(s,a));
        [v_(s),pi(s)] = max(Q(s,:));
        s = s_; a = a_;
        % once reach goal state, start a new episode
        if s == model.goalState
            fprintf('%d: reached the goal state! Start Again!\n', i)
            break;
        end
    end
    if max(abs(v_-v))<0.001
       fprintf('value function converged after %d episodes!', i)
       break;
    end
    v = v_;
end
```

2.4 Q-learning

Similar to SARSA, Q-learning is also a TD learning algorithm. Unlike SARSA, it is an off-policy algorithm, which means the policy it evaluates is different from the action it takes. The backup always uses the maximum Q-function without updating policy. As the algorithm is learning the optimal action-value function directly, it learns the optimal policy and suggests early convergence. As Q-learning is a model-free algorithm, its convergence is slower compared to model-based learning.

The optimal policy is obtained in the smallworld shown in Figure 5. The maxmum number of steps in each episode is 30, as the grid only contains 16 states. If the algorithm couldn't find the goal state in 30 steps it should start again. And maximum munber of episode is 400. The ϵ and α values are set the same as in SARSA for the same reasons.

Algorithm 4: Q-Learning algorithm

```
Initialise $Q$ arbitrarily, $Q(terminal, .) = 0$;
repeat
    Initialise s ;
    repeat
        Choose a $\epsilon$-greedily ;
        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;
until $\max_s |V_{k+1}(s) - V_k(s)| < \theta$;
```

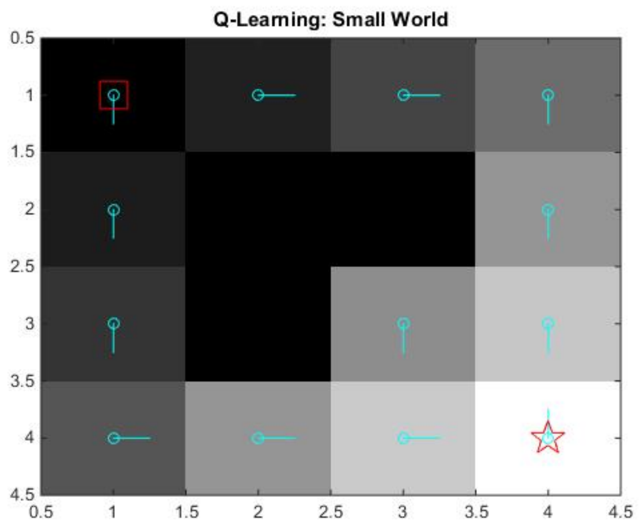
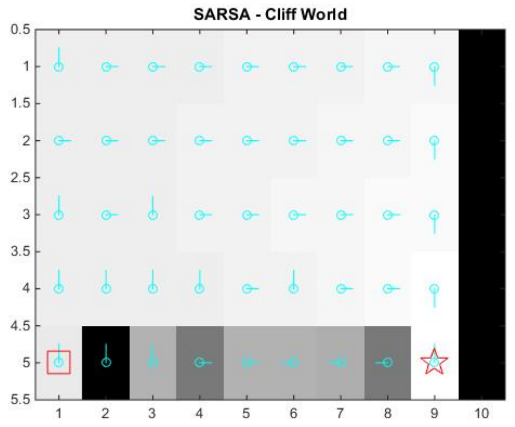


Figure 5: Smallworld: Q-learning

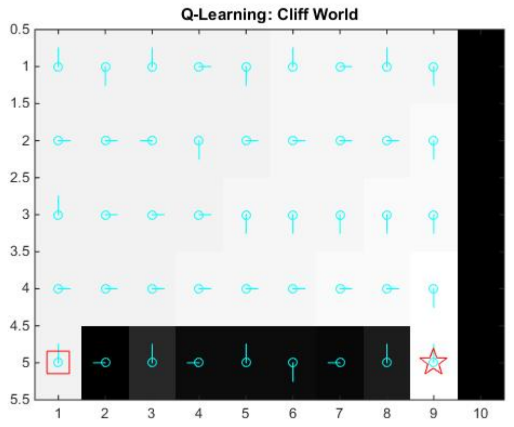
```
function [v, pi, acc_R] = qLearning(model, maxit, maxeps, alpha, epsilon)
rand('seed',287)
% initialize the value function
Q = zeros(model.stateCount, 4);
v = zeros(model.stateCount, 1);
v_ = zeros(model.stateCount, 1);
pi = ones(model.stateCount, 1);
acc_R = zeros(maxeps, 1);

for i = 1:maxeps,
    s = model.startState;

    for j = 1:maxit,
        % PICK AN ACTION
        prob1 = rand;
        if prob1 < epsilon
            a = datasample([1 2 3 4], 1);
        else
            [¬,a] = max(Q(s,:));
        end

        % look for next state s_ after action a
        p = 0;
        r = rand;
        for s_ = 1:model.stateCount,
            p = p + model.P(s, s_, a);
            if r ≤ p,
                break;
            end
        end
        acc_R(i) = acc_R(i) + model.R(s,a);
        Q(s,a) = Q(s,a) + alpha*(model.R(s,a) + model.gamma*max(Q(s_,:)) - ...
            Q(s,a));
        [v_(s), pi(s)] = max(Q(s,:));
        s = s_;

        % start new episode after reaching goal state
        if s == model.goalState
            fprintf('%d: reached the goal state! Start Again!\n', i)
            break;
        end
    end

    if max(abs(v_-v))<0.00001
       fprintf('value function converged after %d episodes!', i)
       break;
    end
    v = v_;
end
```

2.5 SARSA and Q-learning

SARSA and Q-learning are compared against each other in the cliffworld. The different performance is due to different update rule. Q-learning learns directly the optimal policy, which is the path along the edge of the cliff, by using maximum Q-function in its update rule. On the other hand, SARSA considers all actions in a particular state while updating Q-function, thus it learns a safer path further away from the cliff. This ensures that when algorithm chooses action ϵ-greedily, it doesn't fall off the cliff.



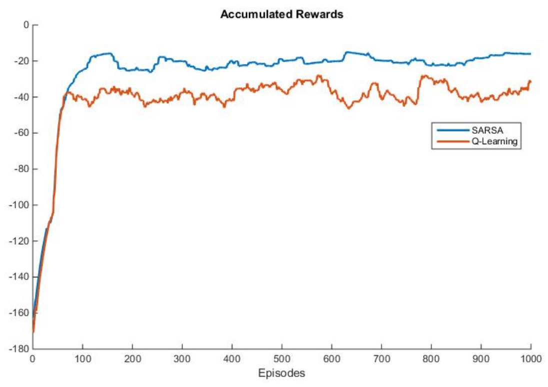
(a) SARSA



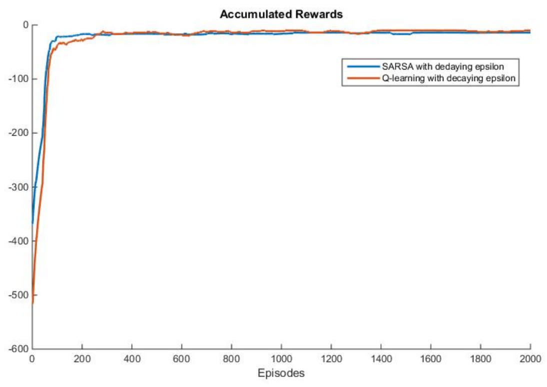
(b) Q-learning

Figure 6: Cliff World

The accumulated rewards show the same results (Figure 7(a)). Although Q-learning learns the optimal policy, it's more risky. When the algorithm explores, it will fall off the cliff resulting in worse overall accumulated reward. On the contrary, SARSA learns a safer path further away from the cliff, so even if the agent explores, the accumulated reward would still be higher. The accumulated reward is calculated over 1000 iterations with a smoothing window size of 80 iterations.



(a) $\epsilon = 0.1$



(b) Decaying ϵ

Figure 7: Accumulated rewards over iterations

If ϵ gradually reduces over time, the algorithm eventually stops exploring. In this case, the accumulated rewards from SARSA and Q-learning are going to merge as shown in Figure 7(b). However, this doesn't change the policy SARSA converges to, unless the algorithm visits all states infinite times.

```
%% --- accumulated reward --- %%
% smoothing accumulated reward cliff world
window_size = 80;
```

```
for i=1:length(R3) % SARSA
    if i ≤ window_size/2
        smooth_R3(i) = mean(R3(1:i+window_size/2));
    elseif i ≥ length(R3)-window_size/2
        smooth_R3(i) = mean(R3(i-window_size/2:length(R3)));
    else
        smooth_R3(i) = mean(R3(i-window_size/2:i+window_size/2));
    end
end

for i=1:length(Q_R3) % Q-learning
    if i ≤ window_size/2
        smooth_Q_R3(i) = mean(Q_R3(1:i+window_size/2));
    elseif i ≥ length(Q_R3)-window_size/2
        smooth_Q_R3(i) = mean(Q_R3(i-window_size/2:length(Q_R3)));
    else
        smooth_Q_R3(i) = mean(Q_R3(i-window_size/2:i+window_size/2));
    end
end
% plot accumulated rewards
figure;hold on
plot(smooth_R3); plot(smooth_Q_R3)
title 'Accumulated Rewards'
xlabel 'Episodes'
```

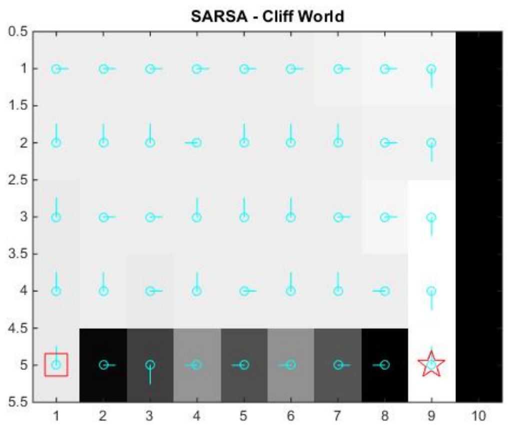
Explore combinations of ϵ and α for SARSA

After trying different combinations of epsilon and alpha, the following observations are made.

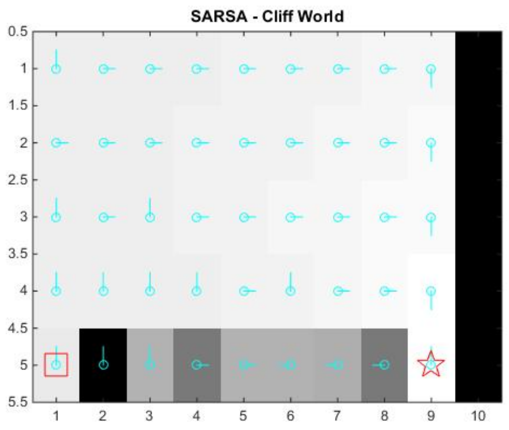
First of all, to converge to a policy using SARSA the scale of ϵ and α should match. This means when using larger ϵ, α should be increased accordingly. The level of exploration and learning rate should match in order to learn from experience. The more an agent explores, the learning rate should be higher in order to take into account all the experiences.

Secondly, the more the algorithm explores, the more it tends to converge to a safer path. This can be shown in figure 8. When $\epsilon = 0.3$ the algorithm has higher chance of falling off the cliff. In order to reach goal state, SARSA takes the path that's furtherest from the cliff. Also, as ϵ decreases, it starts to take paths that are closer to the cliff until the ϵ is small enough. When ϵ is very small, the algorithm hardly explores, it has less chance of falling off cliff, so it converges to the optimal path next to the cliff.

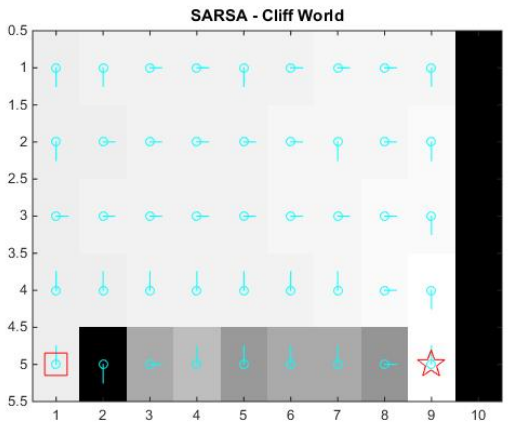
Finally, the more the algorithm explores, the more updates we have on all states. This will give better estimates for the final value function. Ideally, the cliff should be black in the plot. In practice, they tends to be different shades of grey. This is because they are not visited enough times to have their true value function estimated. When ϵ is larger, the algorithm explores more, therefore, the cliff has darker colour. The cliff next to the start state is black because it is very likely to be visited. This means its value was updated more than other cliff positions. So the algorithm has more information to give a better estimate of its value function.



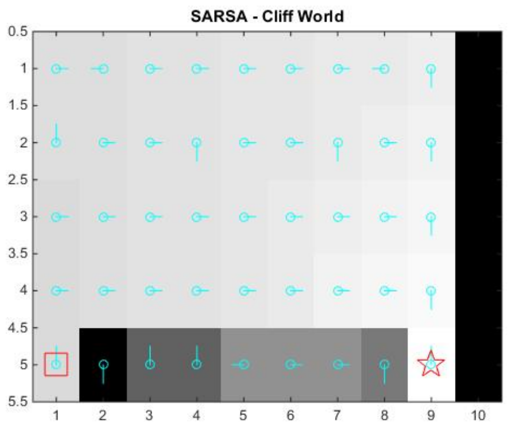
(a) $\epsilon = 0.3, \alpha = 0.5$



(b) $\epsilon = 0.1, \alpha = 0.3$



(c) $\epsilon = 0.1, \alpha = 0.1$



(d) $\epsilon = 0.05, \alpha = 0.05$

Figure 8: Cliffworld:policy under different combinations of ϵ and α

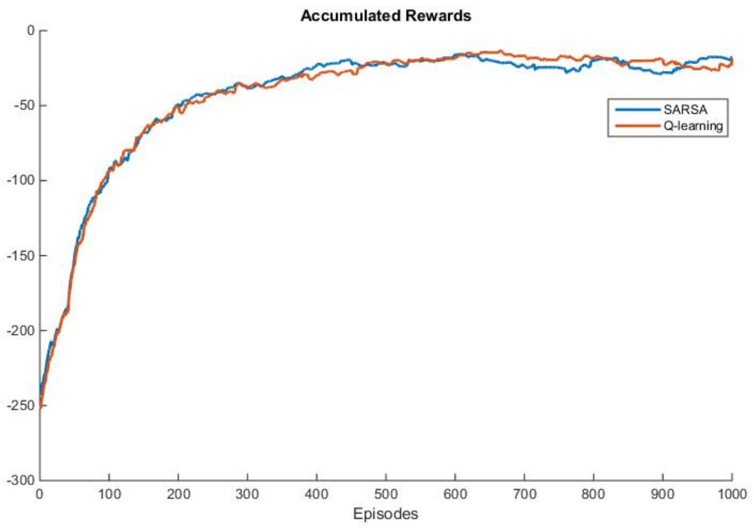


Figure 9: Accumulated reward when $\epsilon = 0.05, \alpha = 0.05$

Figure 9 shows the accumulated rewards from SARSA and Q-learning. In this particular case, SARSA was able to learn the optimal policy like Q-learning. As the epsilon is very small the accumulated reward from the two algorithms have the similar values.

3 Conclusion

There are three major types of reinforcement learning algorithm: dynamic programming, Monte Carlo methods and temporal difference learning. This report explored and implemented four basic algorithms in reinforcement learning, which shows the similarity and difference between the three types of methods mentioned.

Value iteration and policy iteration, examples of dynamic programming method, both learn from a known environment, therefore, they can both learn the optimal policy with fast convergence. Value iteration combines the two steps from policy iteration updating the value function and policy at the same time. This results in a simpler implementation. SARSA and Q-learning are TD learning algorithms, which takes the advantage from dynamic programming and Monte Carlo methods. SARSA is more aware of the risks involved in the actions, unlike Q-Learning which learns the optimal policy directly, it learns a safer non-optimal path. Different combinations of learning rate and epsilon value result in SARSA converging to different policy. This is due to the exploration nature of SARSA and on-policy methods. If SARSA has a decaying epsilon value and it visits all states an infinite number of times, it will eventually converge to the optimal policy and have the same accumulated rewards as Q-learning.

References

[1] Sutton, R S and Barto A G (1998) *Reinforcement Learning: An Introduction.* The MIT Press