Reinforcement Learning

Chris Tegho

March 23, 2017

In this project, the basic building-blocks of Reinforcement Learning are implemented for a discrete grid-world model. Policy Iteration, Value Iteration, SARSA and Q-Learning are considered. The convergence to an optimal policy and value functions is examined for all 4 algorithms. The code implementation in Matlab can be found at the end of the report.

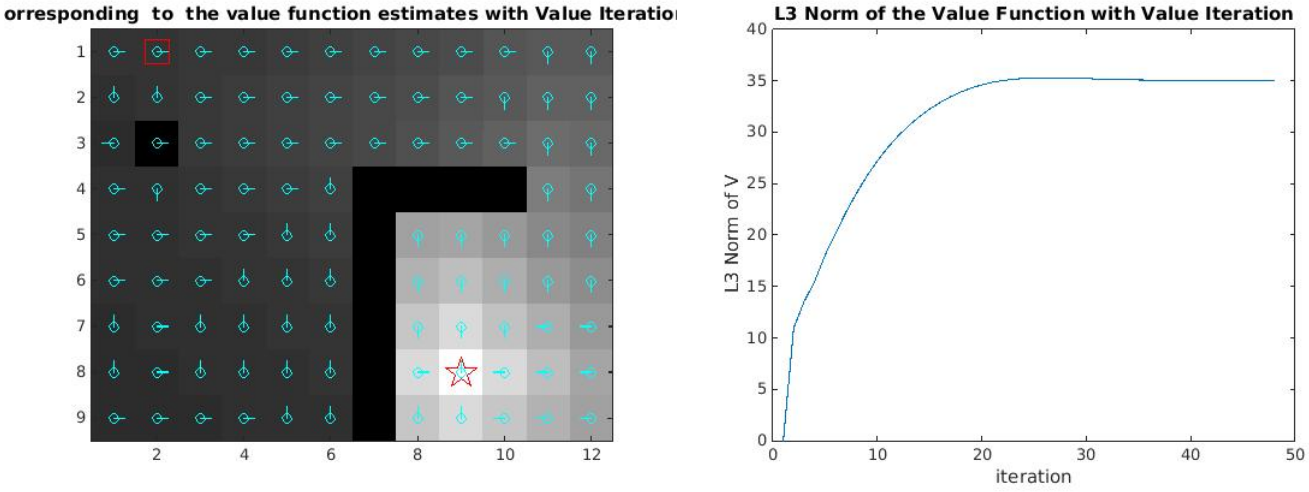
1 Value Iteration

Value iteration finds the optimal policy π^* and optimal value function V^* by turning the Bellman optimality equation into an update rule. The backup operation combines the policy improvement and truncated policy evaluation steps. Value iteration starts with an arbitrary value function $V_0(s)$ and applies successive approximations based on:

$$V_{k+1}(s) = max_a \sum_{s',r} p(s', r|s, a)(r + \gamma V_k(s')) \tag{1}$$

With an infinite number of iterations, it is guaranteed that value iteration converges to an optimal v*. In practice, the algorithm stops when the value function changes by only a small amount that is less than a threshold.

The value iteration algorithm is used to find the optimal policy and value function for the girdworld model. The results are show in figure 1. In the implementation, convergence is achieved when the norm of the difference between V_k and V_{k+1} is less than the threshold of 10^{-3}. For the gridworld, the value function converges after 46 iterations.



(a) (b)

Figure 1: (a) Greedy policy corresponding to the value function estimates with Value Iteration for the gridWorld. (b) Convergence of the L3 norm of the value function for the gridlworld.

2 Policy Iteration

Policy iteration uses an iterative policy evaluation step followed by a policy improvement step. The value function is arbitrarily initialized. Successive approximations of V are then obtained using the Bellman equation for the policy evaluation:

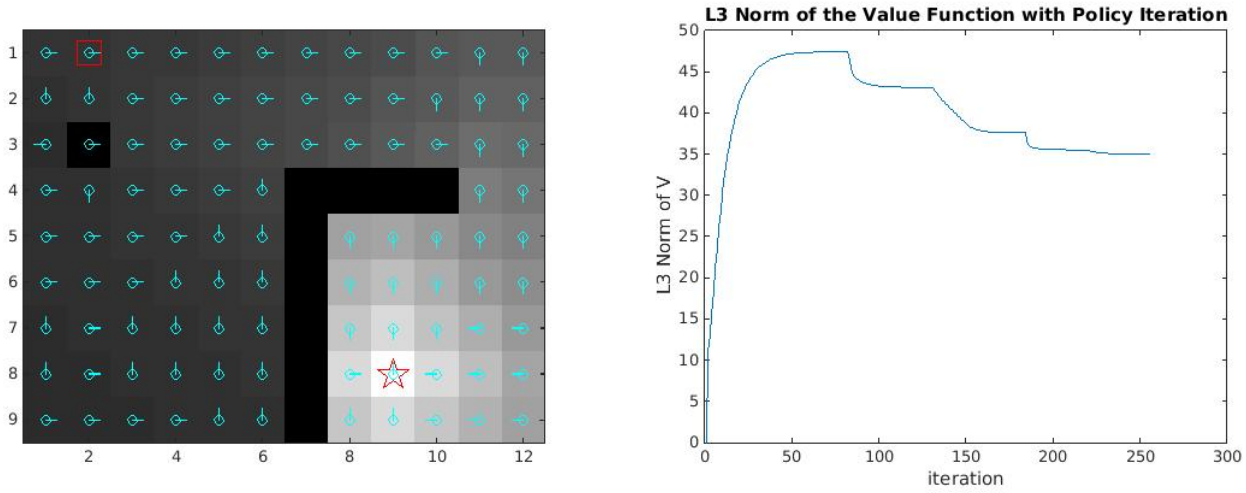
$$V_{k+1}(s) = \sum_a \pi(a, s) \sum_{s',r} p(s', r|s, a)(r + \gamma V_k(s')) \tag{2}$$

V is improved to yield a better policy π than the one obtained from the previous policy improvement iteration. The new, improved, greedy policy is obtained using the value function obtained from the previous policy evaluation:

$$\pi'(s) = argmax_a \sum_{s,r} p(s', r|s, a)(r + \gamma V_\pi(s')) \tag{3}$$

A sequence of monotonically improving policies and value functions is obtained: Each policy is guaranteed to be a strict improvement over the previous one (unless it is already optimal). Because a finite MDP has only a finite number of policies, this process must converge to an optimal policy and optimal value function in a finite number of iterations [1].

When policy iteration was used on the gridworld model, 9 iterations of policy evaluation followed by policy improvement for a total of 247 iterations of policy evaluations were necessary for the policy to become stable.



(a) (b)

Figure 2: (a) Greedy policy corresponding to the value function estimates with Policy Iteration for the gridworld. (b) Convergence of the L3 norm of the value function for the gridlworld.

2.1 Value Iteration vs Policy Iteration

Both algorithms lead to the same optimal policy π^* and value function V^*. In general, the convergence of the value function with Policy Iteration or Value Iteration depends on gamma, on the number of states in the model and the threshold for convergence (which were not investigated). The same gamma, and threshold for convergence were used with both algorithms. While policy iteration converged to the optimal policy after 9 iterations of policy improvements, a total of 247 iterations of policy evaluations were necessary. On the other hand, value iteration converged after 46 iterations in total only, which is much less than the total number of iterations that were necessary for the policy iteration to achieve same optimal policy.

Policy iteration consists of two simultaneous processes, one making the value function consistent with the current policy (policy evaluation), and the other making the policy greedy with respect to the current value function (policy improvement). In policy iteration, the two processes of policy evaluation and policy improvement alternate, each completing before the other begins. In value iteration, only a single iteration of policy evaluation is performed in between each policy improvement.

3 SARSA

SARSA is a Temporal Difference (TD) learning method. TD methods can learn directly from raw experience without a model of the environment's dynamics. SARSA is a model-free method for policy evaluation that bootstraps the value function from subsequent estimates. In SARSA, the value function is bootstrapped from the very next time-step. Rather than waiting until the complete return has been observed (as with Monte Carlo methods), the value function of the next state is used to approximate the expected return [1].

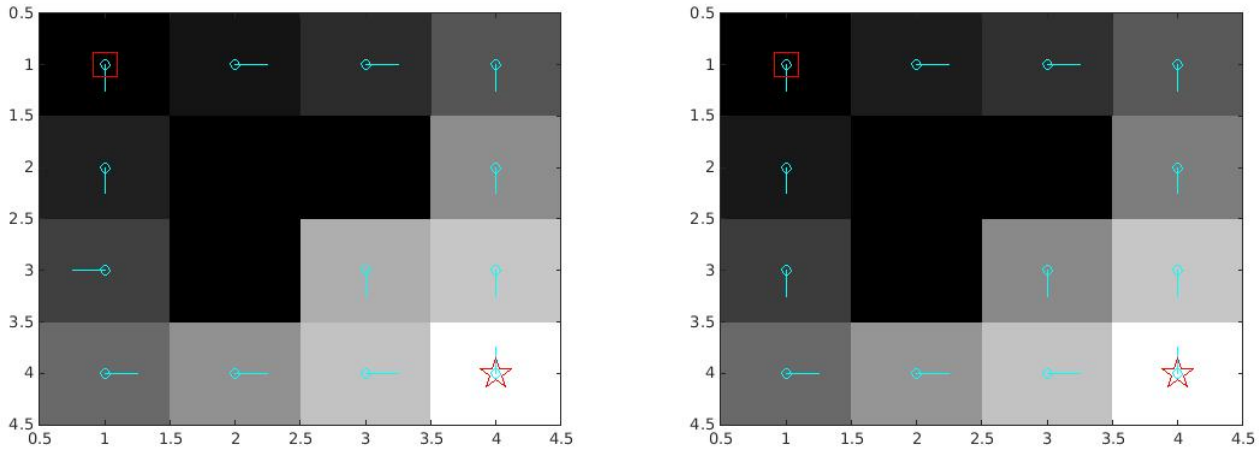
SARSA estimates an action value function, Q(s, a), for each state s and action a. At each time-step t, the next action a_t is selected by an ε-greedy policy with respect to the current action values Q(s,·). The Q function is updated as:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]. \tag{4}$$

This update is done after every transition from a nonterminal state s_t. If s_{t+1} is terminal, then $Q(s_{t+1}, a_{t+1})$ is defined as zero. If all states are visited infinitely many times, and ε decays to zero in the limit, SARSA converges to the optimal action-value function Q^* for all values of λ [2].

3.1 Convergence to the optimal policy

I ran SARSA on the small world model with 1000 maximum iterations within each episode and for 500 maximum episodes. Each episode terminated when the goal state was reached. I tested one version of the algorithm where it stops when the policy does not change after 1, 2 or 3 consecutive episodes, and a version of the algorithm where it runs until it reaches the maximum number of episodes. In the case where the algorithm stops after the policy does not change after 1, 2 or 3 consecutive episodes, the optimal policy obtained always presented only one of the two paths that lead to the goal states (figure 3.a), compared to the two paths obtained with the optimal policy when the algorithm runs until it reaches the maximum number of episodes (figure 3.b).

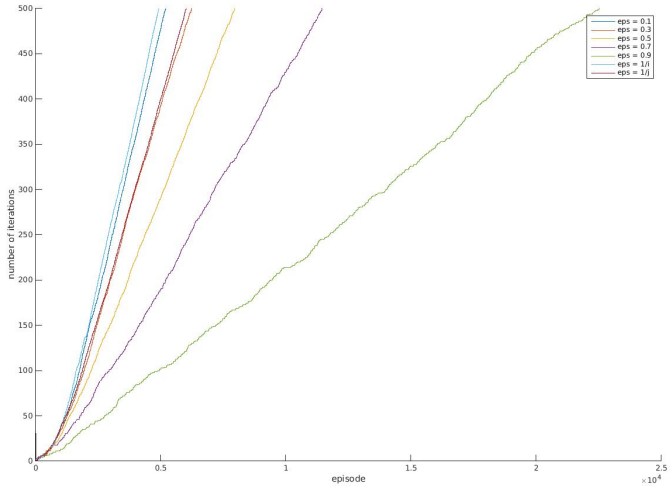


(a) (b)

Figure 3: (a) Optimal policy and value function when SARSA stops when the policy does not change after 2 consecutive episodes. (b) Optimal policy and value function when SARSA runs until the maximum number of episodes is reached.

3.2 Effect of ε

I ran SARSA with different values of ε to examine its effect on reaching the goal state within each episode and convergence to the optimal value function and optimal policy. Results are summarized in figure 4.

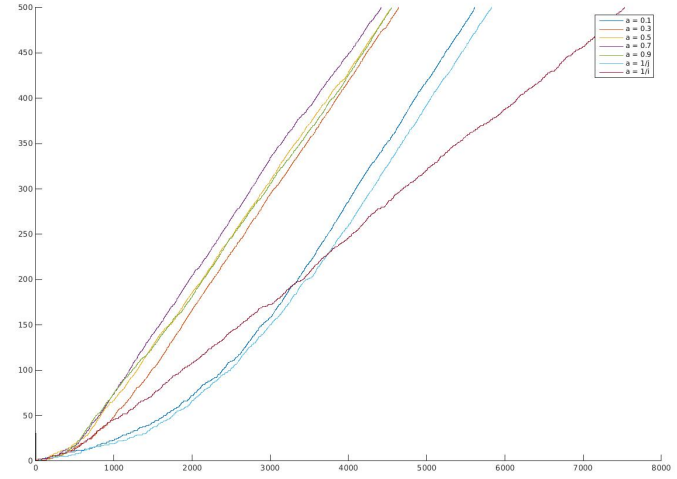


(a)

Figure 4: Number of iterations necessary until convergence of SARSA with different values of α. X axis is the number of iterations. Y axis is the episode number.

A decaying epsilon greedy parameter $\varepsilon = 1/episodenumber$, in which the ε parameter decays with the number of episodes, allowed faster convergence of the value function at later episodes. The lowest total number of iterations after 500 episodes of SARSA was also obtained with a decaying epsilon greedy parameter. At earlier episodes, the agent has not explored all states yet and a decaying ε allows more exploration of the environment, increasing the probability for the agent to explore actions from a uniform distribution. At later stages, the agent has explored the environment and is encouraged to exploit the information it has learned with earlier episodes, which is achieved by a smaller ε value. If ε is to be kept fixed, a lower value of ε (0.2-0.5) is preferable over larger values of ε (0.9 for instance), as larger values of ε will result in too much exploration without ever converging.

3.3 Effect of the learning rate α



(a)

Figure 5: Number of iterations necessary until convergence of SARSA with different values of α. X axis is the number of iterations. Y axis is the episode number.

The learning rate determines to what extent the newly acquired information will override the old information. A factor of 0 will make the agent not learn anything, while a factor of 1 would make the agent consider only the most recent information. Overall, higher values of alpha allowed faster convergence of the value function and lower total

number of iterations for 500 episodes were obtained. A decaying learning rate led to larger number of total iterations for 500 episodes compared to the other values of alpha that were tested.

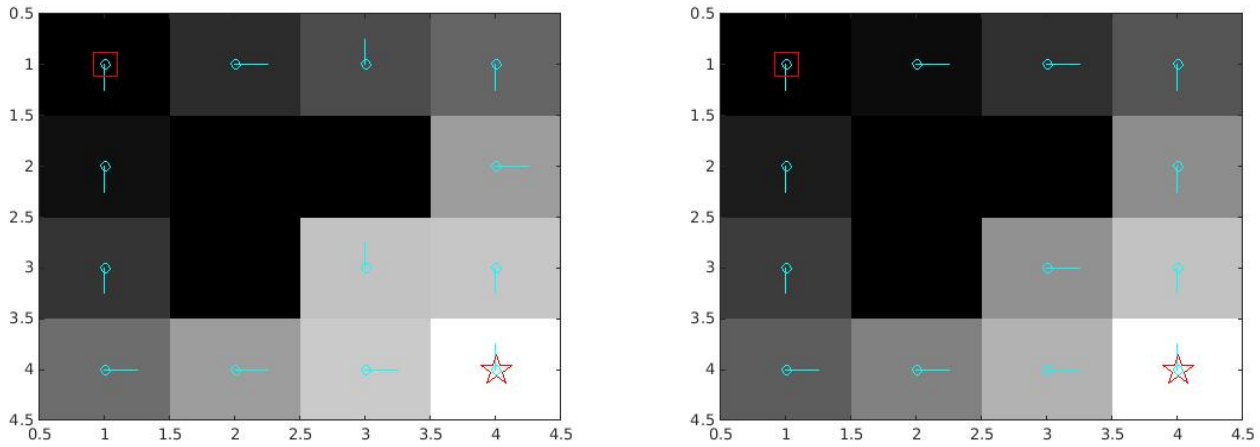
4 Q-Learning

In Q-Learning, the learned action-value function, Q, directly approximates $Q*$, the optimal action-value function using a target policy, independent of the policy being followed - the behavior policy. The update rule for the Q function is as follow:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha(r_{t+1} + \gamma max_{a'} Q(s_{t+1}, a') - Q(s_t, a_t))) \tag{5}$$

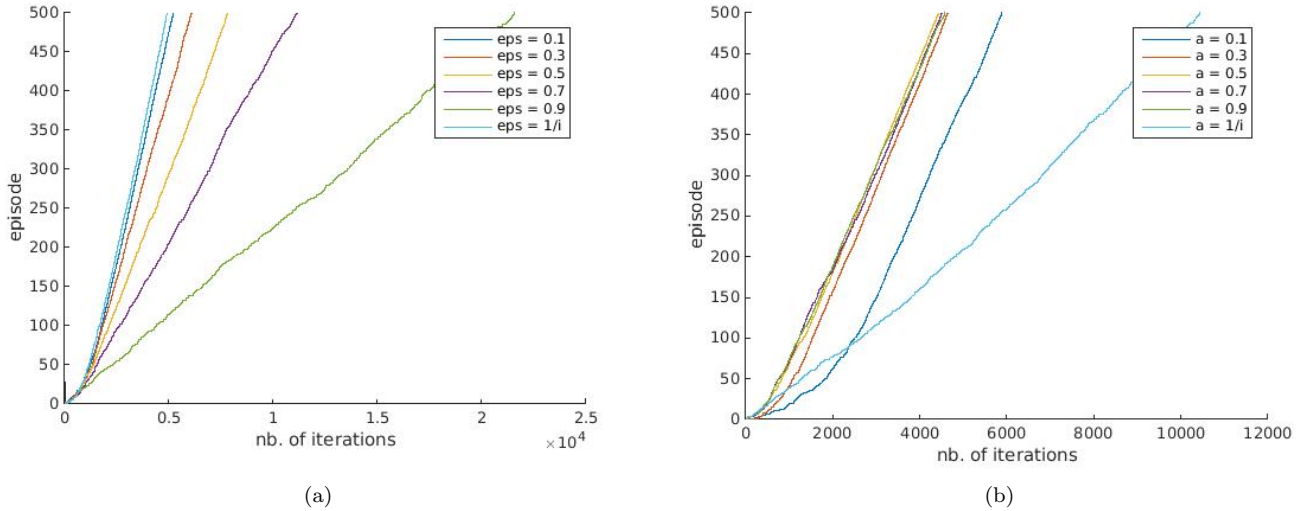
The behavior policy has an effect in that it determines which state–action pairs are visited and updated and is determined by epsilon-greedily, ensuring sufficient exploration, and all that is required for correct convergence is that all pairs continue to be updated [1].

Similar observations to the ones with SARSA were obtained when I varied the learning parameter α and the ε greedy parameter, see figure 7.



(a) (b)

Figure 6: (a) Optimal policy and value function when Q-Learning stops when the policy does not change after 2 consecutive episodes. (b) Optimal policy and value function when Q-Learning runs until the maximum number of episodes is reached.



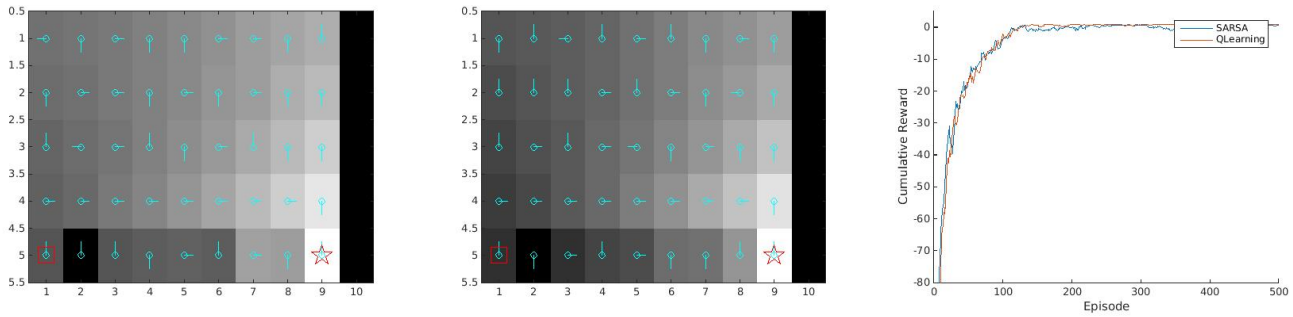
(a) (b)

Figure 7: Number of iterations necessary until convergence of Q-Learning with different values of (a) ε and b) α. X axis is the number of iterations. Y axis is the episode number.

5 CliffWorld

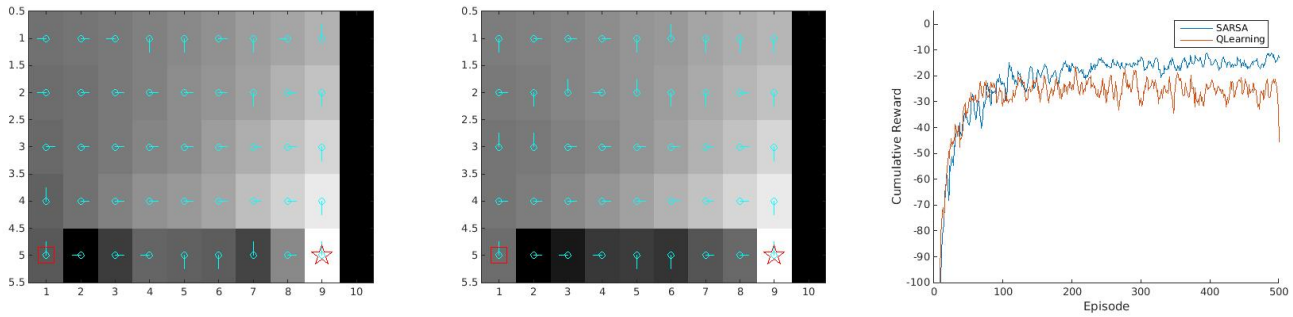
In this section, SARSA and Q-Learning are used with the cliffworld model. In this model, the states on the same line between the start and the goal states (the cliff) incurs a negative reward of -6 and sends the agent instantly back to the start. I changed the reward for the cliff to -100 and compared performances for the original model with a reward of -6. The average cumulative reward over 10 runs of each of the algorithms is plotted (after smoothing) through the learning process to examine convergence of each method to an optimal Q function. Note that the scale of the y axis for the plots of the cumulative reward for the different cases are not the same and one needs to look closely at the y axis to understand differences between the two methods

5.1 Comparison



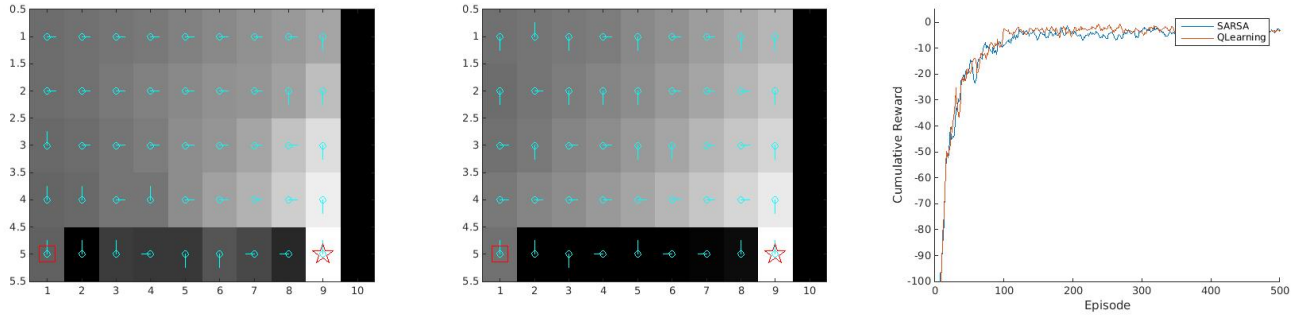
(a) (b) (c)

Figure 8: Performance of the (a) Sarsa and (b) Q-learning methods with ε-greedy action selection. The cumulative reward is plotted (c) for both methods. α **= 0.2,** $\varepsilon = 1/i$**, R for the cliff is -6.**



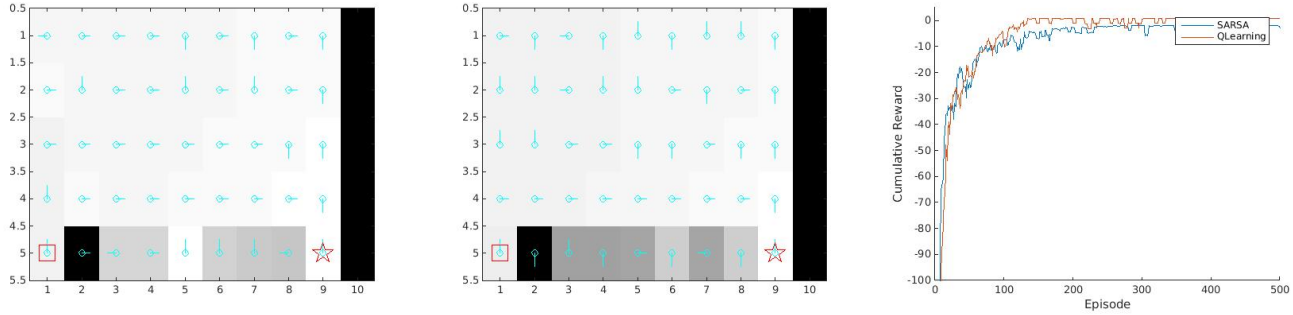
(a) (b) (c)

Figure 9: Performance of the (a) Sarsa and (b) Q-learning methods with ε-greedy action selection. The cumulative reward is plotted (c) for both methods. α **= 0.2,** ε **= 0.1, R for the cliff is -6.**



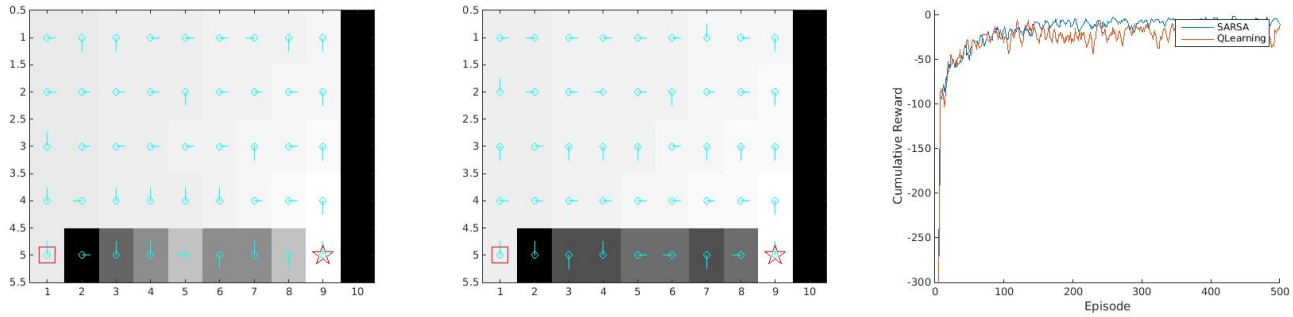
(a) (b) (c)

Figure 10: Performance of the (a) Sarsa and (b) Q-learning methods with ε-greedy action selection. The cumulative reward is plotted (c) for both methods. α **= 0.2,** ε **= 0.4, R for the cliff is -6.**



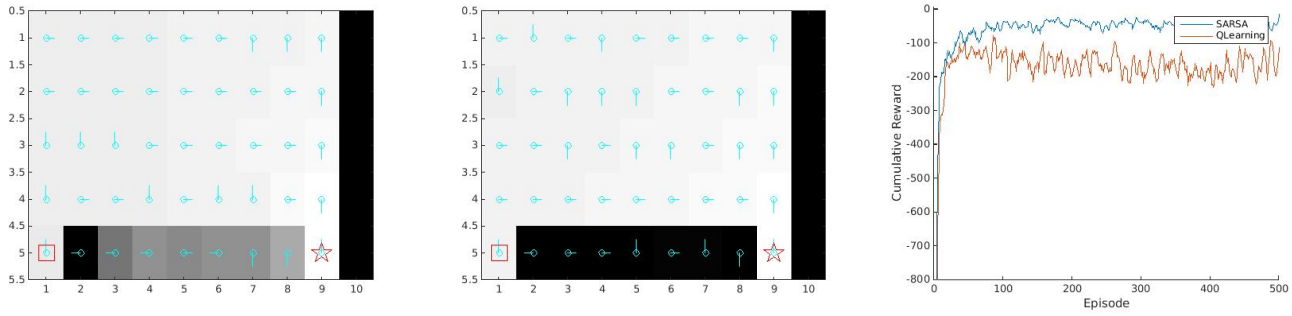
(a) (b) (c)

Figure 11: Performance of the (a) Sarsa and (b) Q-learning methods with ε-greedy action selection. The cumulative reward is plotted (c) for both methods. α **= 0.2,** ε **= 1/i, R for the cliff is -100.**



(a) (b) (c)

Figure 12: Performance of the (a) Sarsa and (b) Q-learning methods with ε-greedy action selection. The cumulative reward is plotted (c) for both methods. α **= 0.2,** ε **= 0.1, R for the cliff is -100.**



(a) (b) (c)

Figure 13: Performance of the (a) Sarsa and (b) Q-learning methods with ε-greedy action selection. The cumulative reward is plotted (c) for both methods. α **= 0.2,** ε **= 0.4, R for the cliff is -100.**

5.2 Varying ε

After an initial transient, both SARSA and Q-learning learn values for the optimal policy. At the beginning, the cumulative reward has very large negative values (these values are larger for larger ε values) since the agent has not yet learned about the environment and is more likely to explore actions that leads to states in the cliff region. Q-Learning learns an optimal policy that travels right along the edge of the cliff, which results in its occasionally falling off the cliff because of the ε-greedy action selection. Sarsa takes the action selection into account and learns a longer but safer path through the upper part of the grid.

For high values of ε, the difference between the two methods is more apparent: for a reward of -100 for the cliff region and with $\varepsilon = 0.4$, SARSA achieves a much higher cumulative reward than Q-Leaning. Sarsa takes a path to the goal that is 3 steps higher than the one obtained with Q-Learning. As ε is reduced (especially when ε is set to $1/i$), i.e. as exploration is reduced, both methods achieve similar cumulative rewards and asymptotically converge to the optimal policy; and SARSA takes a path that is closer to the cliff than the ones it took with a higher values of ε.

Further, SARSA achieves similar cumulative rewards when ε is changed. Whereas Q-Learning achieves lower cumulative rewards for higher values of ε. This is because the Q function in SARSA is updated using the on-policy that maximizes the Q function. Each SARSA update of the Q function is dependent on the states and actions visited. Whereas Q-Learning evaluates and improves a policy different from the one used to explore the environments, and is hence more dependent to the ε greedy action selection, which controls the exploitations/exploration trade-off.

5.3 Varying the reward for the cliff

As the reward for the cliff region decreases from -6 to -100, SARSA achieves much higher cumulative reward than Q-Learning (for a fixed value of ε) (see difference between figure 10.c and 13.c). This indicates that SARSA is better for handling state spaces where exploration can be (extremely) costly.

5.4 Convergence to goal state

Table 1: Average number of iterations per episode until goal state is reached. Numbers are shown for 10 runs of each algorithm, with 500 episodes for each run.

ε	0.4	0.1	1/i
SARSA	35.55	22.694	20.04
Q-Learning	38.14	21.572	19.722

Table 1 shows the average number of iterations per episode (averaged over 500 episodes, and 10 runs of each algorithm) until reaching the goal state. For low values of ε, Q-Learning reaches the goal state at each episode in less iterations than SARSA. This is because Q-Learning takes the path closest to the cliff path, since it is indifferent to the costs incurred by exploratory actions. Whereas SARSA learns to take the slow path because it gains experience on a cost that exploratory action may incur. For higher values of ε, exploration increases and more iterations per episode are necessary for SARSA and even more for Q-Learning to reach the goal state.

5.5 Summary

This practical investigated 4 algorithms in reinforcement learning. When used for the world grid model, Value Iteration converged to the optimal policy and value function in much less iterations of policy- evaluation and policy-improvement than Policy Iteration. With the cliff world model, SARSA, an on-policy learning method achieved a higher cumulative rewards than Q-Learning. Q-Learning however reached the goal state in less iterations per episode than SARSA, for low ε. This is mainly because SARSA takes a safer and longer path than Q-Learning, especially for large values of ε. Q-Learning and SARSA are TD methods and have an advantage over Policy Iteration and Value Iteration methods in that they do not require a model of the environment, of its reward and next-state probability distributions.

References

[1] R. S. Sutton and A. G. Barto. *Introduction to Reinforcement Learning*. MIT Press, 1998.

[2] Silver, David, Richard S. Sutton, and Martin Müller. "Temporal-difference search in computer Go." Machine learning 87, no. 2 (2012): 183-219.

6 Implementation in MATLAB

6.1 Value Iteration

```
function [v, pi, normV, it] = valueIteration(model, maxit)

tol = 0.001;
nstates = size(model.P,1);
v = zeros(1,nstates);
v_ = zeros(1,nstates);
pi = zeros(nstates,1);
maxiter = 0;
diff = inf;
normV = zeros(1,1);
it=0;
while (diff > tol && it < maxit)
    maxiter = max(it,maxiter);

    for state= 1:nstates
        P = reshape(model.P(state,:,:), size(model.P,2),size(model.P,3));
        R = repmat(model.R(state,:), nstates,1);
        [A, I] = max(sum(P.*(R+model.gamma*repmat(v, size(R,2),1)')));
        v_(state) = A;
        pi(state) = I;
    end
    it = it+1;
    diff = norm(v-v_,3);
    normV = [normV, norm(v_,3)];
    v = v_
end
maxiter
end
```

6.2 Policy Iteration

```
function [v, pi, normV, itTotal] = policyIteration(model,maxit)
tol = 0.001;
nstates = size(model.P,1);
v = zeros(1,nstates);
v_ = zeros(1,nstates);

pi = ones(nstates,1);
```

```
maxiter = 0
it2 = 0;
piStable = false;
normV = zeros(1,1);
itTotal = zeros(1,1);

while( it2 < maxit && ~piStable)

    %policy evaluation
    diff = inf;
    it=0;
    while (diff > tol && it < maxit)
        maxiter = max(it,maxiter);

        for state= 1:nstates
            P = reshape(model.P(state,:,:), size(model.P,2),size(model.P,3));
            R = repmat(model.R(state,:), nstates,1);
            pi_prob = zeros(1,size(R,2));
                pi_prob(pi(state)) = 1;
            v_(state)  = sum(pi_prob.*(sum(P.*(R+model.gamma*repmat(v, size(R,2),1)'))));

        end
        it = it+1;
        diff = norm(v-v_,3);
        normV = [normV, norm(v_,3)];
        v = v_
    end
    itTotal = [itTotal, it-1];
    %policy improvement
    piStable = true;
    for state2= 1:nstates

        P2 = reshape(model.P(state2,:,:), size(model.P,2),size(model.P,3));
        R2 = repmat(model.R(state2,:), nstates,1);
        pi_old = pi(state2);
        [A, I] = max(sum(P2.*(R2+model.gamma*repmat(v, size(R2,2),1)')));
        pi(state2) = I;
            if (pi(state2) ~= pi_old)
                piStable = false;
            end

    end
    it2 = it2+1;

end
```

6.3 SARSA

```
function [v, pi, cumulativeR, itEps, epsIt] = sarsa(model, maxit, maxeps)
v = zeros(model.stateCount, 1);
pi = ones(model.stateCount, 1);
% initialize the value function
Q = zeros(model.stateCount, 4);
cumulativeR = zeros(maxeps,1);
itEps = [];
epsIt = [];
for i = 1:maxeps,
    % every time we reset the episode, start at the given startState
    s = model.startState;
    eps = 1/i;
```

```
    if rand < (1-eps)
            [mQ, a] = max(Q(s,:));
    else
            a = randi(4);
    end
    for j = 1:maxit,
        p = 0;
        r = rand;

        for s_ = 1:model.stateCount,
            p = p + model.P(s, s_, a);
            if r <= p,
                break;
            end
        end

        % s_ should now be the next sampled state.
        alpha = 0.2;
        if rand < (1-eps)
            [mQ_, a_] = max(Q(s_,:));
        else
            a_ = randi(4);
        end

        Q(s,a) = Q(s,a) + alpha*(model.R(s,a)+model.gamma*Q(s_,a_) -Q(s,a));
        s=s_;
        a=a_;
        cumulativeR(i) = cumulativeR(i) + model.gamma*model.R(s,a);
        epsIt = [epsIt, i];
        if s == model.goalState
            break;
        end
    end
    itEps = [itEps ,j];
    [A,I] = max(Q');
    pi = I';
    v = A';
    end
end
```

6.4 Q-Learning

```
function [v, pi, cumulativeR,  itEps, epsIt] = qLearning(model, maxit, maxeps)
v = zeros(model.stateCount, 1);
pi = ones(model.stateCount, 1);
% initialize the value function
Q = zeros(model.stateCount, 4);
cumulativeR = zeros(maxeps,1);
itEps = [];
epsIt = [];
a = randi(4);
for i = 1:maxeps,
    % every time we reset the episode, start at the given startState
    s = model.startState;
    eps = 1/i;
    if rand < (1-eps)
            %[mQ, a] = max(Q(s,:));
    else
            %a = randi(4);
    end
```

```
    for j = 1:maxit,
        p = 0;
        r = rand;

        %Choose a from s using policy derived from Q
        if rand < (1-eps)
            [mQ, a] = max(Q(s,:));
        else
            a = randi(4);
        end

        %take action a, observe r and s_
        for s_ = 1:model.stateCount,
            p = p + model.P(s, s_, a);
            if r <= p,
                break;
            end
        end

        %update rule for q
        alpha = 0.2;

        Q(s,a) = Q(s,a) + alpha*(model.R(s,a)+max(model.gamma*Q(s_,:)) -Q(s,a));

        s=s_;
        cumulativeR(i) = cumulativeR(i) + model.gamma*model.R(s,a);
        epsIt = [epsIt, i];
        if s == model.goalState
            break
        end
    end
    itEps = [itEps ,j];
    [A,I] = max(Q');
    pi = I';
    v = A';

end
```