# MLMI7 Practical Report

Word Count: 1994

March 30, 2023

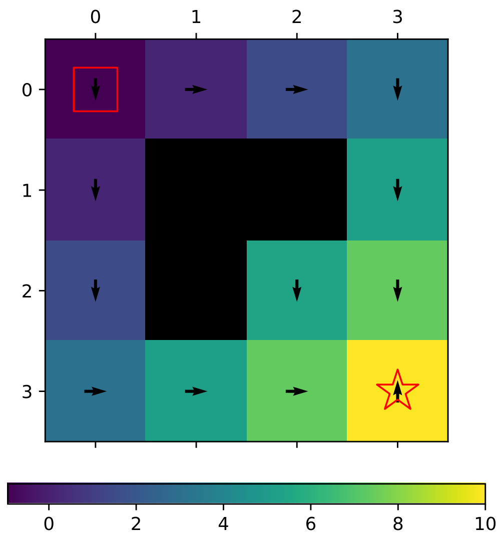
## 1 Problem a)

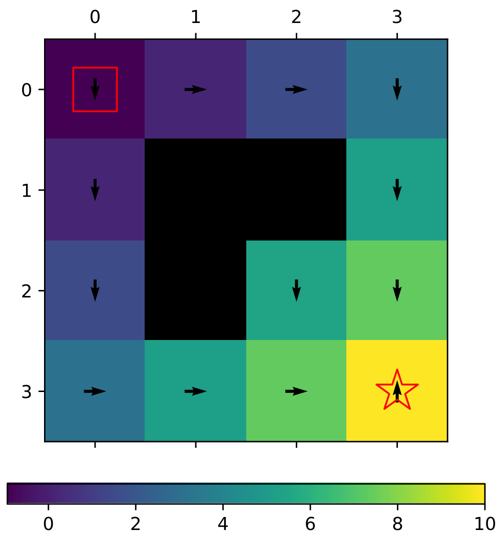
The (synchronous) Value Iteration (VI) algorithm is implemented by the following code:

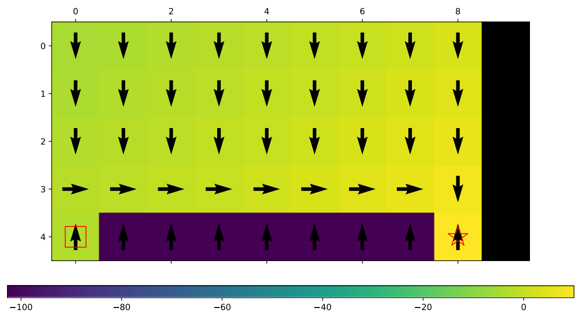
**Listing 1** The code for VI implementation

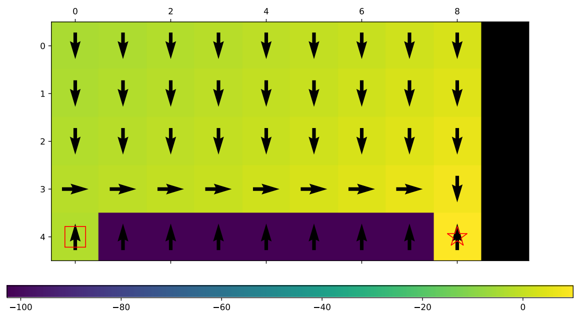
```
def value_iteration(model: Model, maxit: int = 100, tol: float=0.01):
    V = np.zeros((model.num_states,))
    pi = np.zeros((model.num_states,))
    mae = []

    def compute_value(s, a, reward: Callable):
        return np.sum(
            [
                model.transition_probability(s, s_, a)
                * (reward(s, a) + model.gamma * V[s_])
                for s_ in model.states
            ]
        )

    def value_update():
        for s in model.states:
            action_val = np.max(
                [compute_value(s, a, model.reward) for a in Actions]
            )
            V[s] = action_val

    for i in tqdm(range(maxit)):
        V_old = np.copy(V)
        value_update()
        mae.append((abs(V - V_old)).mean())
        if all(abs(V - V_old) <= tol):
            print("breaking")
            break

    for s in model.states:
        action_index = np.argmax(
            [compute_value(s, a, model.reward) for a in Actions]
            )
        pi[s] = Actions(action_index)

    return V, pi, mae
```

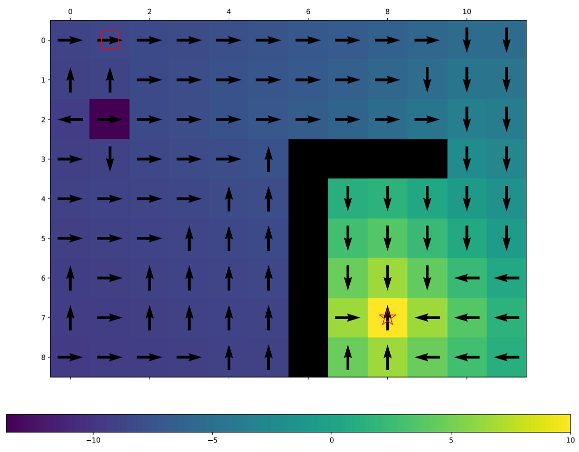
The correctness of our implementation of VI can be compared with the policy returned by Policy Iteration (PI) on `small_world`, `cliff_world` and `grid_world`:


(a) PI on `small_world`


(b) VI on `small_world` (`tol`=0.0001)


(c) PI on `cliff_world`


(d) VI on `cliff_world` (`tol`=0.0001)


(e) PI on `grid_world`

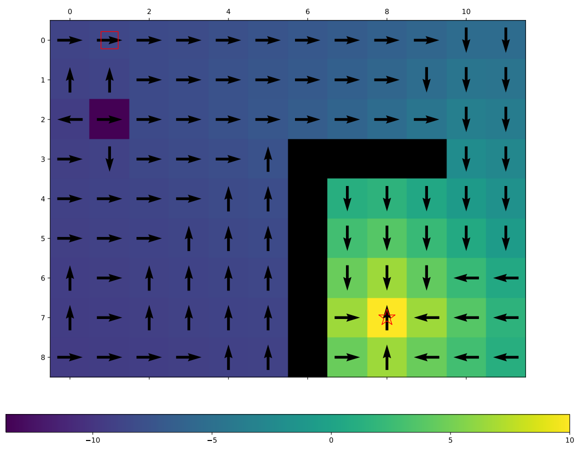
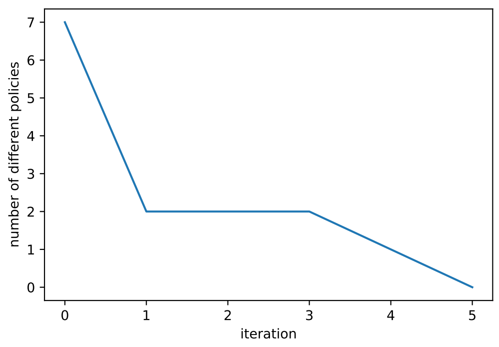

(f) VI on `grid_world` (`tol`=0.0001)

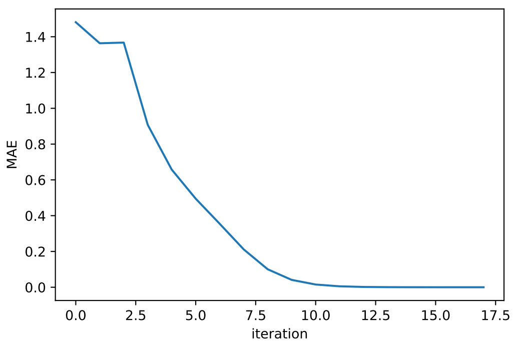
Figure 1: Policy and Value output for VI and PI on different environments

From Figure 1, we can see that exactly the same policies have been learnt from PI and VI in all the three environments.

When it comes to the stopping criteria for both algorithms, PI updates the policy at each iteration (although the updated policy would be then used to update the value function) and VI updates the value function directly. Thus, for PI, the stopping criteria would be that the number of different policies between the updated policy and the old policy in the last iteration drops to 0 (this criteria is implemented in the PI function here) and for VI, the stopping criteria would be that for each state $s \in \mathcal{S}$ ($\mathcal{S}$ is the state space), the absolute value of the difference between its updated value function and its old value function in the last iteration dropped under a threshold `tol` (i.e, $|V(s) - V_{\text{old}}(s)| \leqslant \texttt{tol}, \ \forall s \in \mathcal{S}$, as shown above in Listing 1). Below, we provided plots that shows the convergence of PI and VI on all the three environments, where we use the total number of different policies between the updated policy and the old policy in the last iteration to show the convergence of PI and the Mean Absolute Error (MAE) of the absolute value of the difference between two consecutive value function vectors (the value function vector contains the value of all states $s$ in an iteration) to show the convergence of VI (we choose `tol` to be 0.0001 here as we found that `tol`=0 makes the convergence of VI on `grid_world` to be longer).

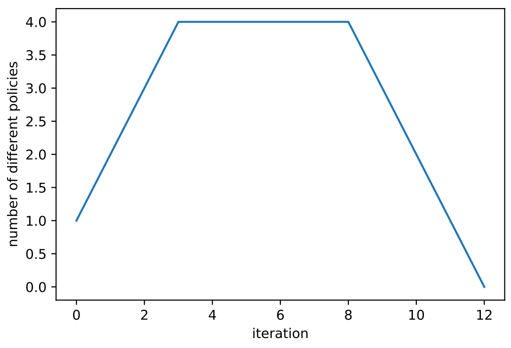


(a) Convergence of PI on `small_world` (no `tol`)

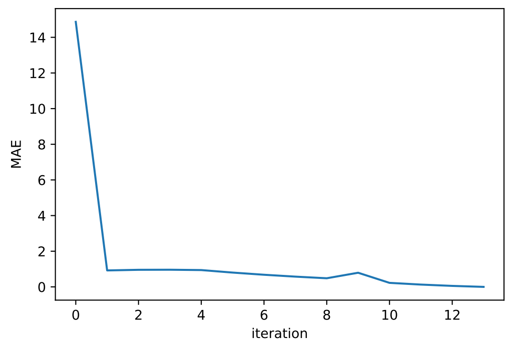


(b) Convergence of VI on `small_world` (`tol`=0.0001)

Figure 2: Convergence of VI and PI on `small_world`

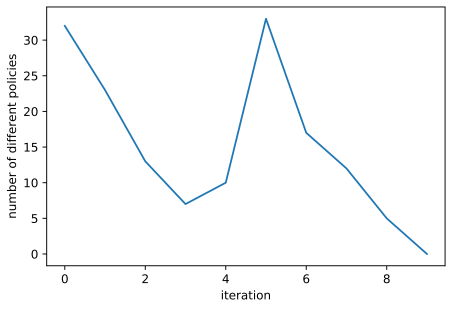


(a) Convergence of PI on `cliff_world` (no `tol`)

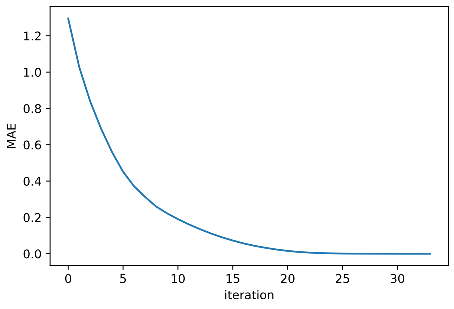


(b) Convergence of VI on `cliff_world` (`tol`=0.0001)

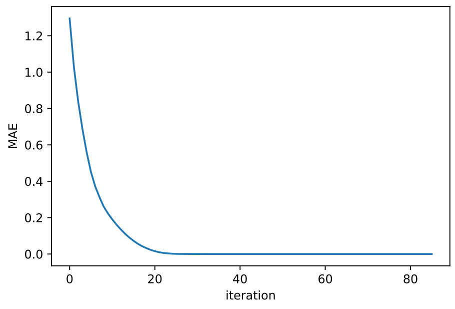
Figure 3: Convergence of VI and PI on `cliff_world`



(a) Convergence of PI on `grid_world` (no `tol`)



(b) Convergence of VI on `grid_world` (`tol`=0.0001)



(c) Convergence of VI on `grid_world` (`tol`=0.0)

Figure 4: Convergence of VI and PI on `grid_world`

From Figure 2, 3, we can see that for `small_world` and `cliff_world`, both VI and PI converges quite fast with VI takes more iteration than PI (each iteration of VI is slower). From Figure 4, we can see that setting `tol` to 0 indeed increases the number of required iterations a lot while `tol`=0.0001 seems to be better.

We have implemented the synchronous VI algorithm in this practical as we have saved two versions of the value function (all states are backed up in parallel) and the update involves operations over the entire state set. Thus, if the state space is very large, a single iteration would be computationally expensive. The asynchronous VI, instead, backs up the values of states in any order using whatever values of other states happen to be available, which means that we do not necessarily need to save two versions of the value function (one copy would be enough, such as in the in-place dynamic programming method) and hence reduce a lot of computation time. This means that the asynchronous VI is more efficient than the synchronous VI.

Let $|\mathcal{S}|$ be the total number of possible states in state space and $|\mathcal{A}|$ be the total number of possible actions in action space $\mathcal{A}$. In each iteration, the time complexity of VI is $\mathcal{O}(|\mathcal{S}|^2|\mathcal{A}|)$ as each time VI iterates over all actions to find maximum values over all states one step ahead for each of the state in the state space. However, in each iteration, PI has to first do policy evaluation for all states in $\mathcal{S}$ with a total time complexity of $\mathcal{O}(|\mathcal{S}|^2)$ and then improve the policy with a time complexity of $\mathcal{O}(|\mathcal{S}|^2|\mathcal{A}|)$. Therefore, the computational cost of PI would be larger for each iteration, but depends on the environment, PI converges with less iterations than VI so the actually time spent for convergence for VI is not always smaller.

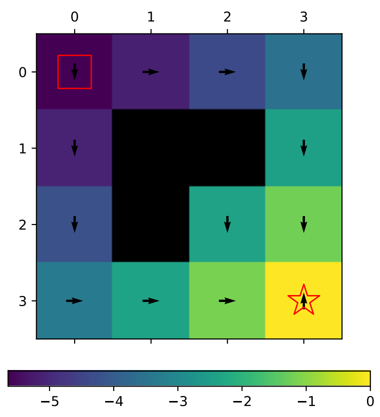
## 2 Problem b)

The SARSA algorithm is implemented by the following code:

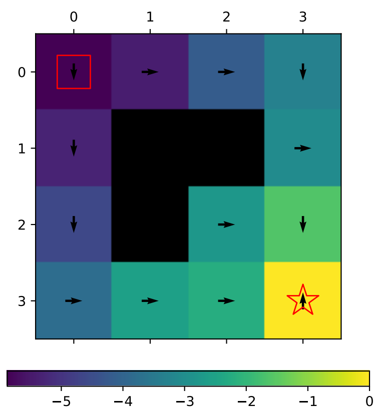
**Listing 2** The code for SARSA implementation

```python
def sarsa(model: Model, maxit: int = 100, num_episode: int = 300, eps: float
                                      =0.1, alpha: float=0.3):
    V = np.zeros((model.num_states,))
    pi = np.zeros((model.num_states,))
    Q = np.zeros((model.num_states, len(Actions)))


    def eps_greedily(s, epsilon):
        unif = np.random.rand()

        if unif < epsilon:
            idx = np.random.randint(0, len(Actions))
            return Actions(idx)

        else:
            return Actions(np.argmax(Q[s]))

    for i in tqdm(range(num_episode)):
        s = model.start_state
        a = eps_greedily(s, eps)
        Q_old = np.copy(Q)
        for j in range(maxit):

            r = model.reward(s, a)
            possible_dict= model._possible_next_states_from_state_action(s,
                                                        a)
            possible_s = list(possible_dict.keys())
            prob = list(possible_dict.values())
            s_next = np.random.choice(possible_s, p = prob)
            a_next = eps_greedily(s_next,eps)
            if s_next == model.goal_state:
                Q[s, a] = Q[s, a] + alpha*(r - Q[s, a])
            else:
                Q[s, a] = Q[s, a] + alpha*(r + model.gamma*Q[s_next, a_next]
                                                            - Q[s, a])
            s = s_next
            a = a_next
            if s == model.goal_state:
                break

        if np.all(abs(Q-Q_old)<=0.0001):
            print("breaking")
            break

    V = np.amax(Q, axis=1)
    pi = np.argmax(Q, axis=1)
    return V, pi
```

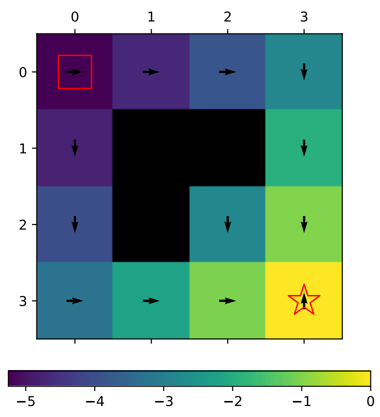
the exploration rate $\epsilon$ controls whether we want SARSA to explore more paths or just to stick to the greedy path for most of the times (episodes). Larger $\epsilon$ means more exploration and here, as we are running on `small_world` with relatively small $\mathcal{S}$ and $\mathcal{A}$, setting $\epsilon$ to be smaller might be better as this might help SARSA converges to the optimal policy more quickly without exploring too much (and being unable to converge within a fixed amount of episodes). Moreover, we also want a modest learning rate $\alpha$ to make sure that the Q function is still being updated in a reasonable pace which is neither too aggressive nor too slow. We will try with $\epsilon \in \{0.1, 0.3, 0.5\}$ and $\alpha \in \{.1, 0.3, 0.5\}$ with the maximum number of iterations per episode (maxit) and the number of episodes (num_episode) being 100 and 600 respectively.



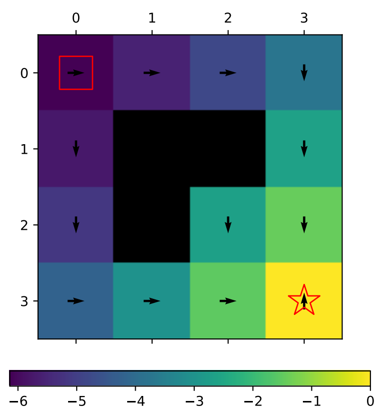
(a) $\epsilon = 0.1$, $\alpha = 0.1$



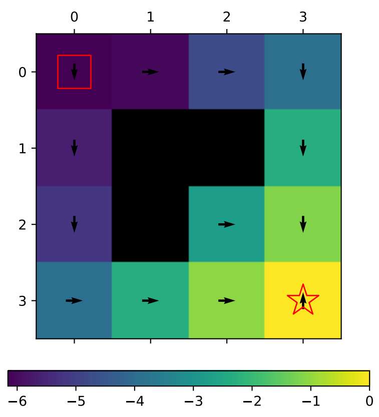
(b) $\epsilon = 0.1$, $\alpha = 0.3$



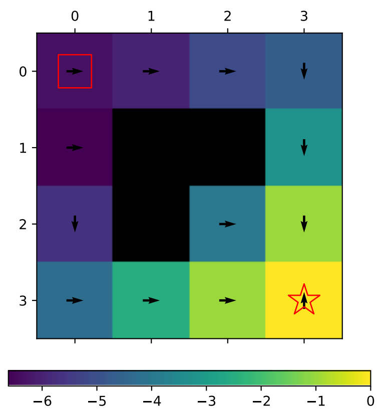
(c) $\epsilon = 0.1$, $\alpha = 0.5$



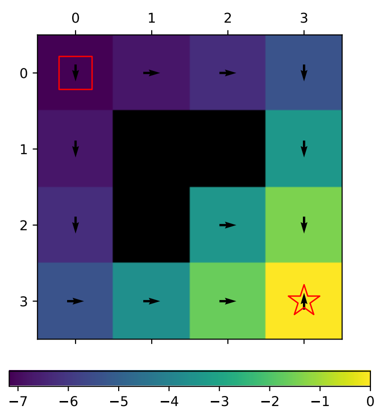
(d) $\epsilon = 0.3$, $\alpha = 0.1$



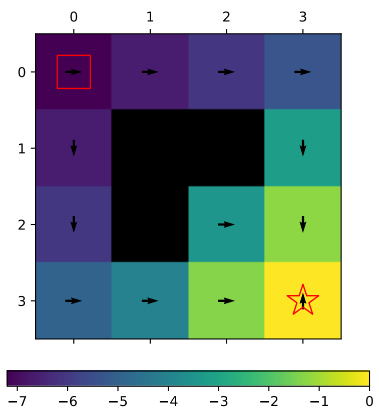
(e) $\epsilon = 0.3$, $\alpha = 0.3$



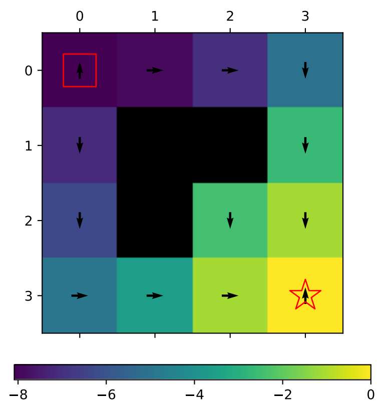
(f) $\epsilon = 0.3$, $\alpha = 0.5$



(g) $\epsilon = 0.5$, $\alpha = 0.1$



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

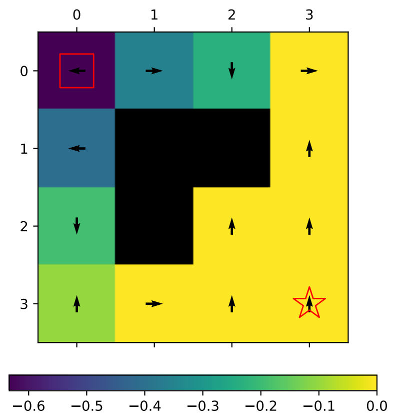


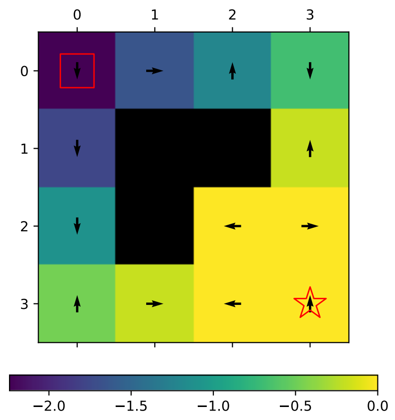
(i) $\epsilon = 0.5$, $\alpha = 0.5$

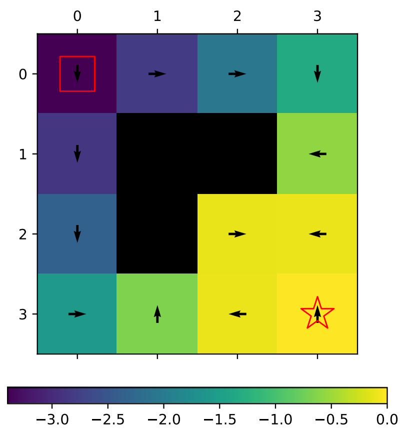
Figure 5: Different $\epsilon$ and $\alpha$ for SARSA on `small_world`

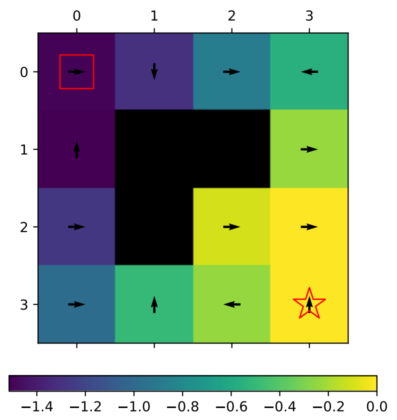
From Figure 5, it seems that the combination $\epsilon = 0.1$, $\alpha = 0.1$, converges to the optimal policy and gives similar estimation of the value function compared to the ground truth in 1 (although not 100% the same). We will stick to this setting for further SARSA experiments (unless specified).

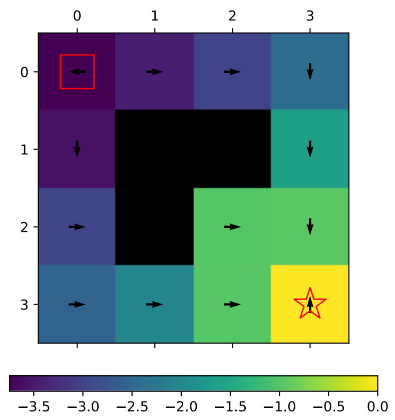
Now we also investigate maxit and num_episode. As the total number of states in `small_world` is small (only 16), maxit should not be too large. Actually, maxit=100 and num_episode=600 used above seems to be quite reasonable already, but to illustrate the effect of different maxit and num_episode, we try num_episode $\in \{10, 50, 100\}$ and maxit $\in \{10, 50, 100\}$ (with $\epsilon = 0.1$, $\alpha = 0.1$).

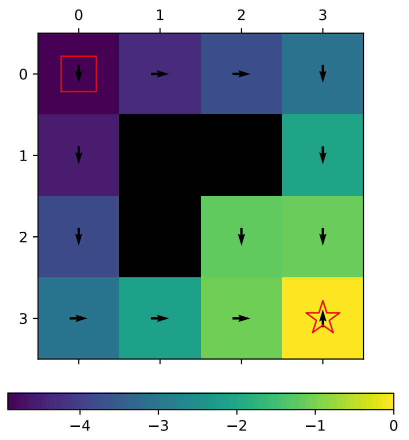

(a) maxit=10, num_episode=10

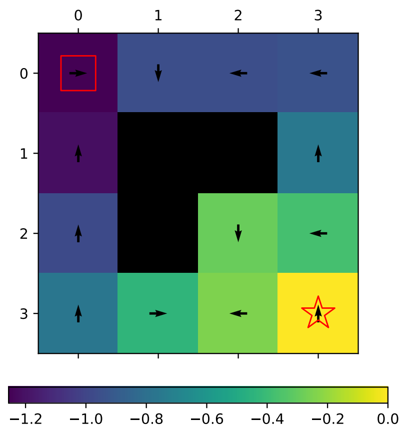

(b) maxit=10, num_episode=50

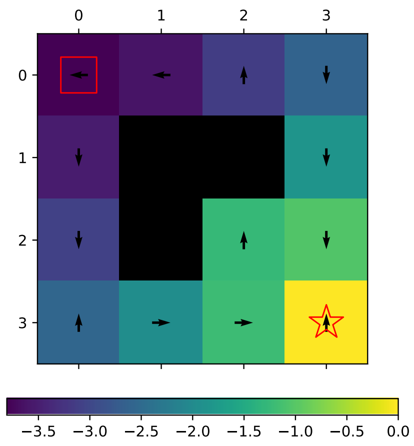

(c) maxit=10, num_episode=100


(d) maxit=50, num_episode=10


(e) maxit=50, num_episode=50


(f) maxit=50, num_episode=100


(g) maxit=100, num_episode=10


(h) maxit=100, num_episode=50

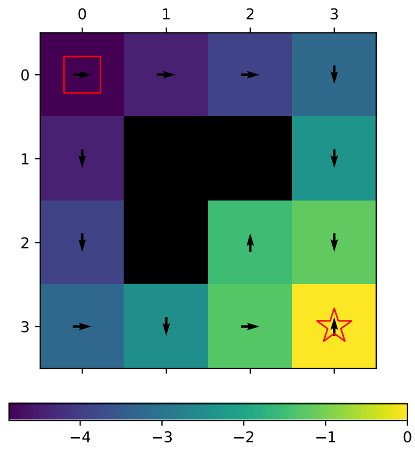
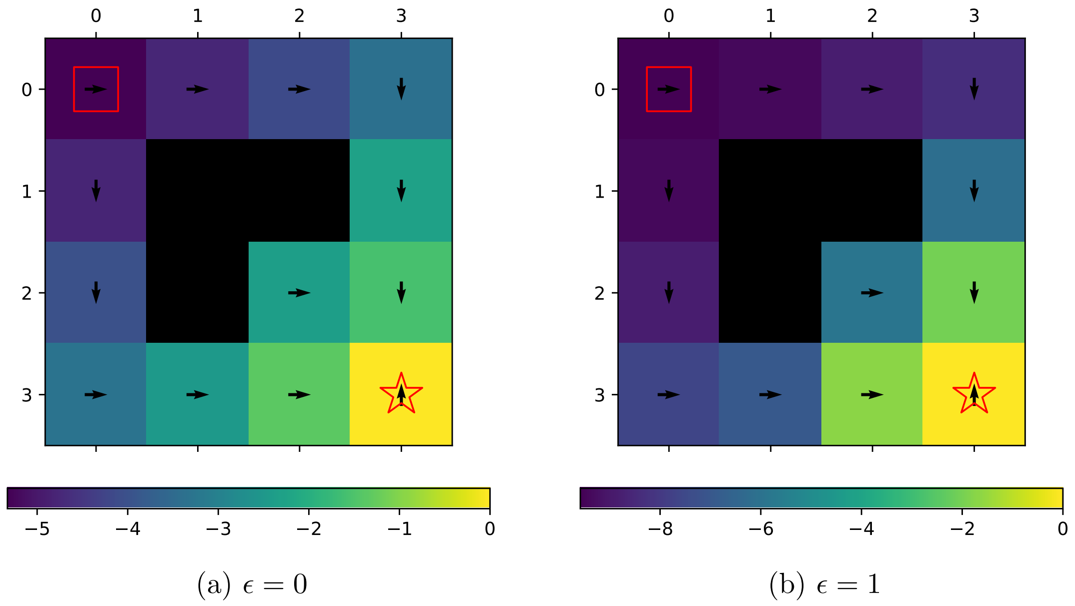

(i) maxit=100, num_episode=100

Figure 6: Different maxit and num_episode for SARSA on `small_world`

From Figure 6, for all these settings SARSA has not converged to the optimal policy as shown in Figure 1. Using too small maxit and num_episode would make SARSA struggle to find the optimal policy. Thus, we will stick to maxit=100 and num_episode=600 in the SARSA experiments below (unless specified).

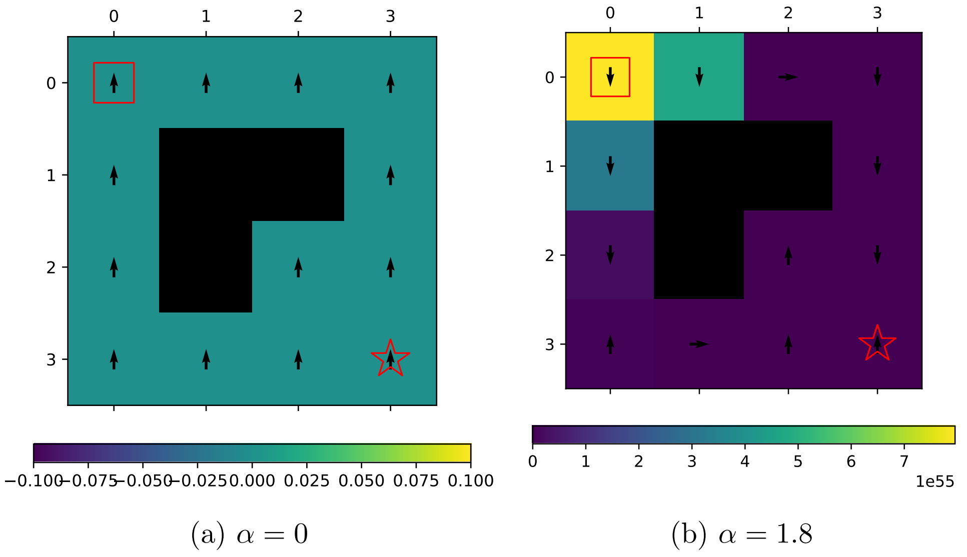
Now let's try with extreme values of $\epsilon$ and $\alpha$.



(a) $\epsilon = 0$ (b) $\epsilon = 1$

Figure 7: extrem $\epsilon$ for SARSA on `small_world` (other parameters are unchanged, maxit=100, num_episode=600, $\alpha = 0.1$)

From Figure 7, we can see that $\epsilon = 0$ and $\epsilon = 1$ has indeed makes the performance of SARSA worse (unable to converge optimal policy and bad estimate of value function).



(a) $\alpha = 0$ (b) $\alpha = 1.8$

Figure 8: extrem $\alpha$ for SARSA on `small_world` (other parameters are unchanged, maxit=100, num_episode=600, $\epsilon = 0.1$)

We can see from Figure 8 that for small learning rate $\alpha$ such as 0, the agent would not learn anything at all (as its not updating Q function at all) while for too large $\alpha$ such as 1.8, the estimate of the Q function would be pretty bad and hence leads to a suboptimal policy.

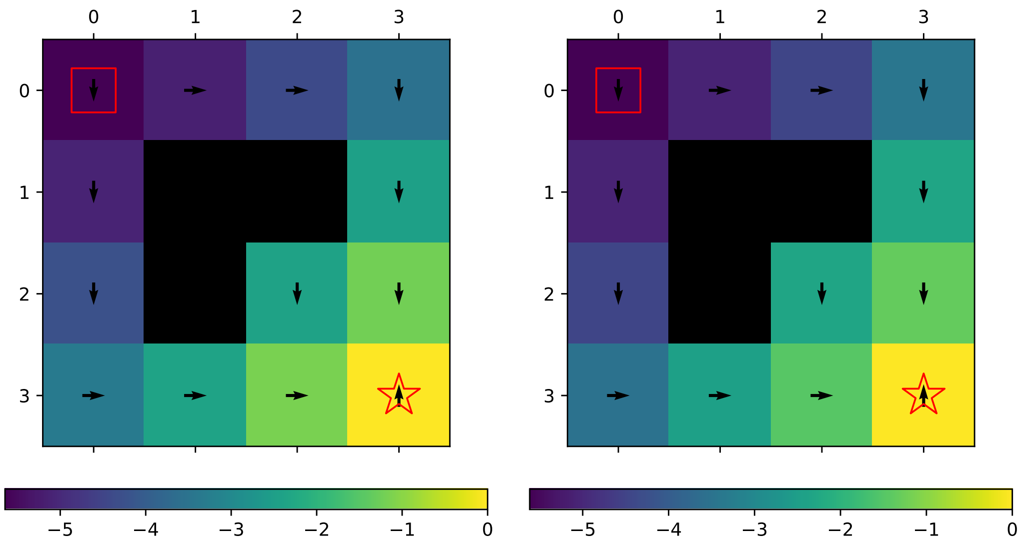
Overall, the learnt policy of SARSA, given suitable parameters, is the optimal policy although the value function not necessarily converge to the true value. Moreover, the variance of SARSA algorithm is large in the sense that it is stochastically sampling episodes (due to $\epsilon$) which might includes "bad" paths and this might slow down SARSA's convergence to optimal policy.

Lastly, we implemented Expected SARSA using the following code:

**Listing 3** The code for Expected SARSA implementation

```
def expected_sarsa(model: Model, maxit: int = 100, num_episode: int = 300,
                                      eps: float=0.1, alpha: float=0.3):
    V = np.zeros((model.num_states,))
    pi = np.zeros((model.num_states,))
    Q = np.zeros((model.num_states, len(Actions)))


    def eps_greedily(s, epsilon):
        unif = np.random.rand()

        if unif < epsilon:
            idx = np.random.randint(0, len(Actions))
            return Actions(idx)

        else:
            return Actions(np.argmax(Q[s]))

    def action_policy(s, epsilon):
        state_dist = [epsilon/len(Actions)]*len(Actions)
        max_Q = np.max(Q[s])
        count = np.sum(Q[s] == max_Q)

        if count > 1:
            indexes = [i for i in range(len(Actions)) if Q[s,i]== max_Q]
            action_idx = np.random.choice(indexes)
        else:
            action_idx = np.where(Q[s] == max_Q)[0].item()

        best_action = Actions(action_idx)
        state_dist[best_action]+= (1-epsilon)

        return state_dist

    for i in tqdm(range(num_episode)):
        s = model.start_state

        Q_old = np.copy(Q)
        for j in range(maxit):

            a = eps_greedily(s, eps)
            r = model.reward(s, a)
            possible_dict= model._possible_next_states_from_state_action(s,
                                                                       a)
            possible_s = list(possible_dict.keys())
            prob = list(possible_dict.values())
            s_next = np.random.choice(possible_s, p = prob)
            state_dist_next = action_policy(s_next, eps)
            E_a = sum([q * p_eps_greedy for q, p_eps_greedy in zip(Q[s_next]
                                                           , state_dist_next)])
            if s_next == model.goal_state:
                Q[s, a] = Q[s, a] + alpha*(r - Q[s, a])
            else:
                Q[s, a] = Q[s, a] + alpha*(r + model.gamma*E_a - Q[s, a])
            s = s_next
            if s == model.goal_state:
                break

        if np.all(abs(Q-Q_old)<=0.0001):
            print("breaking")
            break

    V = np.amax(Q, axis=1)
    pi = np.argmax(Q, axis=1)
    return V, pi
```

Here we run Expected SARSA on `small_world` with maxit=100, num_episode=600, $\epsilon = 0.1$ and $\alpha = 0.1$:



(a) SARSA (b) Expected SARSA

Figure 9: Comparison of Expected SARSA and SARSA on `small_world`

The performance of Expected SARSA and SARSA on `small_world` is similar and both of them seem to converge to the optimal policy under the same setting.

Theoretically speaking, the computational cost of Expected SARSA is higher as it calculate the expected Q values over all actions in the action space. Here, as the number of actions in `small_world` is small, the difference in computational cost is not obvious for Expected SARSA and SARSA. Also, it is worth noting that Expected SARSA has smaller variance and would generally performs better than SARSA as it averages over action spaces instead of randomly sample new actions when updating Q values.

## 3 Problem c)

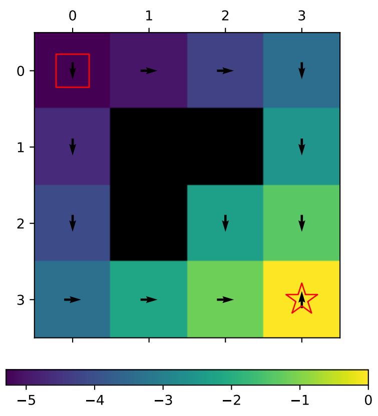
The implementation of Q-learning is as follows:

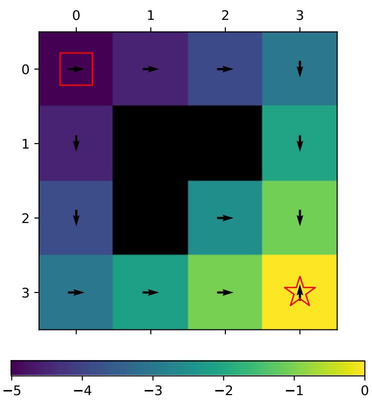
**Listing 4** The code for Q-learning implementation

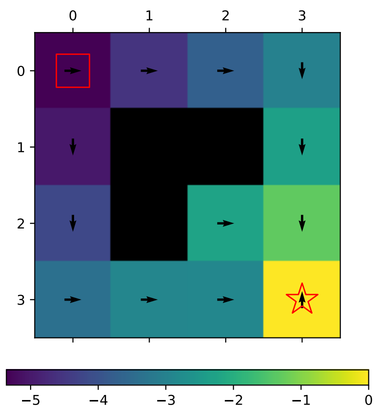
```
def q_learning(model: Model, maxit: int = 100, num_episode: int = 300, eps:
                                float=0.1, alpha: float=0.3):
    V = np.zeros((model.num_states,))
    pi = np.zeros((model.num_states,))
    Q = np.zeros((model.num_states, len(Actions)))


    def eps_greedily(s, epsilon):
        unif = np.random.rand()

        if unif < epsilon:
            idx = np.random.randint(0, len(Actions))
            return Actions(idx)

        else:
            return Actions(np.argmax(Q[s]))

    for i in tqdm(range(num_episode)):

        s = model.start_state
        Q_old = np.copy(Q)
        for j in range(maxit):
            a = eps_greedily(s, eps)
            r = model.reward(s, a)
            possible_dict= model._possible_next_states_from_state_action(s,
                                                        a)
            possible_s = list(possible_dict.keys())
            prob = list(possible_dict.values())
            s_next = np.random.choice(possible_s, p = prob)
            if s_next == model.goal_state:
                Q[s, a] = Q[s, a] + alpha*(r - Q[s, a])
            else:
                Q[s, a] = Q[s, a] + alpha*(r + model.gamma*np.max(Q[s_next])
                                                        - Q[s, a])
            s = s_next
            if s == model.goal_state:
                break

        if np.all(abs(Q-Q_old)<=0.0001):
            print("breaking")
            break

    V = np.amax(Q, axis=1)
    pi = np.argmax(Q, axis=1)
    return V, pi
```

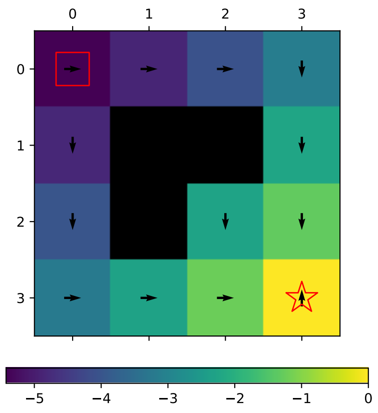
The parameters to be set are still $\epsilon$, $\alpha$, maxit and num_episode. Therefore we will set them using the same way as SARSA. Firstly, we will tune $\epsilon$, $\alpha$. Still, with the same reason as SARSA, we will try with $\epsilon \in \{0.1, 0.3, 0.5\}$ and $\alpha \in \{0.1, 0.3, 0.5\}$ with maxit=100 num_episode=600.

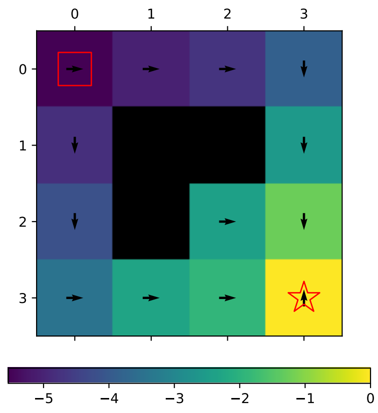
.

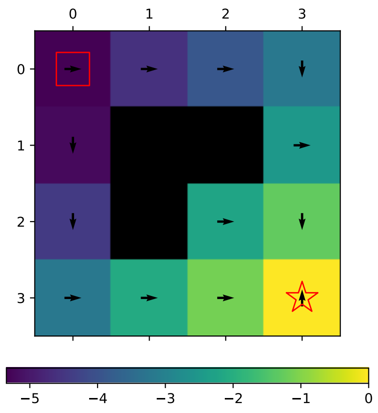

(a) $\epsilon = 0.1,\ \alpha = 0.1$

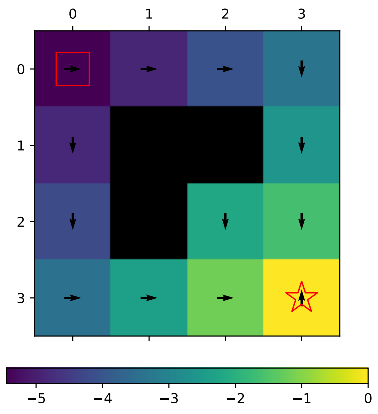

(b) $\epsilon = 0.1,\ \alpha = 0.3$

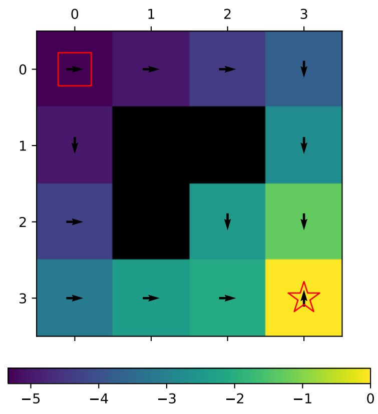

(c) $\epsilon = 0.1,\ \alpha = 0.5$


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


(e) $\epsilon = 0.3,\ \alpha = 0.3$


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


(g) $\epsilon = 0.5,\ \alpha = 0.1$


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

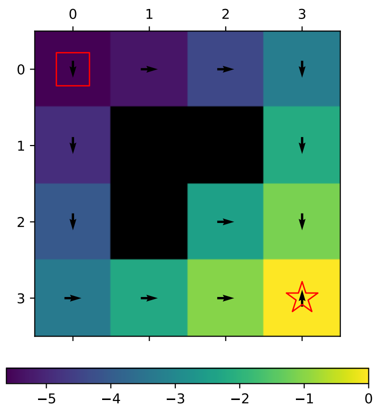
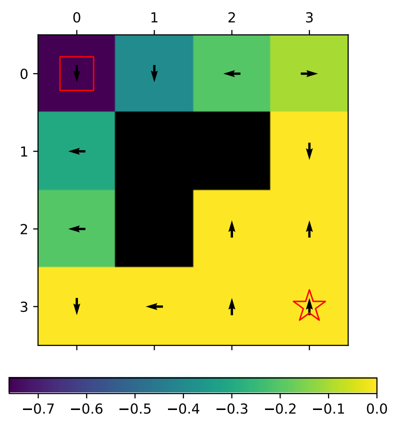
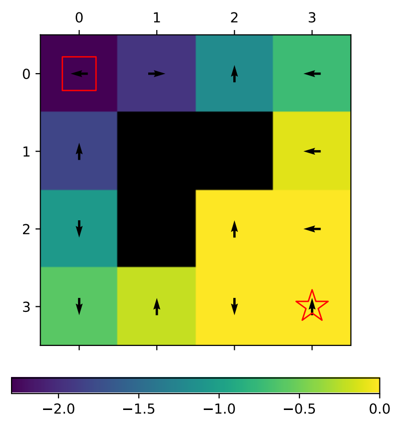

(i) $\epsilon = 0.5,\ \alpha = 0.5$

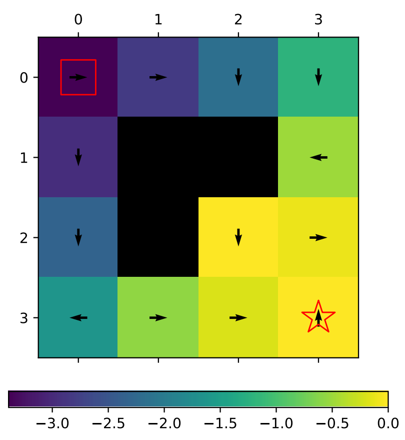
Figure 10: Different $\epsilon$ and $\alpha$ for Q-learning on `small_world`

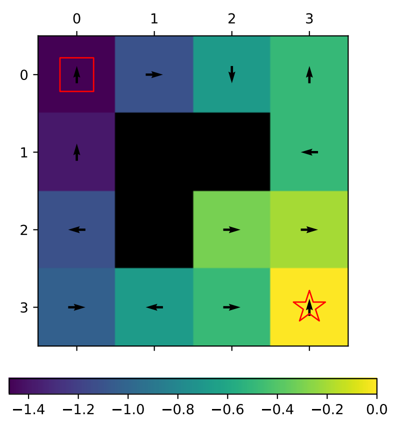
From Figure 10, it seems that $\epsilon = 0.1,\ \alpha = 0.1$ seems to be a good choice as it returns the optimal policy on `small_world` compared to Figure 1.

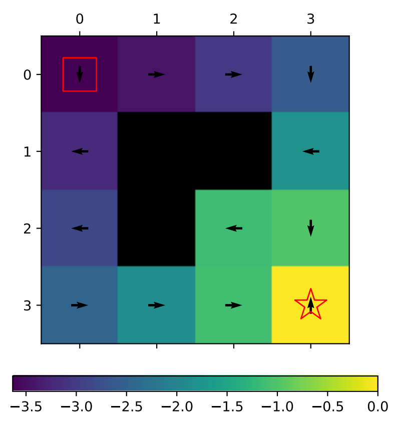
Also, although it seems that maxit=100 num_episode=600 works fine, we still try num_episode $\in \{10, 50, 100\}$ and maxit $\in \{10, 50, 100\}$ (with $\epsilon = 0.1,\ \alpha = 0.1$).

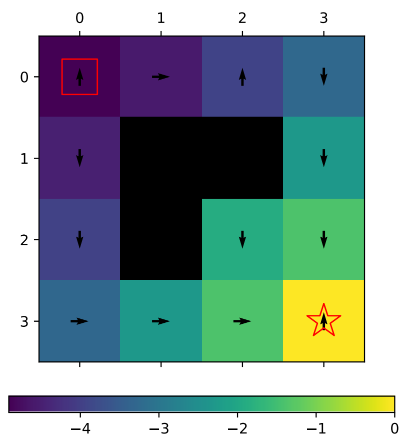

(a) maxit=10, num_episode=10

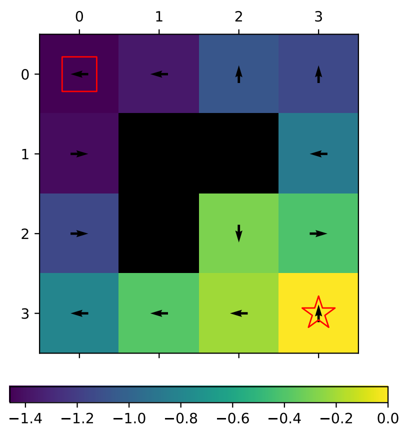

(b) maxit=10, num_episode=50

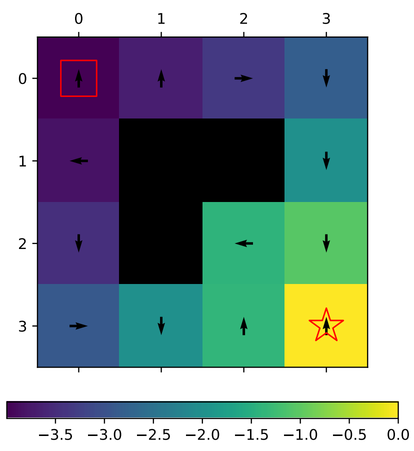

(c) maxit=10, num_episode=100


(d) maxit=50, num_episode=10


(e) maxit=50, num_episode=50


(f) maxit=50, num_episode=100


(g) maxit=100, num_episode=10


(h) maxit=100, num_episode=50

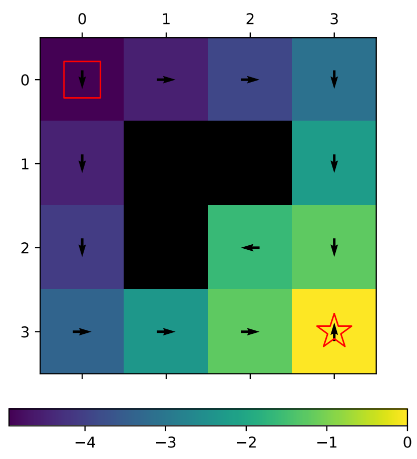

(i) maxit=100, num_episode=100

Figure 11: Different maxit and num_episode for Q-learning on `small_world`

From Figure 11, we can see that using small maxit and num_episode does make the Q-learning algorithm struggle to find the best policy, so we stick to maxit=100 and num_episode=600 now.

The performance of SARSA and Q-learning is quite similar on `small_world` with both algorithms converging to the optimal policy as shown in Figure 1 and they are all very efficient for the simple `small_world` context.

In theory, Q-learning would take longer than SARSA in each update step as Q-learning needs to find the maximum Q value over the whole action space for a given sampled state while SARSA needs only to sample a state and an action and uses the Q value at the updated state and action for Q value update. However, when it comes to the actual implementation, Q-learning may converge faster than SARSA as it explores less than SARSA (Q-learning is an off-policy algorithm that simply updates Q value based on the maximum current Q-values over all actions regardless of the current policy and Q-learning directly learns the optimal policy while SARSA tends to be risk averse and learn safer sub-optimal policy, this will be explored more in Problem d), thus each episode of Q-learning might be smaller. Which one of Q-learning and SARSA being more efficient depends on tasks.

To investigate the performance of these two algorithms on more complex environments, we also apply SARSA and Q-learning on `grid_world` (with the default parameters set as mentioned above) and here SARSA actually takes less time to run.

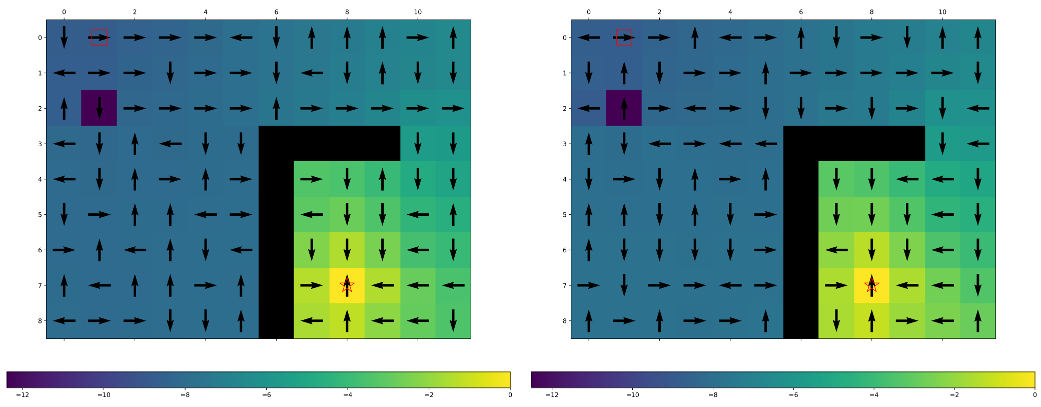

(a) Q-learning (b) SARSA

Figure 12: Comparison of Q-learning and SARSA on `grid_world`

From Figure 12, we can see that the policies learnt by both Q-learning and SARSA are similar themselves but are quite different from the ones learnt by VI and PI in Figure 1. The reason might be that both Q-learning and SARSA are not directly optimising the policies or values of each state, they are learning by optimising through the generated episodes (with $\epsilon$-greedy exploration).

## 4 Problem d)

In this part, we first modify the SARSA and Q-learning algorithm to report cumulative reward per episode. Then, we implement these two algorithms on `cliff_world` with maxit=100 and num_episode=2000 (we use more episodes here just to better illustrate the pattern of cumulative rewards). For $\epsilon$ and $\alpha$, we first try $\epsilon = 0.1$ and $\alpha = 0.1$ for both algorithms and then try further $\epsilon = 0.01$ and $\alpha = 0.1$ for both algorithms. Note that we need to apply mean filter (or uniform filter) to the raw cumulative reward to reduce noise within the raw cumulative reward.

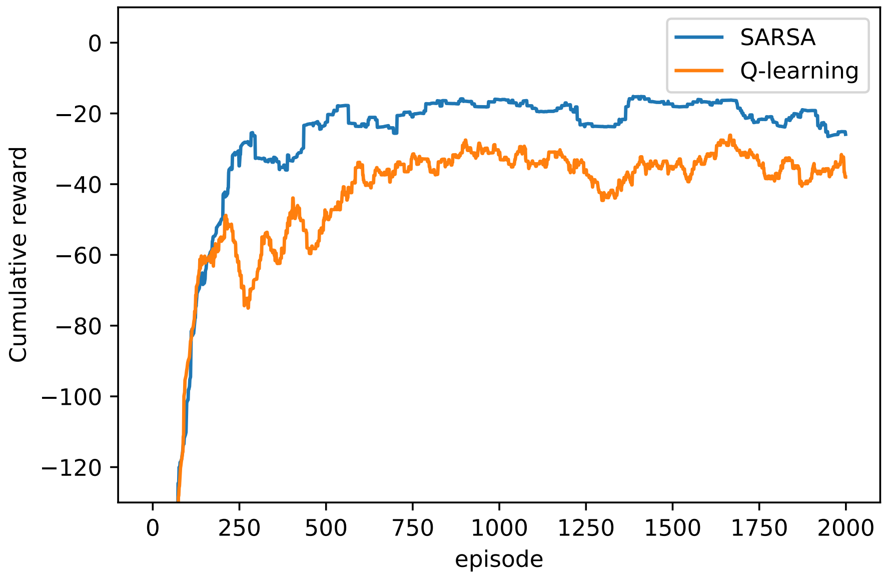


Figure 13: Cumulative reward per episode for SARSA and Q-learning with $\epsilon = 0.1$ and $\alpha = 0.1$

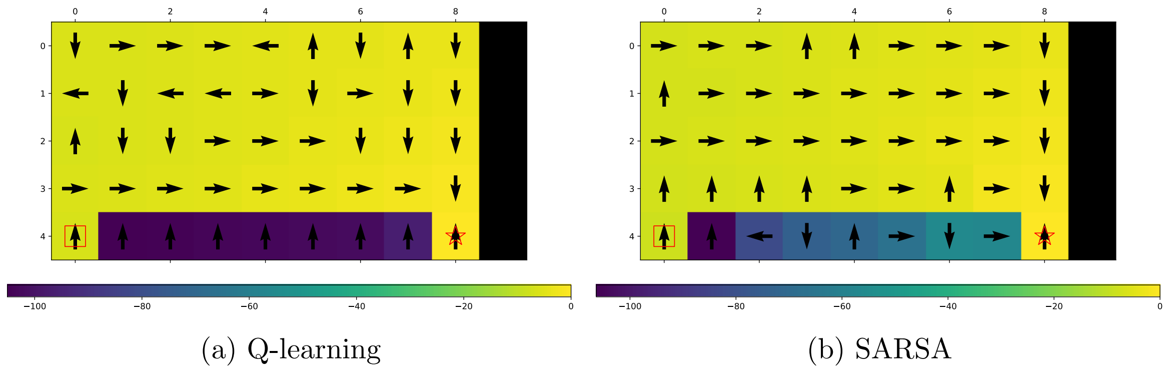


Figure 14: Learnt policy for Q-learning and SARSA on `cliff_world` with $\epsilon = 0.1$ and $\alpha = 0.1$

In each iteration, Q-learning updates Q-values based solely on the current best Q-values regardless of the current policy (Q-learning is an off-policy method) and hence Q-learning directly learns the optimal policy, which is just the path near the cliff edge as shown in Figure 14a and Q-learning has indeed learnt this optimal policy directly as shown in Figure 14a while for SARSA, as SARSA updates the Q-value considering all possible actions over its current state (SARSA is an on-policy algorithm), it tends to explore more and would be highly risk-averse and thus it would learn a safer, sub-optimal path that is far from the cliff, as shown in Figure 14b

As a result, with $\epsilon = 0.1$, Q-learning is much more likely to fall off the cliff and get lower cumulative rewards than SARSA (SARSA is risk-averse and would avoid getting close to cliff), which is exactly what we have observed in Figure 13.

Let's now try implementing both algorithms with $\epsilon = 0.01$ and $\alpha = 0.1$.

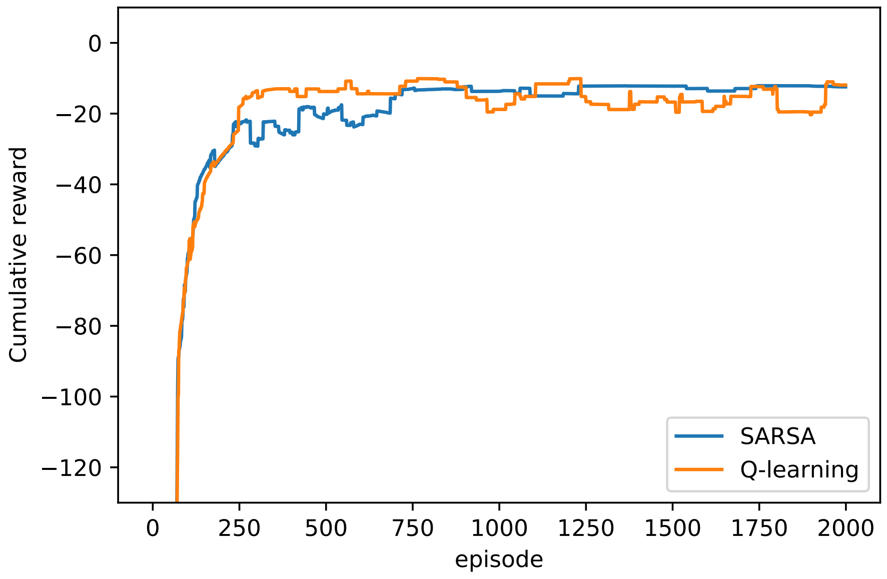
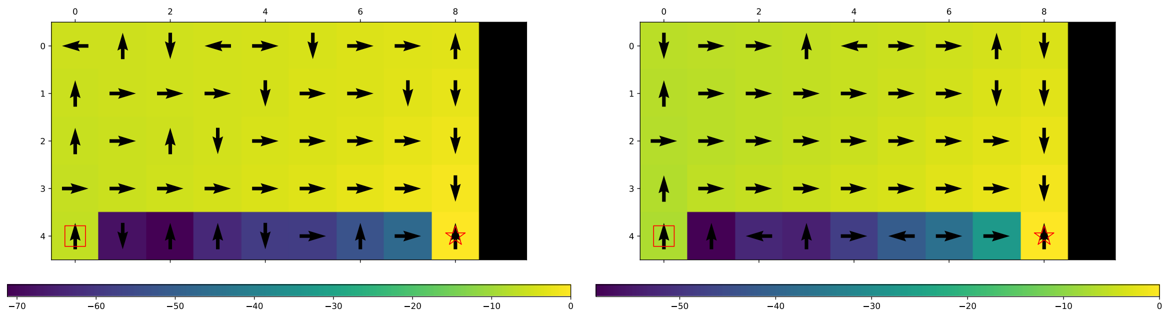


Figure 15: Cumulative reward per episode for SARSA and Q-learning with $\epsilon = 0.01$ and $\alpha = 0.1$



(a) Q-learning

(b) SARSA

Figure 16: Learnt policy for Q-learning and SARSA on `cliff_world` with $\epsilon = 0.01$ and $\alpha = 0.1$

If we change $\epsilon$ to 0.01, then SARSA would not be exploring much and hence it would more likely to directly learn the optimal policy now just as Q-learning (Q-learning does not change its learnt policy as it is an off-policy algorithm), which agrees with the plotted policy in Figure 16.

Moreover, From Figure 15, we can see that the cumulative rewards of SARSA and Q-learning have converged to approximately the same level for $\epsilon = 0.01$, which means that SARSA has indeed learnt the optimal policy as Q-learning does. Moreover, we can see that the converged cumulative reward of Q-learning is also larger than the $\epsilon = 0.1$ situation, which is also reasonable as now $\epsilon$ is more smaller, discouraging exploration and hence Q-learning would be less likely to fall off the cliff.

## 5 Problem e)

Function approximation is useful when we have a very large state space or a continuous state space. Here, the modified `small_world` environment now contains $16 \times 8 = 128$ possible states (whether a particular cell contains barrier or not can be considered as a binary variable), which may heavily increase the computational cost of Reinforcement algorithms such as VI where we need to estimate the value of each cell iteratively. Thus, function approximation might help a lot in this case.

Consider such a linear approximation $\boldsymbol{\phi}(\mathbf{s})$, where $\mathbf{s} = [1, b_1, b_2, b_3, s]^T$, $b_i = 1$ if the $i$th barrier cell contains barrier and 0 otherwise and $s$ denotes the state. If we let $\boldsymbol{\theta} = [\theta_0, \theta_1, \theta_2, \theta_3, \theta_4]^T$ be the weight vector, then the full linear value function approximation would be the following:

$$V(\mathbf{s}, \boldsymbol{\theta}) = \theta_0 + \theta_1 b_1 + \theta_2 b_2 + \theta_3 b_3 + \theta_4 s$$

The above linear value function approximation is certainly not suffice in this case since it cannot take into account any interactions between features. For example, the value function of terminal state should not be affected by the number of barriers present (it should always be 0) while the start state (or states in the upper left corner of `small_world`) would be definitely affected by the presence of barriers (as it blocks several possible paths). However, if we simply use the linear value function approximation as above, the value functions of terminal state and start state would be affected by exactly the same amount given the presence of barriers, compared to the case with no barriers, which is not reasonable. Also, the value function of terminal state should not be affected by the presence of barriers (it should always be 0) and if we use the linear value function approximation as above, the presence of barriers (any number of barriers) would change the value of the terminal state, which is not sensible.