# Coursework report

## MLMI7 - Reinforcement Learning

March 6, 2022

**Candidate no.:** J902C

**Word count:** 1954

UNIVERSITY OF CAMBRIDGE
Department of Engineering

[0]Excluding appendix, tables, footnotes, images, code and titles



Trumpington Street
Cambridge CB2 1PZ
Tlf.: +44 01223 332600
http://www.eng.cam.ac.uk/

# Table of contents

# 1. Question a)

Value Iteration (VI) algorithm is implemented following the pseudocode given in [1], shown in figure 1.1. The Python code can be found in A.1, highlighting the code 1.1.

**Value Iteration, for estimating $\pi \approx \pi_*$**

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

Figure 1.1: Value Iteration pseudocode described in [1]

```
    for i in tqdm(range(n_episodes)):
        V_new = np.zeros((model.num_states,))
        for s in model.states:
            action_values = [compute_value(s, a, model.reward) \
                             for a in Actions]
            V_new[s] = np.max(action_values)
```

Listing 1.1: Value function update in VI algorithm

We can verify the correct implementation of VI comparing its results with those using Policy Iteration (PI). In figure 1.2 the policies of both algorithms can be compared on the `small_world` model. Similarly, figures 1.3 and 1.4 show the final policy after using Policy Iteration and Value Iteration on `cliff_world` and `grid_world` respectively.

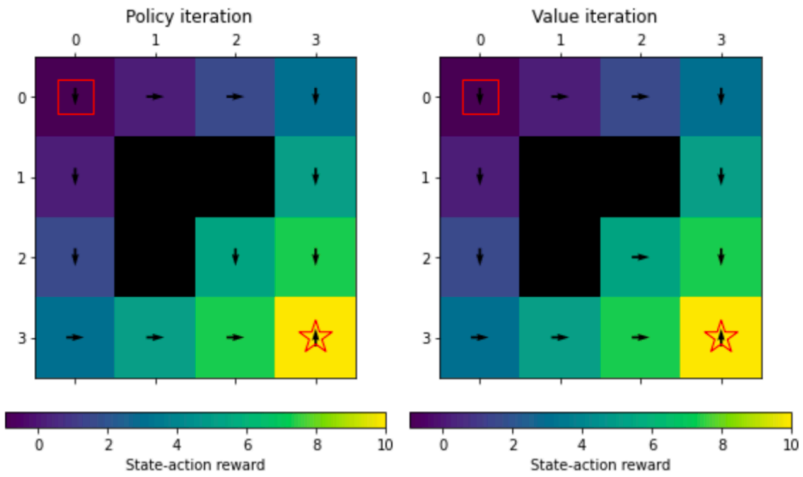



Figure 1.2: Policy Iteration (left) and Value Iteration (right) on small world. Check that both methods perform equally well, finding optimal solutions for the environment.

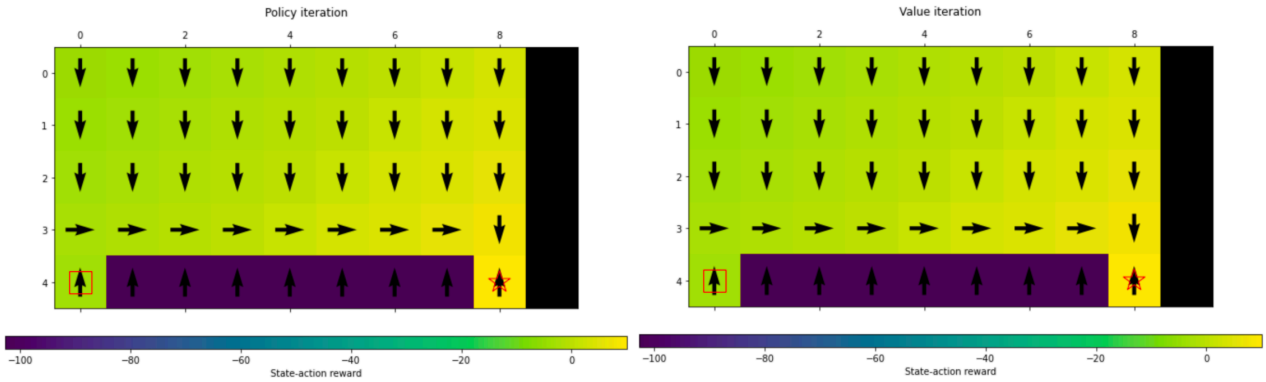



Figure 1.3: Policy Iteration (left) and Value Iteration (right) on cliff world. As in `small_world`, both algorithms produce the exact same resulting policy for each state.

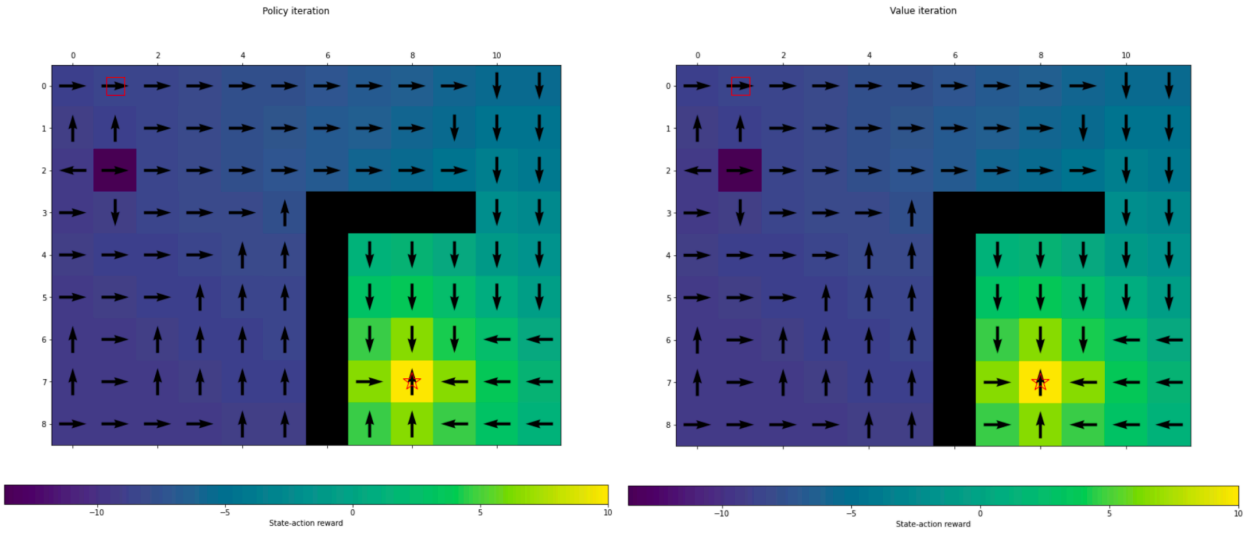



Figure 1.4: Policy Iteration (left) and Value Iteration (right) on grid world. Again, the optimised policies are identical. Assuming that the goal of all models is to find a path from the initial state to the final state while maximising the reward, we can check that VI and PI find the optimal solution due to the fact that the resulted policies describe the shortest path from initial state to the final one.

The convergence of the algorithms can be analysed comparing the optimised object in each algorithm. As the names imply, PI improves the policy and VI optimises the value function in each episode. The convergence can be studied by analysing the differences between the policies (in PI) of two consecutive episodes and between the value function vectors in the VI case. Since in this project the policy is discrete (4 different actions), the difference between two policies is the number of different coordinates between the two policy vectors. On the other hand, the value function $V$ can be represented as a real vector $|S|$ dimensional, where $|S|$ is the number of states in the model. So the difference between two consecutive value functions is the MSE of the vectors.

Figure 1.5 describes the convergence of PI and VI on `small_world` model.

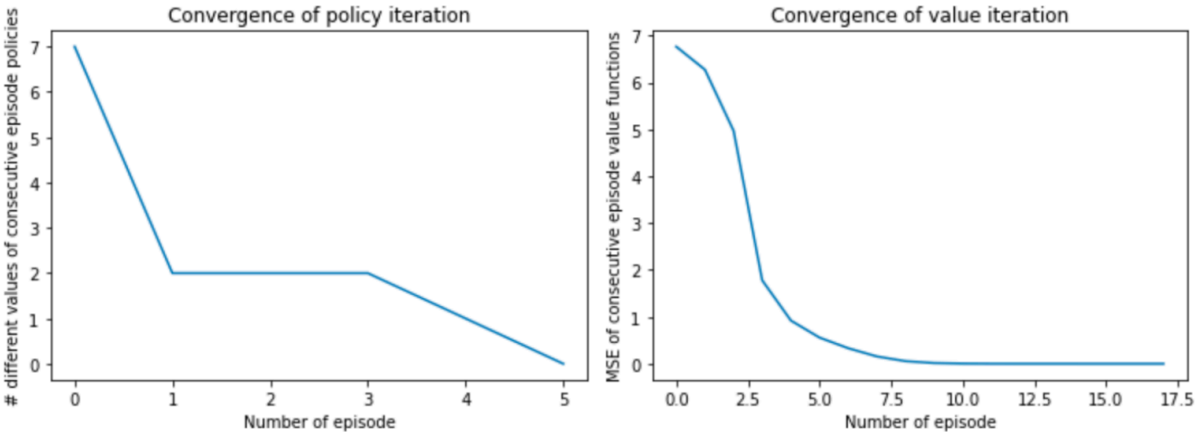



Figure 1.5: Convergence of PI (left) and VI (right) on small world. As the number of episodes increases, the difference between policies and value functions of two consecutive episodes decreases.

In other model, like `cliff_world`, the number of iterations needed to converge might vary. In figure 1.6 we can see how PI changes its policy a lot in the first episodes and then it converges to the final solution. This phenomenon can be observed in a more complex world, like `grid_world` that is represented in figure 1.7. If the tolerance is set to 0, the algorithm might need a really log time to finish, as we can see in the middle picture. If the tolerance is set to a slightly higher value, like 0.01 (right picture), the algorithm can finish sooner and the found solution is optimal as well.

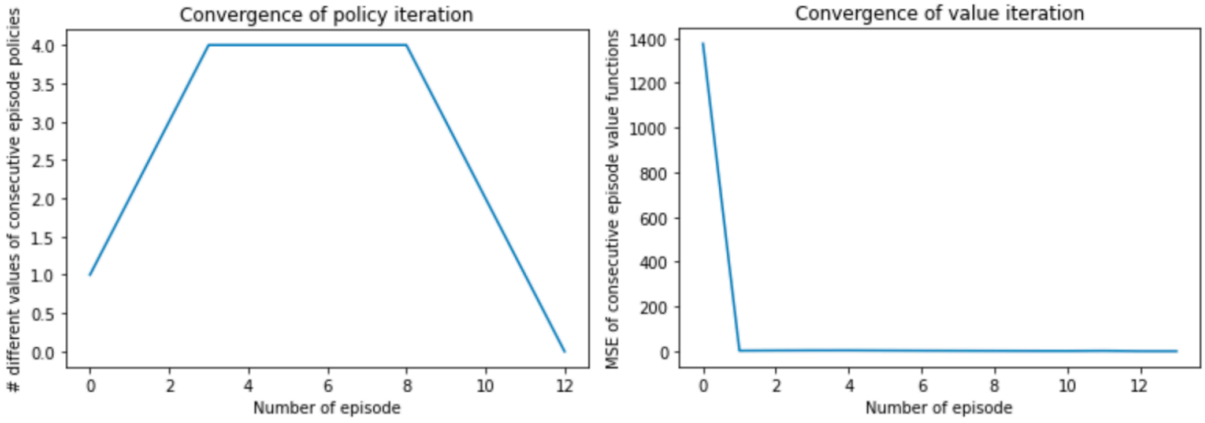



Figure 1.6: Convergence of PI (left) and VI (right) on cliff world. VI seems to converge quite fast in this model, even thought the stopping condition was set to 0, i.e., the MSE between value functions of two consecutive episodes is 0. This is not the typical procedure, since the agent might have already converged to an optimal solution but the value functions might still be changing a bit.

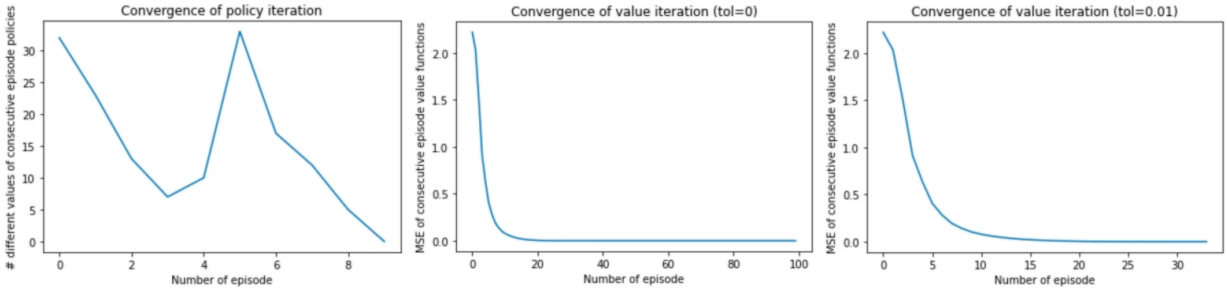



Figure 1.7: Convergence of PI (left) and VI (mid and right) on grid world

To sum up, a good stopping criteria for PI can be that two consecutive policies do not change between episodes (above all if they are discrete), and a MSE between consecutive value functions less than a given tolerance is a good condition to stop VI algorithm.

For this project, synchronous VI has been implemented since it stores two copies of value function (check Python code in A.1) to analyse the convergence. However, this is not the best implementation because asynchronous VI only stores one copy of value function since the update step is done in-place. A major drawback of synchronous VI is that it involves operations over the entire state set, that is, it requires sweeps of the state set. If the state set is very large, then even a single sweep can be prohibitively expensive. Therefore, asynchronous VI uses less memory and does not require to copy the updated value function at the end of each episode, so it would be preferable. Of course, avoiding sweeps does not necessarily mean that we can get away with less computation. It just means that an algorithm does not need to get locked into any hopelessly long sweep before it can make progress improving a policy.

Finally, the computational efficiency of PI and VI can be compared. In each iteration, the PI function evaluates the policy with a cost of $O(|S|^3)$ and then improves it in $O(|A||S|^2)$ operations, where $|A|$ is the number of actions. The VI function covers these two phases by taking a maximum over the utility function for all possible actions, with a computational cost of $O(|A||S|^2)$. The VI algorithm is straightforward. It combines two phases of the PI into a single update operation. However, the VI function runs through all possible actions at once to find the maximum action value. Subsequently, the VI algorithm is computationally heavier.

Both algorithms are guaranteed to converge to an optimal policy in the end. Yet, the PI algorithm converges within fewer iterations. As a result, the PI is reported to conclude faster than the VI algorithm.

# 2. Question b)

SARSA algorithm is implemented using the pseudocode available in [1], illustrated in figure 2.1. The Python code is included in A.2, highlighting the main code 2.1. It's tested on `small_world` and we discuss now how to set properly its hyperparameters.

**Sarsa (on-policy TD control) for estimating $Q \approx q_*$**

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize $S$
  Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
  Loop for each step of episode:
    Take action $A$, observe $R$, $S'$
    Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until $S$ is terminal

Figure 2.1: SARSA pseudocode described in [1]

```
    r = model.reward(s, a)
    acts_probs_dict = possible_next_states_from_state_action(s, a)
    new_s = choice(acts_probs_dict.keys(), p=acts_probs_dict.values())
    new_a = choose_eps_greedily(new_s, epsilon)
    Q[s][a] += alpha*(r + gamma*Q[new_s][new_a] - Q[s][a])
```

Listing 2.1: Q update in SARSA algorithm

First, we investigate the exploration rate $\epsilon$. Figure 2.2 shows the effect of varying $\epsilon$, plotting the number of iterations needed to converge per episode, i.e., in the x-axis it is plotted the cumulative number of iterations.

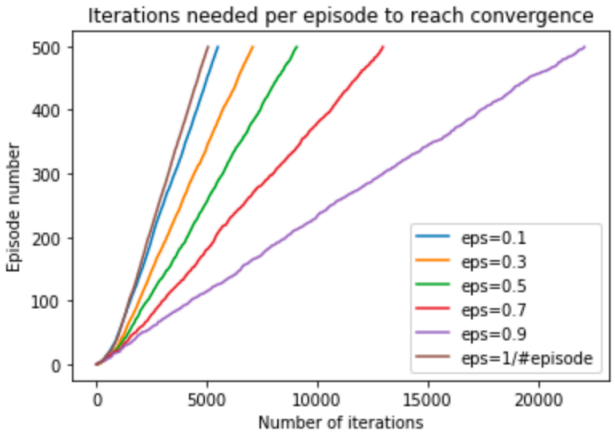



Figure 2.2: Selection of exploration rate $\epsilon$. Smaller values of $\epsilon$ allows the algorithm to converge faster, since the $\epsilon$-greedy policy is exploiting the best next action in the majority of times, but some exploration is allowed to find potential better paths.

The best value for the exploration rate is $\epsilon = 1/\text{episode\_number}$, so SARSA converges faster (fewer number of iterations) using a decaying exploration rate $\epsilon$. Similarly, the learning rate $\alpha$ can be investigated. Figure 2.3 illustrates the convergence velocity of SARSA using several learning rates.

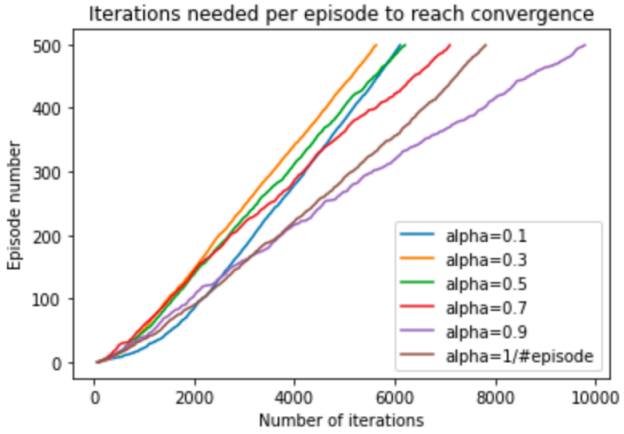



Figure 2.3: Selection of learning rate $\alpha$. Again, smaller values of $\alpha$ performs the best, needing fewer iterations to converge. The best value is $\alpha = 0.3$. For the future experiments, SARSA algorithm is run using a decaying exploration rate $\epsilon$ and $\alpha = 0.3$.

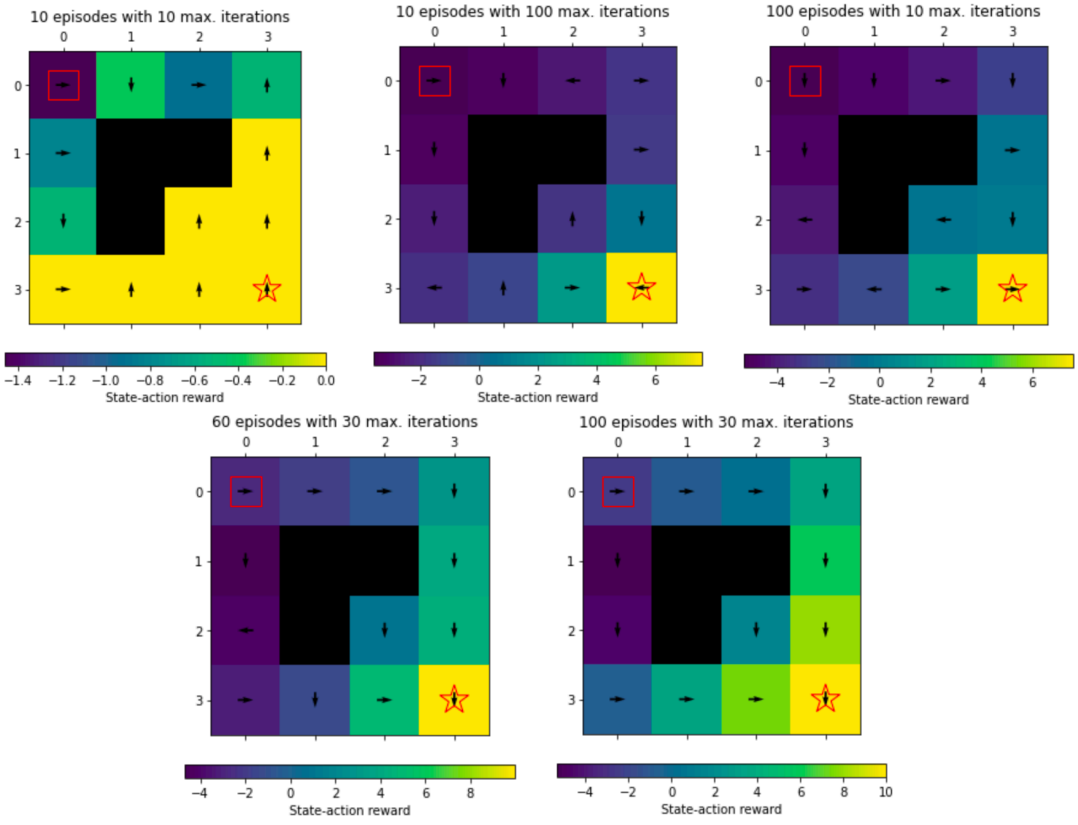



Figure 2.4: Performance (policies) of SARSA for different episodes and iterations

Figure 2.4 collects the results of several experiments using different episodes and iterations. Using just 10 episodes and 10 iterations (as maximum) per episode end up being insufficient to get a solution. Increasing the number of iterations but not the episodes would improve the performance, but it'd be quite poor too. Similarly, if we only increase the number of episodes to 100 but we keep the number of maximum iterations to a small value, like 10, SARSA won't be capable to converge either. If we increase both hyperparameters but by just a small amount; for example, 60 episodes and 30 iterations, we can find a solution but other paths might be still unexplored. Finally, running SARSA using a decent number of episodes and iterations (100 episodes and 30 max. iterations) results in finding both paths for `small_world` model.

It is important to underline that this experiments are run using suitable values for $\alpha$ and $\epsilon$. Figure 2.5 shows what would happen if edge values of $\epsilon$ were used.

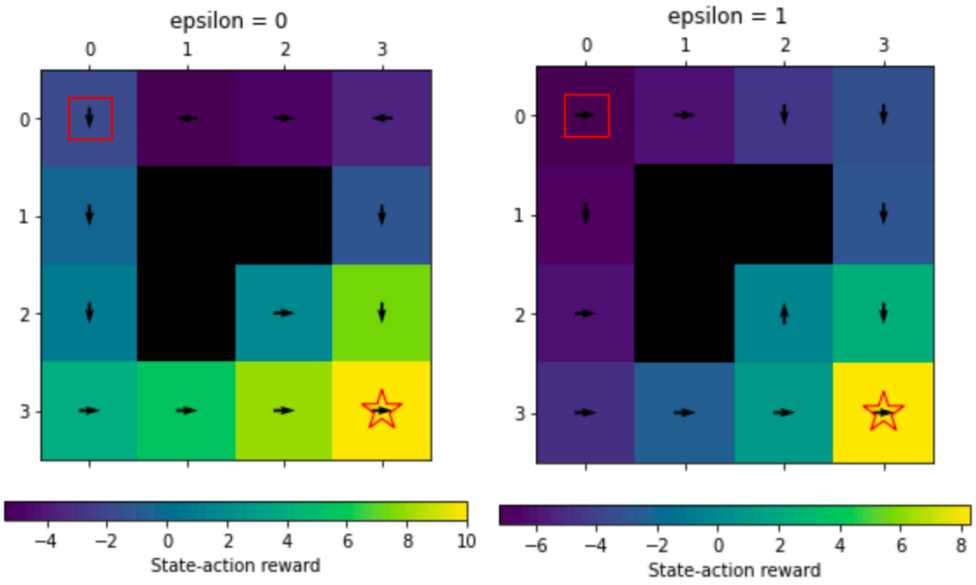



Figure 2.5: Performance of SARSA for small (left) and large (right) values of $\epsilon$. If the exploration rate $\epsilon$ is set to a really small value close to 0, SARSA will fail to find new possible solutions, even though the early rewards are bad and it is a simple model. If $\epsilon$ is set to a large value close to 1, the exploitation is hardly ever occurring and the algorithm looks for a solution basically randomly and it is unable to converge.

The learning rate is even a more sensible hyperparameter. If $\alpha$ is set to a small value, the agent won't learn at all. In contrast, if it is set to a really large value, the $Q$ function that SARSA is trying to optimise will end up overflowing, as illustrated in figure 2.6.

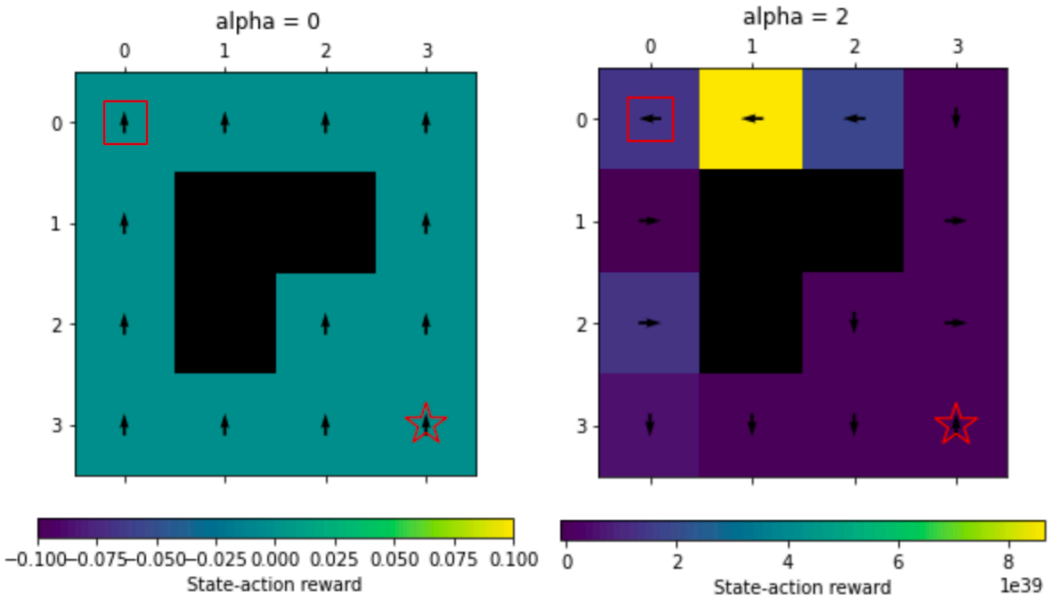



Figure 2.6: Performance of SARSA for small (left) and large (right) values of $\alpha$. For $\alpha = 0$ SARSA couldn't learn any reasonable policy, and for $\alpha = 2$, a solution couldn't be found because all the rewards are huge.

Finally, Expected SARSA is implemented using the pseudocode in [1], and the Python code is in the appendix A.4, highlighting here 2.2.

```
    a = choose_eps_greedily(s, epsilon)
    acts_probs_dict = possible_next_states_from_state_action(s, a)
    new_s = choice(acts_probs_dict.keys(), p=acts_probs_dict.values())
    r = model.reward(s, a)
    # calculate expected value of the action-value function
    next_state_probs = action_probs(new_s)
    expected_q = sum([a*b for a, b in zip(next_state_probs, Q[new_s])])
    Q[s][a] += alpha*(r + gamma*expected_q - Q[s][a])
```

Listing 2.2: Q update in Expected SARSA algorithm

Figure 2.7 shows the policy of Expected SARSA on `small_world` and `grid_world` after a decent amount of episodes and iterations, using tuned $\alpha = 0.3$ and $\epsilon = 0.1$.

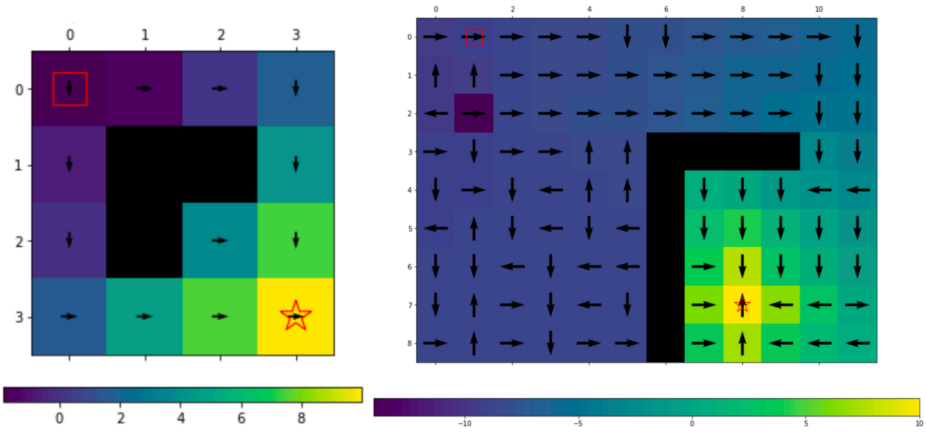
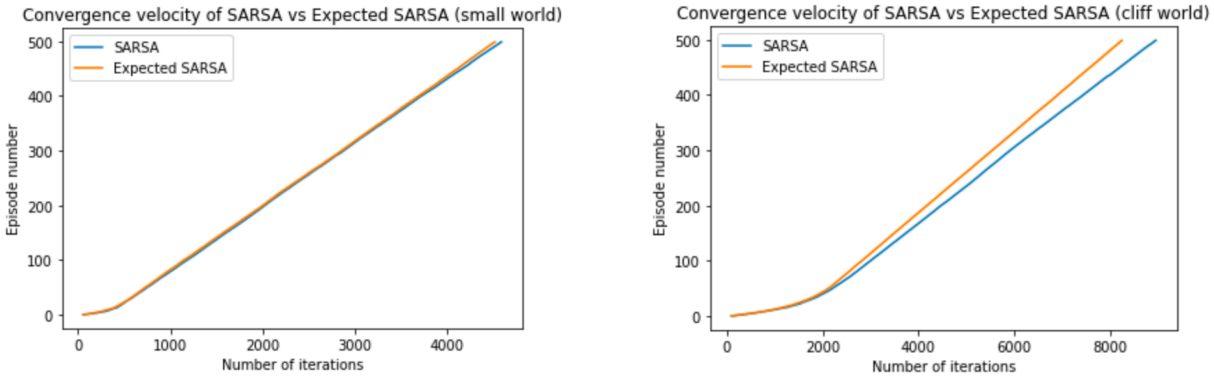


Figure 2.7: Performance of Expected SARSA on small world and grid world. The obtained policy on `small_world` is really similar as the one using SARSA, PI and VI because this is a really simple model. On `grid_world`, the resulted policy is similar as SARSA but not as PI or VI, since SARSA and Expected SARSA don't optimise every state, they just optimise greedily the optimal path, with a small chance of exploration.

Focusing on efficiency, figure 2.8 illustrates the empirical speed of convergence of SARSA and Expected SARSA on `small_world` and `cliff_world`. It heavily depends upon the parameter choice, but Expected SARSA is expected to converge faster than SARSA, above all in models where state transitions are all deterministic and all randomness comes from the policy, because Expected Sarsa can safely set $\alpha \approx 1$ without suffering any degradation of asymptotic performance, whereas SARSA can only perform well in the long run at a small value of $\alpha$ at which short-term performance is poor.




(a) Small world

(b) Cliff world

Figure 2.8: Velocity of convergence between SARSA and Expected SARSA. Since `small_world` is a really simple world, both algorithms converge at the same time. However, on `cliff_world` that it's a more complex model, it's clear that Expected SARSA converges faster (uses fewer iterations in the last episodes). This is due to the fact that while calculating the TD error, Expected SARSA weights the $Q$-value of each state-action pair with the probability of that action occurring.

# 3. Question c)

In this section Q Learning algorithm is investigated. Again, it has been implemented using the pseudocode in [1], and the Python code can be found in appendix A.3, highlighting here 3.1. Analogously to section 2, we study how to properly set the hyperparameters of Q Learning.

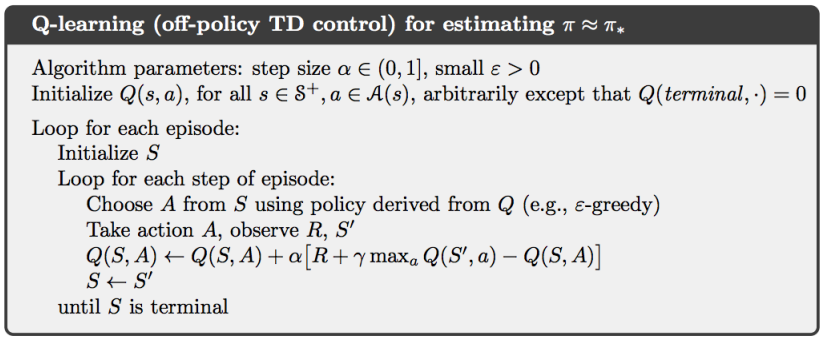



Figure 3.1: Q Learning pseudocode described in [1]

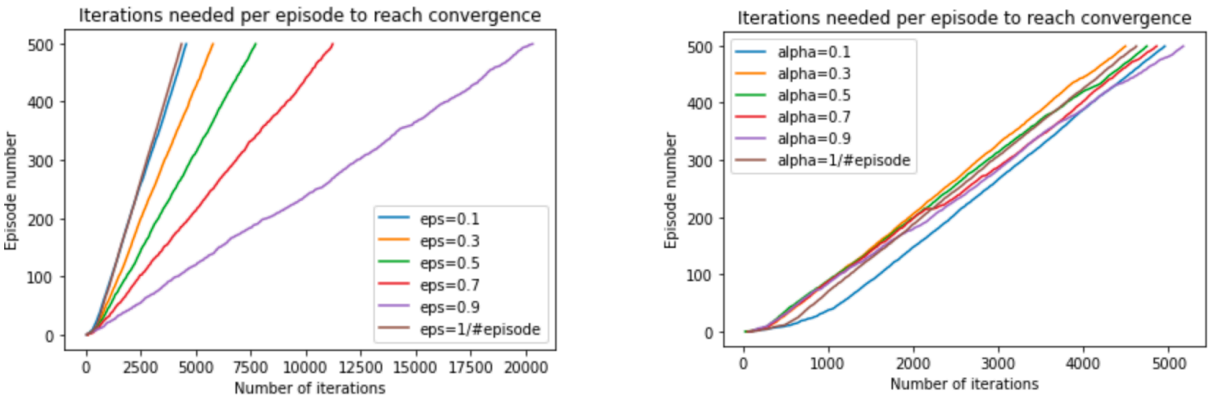
```
    a = choose_eps_greedily(s, epsilon)
    acts_probs_dict = possible_next_states_from_state_action(s, a)
    new_s = choice(acts_probs_dict.keys(), p=acts_probs_dict.values())
    r = model.reward(s, a)
    Q[s][a] += alpha*(r + gamma*max(Q[new_s]) - Q[s][a])
```

Listing 3.1: Q update in Q Learning

Figure 3.2 includes the results of running Q Learning on `small_world` with different exploration rates $\epsilon$ and learning rates $\alpha$. Smaller values of $\epsilon$ usually converge faster than larger values, since the algorithm is often taking the best next value instead of acting randomly (exploration). However, the best exploration rate is when $\epsilon = 1/\text{episode\_number}$ (figure 3.2a) because it allows exploring frequently the environment in the early stages of the algorithm and, once it has found a potential good path, the exploration is reduced to exploit the agent's knowledge more often and find a solution.

Additionally, after seeing figure 3.2b we can conclude that values of $\alpha \in [0.2, 0.5]$ perform well. A decaying learning rate performs also decently and could be used in different problems.

In future experiments, a decaying exploration rate $\epsilon$ and $\alpha = 0.3$ are used.

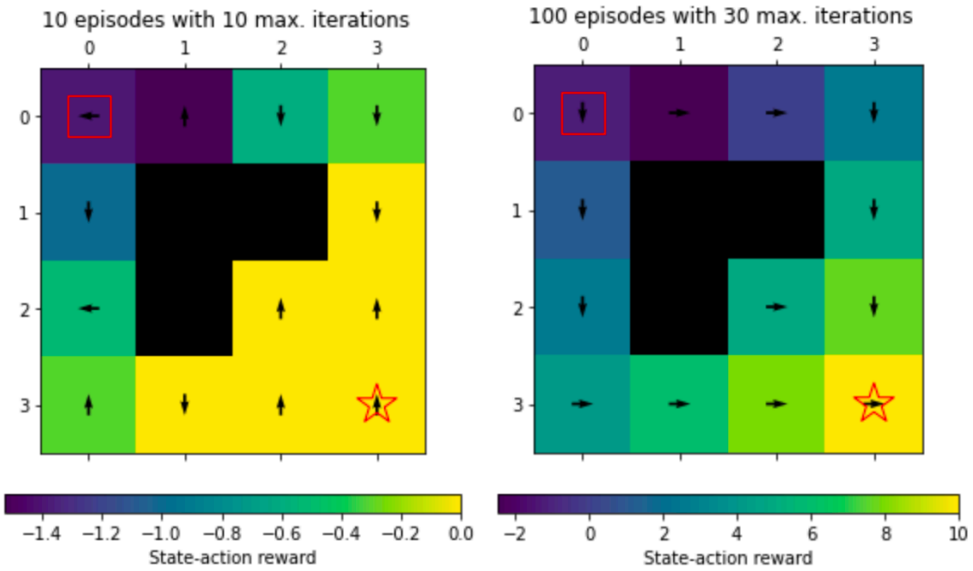



(a) Exploration rate $\epsilon$

(b) Selection of learning rate $\alpha$

Figure 3.2: Selection of $\epsilon$ (left) and $\alpha$ (right) for Q Learning.

The no. of episodes and the no. of maximum iterations per episode are also important hyperparameters to tune for Q Learning. Figure 3.3 illustrates how using a decent amount of episodes and iterations can drastically improve the learning performance. The final policy after Q Learning has converged is straightforward on `small_world` (figure 3.3b). The agent can travel from the initial state to the final state using both possible paths, obtaining similar rewards.

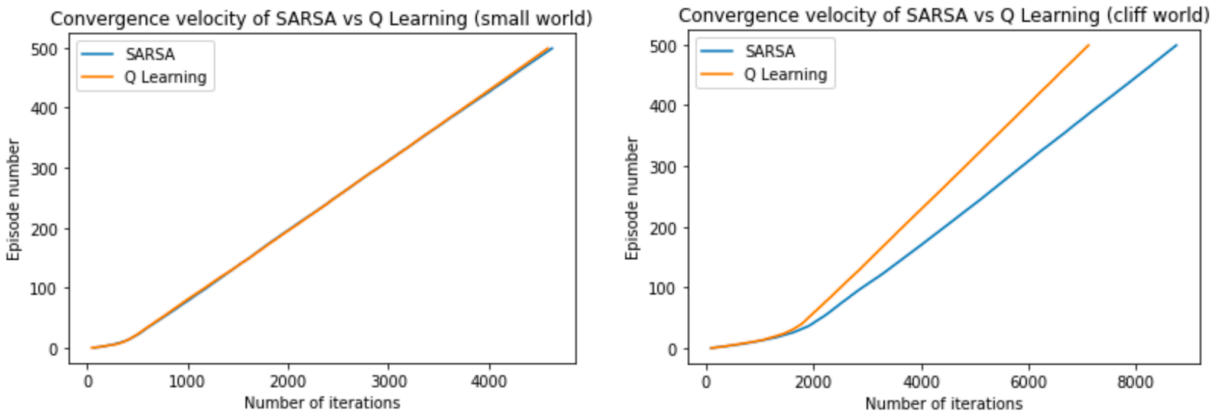



(a) 10 episodes of 10 max. iterations

(b) 100 episodes of 30 max. iterations

Figure 3.3: Performance of Q Learning for different no. episodes and max. iterations. If the number of episodes is insufficient (fig. 3.3a, with 10 episodes and 10 iterations) the algorithm cannot converge. But, if the algorithm runs for more episodes it manages to converge and even finding the two posible solutions (fig. 3.3b).

Focusing on efficiency, figure 3.4 illustrates the speed of convergence of SARSA and Q Learning on `small_world` and `cliff_world`. Q Learning optimises the $Q$ function by approximating the optimal action-value function and it does not rely so much on exploration. SARSA tends to explore more, and since it's an on-policy algorithm it will penalise the really high negative rewards, typical in `cliff_world`. Q-learning directly learns the optimal policy, whilst SARSA learns a near-optimal policy while exploring. Moreover, Q Learning is an off-policy algorithm and it'll find the optimal path ignoring the potential huge negative rewards of falling into the cliff because of a exploration action.




(a) Small world

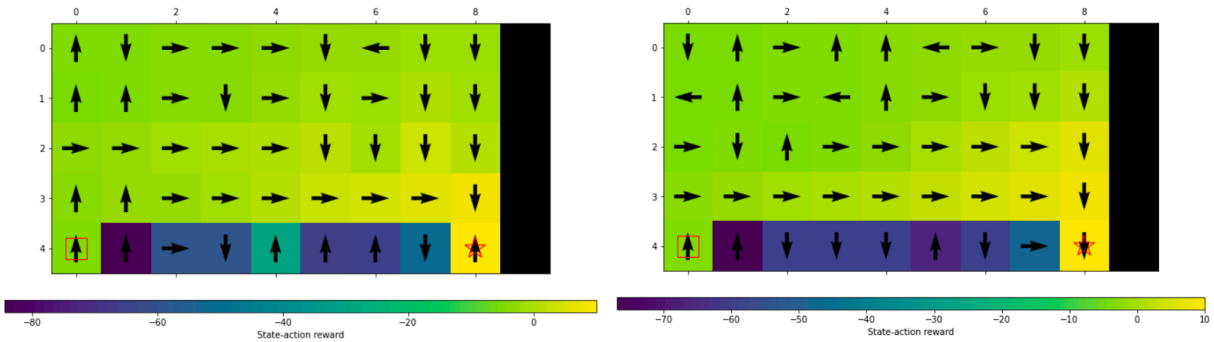
(b) Cliff world

Figure 3.4: Velocity of convergence between SARSA and Q Learning. Since `small_world` is a really simple world, both algorithms converge at the same speed. However, on `cliff_world` that it's a more complex model, it is easy to see that Q Learning converges faster (uses fewer iterations in the last episodes). Although in this environment Q Learning resulted to converge faster, in other models or using other hyperparameters it can be slower. In conclusion, Q Learning can be considered more efficient in some models if you care about the cumulative number of iterations (metric to measure speed of convergence) but, if you really care about the cumulative reward, SARSA can be more efficient since if there is risk of a large negative reward close to the optimal path, that Q-learning will tend to trigger that reward whilst exploring, whilst SARSA will tend to avoid a dangerous optimal path and only slowly learn to use it when the exploration parameters are reduced.

# 4. Question d)

SARSA and Q Learning algorithms are modified to store accumulated reward per episode. If both algorithms are run using decaying $\epsilon$, then its behaviour is pretty similar since they have the same learning process because $\epsilon \to 0$ when n_episodes $\to \infty$. This is shown in figure 4.1, where the accumulated rewards of both algorithms are almost identical; and in figure 4.2, where we can compare the learnt policies and check that Q Learning learns a path right next to the cliff, and SARSA is finding a slightly safer path, but quite close to the cliff [1].


Figure 4.1: Accumulated reward per episode using decaying $\epsilon$, $\alpha = 0.3$




(a) Sarsa

(b) Q Learning

Figure 4.2: Policies of Sarsa and Q Learning using decaying $\epsilon$, $\alpha = 0.3$

[1] SARSA would converge to the optimal path of Q Learning when $\epsilon \to 0$, i.e., when n_episodes $\to \infty$

If both algorithms are executed using a fixed value for $\epsilon$, then their performance is considerably different. For example, in figure 4.3 we can see the cumulative reward using $\epsilon = 0.1$, and the learnt policies are in figure 4.4.

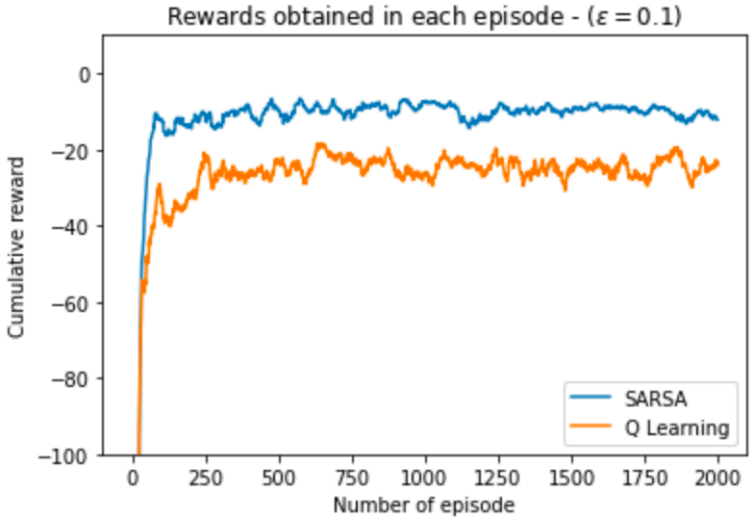
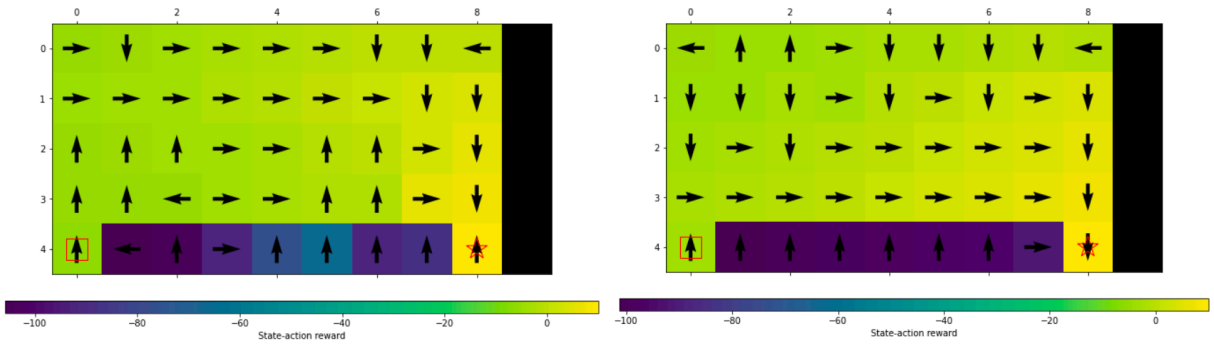



Figure 4.3: Accumulated reward per episode with $\epsilon = 0.1$ and $\alpha = 0.3$. SARSA obtains higher accumulated reward since it an off-policy algorithm and it tries to avoid falling into the cliff because of the exploration nature of $\epsilon$-greedy random action selection. Q learning, since it is a on-policy method, will find the shortest path even though it is dangerous and it can fall into the cliff, resulting in really high negative rewards sometimes.



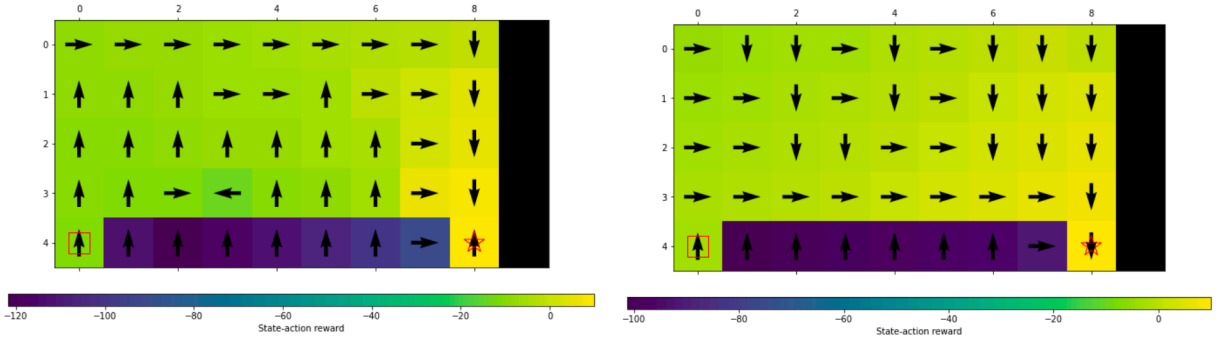

(a) Sarsa

(b) Q Learning

Figure 4.4: Policies of Sarsa and Q Learning with $\epsilon = 0.1$ and $\alpha = 0.3$. As mentioned before, SARSA will approach convergence allowing for possible penalties from exploratory moves, whilst Q-learning will ignore them. That's the reason why SARSA is more conservative and if there is risk of a large negative reward close to the optimal path Q-learning will exploit that reward, whereas SARSA will avoid a potential dangerous optimal path and only slowly learn to use it when the exploration rate is considerably reduced.

To emphasise the disparity between off-policy vs on-policy approaches, we can execute these algorithms using a greater exploration rate: $\epsilon = 0.4$. The cumulative reward plot resulted after running both algorithms is shown in figure 4.5, and the final policies after 1000 episodes are illustrated in figure 4.6.


Figure 4.5: Accumulated reward per episode with $\epsilon = 0.4$ and $\alpha = 0.3$. The cumulative reward disparity is more notorious, since it's more probable to execute a random action. Q Learning suffers from it, and its cumulative reward is much worse than SARSA's.




(a) Sarsa (b) Q Learning

Figure 4.6: Policies of Sarsa and Q Learning with $\epsilon = 0.4$ and $\alpha = 0.3$. For Q Learning, the obtained policy is the same as the ones before (fig. 4.4), since it's a on-policy method. However, the solution found by SARSA is safer than before, walking as far as possible from the cliff. The $Q$ function in SARSA is updated using the policy that maximises the $Q$ function and each update of the $Q$ function is dependent on the actions and states visited before, whereas Q Learning evaluates and improves a different policy from the one used to explore the environments, and is hence more dependent to the randomness (exploration phase) of the $\epsilon$-greedy action selection.

# 5. Question e)

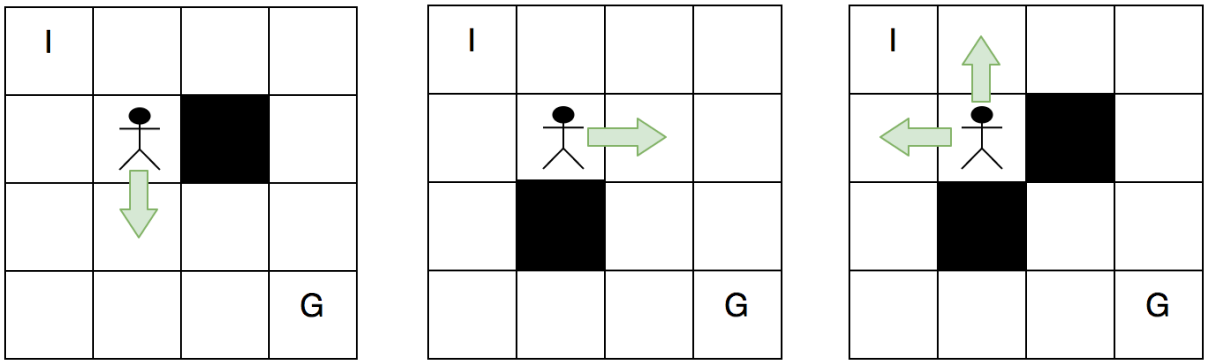
Function approximation is useful in Reinforcement Learning when the state space is large or when it is continuous because it is not possible to represent the the value function as a table. In this question, the new state space is $\hat{S} = S \times \mathbb{Z}_2^3$. If the model `small_world` is used, then $|\hat{S}| = 8 * 16 = 128$ total states. This might seem a small number, but usually the computational cost of reinforcement learning heavily depends on the number of states.

Taking this into account, function approximation is useful for this model since it can linearly approximate the value function $V(\hat{s})$ with $V(\hat{s}, W)$, where $W = \{w_i\}_{i=0}^{N}$ are the weights of the linear approximation, and $\hat{s} = (s, b_1, b_2, b_3)$ is how the state $\hat{s}$ is represented, with $b_i := 1$ if there is a barrier in the $i$-th barrier cell of `small_world`, $b_i = 0$ otherwise.

It is important to underline that $V(\hat{s}, W)$ cannot be fit with a linear function of the form

$$V(\hat{s}, W) = w_0 + w_1 b_1 + w_2 b_2 + w_3 b_3 + w_4 s \tag{5.1}$$

since a single weight just depends on one feature and it is not affected by others. To clarify this we are going to use an example, shown in figure 5.1. The states represented in 5.1a and 5.1b are considered good states ($V$ value should be high), since they would allow the agent to get to the goal state faster than the state represented in 5.1c. However, the approximation using 5.1 wouldn't be able to distinguish between 5.1a, 5.1b and 5.1c since the weights $\{w_j\}_{j=1}^{3}$ would need information from other barrier cells $\{b_i\}_{i=1}^{3}$, but each weight just measures one barrier cell importance.




(a) $b_1 = 0, b_2 = 0, b_3 = 1$ (b) $b_1 = 0, b_2 = 1, b_3 = 0$ (c) $b_1 = 0, b_2 = 1, b_3 = 1$

Figure 5.1: Visual example to argue that the approximation 5.1 is not possible

To handle this, we can still use a linear function approximation to model $V(\hat{s}, W)$,

but using basis functions:

$$V(\hat{s}, W) = \sum_{i=0}^{N} w_i \phi_i(\hat{s}), \tag{5.2}$$

where $\phi_i(\hat{s}) = b_1^{j_1} b_2^{j_2} b_3^{j_3} s^{j_4}$, $j_1, j_2, j_3, j_4 \in \mathbb{Z}$. Basically, a basis function $\phi_i(\hat{s})$ would create a relation between features, and an approximation of $V$ could be fit linearly as in 5.2.

In conclusion: Yes, function approximation would be useful in this model since the number of states is considerably large, and the value function $V$ could be approximated linearly in the parameters $W$, but using basis functions, that would create non-linear relations between features.

# A. Implemented algorithms in Python

## A.1. Value Iteration

```python
import numpy as np

def value_iteration(model, n_episodes=100, tol=0.001):
    # Initialise values arbitrarily, e.g. V_0(s) = 0 for every s state
    V = np.zeros((model.num_states,))
    pi = np.zeros((model.num_states,))

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

    def value_func_update():
        V_new = np.zeros((model.num_states,))
        for s in model.states:
            action_values = [compute_value(s, a, model.reward) \
                             for a in Actions]
            action_val = np.max(action_values)
            V_new[s] = action_val
        return V_new

    for i in range(n_episodes):
        V_new = value_func_update()
        if all((V_new - V) <= tol):
            break
        V = V_new

    # get final policy
    for s in model.states:
        action_values = [compute_value(s, a, model.reward) \
                         for a in Actions]
        action_index = np.argmax(action_values)
        pi[s] = Actions(action_index)

    return V, pi
```

Listing A.1: Value Iteration code

## A.2. SARSA

```python
import numpy as np


def sarsa(model, n_episodes=1000, maxit=100, alpha=0.2, epsilon=0.1):
    V = np.zeros((model.num_states,))
    pi = np.zeros((model.num_states,))
    Q = np.zeros((model.num_states, len(Actions)))

    def choose_eps_greedily(s):
        rand_n = np.random.rand()

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

        idx = np.argmax(Q[s])
        return Actions(idx)

    for i in range(n_episodes):
        # init state
        s = model.start_state
        # init action eps-greedily
        a = choose_eps_greedily(s)

        for _ in range(maxit):
            # get new state after taking action a
            acts_probs_dict = model.next_states_from_state_action(s, a)
            possible_states = list(acts_probs_dict.keys())
            probs = list(acts_probs_dict.values())
            new_s = np.random.choice(possible_states, p=probs)
            # calculate reward
            r = model.reward(s, a)
            # get new action eps-greedily
            new_a = choose_eps_greedily(new_s)
            # update Q using SARSA equation
            Q[s][a] += alpha*(r + model.gamma*Q[new_s][new_a] - Q[s][a])
            # updating state and action
            s = new_s
            a = new_a
            # checking if the new state is terminal
            if s == model.goal_state:
                r = model.reward(s, a)
                Q[s][a] += alpha*(r - Q[s][a])
                break

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

Listing A.2: SARSA code

## A.3. Q Learning

```python
import numpy as np

def q_learning(model, n_episodes=1000, maxit=100, alpha=0.2, epsilon=0.1):
    V = np.zeros((model.num_states,))
    pi = np.zeros((model.num_states,))
    Q = np.zeros((model.num_states, len(Actions)))

    def choose_eps_greedily(s):
        rand_n = np.random.rand()

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

        idx = np.argmax(Q[s])
        return Actions(idx)

    for i in tqdm(range(n_episodes)):
        # init state
        s = model.start_state

        for _ in range(maxit):
            # choose action eps-greedily
            a = choose_eps_greedily(s)
            # get new state after taking action a
            acts_probs_dict = model.next_states_from_state_action(s, a)
            possible_states = list(acts_probs_dict.keys())
            probs = list(acts_probs_dict.values())
            new_s = np.random.choice(possible_states, p=probs)
            # calculate reward
            r = model.reward(s, a)
            # approximating the optimal action-value function
            q = np.max(Q[new_s])
            # update Q using SARSA equation
            Q[s][a] += alpha*(r + model.gamma*q - Q[s][a])
            # updating state
            s = new_s
            # checking if the new state is terminal
            if s == model.goal_state:
                r = model.reward(s, a)
                Q[s][a] += alpha*(r - Q[s][a])
                break

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

Listing A.3: Q Learning code

## A.4. Expected SARSA Learning

```
import numpy as np


def expected_sarsa(model, n_episodes=1000, maxit=100, alpha=0.2, epsilon=0.1):
    V = np.zeros((model.num_states,))
    pi = np.zeros((model.num_states,))
    Q = np.zeros((model.num_states, len(Actions)))

    def choose_eps_greedily(s):
        rand_n = np.random.rand()

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

        idx = np.argmax(Q[s])
        return Actions(idx)

    def greedy_action_selection(s):
        max_q = max(Q[s])
        # there might be cases where there are multiple actions with the same high q_value.
        # Choose randomly then
        count_max_q = np.count_nonzero(Q[s] == max_q)
        if count_max_q > 1:
            # get all the actions with the maxQ
            best_action_indexes = [i for i in range(len(Actions)) \
                                   if Q[s][i] == max_q]
            action_index = np.random.choice(best_action_indexes)
        else:
            action_index = np.where(Q[s] == max_q)[0]

        return Actions(action_index)

    def action_probs(s):
        next_state_probs = [epsilon/len(Actions)] * len(Actions)
        best_action = greedy_action_selection(s)
        next_state_probs[best_action] += (1.0 - epsilon)
        return next_state_probs

    for i in tqdm(range(n_episodes)):
        # init state
        s = model.start_state

        for _ in range(maxit):
            # choose action eps-greedily
            a = choose_eps_greedily(s)
            # get new state after taking action a
            acts_probs_dict = model.next_states_from_state_action(s, a)
```

```
            possible_states = list(acts_probs_dict.keys())
            probs = list(acts_probs_dict.values())
            new_s = np.random.choice(possible_states, p=probs)
            # calculate reward
            r = model.reward(s, a)
            # calculate expected value of the action-value function
            next_state_probs = action_probs(new_s)
            expected_q = sum([a*b for a, b in \
                              zip(next_state_probs, Q[new_s])])
            # update Q using SARSA equation
            Q[s][a] += alpha*(r + model.gamma*expected_q - Q[s][a])
            # updating state
            s = new_s
            # checking if the new state is terminal
            if s == model.goal_state:
                r = model.reward(s, a)
                Q[s][a] += alpha*(r - Q[s][a])
                break

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

Listing A.4: Expected SARSA code

# Bibliography

[1] Sutton, Richard S. and Barto, Andrew G. (2018). *Reinforcement Learning: An Introduction*. The MIT Press. 2nd edition. http://incompleteideas.net/book/the-book-2nd.html