Reinforcement Learning and Decision Making

February 2025

Contents

1 Introduction 3
2 Question A 3
2.1 Policy Iteration 3
2.2 Value Iteration 4
2.3 Comparison of Policy and Value Iteration 5
2.4 Comparison of Synchronous and Asynchronous Value Iteration . . . 8
3 Question B 11
3.1 SARSA algorithm 11
3.2 Expected SARSA algorithm 18
3.3 Comparison of SARSA and Expected SARSA 19
4 Question C 21
4.1 Q-Learning 21
4.2 Comparison of SARSA and Q-Learning 26
5 Question D 28
5.1 Q-Learning in Cliff World 28
6 Question E 30
6.1 Small World Variant 30

1 Introduction

One way that we learn, as humans, is through interaction. This methodology can be extended to machines through reinforcement learning - a framework that learns through interaction with the aim of maximising a reward. Throughout this report we investigate various reinforcement learning methods on discrete grid-world models.

2 Question A

2.1 Policy Iteration

As shown in Figure 1, Policy Iteration (PI) has two key steps: policy evaluation (calculate the value function over each state to evaluate how well the current policy performs) and policy improvement (update the policy to select optimal actions based on the value function). We stop the algorithm, at policy improvement, once the policy is the same as the policy in the previous episode.[1]

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$; $V(terminal) \doteq 0$

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))\big[r + \gamma V(s')\big]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Figure 1: Pseudocode for Policy Iteration [3, Ch.4]. During policy evaluation the value function, $V(s)$, is updated iteratively using the Bellman equation [3, Ch.3]. Policy improvement updates the policy, $\pi(s)$, by selecting the action that maximises the expected return. The process is repeated until the policy stabilises.

[1] In this practical, the code for Policy Iteration was provided as a variant of the Policy Iteration shown in Figure 1. In this code, the policy evaluation was conducted for five iterations as opposed to until the value function under the current policy converges.

2.2 Value Iteration

Value Iteration (VI) stops when the change in the value function is below a small threshold, θ, in a sweep, see Figure 2. In the following experiments this threshold is set to $\theta = 1e{-}4$. Smaller θ, a stricter threshold, trades off potentially better performance with higher computational cost and vice versa for larger θ. Listing 1 shows the key commands implemented.

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

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

Figure 2: Pseudocode for Value Iteration [3, Ch.4]. The value function, $V(s)$, is updated using the Bellman optimality equation until the maximum change in the value function is below the threshold θ [3, Ch.3].

```python
for i in tqdm(range(n_episodes)):
    episode_count += 1
    delta = 0
    V_new = np.copy(V)
    for s in model.states:
        action_values = [compute_value(s, a, model.reward) for a in Actions]
        V_new[s] = max(action_values)
        delta = max(delta, abs(V[s] - V_new[s]))
    V = V_new
    for s in model.states:
        pi[s] = np.argmax(
            [sum(
                    model.transition_probability(s, s_, a)
                    * (model.reward(s, a) + model.gamma * V[s_])
                    for s_ in model.states)
                for a in Actions]
        )
    diff_per_episode.append(delta)
    if delta < threshold:
        break
```

Listing 1: Key commands implemented for Value Iteration

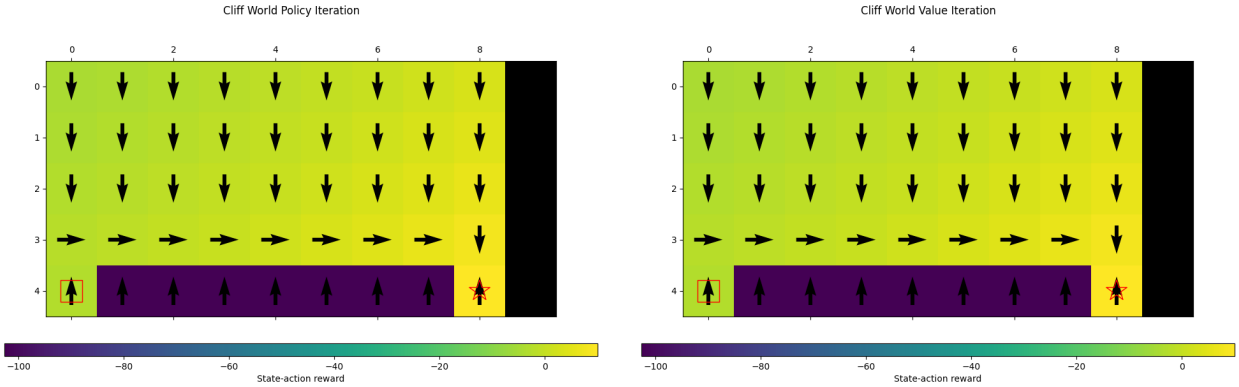
2.3 Comparison of Policy and Value Iteration

The following figures verify the same policy is achieved in each world for both VI and PI.

(a) Policy Iteration policy for small world

(b) Value Iteration policy for small world

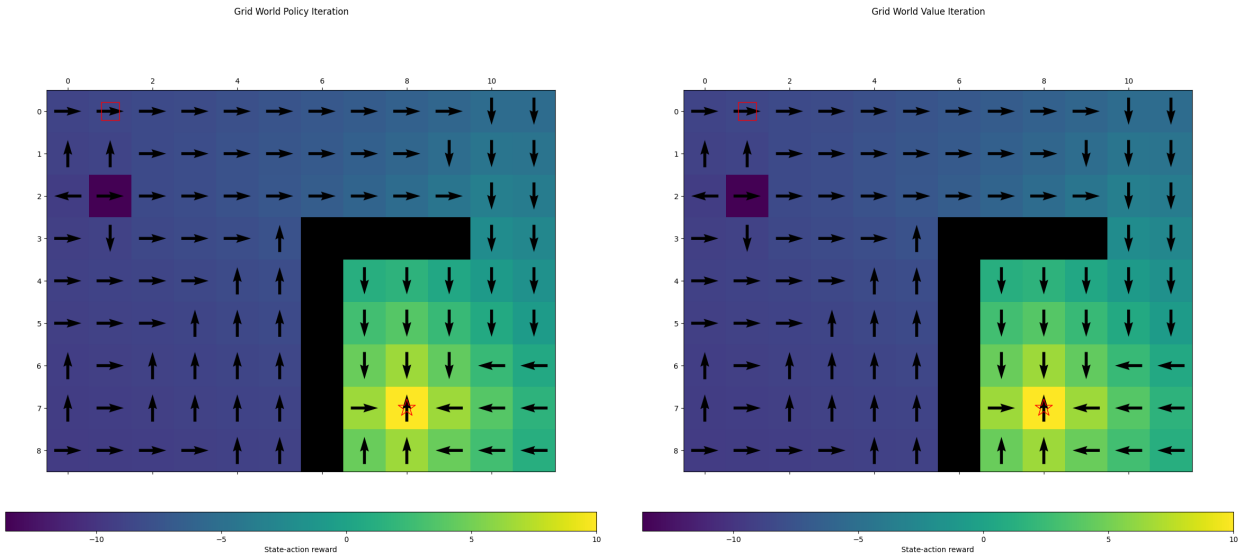
Figure 3: Comparison of policies determined in the small world environment for Policy Iteration and Value Iteration. The square denotes the start position, the star denotes the goal and barriers are shown in black.



(a) Policy Iteration policy for cliff world

(b) Value Iteration policy for cliff world

Figure 4: Comparison of policies determined in the cliff world environment for Policy Iteration and Value Iteration. The square denotes the start position, the star denotes the goal and barriers are shown in black.



(a) Policy Iteration policy for grid world

(b) Value Iteration policy for grid world

Figure 5: Comparison of policies determined in the grid world environment for Policy Iteration and Value Iteration. The square denotes the start position, the star denotes the goal and barriers are shown in black.

Since VI performs policy evaluation once per iteration it is less computationally expensive per iteration - for VI this is $O(|A||S|^2)$ whereas for PI this is $O(|A||S|^2 + |S|^3)$, where A and S denote the number of actions and states respectively [1]. However, Figures 6, 7 and 8 show that VI takes longer to converge than PI. This coincides with the comparisons drawn by Yichen Qiu [2] - it is noted VI can result in slower convergence than PI, particularly in larger state spaces. We see in the largest of our worlds, grid world (Figure 8), that VI requires over five times more iterations than PI before convergence. Overall, total algorithmic costs depend on the size and complexity of the state space, but typically VI is more computationally efficient, particularly in small to medium environments [2].

Since VI only stops once the value function has converged, it is possible that the policy reaches convergence in earlier episodes. For example, in Figure 9 the policy in VI converged at episode 27 but further updates were made, leading to redundant computation.

Note that in cliff world, convergence in VI only requires one more episode than PI due to the nature of the environment, see Figure 7. VI makes a large update instantly - at each iteration the optimal action is chosen allowing the agent to learn to avoid the bad states. Whereas, PI sees initial increases in the policy difference as the value function is computed under the current policy. Hence, the action chosen at the improvement step is restricted by the current policy which may be suboptimal, leading to a similar number of episodes to VI.

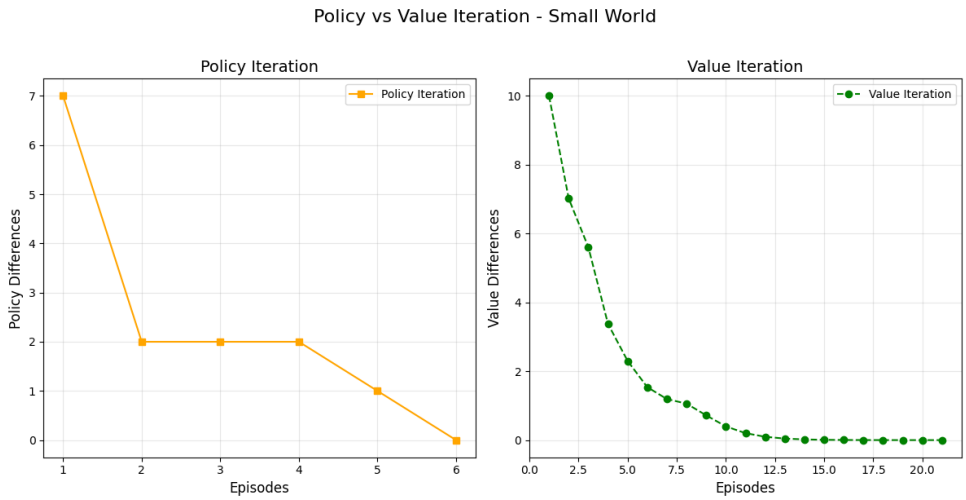


Figure 6: Comparison of the learning behaviour exhibited in small world for Policy Iteration and Value Iteration. For Policy Iteration the number changes to the policy in each episode is plotted. For Value Iteration the final maximum absolute difference between the old and updated value function in each episode is plotted.

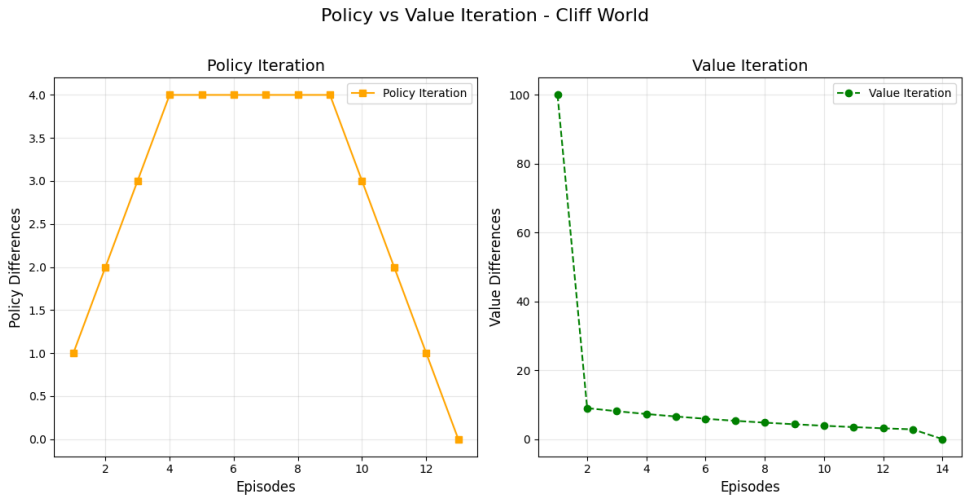


Figure 7: Comparison of the learning behaviour exhibited in cliff world for Policy Iteration and Value Iteration. For Policy Iteration the number changes to the policy in each episode is plotted. For Value Iteration the final maximum absolute difference between the old and updated value function in each episode is plotted. Here, entering a bad state results in an extremely negative reward and moves the agent back to the beginning.

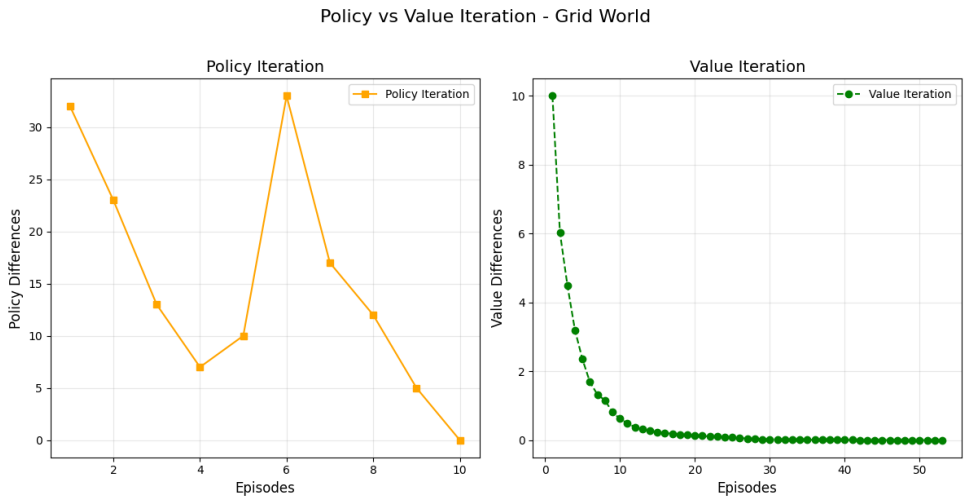


Figure 8: Comparison of the learning behaviour exhibited in grid world for Policy Iteration and Value Iteration. For Policy Iteration the number changes to the policy in each episode is plotted. For Value Iteration the final maximum absolute difference between the old and updated value function in each episode is plotted.

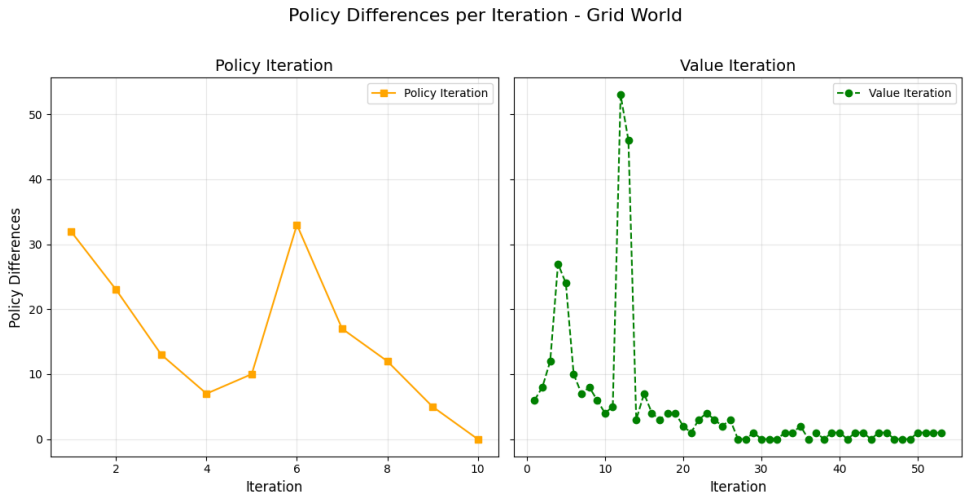


Figure 9: Comparison of the policy behaviour exhibited in grid world for Policy Iteration and Value Iteration. The number changes to the policy in each episode is plotted for both Policy Iteration and Value Iteration.

2.4 Comparison of Synchronous and Asynchronous Value Iteration

The VI algorithm we saw previously is synchronous, meaning that it sweeps over all states before updating the value function. This can be computationally expensive, particularly in large state spaces. Asynchronous algorithms allow updates in any order, using any currently available values. As described by Sutton and Barto [3, Ch.4], one version of asynchronous VI selects one state at random state and updates this. This has been

implemented as shown in Listing 2.

```
for i in tqdm(range(n_episodes)):
    episode_count += 1
    delta = 0
    for _ in range(len(model.states)):
        s = random.choice(model.states)
        action_values = [compute_value(s, a, model.reward) for a in Actions]
        max_value = max(action_values)
        delta = max(delta, abs(V[s] - max_value))
        V[s] = max_value
    for s in model.states:
        pi[s] = np.argmax(
            [sum(
                    model.transition_probability(s, s_, a)
                    * (model.reward(s, a) + model.gamma * V[s_])
                    for s_ in model.states)
                for a in Actions]
        )
    diff_per_episode.append(delta)
    if delta < threshold:
        break
```

Listing 2: Key commands implemented for Asynchronous Value Iteration

Asynchronous VI has the potential to accelerate convergence as shown in Figure 11. However, this is not necessarily always the case, see Figures 10 and 12, because updates may be applied to some states multiple times which may be irrelevant to determining the optimal policy. Additionally, key states (such as the goal and bad states) may not be updated early so key information is not propagated across the state space until later, slowing convergence.

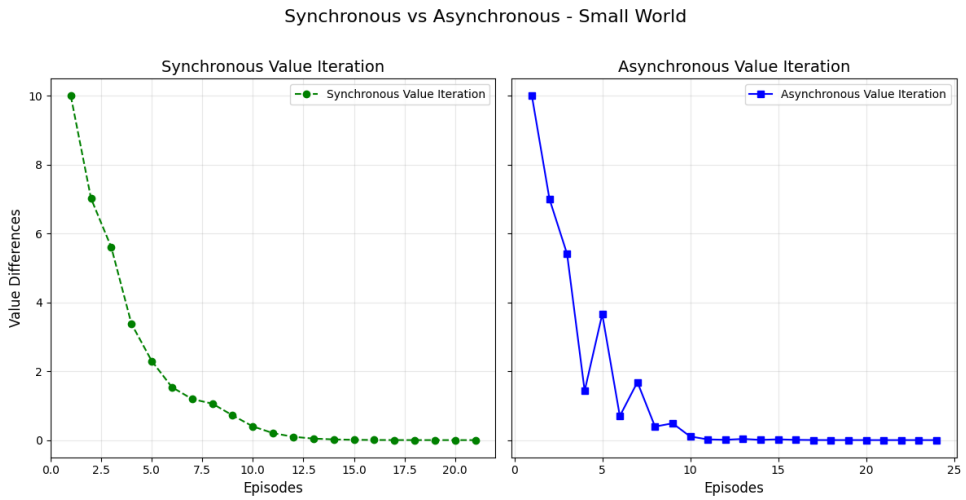


Figure 10: Comparison of Synchronous and Asynchronous Value Iteration in small world. In Synchronous Value Iteration all states are updated simultaneously at the end of each episode, leading to a smooth convergence of value differences over episodes. In Asynchronous Value Iteration random states are updated individually at each iteration within an episode, leading to more unstable learning.

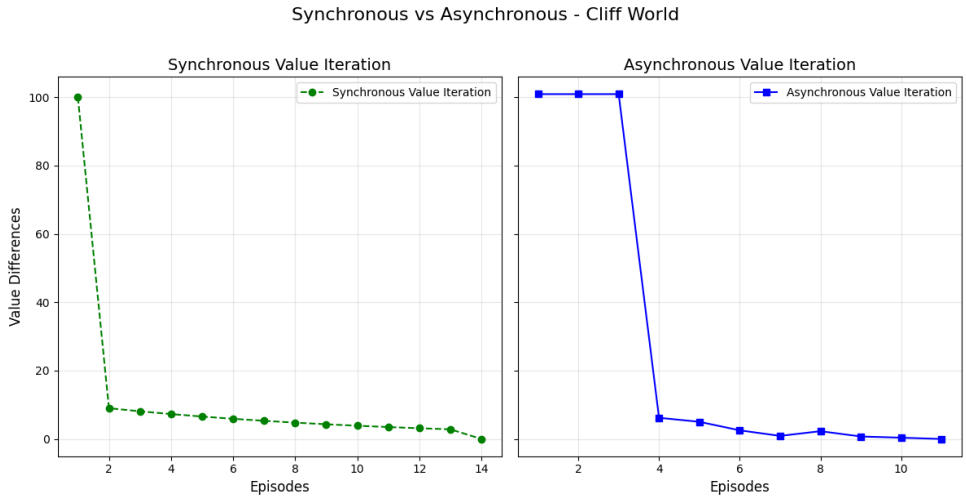


Figure 11: Comparison of Synchronous and Asynchronous Value Iteration in cliff world. In Synchronous Value Iteration all states are updated simultaneously at the end of each episode, leading to a smooth convergence of value differences over episodes. In Asynchronous Value Iteration random states are updated individually at each iteration within an episode. Faster convergence is achieved by Asynchronous Value Iteration, likely due to key states being updated early, at episode 3

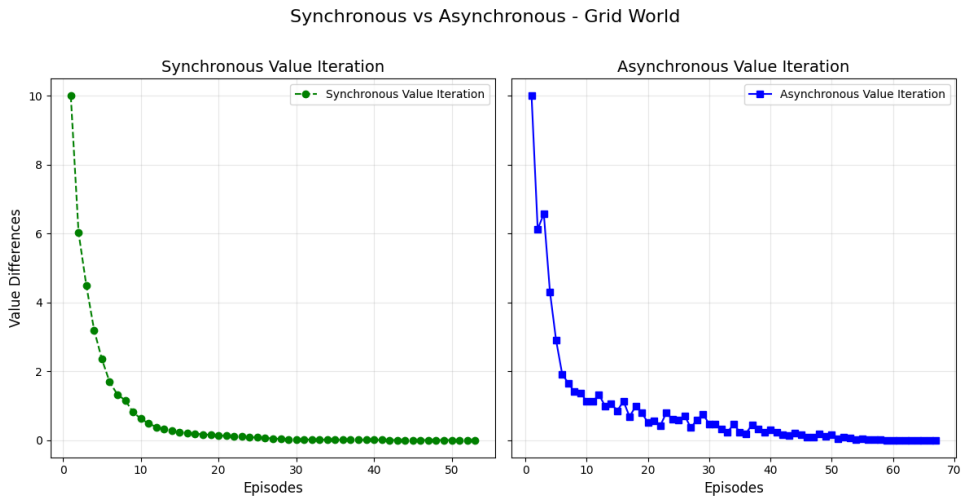


Figure 12: Comparison of Synchronous and Asynchronous Value Iteration in grid world. In Synchronous Value Iteration all states are updated simultaneously at the end of each episode, leading to a smooth convergence of value differences over episodes. In Asynchronous Value Iteration random states are updated individually at each iteration within an episode, leading to more unstable learning.

3 Question B

3.1 SARSA algorithm

In the following sections we investigate Temporal Difference (TD) methods, the first being SARSA (see Figure 13). TD methods learn the value function through bootstrapping and do not require model of the environment as they learn directly from experience. Listing 3 shows the key commands implemented.

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 $Q(\mathit{terminal}, \cdot) = 0$

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

Figure 13: Pseudocode for SARSA, an on-policy TD method [3, Ch.6]. The value function is an estimate that is represented as a Q-table, $Q(s, a)$. It is iteratively updated using the TD error, $[R + \gamma Q(S', A') - Q(S, A)]$, adjusted by the learning rate α. SARSA updates $Q(s, a)$ based on the next action taken under the current policy, following an ϵ-greedy strategy.

```
for i in tqdm(range(n_episodes)):
    s = model.start_state
    a = epsilon_greedy(
        Q, s, epsilon
    )
    total_reward = 0
    for step in range(max_steps):
        s_prime = model.next_state(s, Actions(a))
        R = model.reward(s, Actions(a))
        total_reward += R
        a_prime = epsilon_greedy(
            Q, s_prime, epsilon
        )
        Q[s, a] += alpha * (R + model.gamma * Q[s_prime, a_prime] - Q[s, a])
        if s_prime == model.goal_state:
            break
        else:
            s, a = s_prime, a_prime
```

Listing 3: Key commands implemented for SARSA

To set the hyper-parameters, a combination of reasonable and extreme values for the learning rate (α), exploration parameter for the ϵ-greedy strategy (ϵ), maximum iterations and number of episodes, were explored to evaluate their affect on performance.

Figure 14 shows $\alpha = 0.5$ converges slightly faster than $\alpha = 0.3$. However, $\alpha = 0.3$ converges to slightly higher and more stable average rewards. Given $\alpha = 0.3$ also results in an optimal policy (see Figure 15), this is selected as the best value.

The extreme cases, $\alpha = 0.001$ and $\alpha = 1$, show when α is too small $Q(S, A)$ is adjusted by the TD error update, $\alpha\left[R + \gamma Q(S', A') - Q(S, A)\right]$, slowly i.e. the model takes into account new information learned in the current step very gradually. Hence, the model requires over 3000 episodes to reach convergence. When α is too large the model is unable to stabilise because it is highly sensitive to TD error update from the most recent iteration, leading to higher variance. Figure 15 shows neither $\alpha = 0.001$ or $\alpha = 1$ result in an optimal policy.

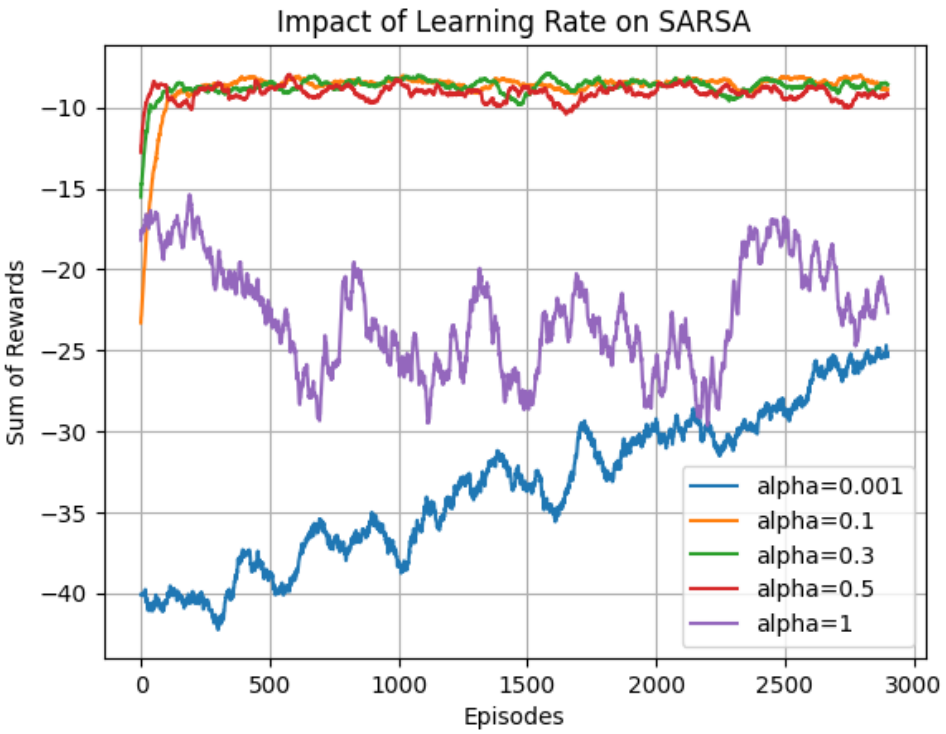


Figure 14: Impact of learning rate, α, on the performance of the SARSA algorithm. The rewards accumulated in each episode are plotted over 3000 episodes for different values of α. When the learning rate is too low ($\alpha = 0.001$) we see slow improvements. When the learning rate is too high ($\alpha = 1$) we see high variance and instability. Note that smoothing has been implemented over the rewards, using a moving average over every 100 episodes, so that trends can be better visualised.

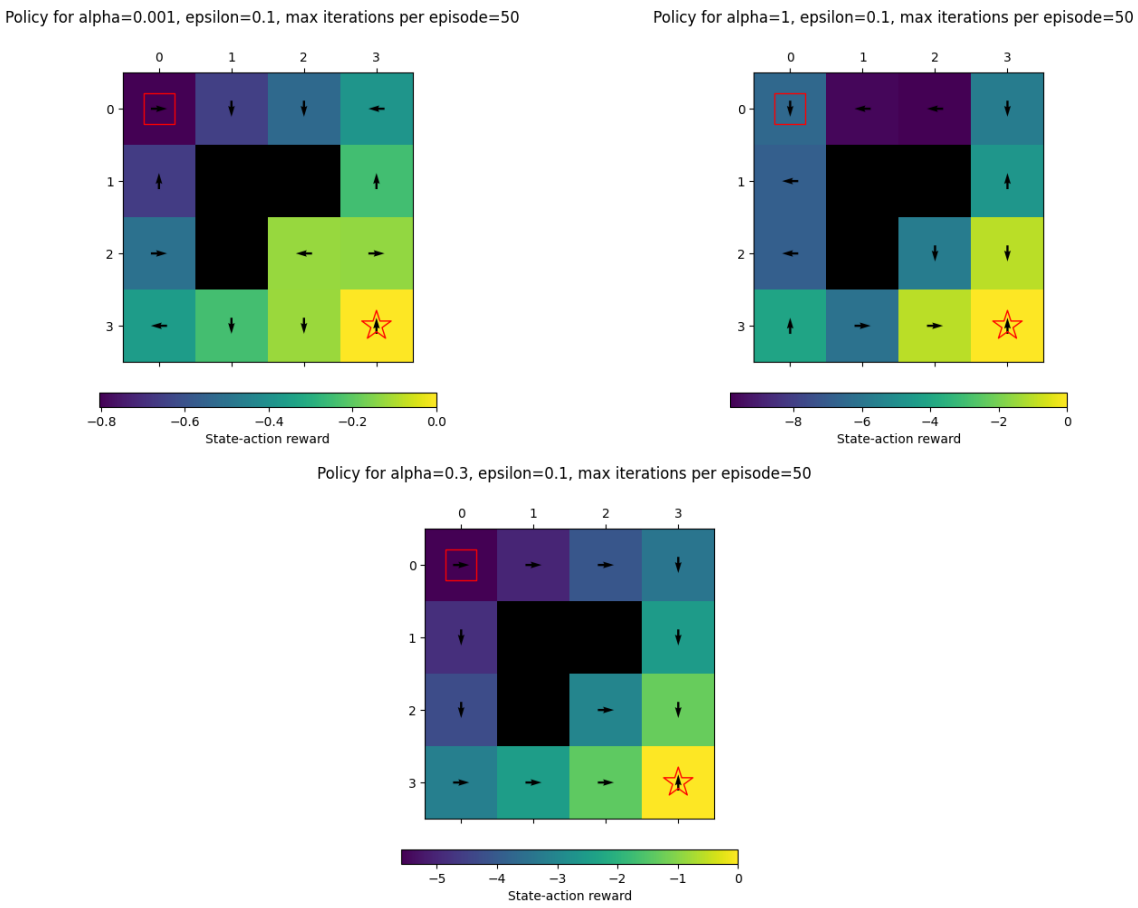


Figure 15: Policies learned by the SARSA algorithm for different values for the learning rate, α, for 500 episodes. The square denotes the start position, the star denotes the goal and barriers are shown in black. (Top right and Top left) When the learning rate is too high (α =1) or too low ($\alpha = 0.001$) an optimal policy is not found. (Bottom) At $\alpha = 0.3$ an optimal policy is found.

The exploration rate, ϵ, determines how often the model takes a random action from the current state instead of the best estimated action. Figure 16 shows $\epsilon = 0.001$ converges to the highest rewards, as it mainly exploits the learned policy. However in larger more, complex worlds, low exploration risks getting stuck in early local optimum state which can result in failure to find a better policy. $\epsilon = 0.01$ provides a good trade-off between exploitation and exploration, achieving only slightly lower rewards than whilst finding the optimal policy (see Figure 17). If ϵ is too high the model mainly explores, often selecting in suboptimal states. This can be seen by the stronger negative rewards in Figure 17 and lower average rewards in Figure 16 for $\epsilon = 0.99$.

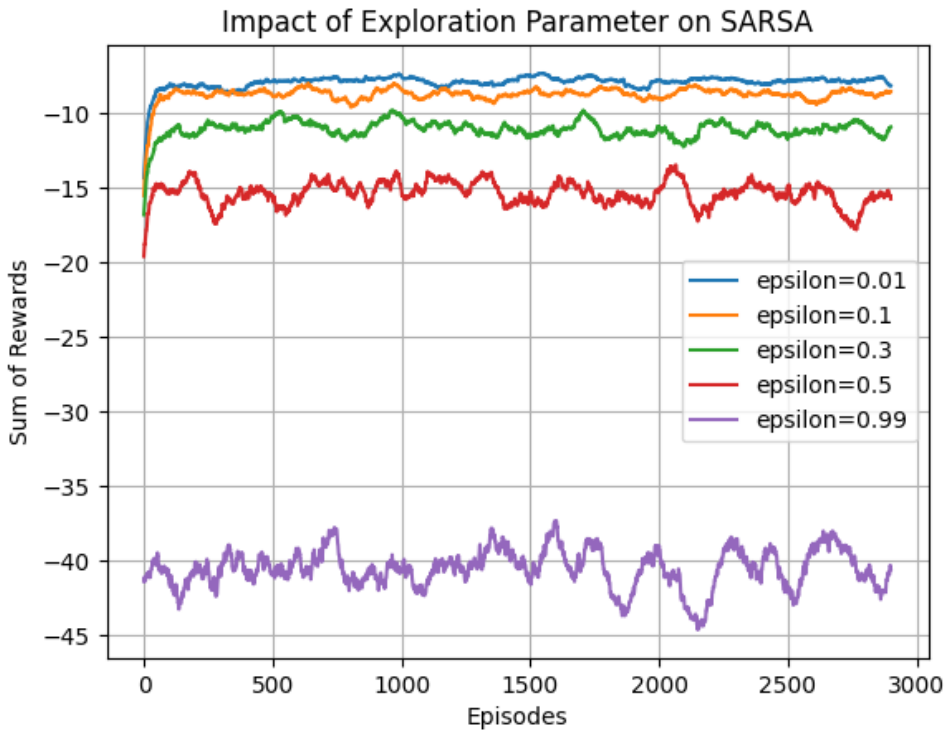


Figure 16: Impact of exploration parameter, ϵ, on the performance of the SARSA algorithm. The rewards accumulated in each episode are plotted over 3000 episodes for different values of ϵ. Note that smoothing has been implemented over the rewards, using a moving average over every 100 episodes, so that trends can be better visualised.

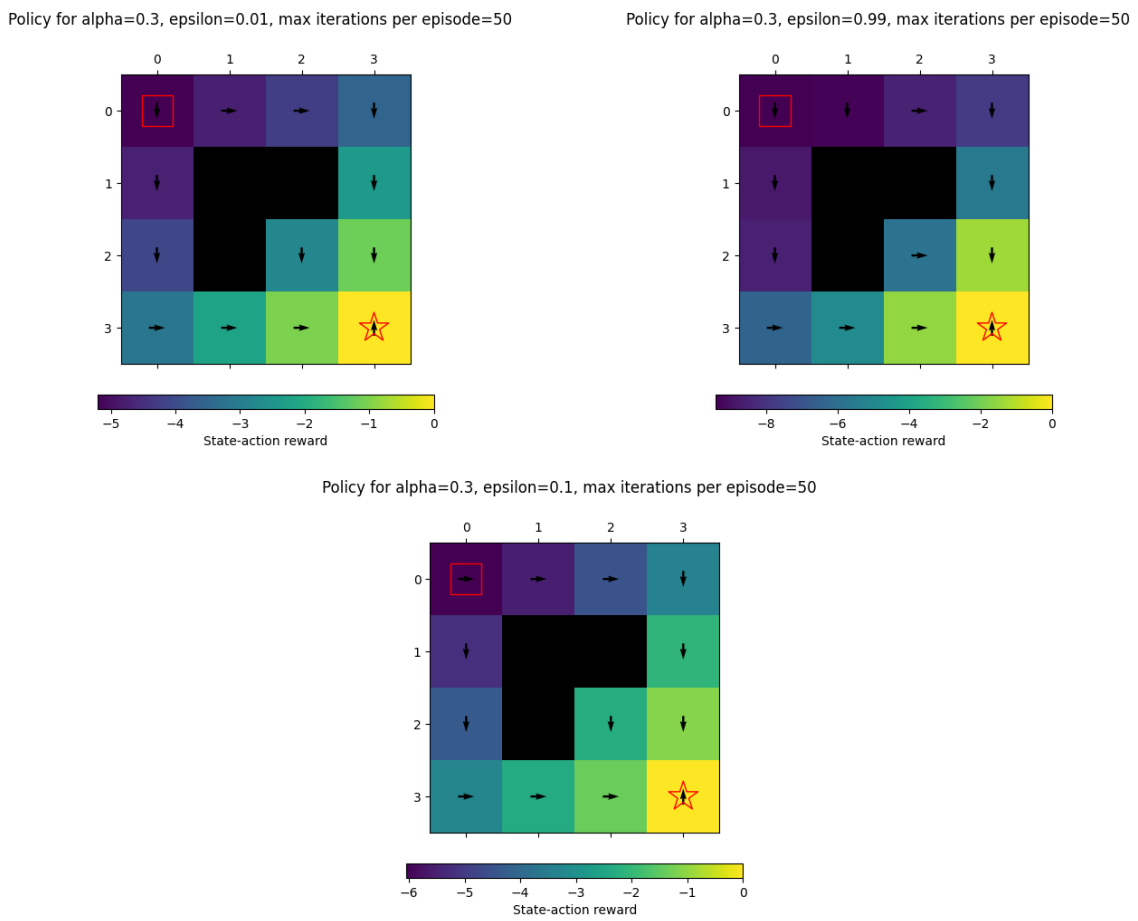


Figure 17: Policies learned by the SARSA algorithm for different values for the exploration parameter, ϵ, for 500 episodes. The square denotes the start position, the star denotes the goal and barriers are shown in black. When the exploration parameter is too high (Top right, $\epsilon = 0.99$), the states are generally darker than when $\epsilon = 0.01$ (Top left), depicting stronger negative rewards.

The number of episodes are considered in conjunction with the maximum iterations. In Figure 18, 5 iterations show no variability in the average rewards, suggesting that the model has not learned effectively. This is evidenced in Figure 19 where an optimal policy is not found. Although 100 iterations achieves an optimal policy and shows quick initial learning and convergence, as shown in Figure 19, compared to 10 iterations, this level of computation shows poorer average rewards and more variance. This due to the agent having too many opportunities to explore and take suboptimal actions. 10 is the best value, achieving sufficient rewards and optimal policy with more stable learning, converging at around episode 250.

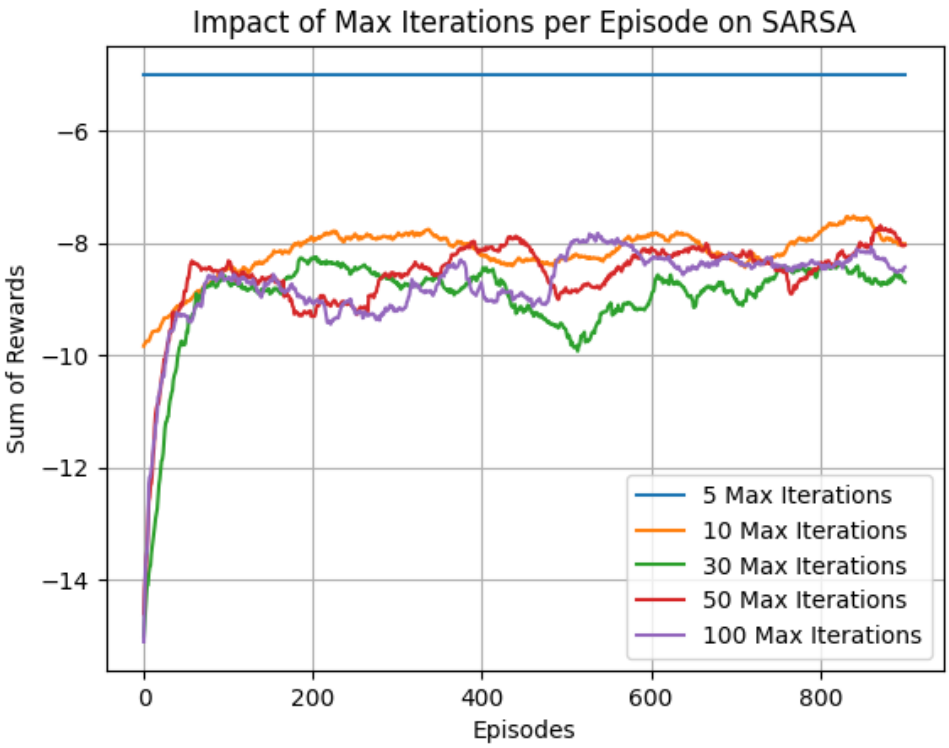


Figure 18: Impact of the maximum number of iterations per episode on the performance of the SARSA algorithm. The rewards accumulated in each episode are plotted over 1000 episodes for different values of of the maximum iterations per episode. A maximum of 10 maximum iterations achieves the best balance between stability and learning efficiency. Note that smoothing has been implemented over the rewards, using a moving average over every 100 episodes, so that trends can be better visualised.

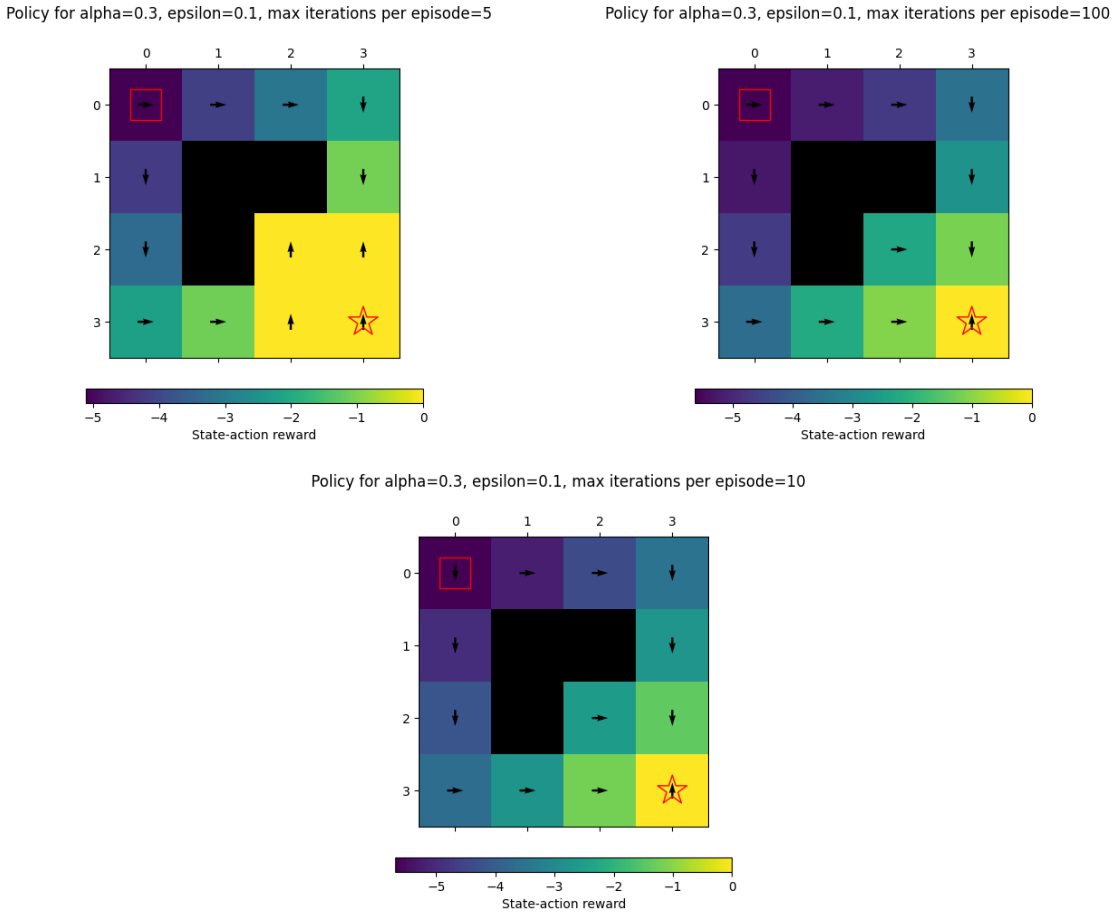


Figure 19: Policies learned by the SARSA algorithm for different values for the maximum iterations per episode for 500 episodes. The square denotes the start position, the star denotes the goal and barriers are shown in black. (Top left) When the maximum number iterations is set to 5, the agent is unable to learn the optimal policy. Near the goal is where the policy fails, as 5 iterations does not give the agent enough time to reach the goal, therefore relying on exploration to learn about the goal and the surrounding states. (Top right and Bottom) The optimal policy is reached when the maximum number of iterations is set to 100 and when set to 10.

3.2 Expected SARSA algorithm

Expected SARSA follows the pseudocode in Figure 13 but takes the expectation of the action-value function when computing the TD error update as shown in Listing 4.

```
for i in tqdm(range(n_episodes)):
    s = model.start_state
    total_reward = 0
    for step in range(max_steps):
        a = epsilon_greedy(
            Q, s, epsilon
        )
        s_prime = model.next_state(s, Actions(a))
        R = model.reward(s, Actions(a))
        total_reward += R
        expected_q = 0
        for a_prime in range(len(Actions)):
            prob_a_prime = (
                epsilon / len(Actions)
                if a_prime != np.argmax(Q[s_prime])
                else 1 - epsilon + (epsilon / len(Actions))
            )
            expected_q += prob_a_prime * Q[s_prime, a_prime]
        Q[s, a] += alpha * (R + model.gamma * expected_q - Q[s, a])
        if s_prime == model.goal_state:
            break
        s = s_prime
```

Listing 4: Key commands implemented for Expected SARSA

3.3 Comparison of SARSA and Expected SARSA

Figure 20 shows both algorithms found an optimal policy. Expected SARSA typically shows less variance however, this is difficult to observe clearly in Figure 21 as small world is a small, simple environment. Expected SARSA usually has less variance is because it averages over all possible actions (Listing 4), smoothing the effect of poor actions. This process of averaging also causes Expected SARSA to be more computationally complex as SARSA only considers the next action when computing the TD error update (Figure 13) [3, Ch.6]. Hence SARSA is more computationally efficient.

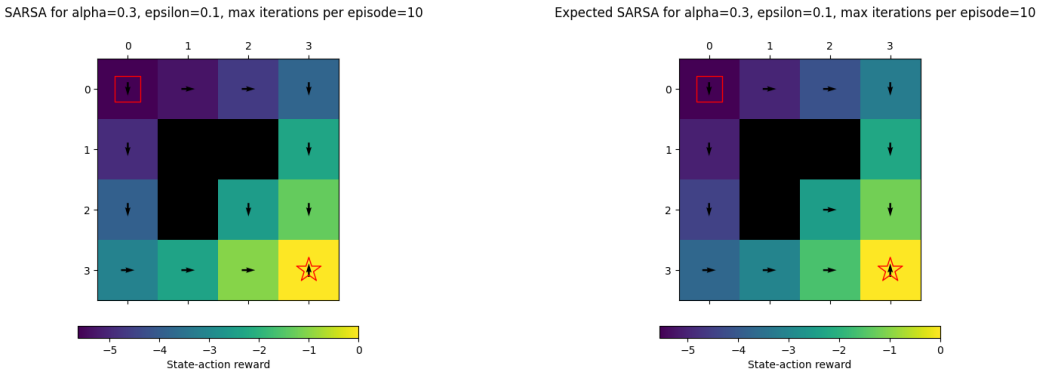


Figure 20: Comparision of policies learned by the SARSA and Expected SARSA algorithm with $\alpha = 0.3, \epsilon = 0.1,$ max iterations $= 10,$ episodes $= 250$. The square denotes the start position, the star denotes the goal and barriers are shown in black.

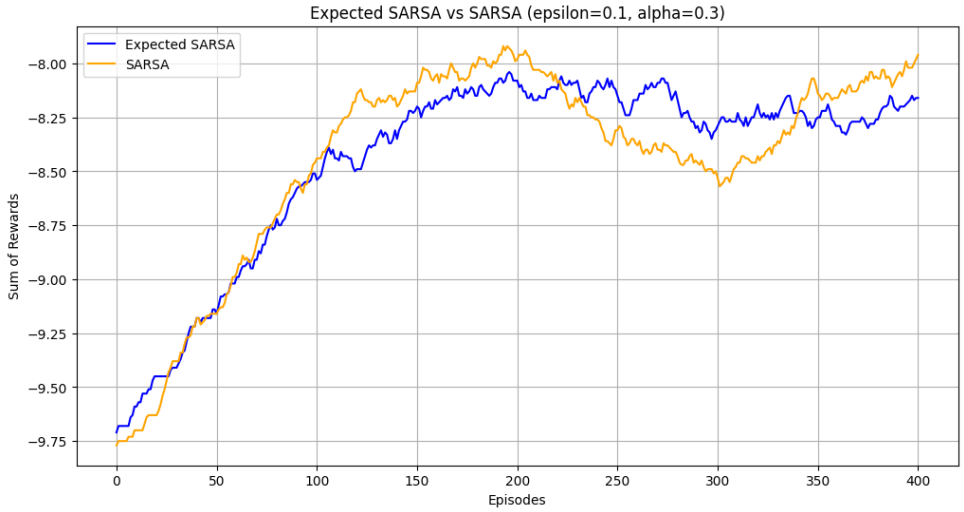


Figure 21: The rewards accumulated in each episode are plotted over 500 episodes for the SARSA and Expected SARSA algorithm in the small world environment, where $\epsilon = 0.1$ and $\alpha = 0.3$. Note that smoothing has been implemented over the rewards, using a moving average over every 100 episodes, so that trends can be better visualised.

The effectiveness of Expected SARSA is more evident in cliff world, shown in Figure 22. Expected SARSA's updates smoothen the affect of poor actions whereas SARSA updates based on the actual action and achieves slightly lower average rewards with more variability.

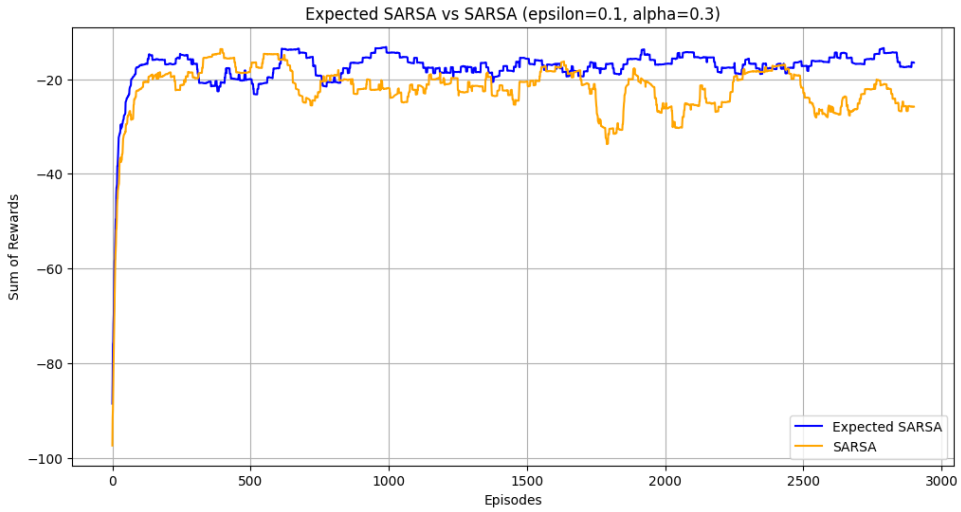


Figure 22: The rewards accumulated in each episode are plotted over 3000 episodes for the SARSA and Expected SARSA algorithm in the cliff world environment, where $\epsilon = 0.1$ and $\alpha = 0.3$. Note that smoothing has been implemented over the rewards, using a moving average over every 100 episodes, so that trends can be better visualised.

4 Question C

4.1 Q-Learning

Q-Learning is similar to SARSA, but makes updates based on the next best action regardless of the current policy (see Figure 23). Listing 5 shows the key commands implemented.

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

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

Figure 23: Pseudocode for Q-Learning, an off-policy TD method [3, Ch.6]. The value function is represented as a Q-table, $Q(s, a)$, which is iteratively updated using TD error, $[R + \gamma \max_a(Q(S', A')) - Q(S, A)]$, adjusted by the learning rate α. Q-Learning updates $Q(s, a)$ based on the action that maximises the reward in the next state, independent of the current policy, following an ϵ-greedy strategy.

```
for i in tqdm(range(n_episodes)):
        s = model.start_state
        total_reward = 0
        for step in range(max_steps):
            a = epsilon_greedy(
                Q, s, epsilon
            )
            s_prime = model.next_state(s, Actions(a))
            R = model.reward(s, Actions(a))
            total_reward += R
            Q[s, a] += alpha * (R + model.gamma * np.max(Q[s_prime, :]) - Q[s, a])
            if s_prime == model.goal_state:
                break
            else:
                s = s_prime
```

Listing 5: Key commands implemented for Q-Learning

The same method as discussed in Question B is conducted to set the hyper-parameters. Figure 24 shows $\alpha = 0.3$ as the best value. This learning rate converges fast whilst achieving high average rewards and with a more stable policy.

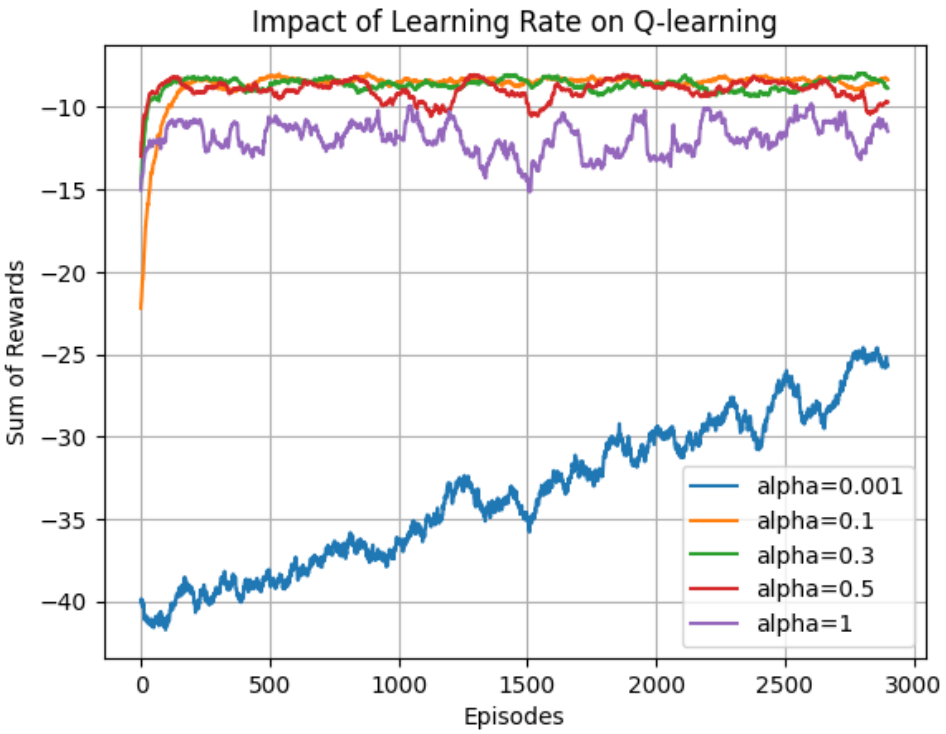


Figure 24: Impact of learning rate, α, on the performance of the Q-Learning algorithm. The rewards accumulated in each episode are plotted over 3000 episodes for different values of α. When the learning rate is too low ($\alpha = 0.001$) we see slow improvements. When the learning rate is too high ($\alpha = 1$) we see high variance and instability. Note that smoothing has been implemented over the rewards, using a moving average over every 100 episodes, so that trends can be better visualised.

Similarly to as discussed in Question B, Figure 25 shows once again $\epsilon = 0.01$ provides a good trade-off, balancing learning efficiency and exploration. Exploration is particularly important in larger, more complex state spaces where there is more risk of getting stuck in a local optimum.

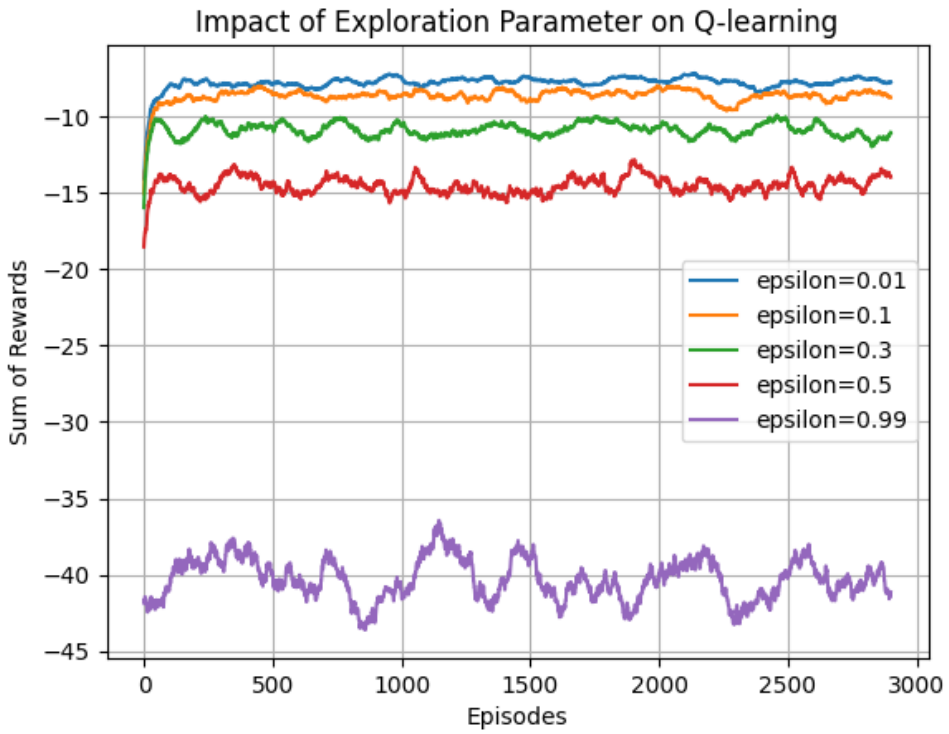


Figure 25: Impact of exploration parameter, ϵ, on the performance of the Q-Learning algorithm. The rewards accumulated in each episode are plotted over 3000 episodes for different values of ϵ. Note that smoothing has been implemented over the rewards, using a moving average over every 100 episodes, so that trends can be better visualised.

Figure 26 and 27 shows the agent does not learn effectively at 5 iterations, as this does not give the agent enough time to learn about the states near the goal. Post quick initial learning, 100, 50 and 30 iterations show high variability due to the exploration parameter. This is due to the agent having too many opportunities to explore within each episode, resulting in more suboptimal actions. 10 iterations with 200 episodes results in higher average rewards and more stable learning therefore, it is the best option.

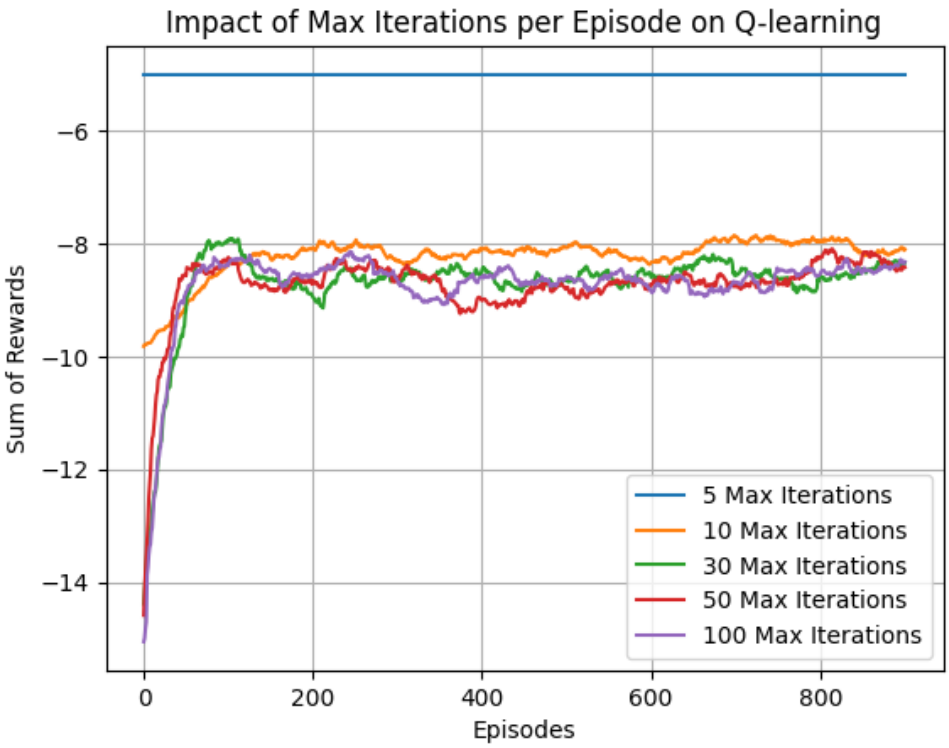


Figure 26: Impact of the maximum number of iterations per episode on the performance of the Q-Learning algorithm. The rewards accumulated in each episode are plotted over 1000 episodes for different values of of the maximum iterations per episode. A maximum of 10 maximum iterations achieves the best balance between stability and learning efficiency. Note that smoothing has been implemented over the rewards, using a moving average over every 100 episodes, so that trends can be better visualised.

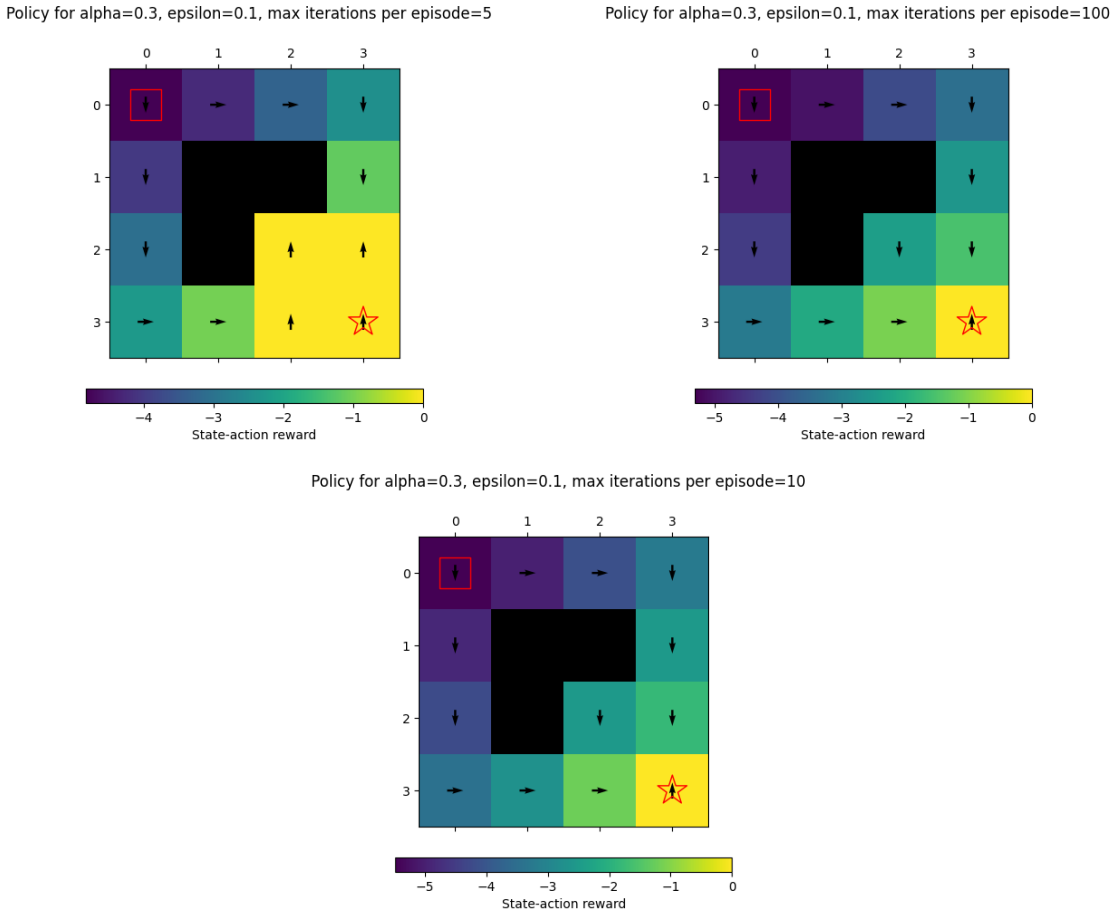


Figure 27: Policies learned by the Q-Learning algorithm for different values for the maximum iterations per episode for 500 episodes. The square denotes the start position, the star denotes the goal and barriers are shown in black. (Top left) When the iterations are too low (5), the agent is unable to learn the optimal policy. Near the goal is where the policy goes wrong, as 5 iterations does not give the agent enough time to reach the goal, therefore relying on exploration to learn about the goal and surrounding states. (Top right and Bottom) The optimal policy is reached when the maximum number of iterations is set to 100 and when set to 10.

4.2 Comparison of SARSA and Q-Learning

Typically Q-Learning is faster and takes riskier paths. However, since small world is a small, simple environment, Q-Learning and SARSA show similar performance in Figure 29 and both find similar optimal policies in Figure 28. Q-Learning is less computationally efficient because it requires considering all possible actions to select the one that provides the maximum reward at the current state. We will delve deeper into Q-learning's behaviour in the next section.

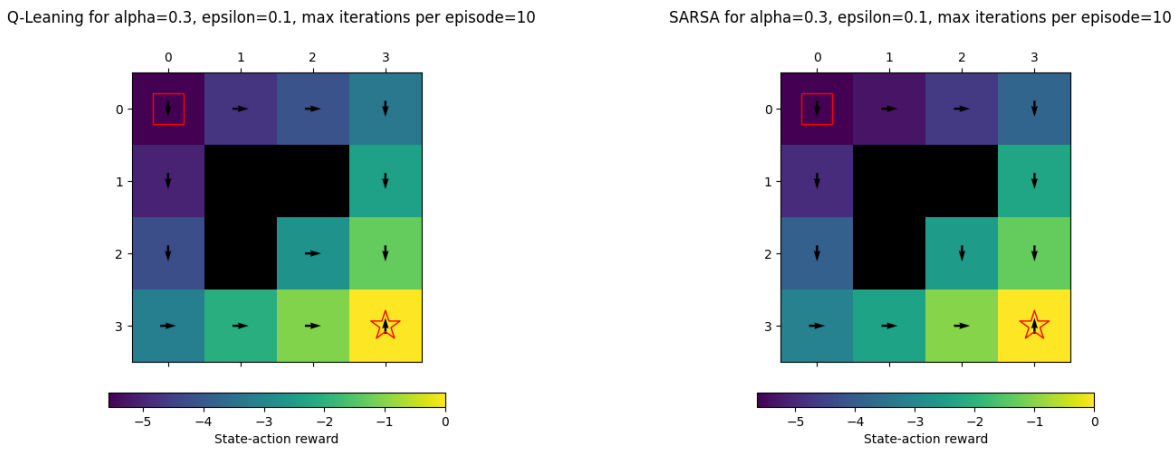


Figure 28: Comparision of policies learned by the SARSA algorithm with $\alpha = 0.3, \epsilon = 0.1,$ max iterations $= 10,$ episodes $= 250$ and the Q-Learning algorithm with $\alpha = 0.3, \epsilon = 0.1,$ max iterations $= 10,$ episodes $= 200$. The square denotes the start position, the star denotes the goal and barriers are shown in black.

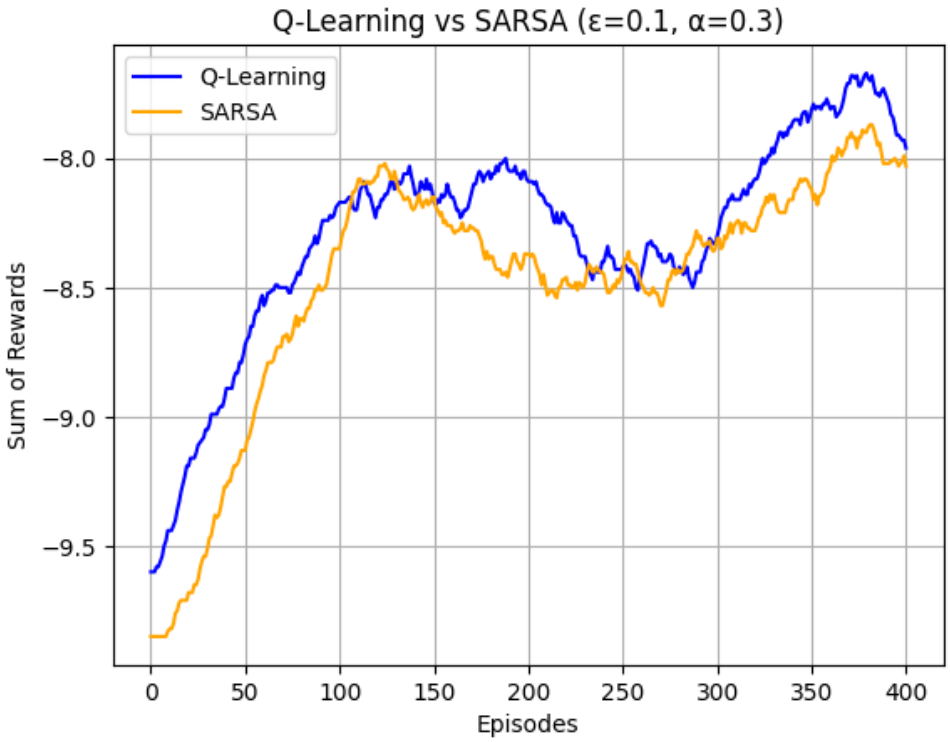


Figure 29: The rewards accumulated in each episode are plotted over 500 episodes for the SARSA and Q-Learning algorithm in the small world environment, where $\epsilon = 0.1$ and $\alpha = 0.3$. Note that smoothing has been implemented over the rewards, using a moving average over every 100 episodes, so that trends can be better visualised.

5 Question D

5.1 Q-Learning in Cliff World

The effects of Q-Learning is more evident in Figure 30. We see Q-Learning converges within less episodes than SARSA. However, Q-learning also shows more variance and achieves lower average rewards.

Since Q-Learning is off-policy and greedily updates $Q(S, A)$ based on the maximum estimated action-value (Figure 23), it is less sensitive to actions resulting in large penalties - Q-values are overestimated as it assumes optimal actions will always be taken. As a result, the agent favours paths that lead to high rewards and in doing so it takes a riskier path, falling off the cliff more often due to the exploration parameter. We see this in Figure 31 (b), where the fastest route is taken and the darkness of the cliff states shows that the model has learnt large penalties occur here.

SARSA takes a risk-averse approach, taking a longer, safer path, see Figure 31 (a). This is because SARSA is on-policy and updates $Q(S, A)$ based on the actual action taken, including actions taken during exploration (Figure 13). Hence, SARSA behaves more cautiously since it is more sensitive to actions resulting in large negative rewards. The darkness of cell $(1, 4)$ indicates the model has learnt large penalties occur here and avoids this state. As a result, it explores the other bad states at the cliff edge less (shown by the blue-green colours), resulting in more stable performance (shown in Figure 30).

Selecting the best algorithm depends on the scenario. Suppose we have a safety-critical task e.g. a self-driving car navigating busy roads. A risk-averse approach such as SARSA would be preferred over Q-Learning as poor actions can lead to dangerous outcomes. Stable learning is also important as sudden changes in behaviour, when driving a car, could be dangerous.

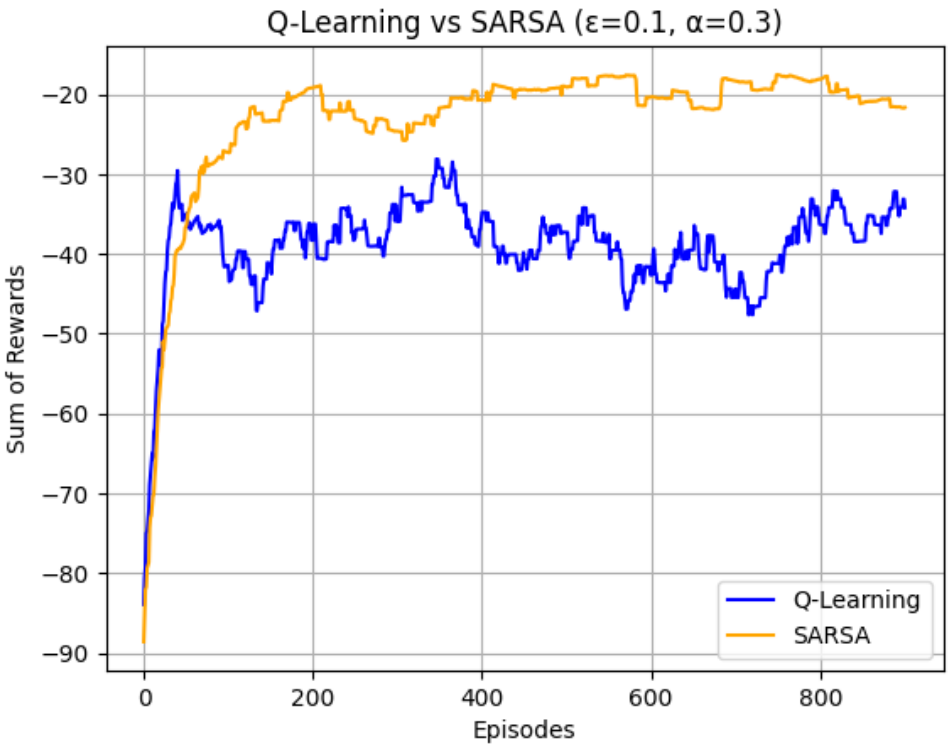
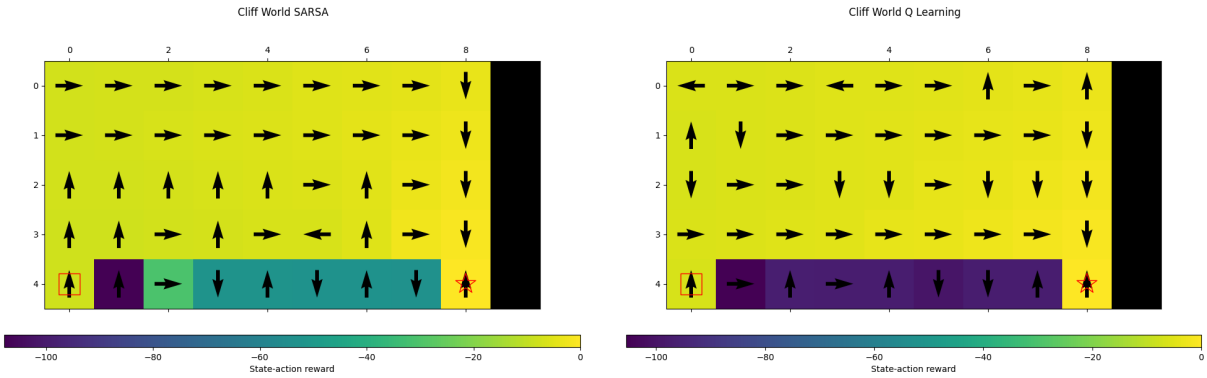


Figure 30: The rewards accumulated in each episode are plotted over 1000 episodes for the SARSA and Q-Learning algorithm in the cliff world environment, where $\epsilon = 0.1$ and $\alpha = 0.3$. SARSA shows better performance than Q-Learning. Although if ϵ were set to decrease gradually, say $\epsilon = 1/(\textit{episode number})$, both methods would asymptotically converge to the optimal policy [3, Ch.6]. This is due exploration parameter leading the agent exploring less as it learns more information about the environment. Note that smoothing has been implemented over the rewards, using a moving average over every 100 episodes, so that trends can be better visualised.



(a) Policy under SARSA

(b) Policy under Q-Learning

Figure 31: The policy found for SARSA and Q-Learning in cliff world where $\epsilon = 0.1$ and $\alpha = 0.3$. The square denotes the start position, the star denotes the goal and barriers are shown in black. SARSA takes the longer, safer path to the goal whereas Q-Learning takes the quicker, riskier path.

6 Question E

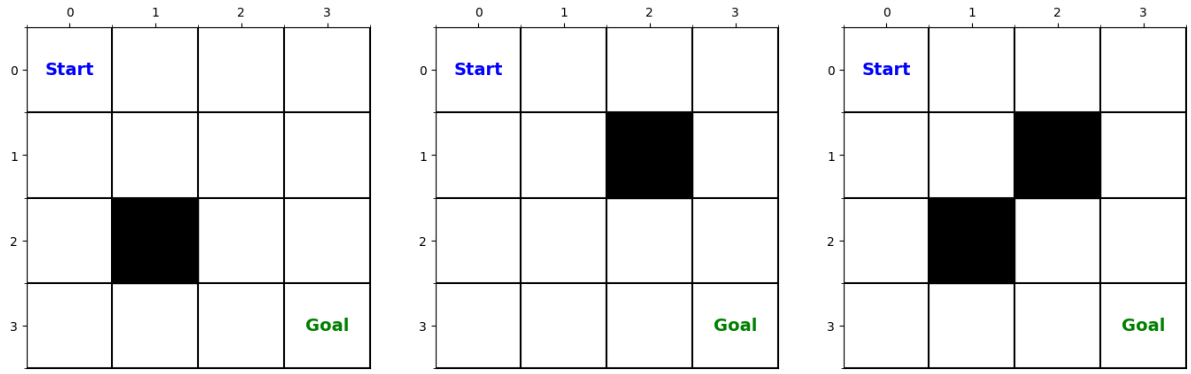
6.1 Small World Variant

The implementations of SARSA and Q-Learning we saw in previous sections used tabular representations to approximate the value function ($Q(S, A)$). However, our new variant of small world expands the state space, $S = 16$, by 8-fold, $\bar{S} = S \times \mathbb{Z}_2^3 = 16 \times 2^3 = 128$, encoding information about the current location and which cells are barriers in the current episode. Hence, the same location can be in multiple states depending on where the barriers are. Due to the larger state space and the complexities the barrier configurations introduce, the value function cannot be represented as a table so function approximation is useful.

Function approximation uses a parametrised function with the weight vector $\boldsymbol{\theta} \in \mathbb{R}^n$. In this scenario it can be represented as follows:

$$V_\pi(\mathbf{s}) \approx \hat{V}_\pi(\mathbf{s}, \boldsymbol{\theta})$$

where $\boldsymbol{s} = (b_1, b_2, b_3, l, 1)$ and $b_i \in \{0, 1\}$ is one of the three barrier cells, l is the current state location and 1 is the bias term. When one state is updated, other states are affected by the new information, allowing the model to generalise [3, Ch.9].



(a) position = (1,1), $b_1 = 1, b_2 = 0, b_3 = 0$

(b) position = (1,1), $b_1 = 0, b_2 = 0, b_3 = 1$

(c) position = (1,1), $b_1 = 1, b_2 = 0, b_3 = 1$

Figure 32: Possible barrier configurations in the small world variant. The current position of the agent in all configurations is $(1, 1)$. The presence or absence of a barrier is denoted by $b_i \in \{0, 1\}, i \in \{1, 2, 3\}$

Linear function approximation is defined as follows:

$$\hat{V}_\pi(\mathbf{s}, \boldsymbol{\theta}) = \sum_i \theta_i \phi_i(\mathbf{s})$$

where $\phi_i(\boldsymbol{s}) : \mathcal{S} \to \mathbb{R}, i = 1, ..., n$ are feature functions.

Linear methods can suffice if the feature functions are non-linear and capture the dependencies correctly [3, Ch.9]. Consider Figure 32, where the agent is currently at position $(1, 1)$. The optimal actions are as follows:

- Figure 32 (a): Right to $(2, 1)$.
- Figure 32 (b): Down to $(1, 2)$.
- Figure 32 (c): Up to $(1, 0)$ or left to $(0, 1)$.

In Figure 32 (c) the combination of both barriers block direct paths to the goal.

The linear method with linear feature functions is represented as follows:

$$\hat{V}_\pi(\mathbf{s}, \boldsymbol{\theta}) = \theta_1 b_1 + \theta_2 b_2 + \theta_3 b_3 + \theta_4 l + \theta_5$$

This representation cannot take into account the effect of interactions between features, such as the presence of b_1 and b_3, since each barrier is treated independently. Hence, non-linear feature functions are needed in order to capture these effects. For example, higher order polynomials, such as quadratics.

Least-Squares TD (LSTD) is also a linear method that would not be suitable. Since LSTD has no step-size parameter it never forgets [3, Ch.9]. This is problematic as the target policy changes each episode due to the changing barrier configurations.

Overall, linear methods can suffice if they capture the nature of the environment appropriately.

Bibliography

[1] Matt Gormley. Lecture 23: Reinforcement learning. Lecture slides, Carnegie Mellon University, 2020. URL `https://www.cs.cmu.edu/~mgormley/courses/10601-s20/slides/lecture23-rl.pdf`. Accessed: 19 Mar. 2025.

[2] Yichen Qiu. Comparative analysis of value iteration and policy iteration in robotic decision-making. *Applied and Computational Engineering*, 83:140–147, 10 2024. doi: 10.54254/2755-2721/83/2024GLG0073.

[3] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 2nd edition, 2020.