ECE 493 Assignment 2 Report

Akshay Patel | Student ID: 20732500

Unless otherwise noted, all figures and algorithm explanations are from the second edition Reinforcement Learning textbook used for this course, written by Richard S. Sutton and Andrew G. Barto.

Domain Description

Maze World is the deterministic environment that was used for this assignment. States, actions, and dynamics are defined as follows. States are defined as the agent being in a specific cell of the maze, and they are represented by four coordinates. Tasks can have walls that the agent cannot pass through, pits that are terminal states where the agent receives a large negative reward, and all tasks have a goal state for which the agent receives a positive reward. The action space in this environment consists of {up, down, right, left} represented by integers {0, 1, 2, 3} respectively. The dynamics of this environment are that given a state *s*, and an action *a*, taking action *a* from a state *s* will result in the agent being in a state *s'*. However, if the action takes the agent into a wall or takes the agent out of the grid, then the state will remain the same. If the agent reaches the goal or runs into a pit, then the game ends as those are terminal states. The reward function is that reaching the goal gives you a reward of +1 and ends the episode, running into a pit yields a penalty of -10 and ends the episode, running into a wall will give you a penalty of -0.3, and apart from these, the agent will receive a -0.1 reward for reaching any other cell in the grid.

Maze World is deterministic, meaning that if the agent takes a specific action from a specific state, it will always go to the same next state as well as receive the same reward value. The objective of this environment given this reward function is to reach the goal state as quickly as possible to maximize the reward.

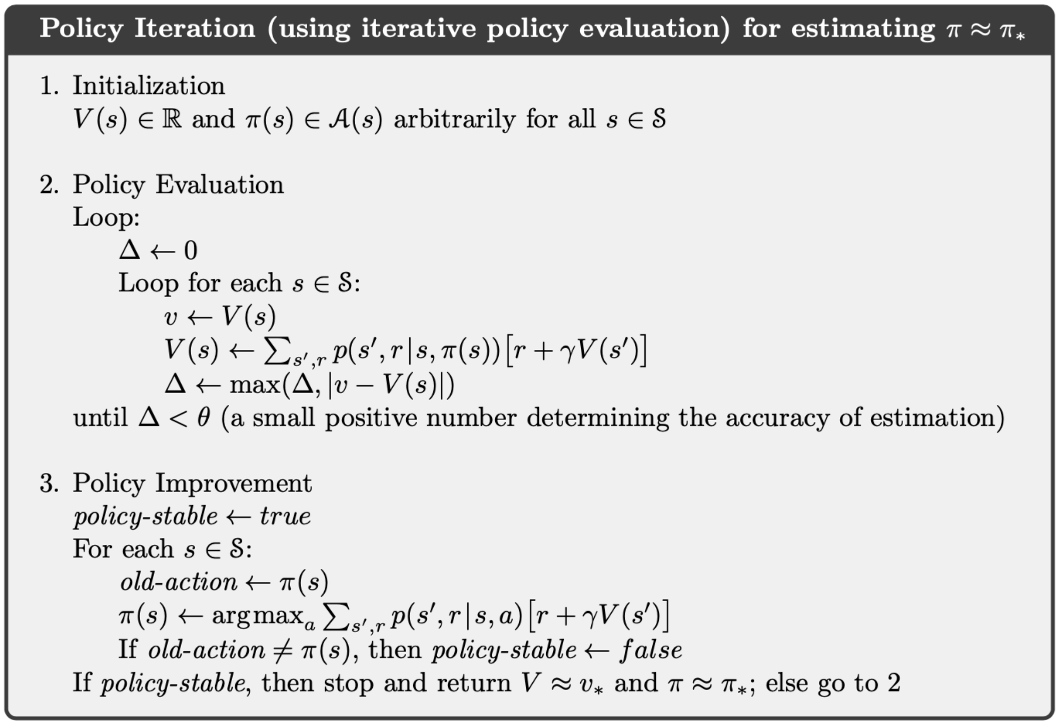
Algorithm Explanations

Asynchronous Policy Iteration

Policy Iteration is an algorithm that helps you find an optimal policy through iterative improvement. This is done through using the policy and the value function together to improve both of them, eventually leading to a sequence of monotonically improving policies and value functions that will bring us closer to the optimal policy after each iteration. Policy iteration will modify the existing policy at each iteration and compute the value function according to this new policy until the policy converges. Policy iteration is guaranteed to converge to the optimal policy, and compared to the next algorithm, value iteration, it often takes less iterations to converge.

Policy iteration consists of two main components, namely policy evaluation and policy improvement. Policy evaluation is where you evaluate and existing policy and update the value function based on that. After evaluating an existing policy, you can then improve the policy by using the updated value function from policy evaluation to push the policy towards the optimal policy. Once the policy is no longer changing, and thus is stable, we can stop trying to improve the policy to save computational resources.

Some sample pseudocode for policy iteration, including policy evaluation, policy improvement, and Bellman equation updates is given below.



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 $s \in \mathcal{S}$

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

In my implementation, I used an asynchronous variant of policy iteration instead of the one shown above. I do not sweep the entire state space multiple times before interacting with the environment and improving my policy. Instead, I sweep the states that the agent has already visited so far during each step. In my asynchronous variant of policy iteration (which I learned of from https://ocw.mit.edu/courses/aeronautics-and-astronautics/16-410-principles-of-autonomy-and-decision-making-fall-2010/lecture-notes/MIT16_410F10_lec23.pdf slide 19), at each step, I will alternate between performing policy evaluation and policy improvement. I keep repeating this until my policy is stable, after which I do not evaluate my policy or improve it to limit wasteful use of computational resources as the policy does not change. In the case of a very bad initial policy, to prevent the agent from being stuck in an episode, I use an epsilon-greedy action-selection method (with epsilon = 0.1) to encourage exploration when the agent is inexperienced. However, as the agent becomes more and more experienced and reaches a stable policy, I do not use epsilon as the policy is stable and will give the agent the best move to make whereas epsilon in later changes might encourage suboptimal behaviour.

Asynchronous Value Iteration

Value iteration is an algorithm developed from truncating the policy evaluation step of policy iteration without losing the convergence guarantees of policy iteration. The value iteration algorithm is obtained simply by turning the Bellman optimality equation given below into an update rule.

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s', r} p(s', r \mid s, a) \Big[r + \gamma v_*(s') \Big], \end{aligned}$$

We can also see that the value iteration update is the same as the policy evaluation update with the exception that the value iteration update needs the maximum value to be taken over all actions. Value iteration terminates formally after an infinite number of iterations to converge to the exact optimal value function. In practice however, we can terminate the function when it is not changing at all or by a significant amount anymore.

Sample pseudocode for value iteration is given below for estimating the optimal policy.

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

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

In my implementation, I used asynchronous value iteration which consists of only updating a single state's value in each step made by the agent instead of having to perform full sweeps of the entire state space. As a result of this, asynchronous value iteration can save computational resources per step, and it also allows you to work towards an optimal policy by improving it iteratively while simultaneously interacting with the environment. This can also prevent the agent being hopelessly stuck in an episode due to a suboptimal policy being used for the entirety of that episode. Moreover, to encourage more exploration in the beginning of the agent's experience, I use an epsilon-greedy action-selection mechanism (with epsilon = 0.1). However, as the agent completes more episodes and thus improves its value function and policy, I use an epsilon decay mechanism which discourages exploration in later stages to prevent suboptimal actions. After each episode, I will decrease the value of epsilon by 1% of the current value.

State Action Reward State Action (SARSA)

Sarsa is an on-policy TD control algorithm that follows the pattern of generalized policy iteration but uses TD methods for the evaluation/prediction parts. Sarsa uses an action-value function and it is model-free. Unlike dynamic programming methods, Sarsa does not require explicit knowledge of the environment. Pseudocode for the Sarsa algorithm is given below.

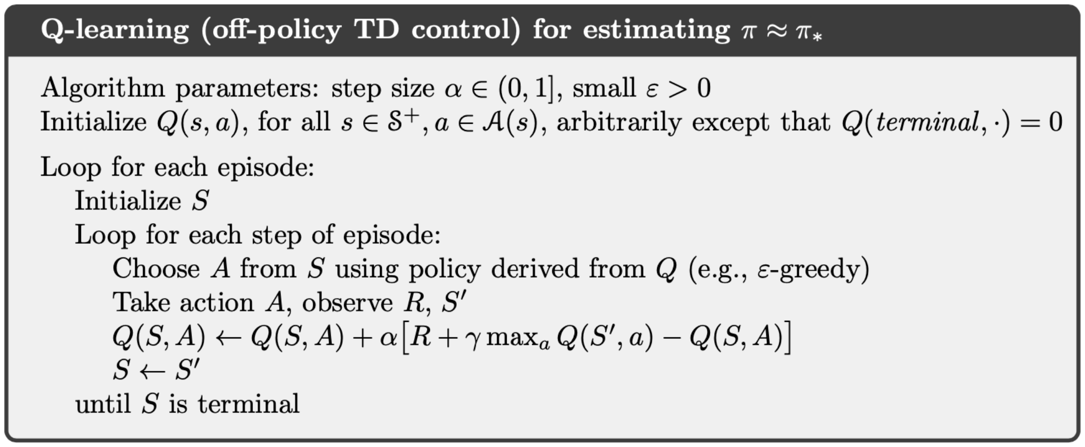
```
Sarsa (on-policy TD control) for estimating $Q \approx q_*$

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
```

Sarsa is an on-policy algorithm because it updates its Q-values using the Q-value of the next states and the greedy actions from those next states. Sarsa estimates the return for state-action pairs assuming that the current policy continues to be followed. Sarsa updates its Q-values using the Q-value of the next state *s'* and the current policy's action *a'* (explanation for this was from Statistics StackExchange). My implementation is essentially the same as the pseudocode above.

Q-Learning

Q-Learning is an off-policy TD control method where the learned action-value function Q, approximates the optimal action-value function independent of the policy being followed. Like Sarsa, Q-Learning is also model-free and does not require explicit knowledge of the domain/environment. The reason Q-Learning is referred to as off-policy is because it updates its Q-values using the Q-value of the next state *s'* and the greedy action *a''* estimating the total discounted future reward for state-action pairs assuming a greedy policy is being followed even thought it's not following a greedy policy (same source from StackExchange as above).

The pseudocode for Q-Learning is given below, with the appropriate updates for the action-value function. My implementation is essentially the same as the pseudocode below.

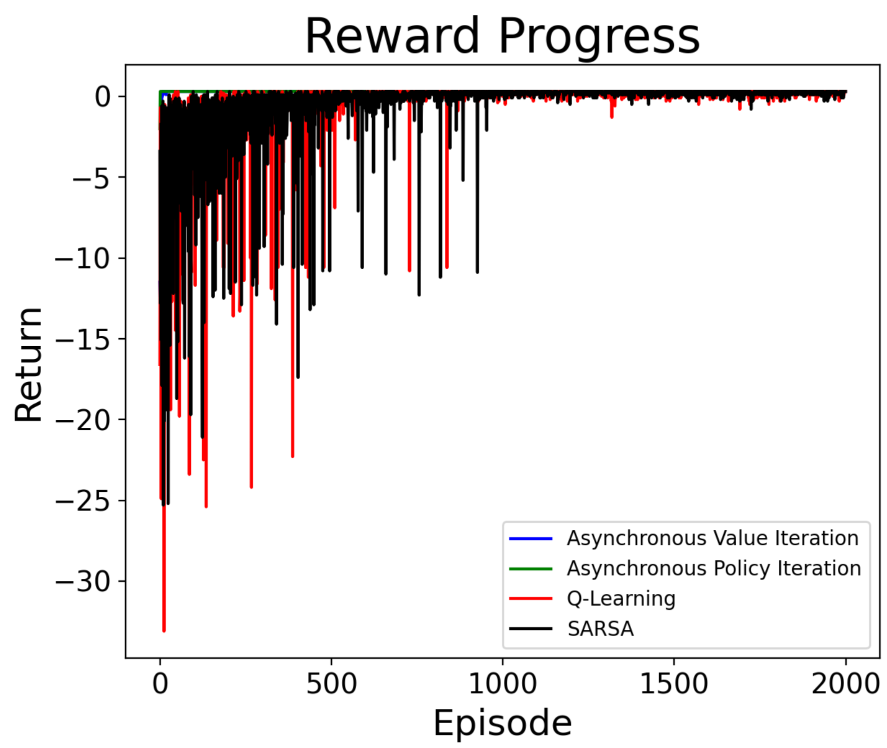


Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

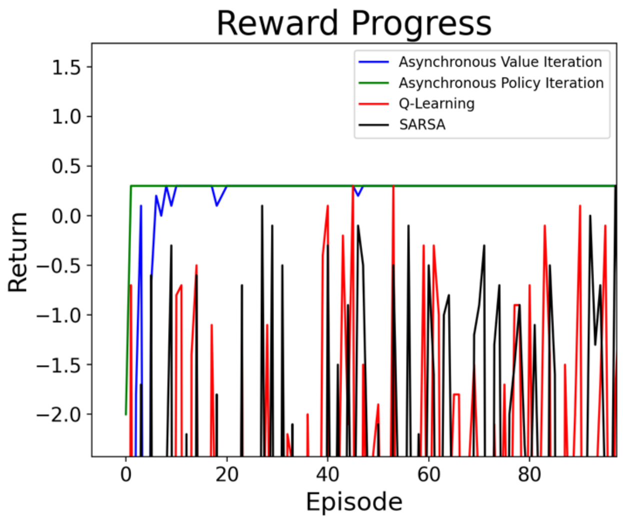
```
Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
    Initialize $S$
    Loop for each step of episode:
        Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-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
```

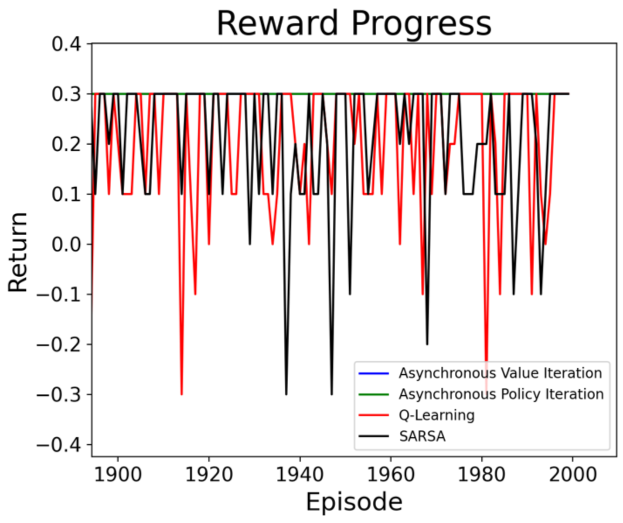
Results and Analysis

Task 1



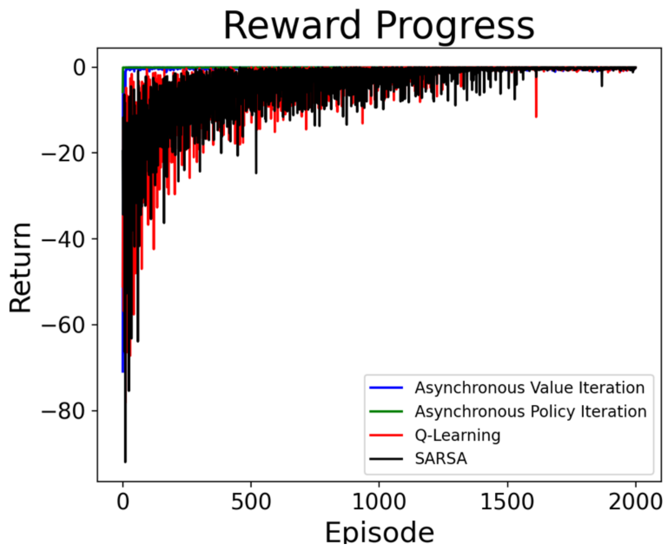
As you can see in the plot above, all 4 of the algorithms perform quite well in Task 1, with all of them converging in the later episodes. All four algorithms converge in order to obtain the maximum reward of +0.3 from the environment.



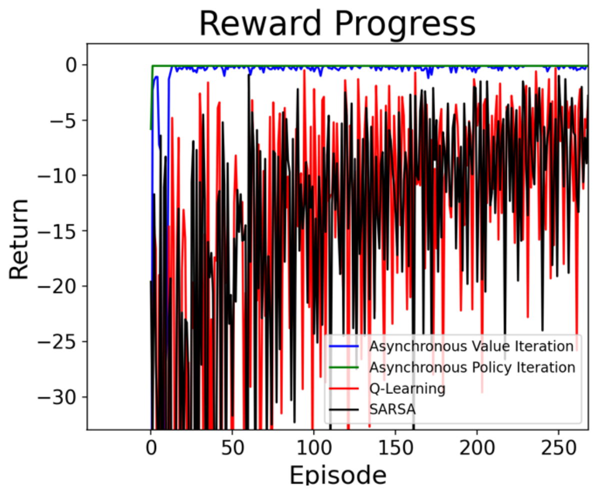


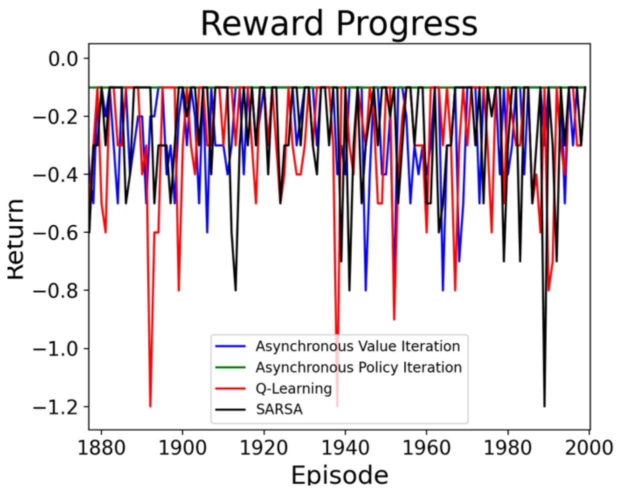
On the left, you can see the episode vs reward values of the 4 algorithms in the beginning of Task 1. We can see that Policy Iteration and Value Iteration converge after very few episodes compared to Q-Learning and Sarsa which do not converge until after approximately 1000 episodes. On the right, you can see the values of the 4 algorithms at the end of Task 1. We see that the dynamic programming methods in which I do not use the epsilon-greedy action-selection mechanism in the later episodes follow the optimal policy consistently. This is because my policy iteration algorithm follows the optimal policy every time after it is stable and my value iteration algorithm decreases epsilon after each episode to reduce suboptimal actions. If I did not decrease the use of epsilon action-selection in the DP methods, then we would see a lot more suboptimal actions being taken in the later episodes. In my Sarsa and Q-Learning implementations, I also have an epsilon-greedy action-selection mechanism, but the epsilon stays constant throughout all episodes, and so they sometimes take suboptimal actions even in the later episodes, resulting in the rewards obtained having a higher variance.

Task 2



As you can see in the plot above, all 4 of the algorithms perform quite well in Task 2, with all of them converging in the later episodes. All four algorithms converge in order to obtain the maximum reward of -0.1 from the environment. However, Sarsa and Q-Learning take much longer to converge compared to the dynamic programming algorithms.

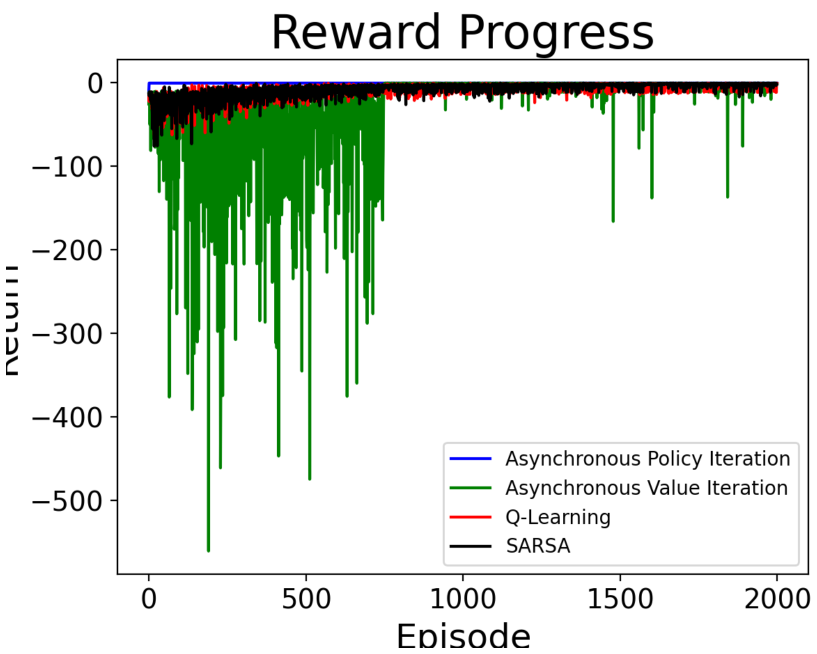




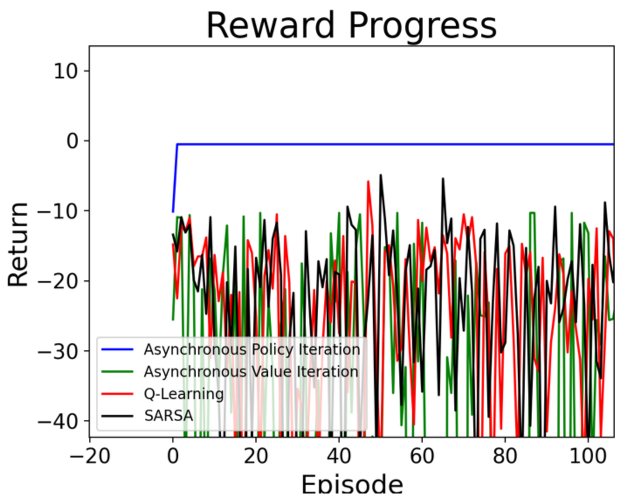
On the left, you can see the episode vs reward values of the 4 algorithms in the beginning of Task 2. We can see that Policy Iteration and Value Iteration converge after very few episodes compared to Q-Learning and Sarsa which do not converge until after approximately 1500 episodes. On the right, you can see the values of the 4 algorithms at the end of Task 2. We see that the dynamic programming methods in which I do not use the epsilon-greedy action-selection mechanism as much in the later episodes follow the optimal policy more consistently compared to Q-Learning and Sarsa, which take more suboptimal actions because of the constant epsilon values. We can see some interesting behaviour as a result of the epsilon values in the dynamic programming algorithms. With policy iteration, epsilon is essentially 0 after the stable policy is obtained. As we can see in the beginning and at the end of Task 2, policy iteration always takes the optimal action. Value iteration however still has an epsilon value, albeit very small, at the end of Task 2, which likely results in the inconsistency compared to policy iteration. It could also be the case that value iteration hasn't completely reached the optimal value function in the 2000 episodes. Increasing value iteration's epsilon decay factor would likely result in a more consistent behaviour at the end of the Task.

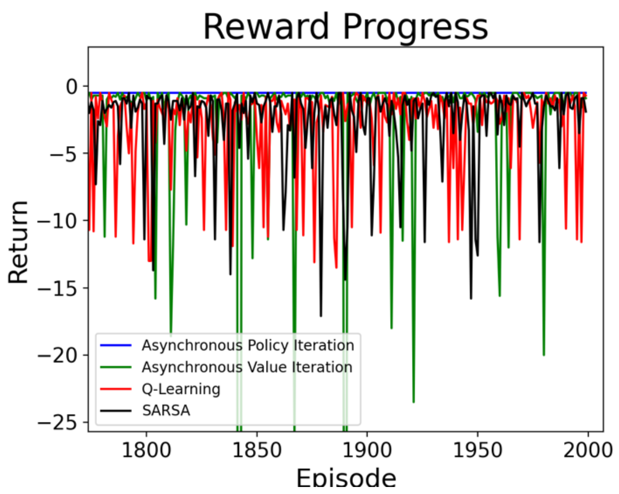
Another interesting thing I noticed as a result of Task 2 (and Task 3) with policy and value iteration is the importance of the epsilon value not being too small. If the epsilon value is too small in the earlier iterations, resulting in very little exploration, then there is a high possibility that the agent could become stuck in repetitive actions due to a very bad policy being used. Thus, an epsilon value of at least 0.1 is recommended with these implementations in order to prevent the aforementioned issue through exploration.

Task 3



As you can see in the plot above, all 4 of the algorithms perform quite well at the end of Task 3, with all of them converging in the later episodes even though this is probably the most challenging task out of the three tasks in this assignment. All four algorithms converge in order to obtain the maximum reward of -0.5 from the environment. However, Value Iteration, Sarsa and Q-Learning take much longer to converge compared to Policy Iteration.





On the left, you can see the episode vs reward values of the 4 algorithms in the beginning of Task 2. We can see that Policy Iteration converges after very few episodes compared to Q-Learning, Sarsa, and Value Iteration which do not converge until later episodes. On the right, you can see the values of the 4 algorithms at the end of Task 2. We see that the Policy Iteration in which I do not follow the epsilon actions in later episodes and just follow the optimal policy behaves more consistently compared to Value Iteration, Q-Learning, and Sarsa, which take more suboptimal actions because of the constant epsilon values. Interestingly, there seems to be more variance in the rewards of Value Iteration than Q-Learning and Sarsa, even though I have epsilon decay in Value Iteration. This seems to imply that perhaps value iteration hasn't fully converged in the 2000 episodes, or maybe it just doesn't do as well on this challenging Task 3.

Something very weird we see in the overall plot for Task 3 with Value Iteration is the sudden improvement in rewards per episode after about 750 episodes. I am not exactly sure what caused this though.

Overall Algorithm Comparison

Although all of the algorithms perform very well on all three Maze World tasks above, there are definitely advantages to some of them. The model-free TD algorithms, Q-Learning and Sarsa, provide great flexibility in their use because they do not require any specific information from the environment other than the rewards and states, and they are quite simple to implement. Also, they do not require sweeping the entire state space like the (non-asynchronous) DP algorithms. The DP algorithms however require information from the environment like the dynamics and they require sweeping the entire state space. In domains with a very large state space and action space, the DP algorithms would likely require much more computational power per episode and information to obtain the optimal policies compared to the model-free algorithms, and so in such environments, Q-Learning or Sarsa is preferred over Policy Iteration or Value Iteration. In smaller domains however, the DP algorithms can quickly obtain the optimal policy faster than the model-free algorithms, so in these environments, DP algorithms, specifically Policy Iteration since it converges in fewer iterations compared to Value Iteration, are preferred for maximizing total reward over all episodes. In a case where you need to minimize computational resource usage per episode in a small domain in which DP methods could excel, then Value Iteration is the preferred choice as it performs less operations per episode.

Between Sarsa and Q-Learning, I prefer Sarsa based on the data we see above in the various plots. Sarsa converges just as fast, if not faster, than Q-Learning, while having a lower variance in the rewards at any given range of episodes we see above compared to Q-Learning. Also, Sarsa performs less operations per step compared to Q-Learning which uses the max of all action-values.

Something I would have liked to measure but was not able to is the amount of CPU Usage per algorithm and the amount of time in seconds that each algorithm took to complete the 2000 episodes per task. This information could have been very insightful in comparing the algorithms' performance from a compute-based perspective.