ECE493 Assignment 2 Report

Akshay Patel | 20732500

Note that all figures are from the RL Textbook.

For each of the following algorithms with respect to Maze World, states are defined as the agent being in a specific cell of the maze. Some tasks may have walls which the agent cannot go to, and other tasks can also have pits which are also terminal states like the goal state, but the pits punish the agent. The action space consists of 4 actions, namely up, right, down, left, represented by integers from 0 to 3 inclusive.

Value Iteration (Asynchronous)

Value Iteration is an algorithm that is 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 (see below)

$$\begin{aligned} v_*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= \max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_*(s')\Big], \end{aligned} \tag{4.1}$$

into an update rule. 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 to be taken over all actions. Value iteration terminates formally after an infinite number of iterations to converge to the exact optimal value function. However, in practice, we can terminate the function when it is not changing by much anymore.

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
    $\pi(s) = \mathrm{argmax}_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
```

Above, you can see pseudocode for the value iteration algorithm. I used the asynchronous value iteration alternative, which consists of only updating a single state's value in each step made by the agent. Instead of having to do full sweeps of the entire state space, the asynchronous version allows you to save computational resources per step, and it also allows you to work towards a policy while simultaneously stepping through the environment. This allows for the value function, and thus the policy function, to be improved iteratively throughout the steps the agent takes instead of being hopelessly stuck in an episode due to a not-so-optimal policy.

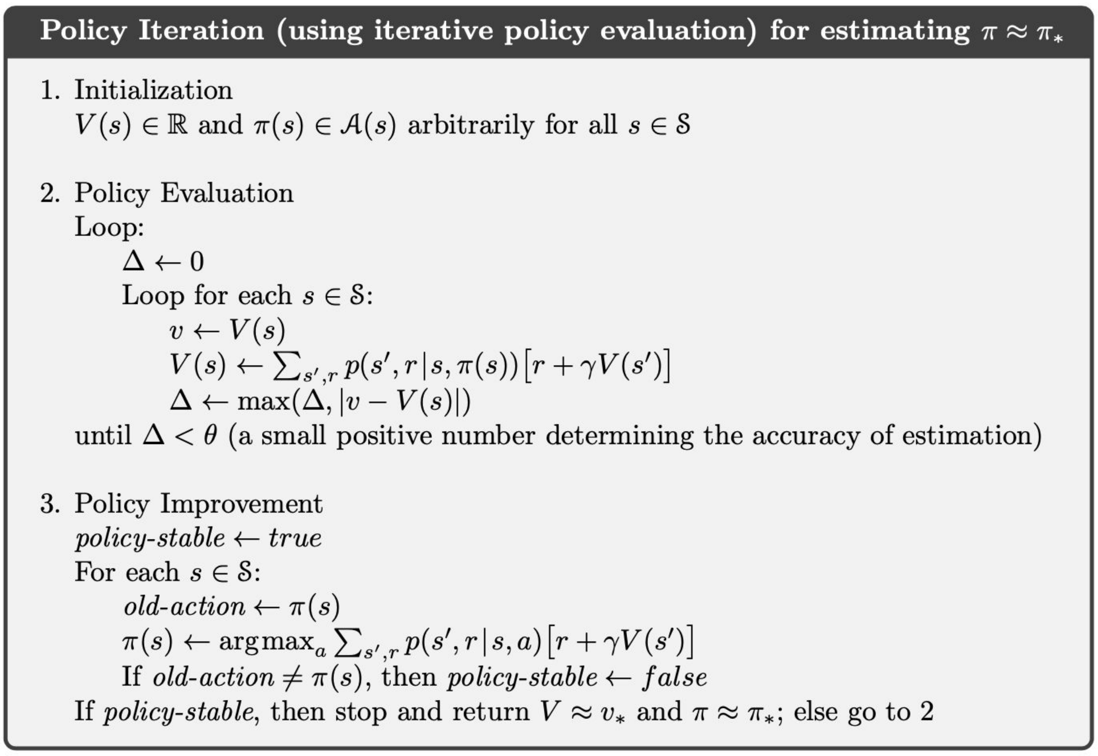
My implementation still uses the same deterministic policy from above. The only difference is that my policy is changing every step and getting better each step. Another slight difference is that I use an epsilon-greedy method to choose the next action to encourage exploration.

Policy Iteration (Asynchronous)

Policy Iteration is an algorithm which helps you find an optimal policy through iterative improvement. This is done through using the policy and the state-value function together to improve both of them, eventually leading to a sequence of monotonically improving policies and value functions that will lead us to the optimal policy!

Policy Iteration consists of two main components, namely Iterative Policy Evaluation, and Policy Improvement. Policy Evaluation allows you to evaluate an existing policy and update a 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!

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



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

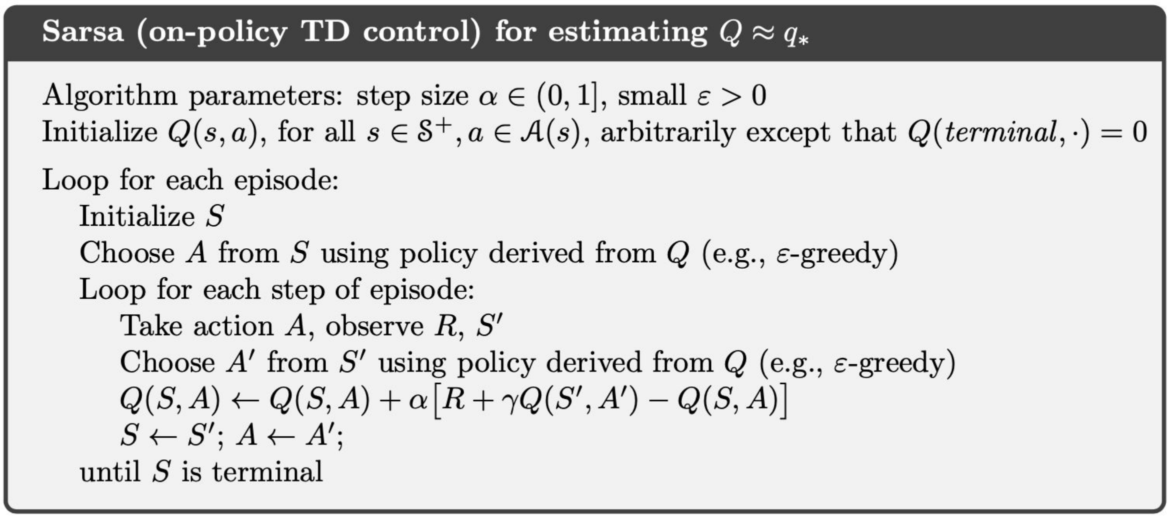
```
1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

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

3. Policy Improvement
   $policy\text{-}stable \leftarrow true$
   For each $s \in \mathcal{S}$:
      $old\text{-}action \leftarrow \pi(s)$
      $\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
      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
```

I used the asynchronous alternative of policy iteration, and so instead of doing a full policy evaluation and a full policy improvement, I only update the value function and policy for the state that the agent was most recently in using the reward it received. I do the exact same Bellman equation update as in Policy Evaluation and the same argmax behaviour for the Policy Improvement.

<u>SARSA</u>

Sarsa is an on-policy TD control algorithm that follows the pattern of generalized policy iteration but uses TD methods for the evaluation/prediction part. Sarsa uses an action-value function instead of a state-value function, and it is model-free! Sarsa does not require explicit knowledge of the environment, unlike dynamic programming methods.

The pseudocode for Sarsa can be seen 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 $Q(\textit{terminal}, \cdot) = 0$

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
```

My implementation of Sarsa is essentially the same as the pseudocode above.

<u>Q-Learning</u>

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. That is why it's referred to as off-policy as well. Like Sarsa, Q-Learning is model-free! It does not require explicit knowledge of the environment, unlike DP methods.

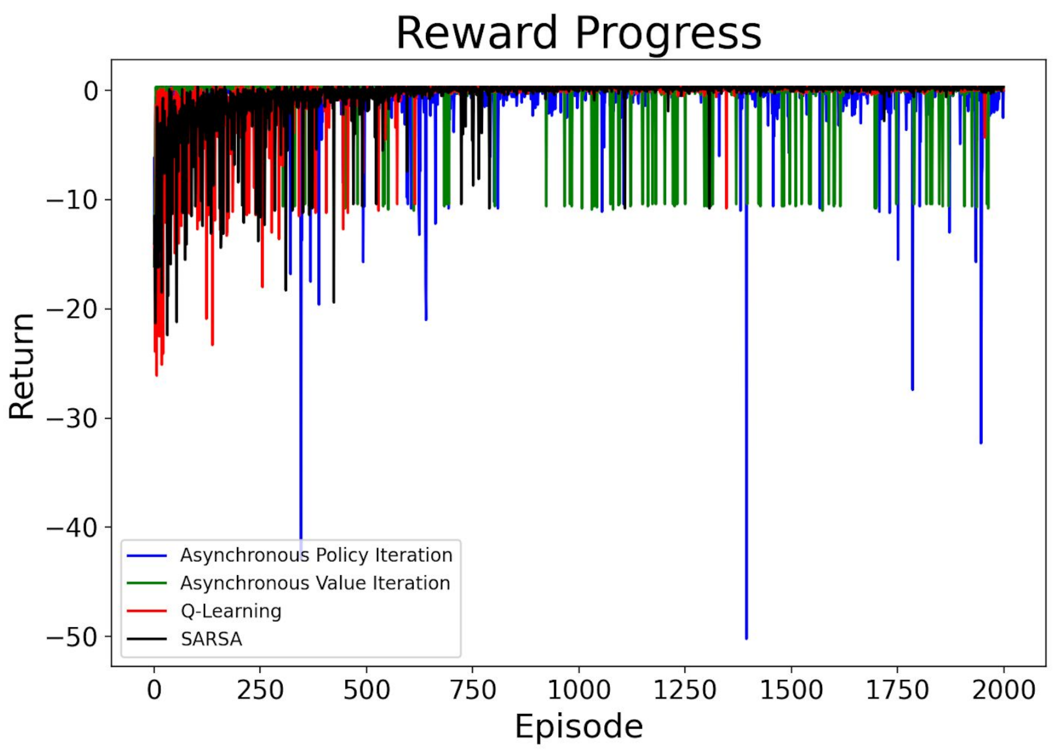
The pseudocode for Q-learning is given below, with the appropriate updates for the action-value function.

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
```

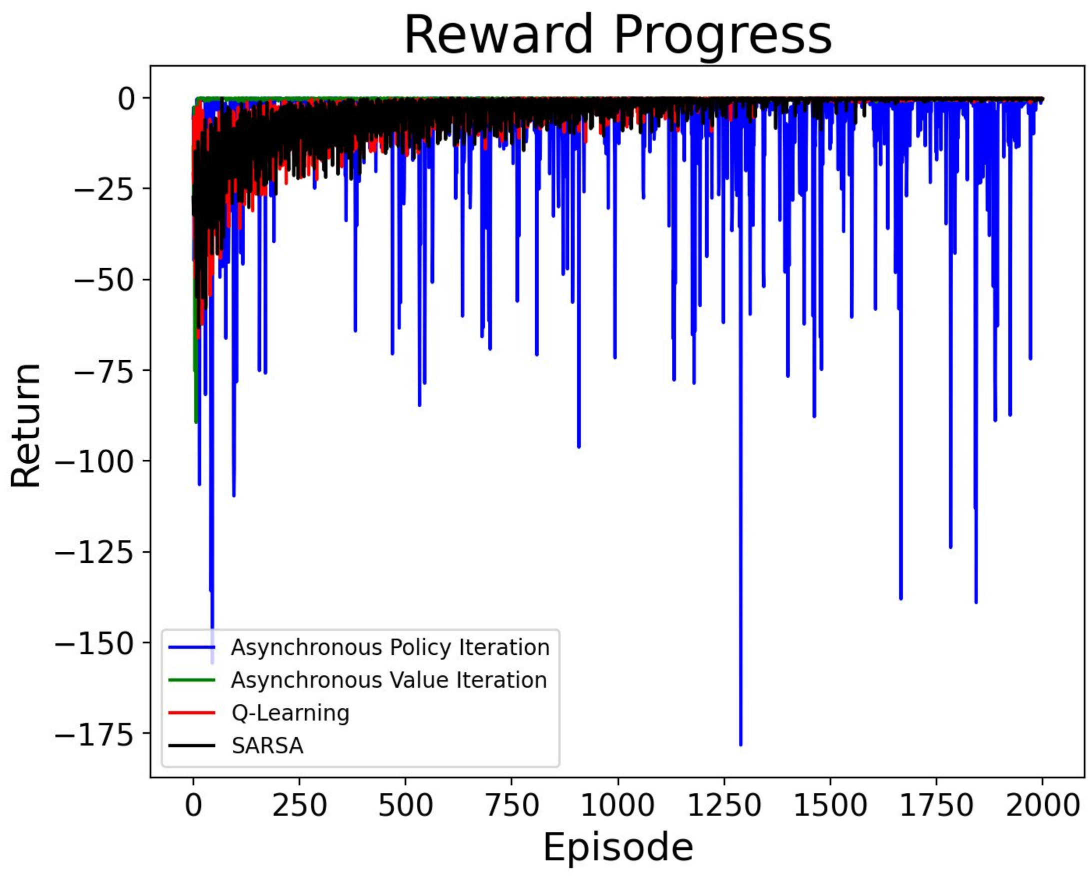
Results and Analysis

Task 1



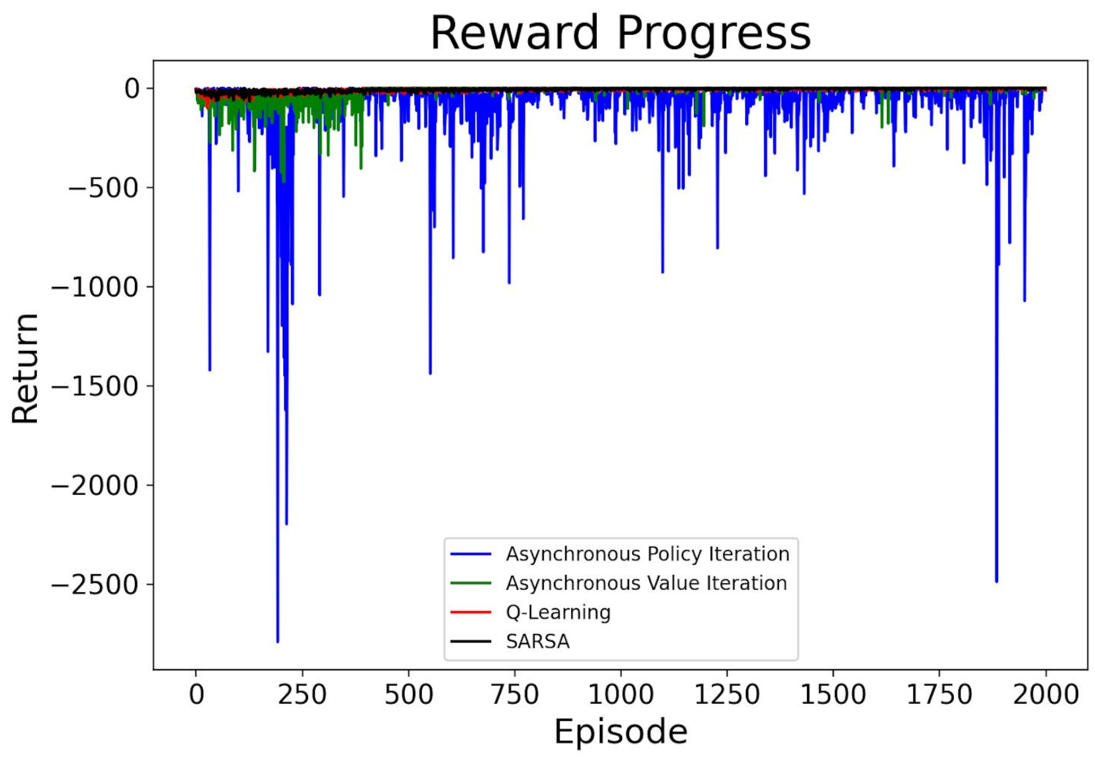
As you can see in the plot above, all of the 4 algorithms do a fairly good job at learning which path is best to take towards the goal state to maximize reward. However, we must note that Task 1 in Maze World is a bit tricky, because one of the optimal paths to the goal state passes by a pit, in which DP methods like Policy and Value Iteration sometimes fall, possibly due to the epsilon-greedy action selection. Sarsa and Q-Learning perform very well, and they do not fall victim to the same mistakes as the DP methods make by falling into the pit after a high number of episodes.

<u>Task 2</u>



As you can see above, all of the algorithms except for Asynchronous Policy Iteration perform very well on this task of Maze World. Asynchronous Policy Iteration does not seem to converge, and so perhaps the interaction between the policy and the state-value functions is not occurring correctly leading to the convergence expected. Q-Learning, Sarsa, and Asynchronous Value Iteration perform quite similarly at the later episodes. Asynchronous Value Iteration seems to converge a little faster however.

Task 3



Similar to Task 2, Asynchronous Policy Iteration does not converge due to what is most likely an implementation error. The other 3 algorithms, Q-Learning, Sarsa, and Asynchronous Value Iteration perform very well on this task of Maze World. Interestingly though, Asynchronous Value Iteration takes longer than the model-free algorithms to converge. Out of the three tasks, this third one is definitely the most challenging due to the wall placement and the pits. In this challenging task, the model-free algorithms were able to perform quite well right from the beginning of the 2000 episodes.

Further Analysis

Although Q-Learning, Sarsa, and Asynchronous Value Iteration all perform quite well on the three Maze World tasks above (Asynchronous Policy Iteration did not converge in Task 2 or Task 3 due to most likely an implementation error), I most definitely prefer the model-free TD methods due to ease of implementation and the lack of requiring specific information from the environment to learn. Moreover, these model-free TD methods also require much less computational resources compared to the DP methods which can result in sweeping the entire state space multiple times to acquire an optimal policy.

When I refer to computational resources though, I’m inferring from the number of floating point operations required to compute the necessary information for updates in each algorithm. I was not able to measure the amount of CPU Usage or the amount of time in seconds that each algorithm took to complete the 2000 episodes for each task, but this information would have been very insightful in comparing the algorithms’ performance from a compute-based perspective.