University of Waterloo
Faculty of Engineering
Department of Electrical and Computer Engineering

Assignment 2 Report

Maze World

ECE 493 T25
University of Waterloo

Prepared by
Zhidong Zhang
20619543
z498zhan@edu.uwaterloo.ca
4A Computer Engineering

1 August 2019

Table of Contents

1 Algorithms

This section contains the introduction of four algorithms that I used in this maze world assignment.

1.1 SARSA

In the SARSA algorithm, the state is "X, Y", the action is 'u', 'd', 'l', 'r'. SARSA is an on-policy Temporal Difference (TD) control algorithm. Instead of learning the state-value function, SARSA learns the action-value function. It starts off in a state-action pair (S, A) and ends up in a new state S'. Then, it applies the epsilon-greedy policy on the state-action values for S' to choose the next action A'. Next, it updates the state-action value Q(S, A) by applying SARSA update, which is $\mathbf{Q(S,A)} \leftarrow \mathbf{Q(S,A)} + \boldsymbol{\alpha}\,[\boldsymbol{R} + \boldsymbol{\gamma Q(S',A')} - \boldsymbol{Q(S,A)}]$. More specifically, based on next action A' and next state S', it calculates the TD target which is $R + \gamma Q(S', A')$ by adding up the reward of the current action A and a decayed state-action value Q(S', A'). Then, it calculate the TD error by subtracting old estimated value Q(S, A) from the TD target, which is $R + \gamma Q(S', A') - Q(S, A)$. Next, it applies the SARSA update to the old estimated Q (S, A) by multiplying TD error and the learning rate α and add it to the old estimated Q (S, A). After applying the SARSA update, it updates the current state S to the next state S' and the current action A to the next action A'. And then, it goes into the next iteration and do the bellman update iteratively. Finally, the state-action value Q will converge to the optimal policy.

```
def learn(self, s, a, r, s_):
    self.check_state_exist(s_)

    if s_ != 'terminal':
        a_ = self.choose_action(str(s_)) # argmax action
        q_target = r + self.gamma * self.q_table.loc[s_, a_] # max state-action value
    else:
        q_target = r  # next state is terminal

    self.q_table.loc[s, a] = self.q_table.loc[s, a] + self.lr * (q_target - self.q_table.loc[s, a])

    return s_, a_
```

1.2 Q-Learning

In the Q-Learning algorithm, the state is "X, Y", the action is 'u', 'd', 'l', 'r'. Q-Learning is an off-policy Temporal Difference (TD) control algorithm. Since it is an off-policy algorithm, it does not follow the policy; instead, it directly approximates the optimal state-action value. It starts in a state-action pair (S, A) and ends up in a new state S'. Instead of using epsilon-greedy policy, it directly chooses the best possible state-action value for state S' by just doing a max. Next, it updates the state-action value Q(S, A) by applying Bellman update, which is $\mathbf{Q(S,A)} \leftarrow \mathbf{Q(S,A)} + \boldsymbol{\alpha}\,[\boldsymbol{R} + \boldsymbol{\gamma}\, \boldsymbol{max_a Q(S',a)} - \boldsymbol{Q(S,A)}]$. More specifically, based on next action A' and next state S', it

calculates the TD target which is $R + \gamma max_a Q(S', a)$ by adding up the reward of the current action A and the maximum state-action value Q(S', a). Then, it calculate the TD error by subtracting old estimated value Q(S, A) from the TD target, which is $R + \gamma max_a Q(S', a) - Q(S, A)$. Next, it applies the Bellman update to the old estimated Q (S, A) by multiplying TD error and the learning rate α and add it to the old estimated Q (S, A). After applying the Bellman update, it updates the current state S to the next state S' and the current action A to a new action chosen using epsilon-greedy policy on the state-action values for S'. And then, it goes into the next iteration and do the bellman update iteratively. Finally, the state-action value Q will converge to the optimal policy.

```
def learn(self, s, a, r, s_):
    self.check_state_exist(s_)

    if s_ != 'terminal':
        q_target = r + self.gamma * self.q_table.loc[s_, :].max() # max state-action value
    else:
        q_target = r  # next state is terminal

    self.q_table.loc[s, a] = self.q_table.loc[s, a] + self.lr * (q_target - self.q_table.loc[s, a])

    return s_, self.choose_action(str(s_))
```

1.3 Expected SARSA

Expected SARSA is similar to Q-Learning, expected that instead of choosing the maximum of the Q(S', a) it uses the expected value of Q(S', a) for all the actions. The Bellman update rule is as follows:

$$\mathbf{Q(S, A)} \leftarrow \mathbf{Q(S, A)} + \boldsymbol{\alpha}\,[\boldsymbol{R} + \boldsymbol{\gamma}\, \boldsymbol{E}[\boldsymbol{Q(S', A')} \mid \boldsymbol{S'}] - \boldsymbol{Q(S, A)}]$$

$$\leftarrow \mathbf{Q(S, A)} + \boldsymbol{\alpha}\,[\boldsymbol{R} + \boldsymbol{\gamma}\, \textstyle\sum_{\boldsymbol{a}} \boldsymbol{\pi(a \mid S')}\, \boldsymbol{Q(S', a)} - \boldsymbol{Q(S, A)}]$$

Now, we have the Bellman update rule, the next step is to determine the policy and its probabilistic distribution. Since we are using epsilon-greedy policy, the probabilistic distribution is as follows:

$$\boldsymbol{\pi(a \mid s)} = \begin{cases} \frac{\epsilon}{m} + 1 - \epsilon & \boldsymbol{if\ a^* = argmax_a\, Q(s, a)} \\ \frac{\epsilon}{m} & \boldsymbol{otherwise} \end{cases}$$

Based on the probability for each action, we can calculate the expected state-action value for state S' for all actions by multiplying the state-action value with its probability and sum them up.

The algorithm starts in a state-action pair (S, A) and ends up in a new state S'. Instead of using the epsilon-greedy policy or choosing the max Q value, it calculates the expected value for the next state-action value using the method mentioned above. Next, it updates the state-action value Q(S, A) by applying Bellman update, which is $\mathbf{Q(S, A)} \leftarrow$

$\mathbf{Q(S, A)} + \boldsymbol{\alpha}\,[\boldsymbol{R} + \boldsymbol{\gamma}\,\boldsymbol{E}[\boldsymbol{Q}(\boldsymbol{S}', \boldsymbol{A}') \mid \boldsymbol{S}'] - \boldsymbol{Q}(\boldsymbol{S}, \boldsymbol{A})]$. More specifically, based on next action A' and next state S', it calculates the TD target which is $R + \gamma\, E[Q(S', A') \mid S']$ by adding up the reward of the current action A and the expected value of the next state-action value. Then, it calculates the TD error by subtracting old estimated value Q(S, A) from the TD target, which is $R + \gamma\, E[Q(S', A') \mid S'] - Q(S, A)$. Next, it applies the Bellman update to the old estimated Q (S, A) by multiplying TD error and the learning rate α and add it to the old estimated Q (S, A). After applying the Bellman update, it updates the current state S to the next state S' and the current action A to a new action chosen using epsilon-greedy policy on the state-action values for S'. And then, it goes into the next iteration and does the bellman update iteratively. Finally, the state-action value Q will converge to the optimal policy.

```
def learn(self, s, a, r, s_):
    self.check_state_exist(s_)
    q_current = self.q_table.loc[s, a]

    if s_ != 'terminal':
        # calculate expected value according to epsilon greedy policy
        state_action_values = self.q_table.loc[s_,:]
        value_sum = np.sum(state_action_values)
        max_value = np.max(state_action_values)
        max_count = len(state_action_values[state_action_values == max_value])
        k = len(self.actions) # total number of actions

        expected_value_for_max = max_value * ((1 - self.epsilon) / max_count + self.epsilon / k) * max_count
        expected_value_for_non_max = (value_sum - max_value * max_count) * (self.epsilon / k)

        expected_value = expected_value_for_max + expected_value_for_non_max

        q_target = r + self.gamma * expected_value # max state-action value
    else:
        q_target = r  # next state is terminal

    self.q_table.loc[s, a] += self.lr * (q_target - q_current)  # update current state-action value

    return s_, self.choose_action(str(s_))
```

1.4 SARSA(λ)

SARSA(λ) is SARSA with eligibility trace. In SARSA, after doing the one-step look ahead, it only propagates information back by one step. However, in SARSA lambda, it builds up eligibility along the trajectory. For each state, it proper gates the information all the way to the starting point with proportion to the eligibility trace, which allows faster flow of information backward through time.

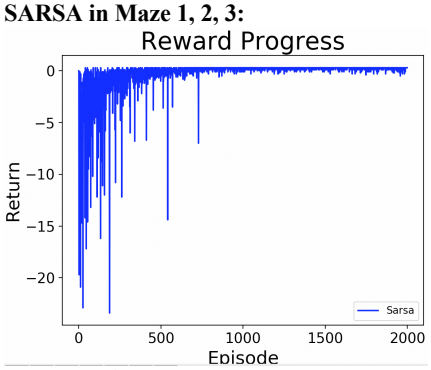
For each episode, initialize the eligibility traces for each state-action pair to 0. It starts in a state-action pair (S, A) and ends up in a new state S'. Then, it applies the epsilon-greedy policy on the state-action values for S' to choose

the next action A'. Next, it computes the TD error δ, by subtracting the old estimate Q (S, A) from the TD target ($\boldsymbol{R} + \boldsymbol{\gamma Q}(\boldsymbol{S'}, \boldsymbol{A'})$). After that, it increments the eligibility trace E (S, A) by 1. The next step is to do the SARSA update with proportion to the eligibility trace for all the states and actions. This SARSA update is $\mathbf{Q}(\mathbf{S}, \mathbf{A}) \leftarrow \mathbf{Q}(\mathbf{S}, \mathbf{A}) + \boldsymbol{\alpha}\, \boldsymbol{\delta}\, \mathbf{E}(\mathbf{S}, \mathbf{A})$. Then, it decays the eligibility trance value for all state and actions. Finally, it updates the current state S to the next state S' and the current action A to the next action A' and goes to the next iteration. In the end, the state-action value Q will converge to the optimal policy.

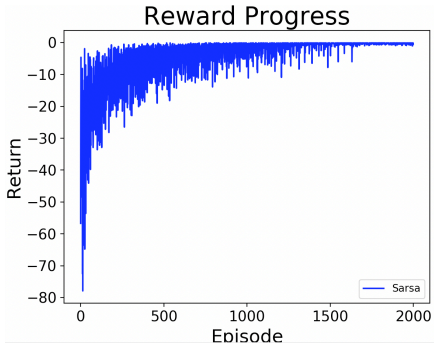
```python
def learn(self, s, a, r, s_):
    self.check_state_exist(s_)

    # determine q_target
    if s_ != 'terminal':
        a_ = self.choose_action(str(s_)) # argmax action
        q_target = r + self.gamma * self.q_table.loc[s_, a_] # max state-action value
    else:
        q_target = r  # next state is terminal

    # update q_table using eligibility trace
    error = q_target - self.q_table.loc[s, a]
    self.e_table.loc[s, a] += 1

    self.q_table += self.lr * error * self.e_table # update state-action value for all states and actions

    # update eligibility trace
    if s_ != 'terminal':
        self.e_table *= self.gamma * self.lambda_decay # decay the eligibility trace for all states and actions
    else:
        self.e_table = pd.DataFrame(columns=self.actions, dtype=np.float64) # clear the eligibility

    return s_, a_
```

2 Quantitative Analysis

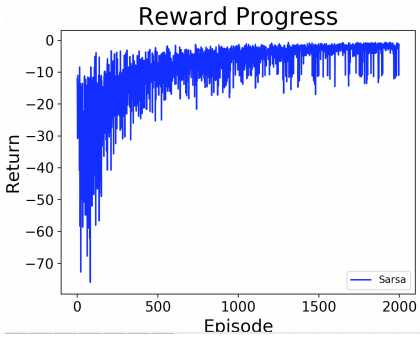
SARSA in Maze 1, 2, 3:



max reward = 0.30000000000000004
medLast100 = 0.30000000000000004
varLast100 = 0.02901899999999999

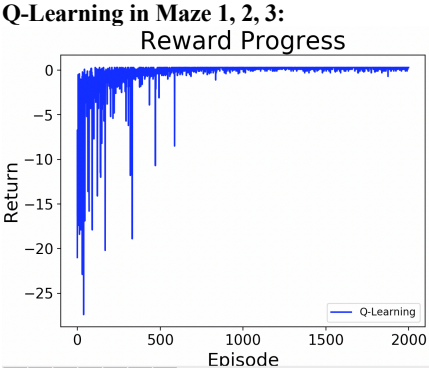


max reward = -0.09999999999999987
medLast100 = -0.30000000000000004
varLast100 = 0.05209100000000009

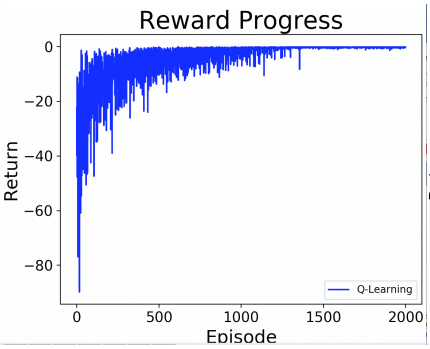


max reward = -0.7000000000000004
medLast100 = -1.9000000000000012
varLast100 = 4.780524000000001

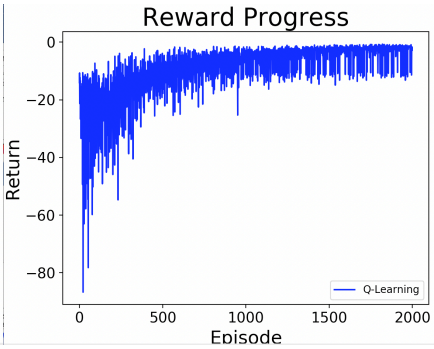
Q-Learning in Maze 1, 2, 3:



max reward = 0.30000000000000004
medLast100 = 0.30000000000000004
varLast100 = 0.013323999999999996

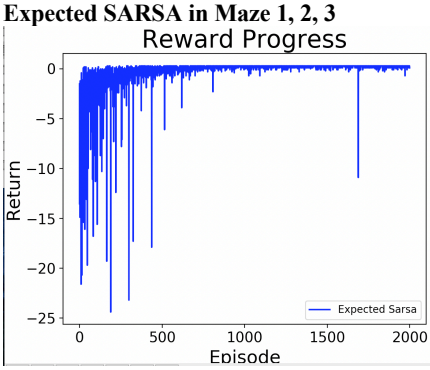


max reward = -0.09999999999999987
medLast100 = -0.25
varLast100 = 0.046924000000000084

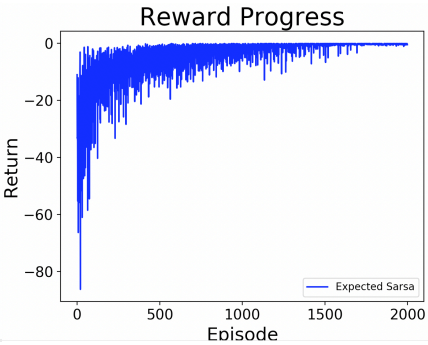


max reward = -0.7000000000000004
medLast100 = -1.9000000000000012
varLast100 = 10.147770999999999

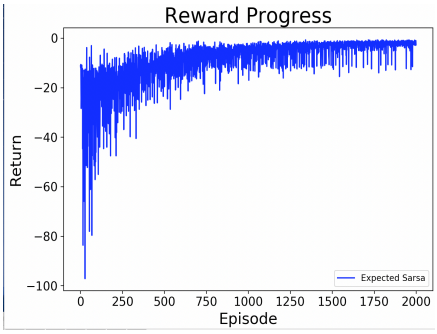
Expected SARSA in Maze 1, 2, 3



max reward = 0.30000000000000004
medLast100 = 0.30000000000000004
varLast100 = 0.020876000000000002

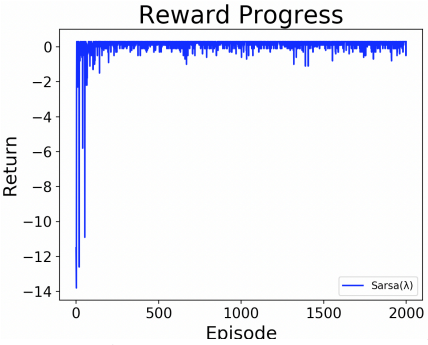


max reward = -0.09999999999999987
medLast100 = -0.30000000000000004
varLast100 = 0.04598400000000008

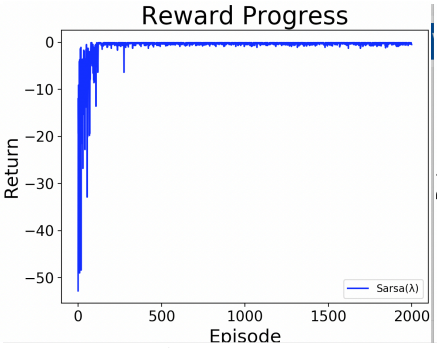


max reward = -0.7000000000000004
medLast100 = -1.7500000000000009
varLast100 = 3.7654590000000003

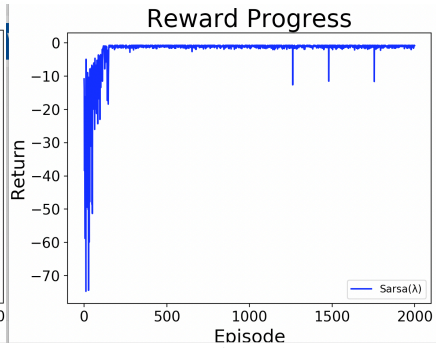
SARSA(λ) in Maze 1, 2, 3



max reward = 0.30000000000000004
medLast100 = 0.30000000000000004
varLast100 = 0.05129100000000001



max reward = -0.09999999999999987
medLast100 = -0.30000000000000004
varLast100 = 0.04300000000000008

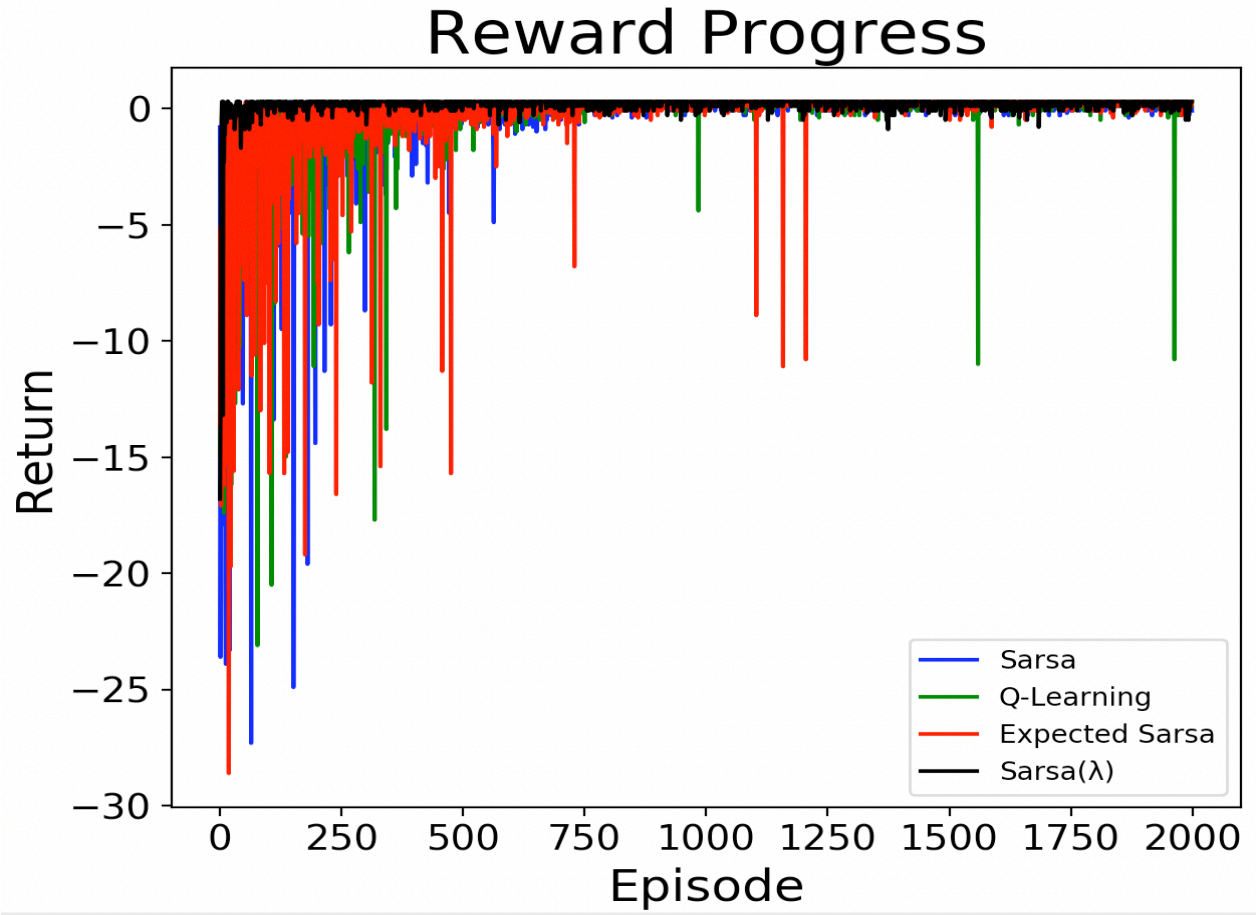


max reward = -0.7000000000000004
medLast100 = -0.9000000000000006
varLast100 = 0.06670000000000008

From the plots above, it is evident that from maze 1 to maze 3, those four algorithms converge slower and slower and has lower max reward and higher variance. It is because the maze is getting more complex and finding the goal is getting harder and harder from maze 1 to maze 3. More specifically, in maze 1, there are only two walls blocks and two pits, which means the agent can easily find the optimal policy. However, in maze 2, there are no pits, but nine wall blacks hiding the goal. The agent needs to go around all the walls to find the goal, which makes it more difficult. In maze 3, there are not only 13 walls hiding the goal, but also three pits, which makes finding the goal

harder. Therefore, maze 3 is harder than maze 2 and maze 2 is harder than maze 3. The more different maze is, the worse performance the RL algorithms get.

Task 1:



SARSA: max reward = 0.30000000000000004 medLast100=0.30000000000000004 varLast100=0.025203999999999983
Q-Learning: max reward = 0.30000000000000004 medLast100=0.30000000000000004 varLast100=1.2213840000000005
Expected SARSA: max reward = 0.30000000000000004 medLast100=0.30000000000000004 varLast100=0.015855999999999995
SARSA (λ): max reward = 0.30000000000000004 medLast100=0.30000000000000004 varLast100=0.026196

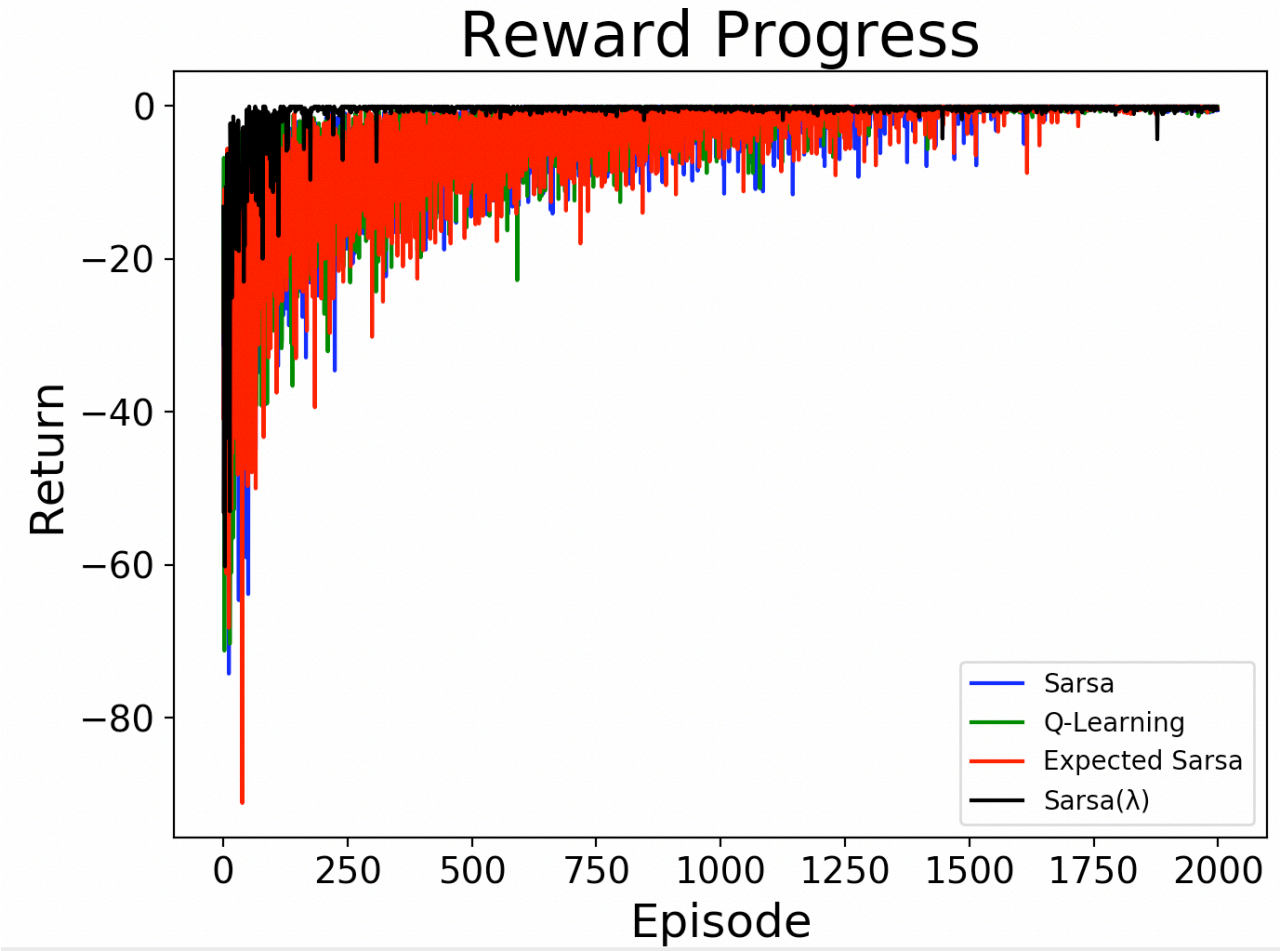
As the plot shows above, Sarsa started to converge around 500 episodes. Q-Learning started to converge around 400 episodes. Expected Sarsa started to converge around 700 episodes. Sarsa(λ) started to converge around 50 episodes. Q-Learning converges faster because Q-Learning always learns the best policy by using max Q. However, Sarsa only learns a near-optimal policy. Sarsa(λ) has the earliest convergence. Because with eligibility trace, Sarsa(λ) allows faster flow of information backward through time. With frequency and recency heuristic, Sarsa(λ) has the best performance than the other three.

Let us talk a look at the variance. Expected SARSA has the lowest variance because it exploits knowledge about stochasticity in the behaviour policy to perform updates by using the expected value instead of a greedy policy.

Convergence (From fast to slow): SARSA(λ) > Expected SARSA > SARSA > Q-Learning
Variance (From low to high): Expected SARSA, SARSA, SARSA(λ), Q-Learning

Best Algorithm for Task 1: SARSA(λ)

Task 2:



SARSA: max reward = -0.09999999999999987 medLast100=-0.30000000000000004 varLast100=0.04565100000000008
Q-Learning: max reward = -0.09999999999999987 medLast100=-0.19999999999999996 varLast100=0.05885600000000012
Expected Sarsa: max reward = -0.09999999999999987 medLast100=-0.09999999999999987 varLast100=0.02901600000000006
SARSA (λ): max reward = -0.09999999999999987 medLast100=-0.30000000000000004 varLast100=0.04647500000000008

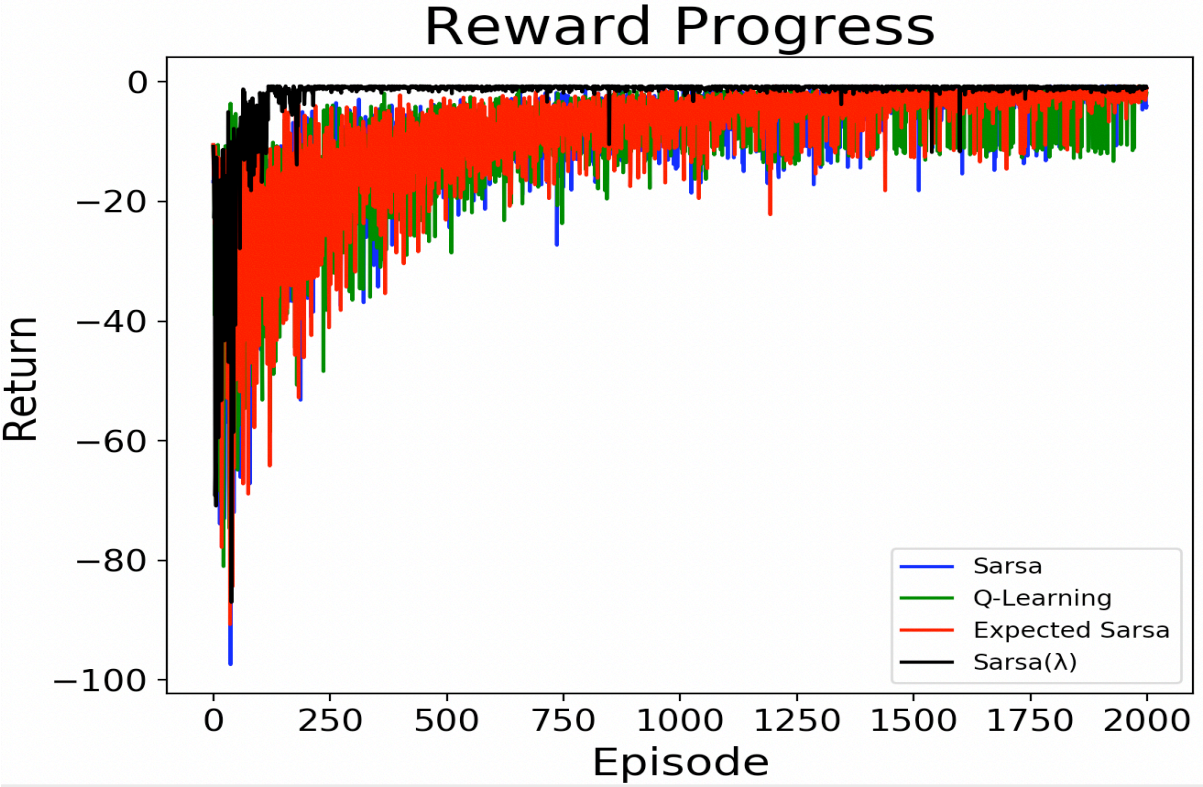
As the plot shows above, Sarsa started to converge around 1500 episodes. Q-Learning started to converge around 1250 episodes. Expected Sarsa started to converge around 1750 episodes. Sarsa(λ) started to converge around 125 episodes. Q-Learning converges faster because Q-Learning always learns the best policy by using max Q. However, Sarsa only learns a near-optimal policy. Sarsa(λ) has the earliest convergence. Because with eligibility trace, Sarsa(λ) allows faster flow of information backward through time. With frequency and recency heuristic, Sarsa(λ) has the best performance than the other three.

Let us talk a look at the variance. Expected SARSA has the lowest variance because it exploits knowledge about stochasticity in the behaviour policy to perform updates by using the expected value instead of a greedy policy.

Convergence (From fast to slow): SARSA(λ), Q-Learning, SARSA, Expected SARSA
Variance (From low to high): Expected SARSA, SARSA, SARSA(λ), Q-Learning

Best Algorithm for Task 2: SARSA(λ)

Task 3



SARSA: max reward = -0.7000000000000004 medLast100=-1.700000000000001 varLast100=1.4552109999999996
Q-Learning: max reward = -0.7000000000000004 medLast100=-1.8000000000000012 varLast100=8.359331
Expected SARSA: max reward = -0.7000000000000004 medLast100=-1.8000000000000012 varLast100=1.0731389999999978
SARSA (λ): max reward = -0.7000000000000004 medLast100=-1.0000000000000004 varLast100=0.07754400000000009

As the plot shows above in 2000 episodes, Sarsa didn't converge. Q-Learning didn't converge. Expected Sarsa just starting to converge around 2000. Sarsa(λ) started to converge around 250 episodes. Sarsa(λ) converge earlier before Sarsa, Q-Learning and Expected Sarsa. Because with eligibility trace, Sarsa(λ) allows faster flow of information backward through time. With frequency and recency heuristic, Sarsa(λ) has the best performance than the other three.

Let us talk a look at the variance. Expected SARSA has the lowest variance because it exploits knowledge about stochasticity in the behaviour policy to perform updates by using the expected value instead of a greedy policy.

Convergence (From fast to slow): SARSA(λ), Expected SARSA, Q-Learning, SARSA,
Variance (From low to high): SARSA(λ), Expected SARSA, SARSA, Q-Learning

Best Algorithm for Task 3 is SARSA(λ)

3 Conclusion

SARSA(λ) has the best performance than Sarsa, Q-Learning and Expected Sarsa. As the maze is getting more complicated. SARSA(λ) show the consistent performance of convergence time and even better and better variance. Expected Sarsa is the second-best algorithm. It does not converge as fast as SARSA(λ), but it has the same convergence speed as Sarsa and q-learning, and with lower variance (less noise).