ME5406-Deep Learning for Robotics

Project 1
Single Agent Pathfinding in
OpenAI Gym Frozen Lake Environment

Reinaldy Maslim

A0225239E

e0575873@u.nus.edu

Department of Mechanical Engineering

National University of Singapore

Singapore

October 2021

I
Implementation

1.1 First Visit Monte Carlo Control without exploring starts

The first visit Monte Carlo without exploring starts is a variant of the MC control. An advantage of first visit compared to every-visit MC is that the return sample G is i.i.d since they were drawn from separate episodes. Consequently, the estimated variance of the action values drops with 1/n with n: available samples, this is called *bias-free*. In our agent traversing in the Frozen Lake environment case, the assumption of exploring starts is discarded although it is beneficial so that every state-action pair a nonzero probability of being selected, because for the problem at hand every game or episode should start from the starting state denoted S which is the state 0 in the gridworld.

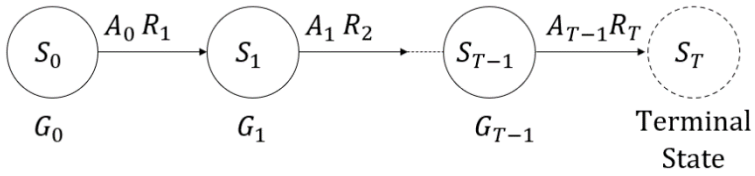

Figure 1 An episodic MDP

For First-visit Monte Carlo policy evaluation the average returns are considered only for first time visits of the state-action pair (s,a) is visited in an episode. The pseudocode for On-policy First-visit MC control without exploring starts is given below:

Initialize:
FrozenLake environment, env
n_episodes, number of total episodes
epsilon, 0<epsilon<1
π(s,a)←a 2-d dictionary or a 2-d array with random policy
Q(s,a)←a 2-d dictionary or a 2-d array with arbitrary values
Returns(s,a)←empty dictionary to be filled with key (s,a) pairs

For j=0,…,(n_episodes-1) episodes do:
Generate an episode following $\pi : s_0, a_0, r_1, \ldots, s_{T-1}, a_{T-1}, r_T, s_T$
set G=0
Initialize gamma, with 0<gamma<1 for convergence
for t=T-1,T-2, down to 0 time steps, do:
G←gamma*G+r_{t+1}
if pair (s_t, a_t) does **not** appear in $s_0, a_0, \ldots, s_{t-1}, a_{t-1}$ then:
Append G to Returns(s_t, a_t)
Q$(s_t, a_t) \leftarrow Average$(Returns$(s_t, a_t)$)
A*← $argmax_a Q(s_t, a)$
policy update using:
For all $a \in A(s_t)$, $\pi(a|s_t)$:$\begin{cases} 1 - \varepsilon + \frac{\varepsilon}{Cardinality\ of\ A(s_t)} & if\ a = A* \\ \frac{\varepsilon}{Cardinality\ of\ A(s_t)} & if\ a \neq A* \end{cases}$
return policy, π(s,a)

In computing the returns G from the equation,

$$G_t = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1} \; for\; t = T-1, T-2, \ldots, 0$$

we can make use of Horner's rule:

$$G_t = R_{t+1} + \gamma G_{t+1} \; as\; we\; move\; from\; t = T-1, T-2, \ldots, 0$$

Using this dynamic programming setting, less computation is needed to calculate G_t.

1.1.1 MC control 4x4 Grid

The 4x4 Frozen Lake gridworld for the agent to traverse is shown in the next figure. The same environment was also used for implementing SARSA and Q-learning.

S	F	F	F
F	H	F	H
F	F	F	H
H	F	F	G

Figure 2 4x4 Frozen Lake Gridworld

set of holes, H:(5,7,11,12)
set of goal, G=(15,)

The monteCarloControl function trains a policy by running 1000 episodes. The resulting policy can then be tested by calling the runEpisode function 1000 times of which the resulting ratio (score) of wins (reaching the goal) to the total number of episodes can be recorded. The process can be repeated a number of times e.g. 1000 trials and the resulting histogram can be plotted in Figure 3.

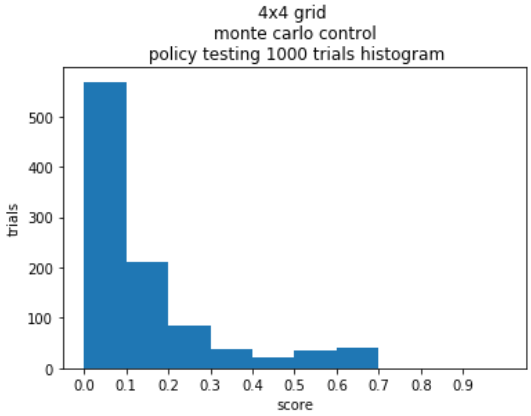

Figure 3 4x4 Grid On-Policy MC control policy testing score histogram

An example of output from mc.py python code is given below.

Output:

```
policy= {0: {0: 0.0025, 1: 0.0025, 2: 0.9925, 3: 0.0025}, 1: {0: 0.0025, 1: 0.0025, 2:
0.0025, 3: 0.9925}, 2: {0: 0.9925, 1: 0.0025, 2: 0.0025, 3: 0.0025}, 3: {0: 0.0025, 1:
0.9925, 2: 0.0025, 3: 0.0025}, 4: {0: 0.9925, 1: 0.0025, 2: 0.0025, 3: 0.0025}, 5: {0: 0.2,
1: 0.2, 2: 0.2, 3: 0.2}, 6: {0: 0.0025, 1: 0.0025, 2: 0.9925, 3: 0.0025}, 7: {0: 0.2, 1:
0.2, 2: 0.2, 3: 0.2}, 8: {0: 0.0025, 1: 0.0025, 2: 0.9925, 3: 0.0025}, 9: {0: 0.9925, 1:
0.0025, 2: 0.0025, 3: 0.0025}, 10: {0: 0.0025, 1: 0.0025, 2: 0.9925, 3: 0.0025}, 11: {0:
0.2, 1: 0.2, 2: 0.2, 3: 0.2}, 12: {0: 0.2, 1: 0.2, 2: 0.2, 3: 0.2}, 13: {0: 0.0025, 1:
```

```
0.9925, 2: 0.0025, 3: 0.0025}, 14: {0: 0.0025, 1: 0.9925, 2: 0.0025, 3: 0.0025}, 15: {0:
0.2, 1: 0.2, 2: 0.2, 3: 0.2}}
 >  ^  <  v

 <  H  >  H

 >  <  >  H

 H  v  v  G
score: 0.099
```

The states in which the holes are in has the same policy of $\frac{\varepsilon}{|A(s)|}$ probability throughout all the action space, in this case 0.2 as was assigned in createRandomPolicyDict function.

1.1.2 MC control 10x10 Grid

The 10x10 Frozen Lake environment for the agent to traverse is shown in the next figure. The same environment was also used for implementing SARSA and Q-learning.

S	F	F	F	F	H	F	F	F	F
F	F	H	F	F	F	H	F	F	F
F	H	F	F	H	F	F	F	F	F
F	F	F	F	H	F	F	H	F	F
H	F	F	F	F	H	F	F	F	H
F	F	H	H	F	F	H	H	F	F
F	H	F	F	F	F	H	F	H	F
F	F	H	F	F	H	F	H	F	F
H	F	F	F	F	H	F	F	H	F
H	F	F	H	F	F	F	F	F	G

Figure 4 10x10 Frozen Lake Gridworld

set of holes, H: (5,12,16,21,24,34,37,40,45,49,52,53,56,57,61,66,68,72,75,77,80,85,88,90,93)
set of goal, G: (99,)

An example of output from mc_extended.py python code is given below.
Output:

```
policy= {0: {0: 0.0025, 1: 0.0025, 2: 0.0025, 3: 0.9925}, 1: {0: 0.0025, 1: 0.0025, 2:
0.0025, 3: 0.9925}, 2: {0: 0.0025, 1: 0.0025, 2: 0.0025, 3: 0.9925}, 3: {0: 0.0025, 1:
0.0025, 2: 0.0025, 3: 0.9925}, 4: {0: 0.9925, 1: 0.0025, 2: 0.0025, 3: 0.0025}, 5: {0: 0.2,
1: 0.2, 2: 0.2, 3: 0.2}, 6: {0: 0.0025, 1: 0.0025, 2: 0.0025, 3: 0.9925}, 7: {0: 0.0025, 1:
0.0025, 2: 0.9925, 3: 0.0025}, 8: {0: 0.9925, 1: 0.0025, 2: 0.0025, 3: 0.0025},   …

             90: {0: 0.2, 1: 0.2, 2: 0.2, 3: 0.2}, 91: {0: 0.2, 1: 0.2, 2: 0.2, 3: 0.2},
92: {0: 0.2, 1: 0.2, 2: 0.2, 3: 0.2}, 93: {0: 0.2, 1: 0.2, 2: 0.2, 3: 0.2}, 94: {0: 0.2, 1:
0.2, 2: 0.2, 3: 0.2}, 95: {0: 0.2, 1: 0.2, 2: 0.2, 3: 0.2}, 96: {0: 0.2, 1: 0.2, 2: 0.2, 3:
0.2}, 97: {0: 0.2, 1: 0.2, 2: 0.2, 3: 0.2}, 98: {0: 0.2, 1: 0.2, 2: 0.2, 3: 0.2}, 99: {0:
0.2, 1: 0.2, 2: 0.2, 3: 0.2}}
 ^  ^  ^  ^  <  H  ^  >  <  <

 <  <  H  >  <  >  H  <  ^  >

 <  H  ^  ^  H  <  <  <  <  ^

 ^  <  >  ^  H  >  <  H  <  >

 H  ^  v  v  ^  H  >  >  <  H

 >  >  H  H  <  >  H  H  <  <
```

```
 ^  H  <  <  v  <  H  <  H  <

 v  >  H  ^  v  H  <  H  <  <

 H  <  <  v  >  H  <  <  H  <

 H  <  <  H  <  <  <  <  <  G

score: 0.0
```

To be more concise, the policy of states 9-89 is not shown. The policy output by the mc_extended.py program was trained by running 20000 episodes in the 10x10 gridworld. It was observed that most of the later states and state-action pairs are not visited by the agent as evident from the un-updated policy value of 0.2 probability (the probabilities do not sum to one but the code still works as intended because every action for the particular state has the same chances of being chosen). *Note that the states that have not been visited by the agent was assigned action '<' by default*.

1.2 SARSA with an ϵ-greedy behaviour policy

SARSA is an On-Policy TD control algorithm in which the name comes from the need to have the quintuple of the current state S, the action that would be executed A, the reward received after executing action A from state S, the next state S' and next action A'. It is called On-Policy because in SARSA both the behaviour policy and the target policy are ϵ-greedy. As opposed to MC where update of the estimate is done using G_t, TD uses the moving target estimate $Q(S_{t+1}, A_{t+1})$, hence it is called a *bootstrapping* method. Advantages of TD control to MC control include: TD control is applicable to both episodic and continuing tasks, updates policy evaluation and improvement in an online fashion, requires less memory, can learn without the final outcome from incomplete sequences, and often TD(0) converges faster than MC but there is no guarantee.

TD-based On-policy SARSA pseudocode is given below:

```
Initialize:
    FrozenLake environment, env
    n_episodes, number of total episodes
    ε, for epsilon-greedy policy
    α, learning rate
    γ, discount rate gamma, with 0<γ<1 for convergence
    Q(s,a) ←a 2-d array with arbitrary values
    set Q(s,a)=0 for s ϵ Ŝ, with Ŝ being the set of terminal states
    decay_constant, an exponential decay rate constant for epsilon and alpha
For j=0,...,(n_episodes-1) episodes do:
    Initialize S
    Choose A from S using policy derived from Q (e.g., ε − greedy)
    Loop for each step=0,…,(max_steps-1):
        Take action A; observe R, S'
        Choose A' from S' using policy derived form Q (e.g., ε − greedy)
        Q(S,A) ←Q(S,A)+ α[R+γQ(S',A')-Q(S,A)]
        S←S';A←A'
        if S ϵ Ŝ:
            break loop
    update ε for next episode such that it decays
    decrease learning rate α for next episode
return Q
```

1.2.1 SARSA 4x4 Grid

The sarsa function returns the state-action values prediction array, Q and a list of rewards obtained after undergoing a total of n_episodes. An optimal policy following the ε-greedy policy (e.g. $\varepsilon = 0.01$). can be extracted/constructed based on the array Q. The resulting policy can then be tested by running 1000 episodes of which the resulting ratio (score) of wins (reaching the goal) to the total number of episodes can be recorded. The process can be repeated a number of times e.g. 1000 trials and the resulting histogram can be plotted in Figure 5.

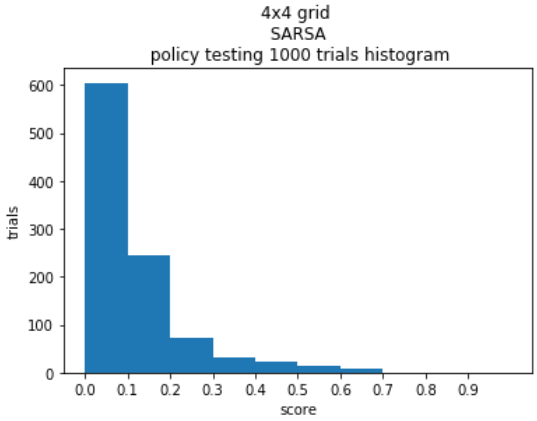

Figure 5 4x4 Grid SARSA policy testing score histogram

An example of output from sarsa.py python code is given below.

Output:

```
gamma: 0.9
epsilon: 1.0-0.01
alpha: 0.8
 v  >  ^  ^

 <  H  <  H

 ^  v  <  H

 H  >  ^  G

score over time: -0.966

Q is not shown here for conciseness.
```

1.2.2 SARSA 10x10 Grid

Implementation of SARSA in the 10x10 grid defined as in section 3.1.2, an example output from sarsa_extended.py python code is given below.

```
gamma: 0.8
epsilon: 1.0-0.001
alpha: 1.0-0.8
64 frisbees obtained in 20000 episodes
average timesteps taken: 57.1174
score over time: -0.79765
 v  ^  ^  v  <  H  >  >  ^  ^

 <  <  H  >  ^  v  H  >  <  ^

 <  H  >  <  H  >  v  ^  v  >

 ^  v  v  <  H  >  <  H  >  ^

 H  >  ^  ^  v  H  >  ^  <  H

 v  <  H  H  >  >  H  H  ^  v

 <  H  ^  v  <  ^  H  v  H  >

 ^  <  H  >  <  H  <  H  <  >
```

```
 H  >  v  ^  <  H  ^  ^  H  >

 H  >  <  H  >  v  v  v  v  G

Q is not shown here for conciseness.
```

The function sarsa_extended returns an array Q which can be used to extract an ε-greedy policy (e.g. $\varepsilon = 0.01$). The policy can be tested by running 1000 episodes and the score obtained be recorded. The process can be repeated a number of times e.g. 1000 trials and the resulting histogram can be plotted in Figure 6.

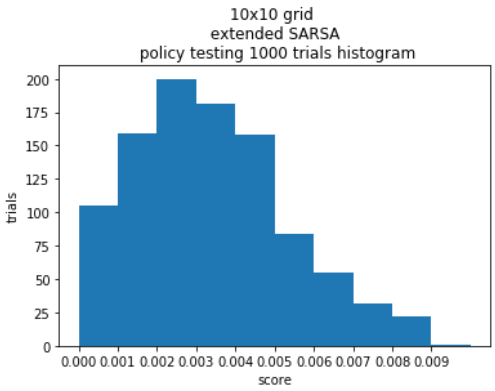

Figure 7 SARSA extended grid policy testing score histogram

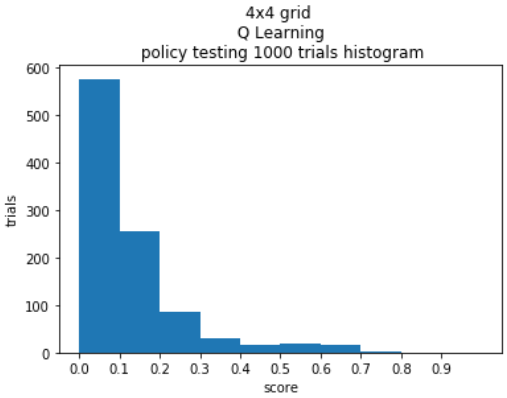

Figure 6 4x4 Grid Q-learning policy testing score histogram

1.2.3 SARSA 10x10 Grid Not Slippery Environment

```
gamma: 0.8
epsilon: 1.0-0.001
alpha: 1.0-0.8
3328 frisbees obtained in 20000 episodes
average timesteps taken: 86.0314
score over time: 0.15525
 >  v  <  <  v  H  >  <  >  <

 ^  ^  H  ^  <  v  H  >  v  >

 <  H  v  ^  H  v  v  >  v  v

 ^  v  >  v  H  >  <  H  v  >

 H  >  >  >  v  H  ^  >  v  H

 >  ^  H  H  v  <  H  H  >  v

 ^  H  >  >  v  ^  H  ^  H  v

 >  <  H  v  v  H  v  H  >  v

 H  >  >  >  v  H  ^  <  H  v

 H  v  v  H  >  >  >  >  >  G

Q is not shown here for conciseness.
```

1.3 Q-learning with an ϵ-greedy behaviour policy

Q-learning differs from SARSA in that the policy used to learn the action-values (target policy) are greedy instead of ε-greedy. Hence, Q-learning is an Off-Policy method because its behaviour policy is ε-greedy while its target policy is greedy policy.

Off-policy Q-learning pseudocode:

Initialize:
    FrozenLake environment, env
    n_episodes, number of total episodes
    ε, for epsilon-greedy policy
    α, learning rate
    γ, discount rate gamma, with $0<\gamma<1$ for convergence
    Q(s,a) ←a 2-d array with arbitrary values
    set Q(s,a)=0 for $s \in \hat{S}$, with $\hat{S}$ being the set of terminal states
    decay_constant, an exponential decay rate constant for epsilon and alpha
For j=0,...,(n_episodes-1) episodes do:
    Initialize S
    Loop for each step=0,…,(max_steps-1):
        Choose A from S using policy derived form Q (e.g., $\varepsilon-$ greedy)
        Take action A; observe R, S'
        $Q(S,A) \leftarrow Q(S,A) + \alpha[R+\gamma \max_{a'} Q(S', a') - Q(S,A)]$
        S←S'
        if $S \in \hat{S}$:
            break loop
    update ε for next episode such that it decays
    decrease learning rate α for next episode
return Q

1.3.1 Q-learning 4x4 Grid

Policy testing of the policy extracted from optimal action values from Q_learning function results in a histogram plot in Figure 7, the data points represent score from 1000 different generated policies with each policy being used to run 1000 episodes.

An example of output from Q_learning.py python code is given below.

Output:

```
gamma: 0.9
epsilon: 1.0-0.01
alpha: 0.8
 v  >  v  ^

 <  H  <  H

 ^  v  >  H

 H  v  v  G

score over time: -0.948

Q is not shown here for conciseness.
```

1.3.2 Q-learning 10x10 Grid

Implementation of Q-learning in the 10x10 grid defined as in section 3.1.2, an example output from Q_learning_extended.py python code is given below.

```
gamma: 0.8
epsilon: 1.0-0.001
alpha: 1.0-0.8
34 frisbees obtained in 20000 episodes
average timesteps taken: 49.9888
score over time: -0.84275
 <  ^  ^  >  <  H  <  >  >  ^

 <  <  H  >  ^  <  H  >  <  >

 <  H  <  <  H  >  v  ^  ^  ^
```

```
 ^  v  v  <  H  <  <  H  >  ^

 H  >  ^  ^  <  H  <  <  <  H

 v  <  H  H  >  <  H  H  <  v

 <  H  <  v  <  <  H  <  H  ^

 ^  <  H  >  <  H  ^  H  <  <

 H  >  v  ^  <  H  >  <  H  >

 H  >  <  H  >  v  v  v  v  G

Q is not shown here for conciseness.
```

When the function Q_learning_extended is called, it returns an array Q which can be used to extract an ε-greedy policy (e.g. $\varepsilon = 0.01$). The policy can be tested by running 1000 episodes and the score obtained be recorded. The process can be repeated a number of times e.g. 1000 trials and the resulting histogram can be plotted in Figure 8.

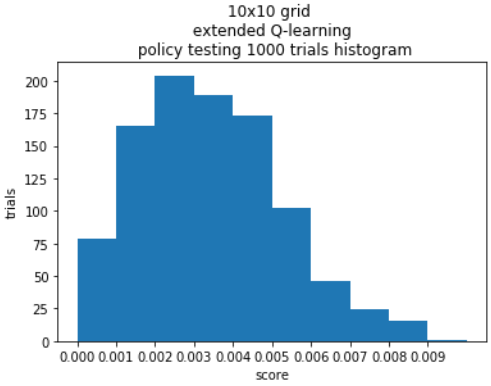

Figure 9 Q-learning extended grid policy testing score histogram

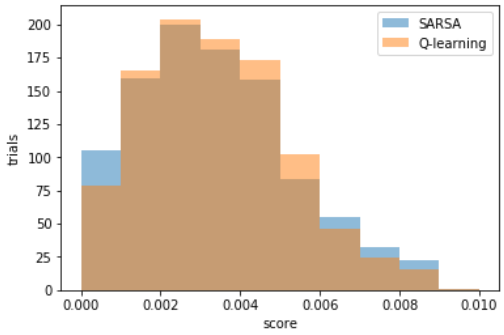

Figure 8 Superposed histogram of Fig. 6 and Fig. 8

1.3.3 Q-learning 10x10 Grid Not Slippery Environment

```
gamma: 0.8
epsilon: 1.0-0.001
alpha: 1.0-0.8
17629 frisbees obtained in 20000 episodes
average timesteps taken: 17.3081
score over time: 0.7629
 >  >  >  v  v  H  >  <  >  v

 >  ^  H  >  >  v  H  ^  v  v

 <  H  >  ^  H  v  v  >  v  <

 >  v  v  ^  H  >  v  H  v  <

 H  ^  >  >  v  H  >  >  v  H

 <  <  H  H  v  <  H  H  >  v

 <  H  >  <  v  <  H  ^  H  v

 <  v  H  >  v  H  <  H  >  v

 H  >  >  <  v  H  v  v  H  v

 H  ^  ^  H  >  >  >  >  >  G

Q is not shown here for conciseness.
```

II
Discussions

2.1 4x4 Grid

By comparing histogram plots of Figure 3, Figure 5 and Figure 7, it can be observed that in the case of traversing in the 4x4 Frozen Lake Gridworld, the policy generated from MC control outperforms that of SARSA and Q-learning. An explanation for this is that in the small gridworld, MC control can achieve sufficient exploration. Once the MC control algorithm has sufficient state-action visits, the value functions will 'grow' to the right direction and results in an optimal policy. It can be concluded that On-Policy MC control is most suitable for problems with fewer number of states or state-action pairs.

The performances of SARSA and Q-learning are very much similar, but closer inspection will suggest that Q-learning is slightly better as witnessed from Figure 5 and 7 as well as comparing the scores over time obtained from Q-learning are generally better than that of SARSA which are closer to -1.

2.2 10x10 Grid

Because convergence for MC control needs the assumption for infinite visits or episodes it renders the method not practical even for a 10x10 grid with 25 holes and 4 actions available at each state as there are 300 unique state-action pairs that should be visited by the agent. For MC control without exploring starts, convergence for later states become even more difficult. This experiment shows that On-Policy MC control without exploring starts has a problem with exploration. It is because the same ε-greedy policy is used for learning and exploration, hence the trade-off between optimality and exploration cannot be resolved easily. Another approach is to use Off-Policy MC control where two separate policies are involved namely the behaviour policy $b(a|s)$ and target policy $\pi(a|s)$. b is an exploratory policy to generate episodes while π is used to store the knowledge that was learned. For example, π is either deterministic or greedy policy (and ultimately the optimal policy) while b is exploratory (e.g. ε-soft). By using Off-Policy MC control it is hoped that sufficient exploration is achieved.

By comparing the scores over time obtained by SARSA and Q-learning one would observe that the scores over time obtained from the policy derived from SARSA are generally better (higher) compared to the optimal policy derived from Q-learning. This is a curious phenomenon since the policies obtained by SARSA are generally sub-optimal policies unlike Q-learning which learns the optimal action values q*(s,a) and hence the optimal policy π*. A possible reason for this is that SARSA generates a safer policy for the agent to traverse in the environment (by following ε-greedy target policy) which was demonstrated from the classical example of the cliff-walking problem.

From Figure 9 one may observe that SARSA outperforms Q-learning shown by the higher frequency visible on the higher right end of the score axis. Given the treacherous nature of the Frozen Lake environment the agent managed to reach the goal 10x10 gridworld at most 2 times in 1000 episodes in 200 trials using the policy generated from SARSA as well as Q-learning.

Policy testing was not carried out to MC control for the 10x10 grid case because training with 20000 episodes, the later episodes of the 10x10 gridworld was not even visited by the agent, therefore no usable policy results from the implementation of MC control for the extended grid case.

2.3 10x10 Grid Not Slippery Environment

With the parameter is_slippery set to False when instantiating the object env, the transition function is now set to deterministic. In this case, Q-learning implementation wins against SARSA by a long shot, 0.7629 to 0.15525 score over time in the example.

2.4 Difficulties

Determining whether the algorithms work as was intended with the transition function of the environment being non-deterministic was one of the difficulties. This was resolved by doing sanity checks by setting the is_slippery environment parameter to False which will set the transition function to a deterministic one. Another difficulty was to tune the parameters involved; discount factor γ, epsilon ε, learning rate α along with how each of them decays w.r.t increasing episodes so that the results obtained are optimal.

2.5 Initiatives

Implementation of MC control using numpy (if possible) to speed up computation, implementation of Off-Policy MC control, expected SARSA, and TD-based Off-Policy control with double learning. Learn and implement non-tabular methods.

2.6 Issues with current Implementation

The current implementation of SARSA and Q-learning uses numpy which is a popular library for array/matrix manipulation and computation. For a 10x10 gridworld with 4 actions in each non-terminating state, the time elapsed after running 20000 episodes for the function sarsa_extended was around 41s, and 54s for Q_learning_extended. In addition, for tabular methods, memory requirements can be problematic. The implication of the number of states and actions on the memory requirements and running times of tabular algorithms is often called the curse of dimensionality.

III
Conclusions

a) On-Policy First-visit MC control without exploring starts has issues with exploration, hence it is most suitable for episodic MDP problems with small number of state-action pairs (e.g. 4x4 Frozen Lake Gridworld). Possible improvement includes using Off-Policy MC control (Sutton-Barto sec 5.6)
b) SARSA may perform better than Q-learning as demonstrated in the 10x10 Frozen Lake Gridworld case. A plausible explanation is because SARSA provide a ‘safer’ policy for the agent to traverse in the environment.
c) For an environment with deterministic transition function, Q-learning is preferred over SARSA and MC control. Due to the stochastic nature of action selection in MC control (A*=random.choice(optimal_actions)) MC control can perform very goodly or very badly.

References

1. P. C. Y. Chen, “Deep Learning for Robotics ME5406, Lecture Notes: Part I Tabular Methods for Reinforcement Learning”, NUS, 2021.
2. Sutton, Richard S. and Barto, Andrew G. Reinforcement Learning: An Introduction, MIT Press, 2ed., 2018.