CS6700: Reinforcement Learning (Jan-May 2024) Programming Assignment 3

Vivek Sivaramakrishnan NS24Z170 Pooja Pandey NS24Z172

Table of contents

1 Defining Options

An option consists of 3 components:

- An initiation set
- A policy
- A termination set ($\beta = 0$ or 1 for any state, can be considered as a set of states)

In the Taxi-v3 problem, we design options to be initiated from anywhere in the state space. Thus, we only have 2 varying components among the options: the option *policy* and *termination* set. We use a `collections.namedtuple` structure to store the policy (a callable function) and termination set of states for our option:

```
```{python}
Option = namedtuple('Option', ['policy', 'terminate'])
```
```

1.1 Primitive Actions

In line with the Recent Advances in Hierarchical Reinforcement Learning paper, we define primitive actions to be *one-step* options; that is:

- For primitive action A, `Option_A.policy(s) = A` for all `s`.
- The termination set is the entire state space.

Following the above, we define our *one-step* options as follows:

```
```{python}
grid = [(i, j) for i in range(5) for j in range(5)]

S = Option(lambda state: 0, grid)
N = Option(lambda state: 1, grid)
E = Option(lambda state: 2, grid)
W = Option(lambda state: 3, grid)
P = Option(lambda state: 4, grid)
D = Option(lambda state: 5, grid)
```
```

1.2 Option `Goto X`

The option `Goto X` moves the taxi to `X`.

Defining the termination set for the option is straightforward; we store only the first 2 values of the 4-tuple state. This can be obtained from an encoded state using the following `decode` function:

```
```{python}
decode = lambda state: tuple(env.unwrapped.decode(state))[:2]
```
```

One can set

```
```{python}
Option_Goto_R.terminate = {(0, 0)}
```
```

and then run

```
```{python}
decode(state) in Option_Goto_R.terminate
```
```

to check if option has terminated.

The policy for the option however has to be learned. For this, we employ Vanilla Q-Learning.

1.2.1 Vanilla Q-Learning to learn `Goto X` option policies

We use Vanilla Q-Learning over primitive actions (or, equivalently SMDP over only *one-step* options) to solve the Taxi environment. We then extract policies for `Goto X` from the learnt Q-Function as follows:

- In the 4-tuple representation of the state, fix passenger location to X, and the destination to any $k \neq X$ (we use $(X+1) \mod 4$). Only 25 states, of the form $(i, j, X, (X+1) \mod 4)$ satisfy the constraint.
- Taking a cross-section of the Q-Table on this set of states, reshaping result to $5 \times 5 \times 6$, and $\arg\max$ (or ϵ-greedy) along the 3rd axis gives the policy for this option.

```{python}
policy_grids = []
for k in range(4):
    policy_grid = Q[[env.unwrapped.encode(i, j, k, (k+1)%4) for i in
    ↪ range(5) for j in range(5)]].reshape(5, 5, 6).argmax(axis=2)

    policy_grids.append(policy_grid)
```

Performing the above gives the following (deterministic) option policies:

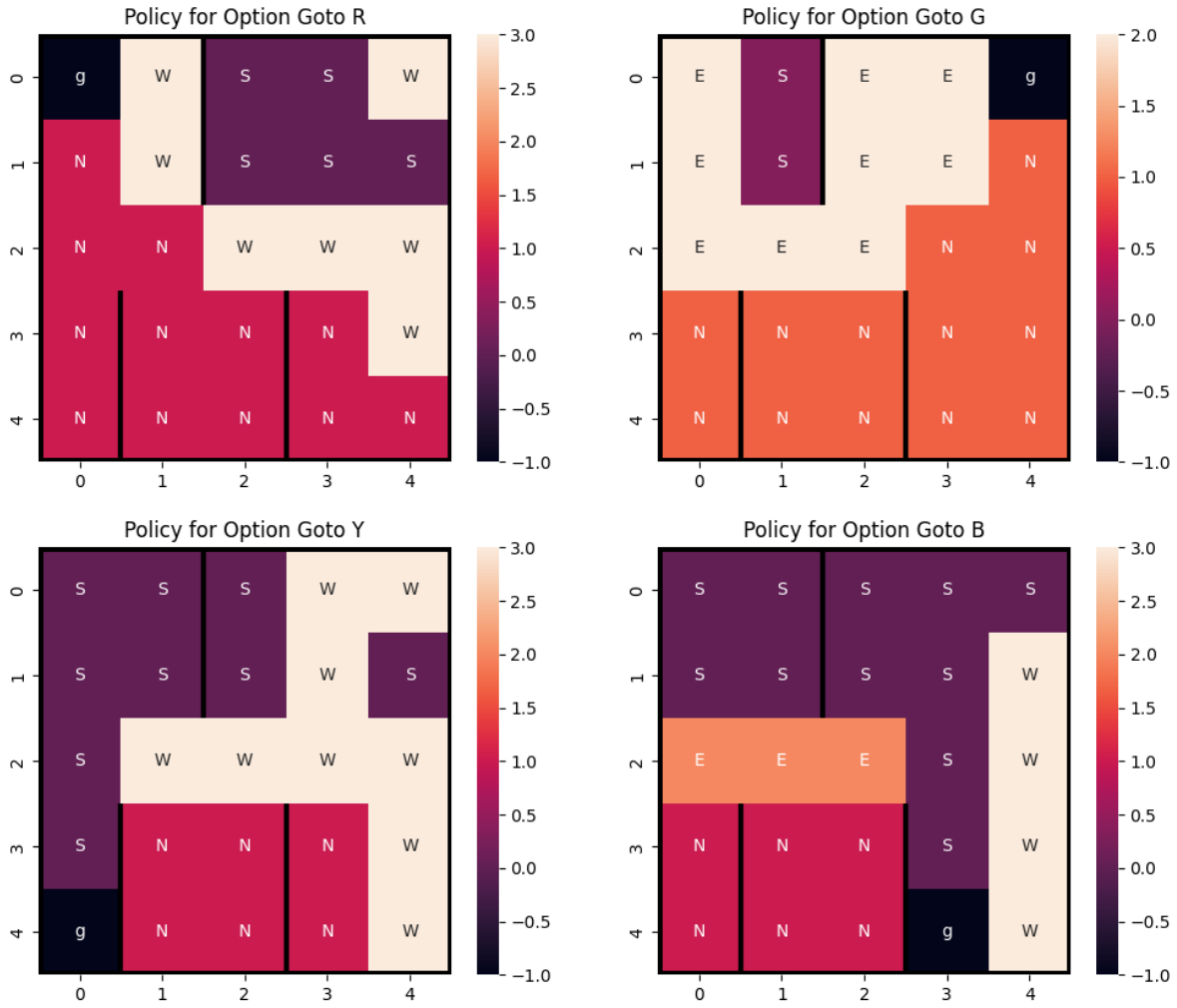

Figure 1: Option Policies

We pass obtained policy grids through a function decorator to generate the policy function. We further define our `Goto X` options using the generated policy functions and termination states as follows:

```{python}
macro_options = []
option_policy_generator = lambda grid: (lambda state:
 ↪  (grid[decode(state)]))
option_terminations = [(0, 0), (0, 4), (4, 0), (4, 3)]

for policy_grid, term_state in zip(policy_grids, option_terminations):
    policy_func = option_policy_generator(policy_grid)
    option = Option(policy_func, [term_state])
    macro_options.append(option)

R, G, Y, B = macro_options
```

1.3 Mutually Exclusive Options

We define options `NORTH`, `SOUTH`, `EAST` and `WEST` which keep moving the taxi in their respective direction till they encounter an *obstruction*. These options are mutually exclusive because the option policies are pairwise non-consistent on all states s.

In our design of the options, we set the termination states to be the states adjacent to these obstructions:

```{python}
SOUTH = Option(lambda state: 0, [(4, i) for i in range(5)])
NORTH = Option(lambda state: 1, [(0, i) for i in range(5)])

EAST = Option(lambda state: 2, [(i, 4) for i in range(5)] + [(0, 1), (1,
 ↪  1), (3, 0), (4, 0), (3, 2), (4, 2)])
WEST = Option(lambda state: 3, [(i, 0) for i in range(5)] + [(0, 2), (1,
 ↪  2), (3, 1), (4, 1), (3, 3), (4, 3)])
```

2 Implementation of Algorithms

2.1 Intra-option Q-Learning

Truncated code highlighting the intra-option updates along with option executions and terminations is given below:

```
```{python}
def ioq_learning(options, episodes=10000, print_freq=2000, epsilon=0.01,
 ↪  alpha=0.1, gamma=0.9, ioql = True, seed=1, track=True):

...

while not done:
  option_no = choose_action(Q[state], epsilon)
  option = options[option_no]

  option_done = False
  option_reward = 0
  option_init = int(state)
  discount = 1

  while not option_done:
    action = option.policy(state)
    state_next, reward, terminated, truncated, _ = env.step(action)

    if ioql:
      # Update all options (including one-step) with consistent policy
       ↪  action at state
      for c_option_no in range(len(options)):
        c_option = options[c_option_no]
        # Check if consistent
        if c_option.policy(state) == action:
          # Update state, action pairs' value
          if decode(state_next) in c_option.terminate:
            Q[state, c_option_no] = (1 - alpha) * Q[state, c_option_no]
 ↪  + alpha * (reward + gamma * np.max(Q[state_next]))
          else:
            Q[state, c_option_no] = (1 - alpha) * Q[state, c_option_no]
 ↪  + alpha * (reward + gamma * Q[state_next, c_option_no])
```
```

```
    done = terminated or truncated
    option_done = decode(state_next) in option.terminate

    option_reward += discount * reward
    discount *= gamma
    tot_reward += reward
    state = state_next

  Q[option_init, option_no] = (1 - alpha) * Q[option_init, option_no] +
 ↪  alpha * (option_reward + discount * np.max(Q[state_next]))

...

return Q, episode_rewards, avg_rewards
```

2.2 SMDP Q-Learning

SMDP Q-Learning is simply intra-option q-learning, but without the intra-option updates. Exploiting this subsumption, SMDP is defined as the following:

```
```{python}
def smdp_learning(options, episodes=10000, print_freq=2000,
 ↪  epsilon=0.01, alpha=0.1, gamma=0.9, seed=1):
  return ioq_learning(options, episodes, print_freq, epsilon, alpha,
   ↪  gamma, ioql = False, seed=seed)
```
```

3 Tasks

3.1 Hyperparameter tuning

The Optuna Framework is used to search for the optimal values of the following hyperparameters:

- Learning rate α
- ϵ in ϵ-greedy selection

50 Trials were run on each of the algortihm configurations (SMDP, IOQL on 2 option configurations). The results are stored in `hpt_results.db` which can be browsed through using *Optuna Dashboard*. The hyperparameters found is listed below:

Algorithm	Options	Alpha	Epsilon
SMDP	`Goto X`	0.5777660	0.0046396
IOQL	`Goto X`	0.7365430	0.0014736
SMDP	Mutually Exclusive	0.8030125	0.0005293
IOQL	Mutually Exclusive	0.8028770	0.0002563

3.2 One-Step + `Goto X` Options

3.2.1 Plot

The running average over 100 episodes is plotted. The 2 plots below are just zoomed in to the original plot (performance on first and last 500 out of 10000 episodes):

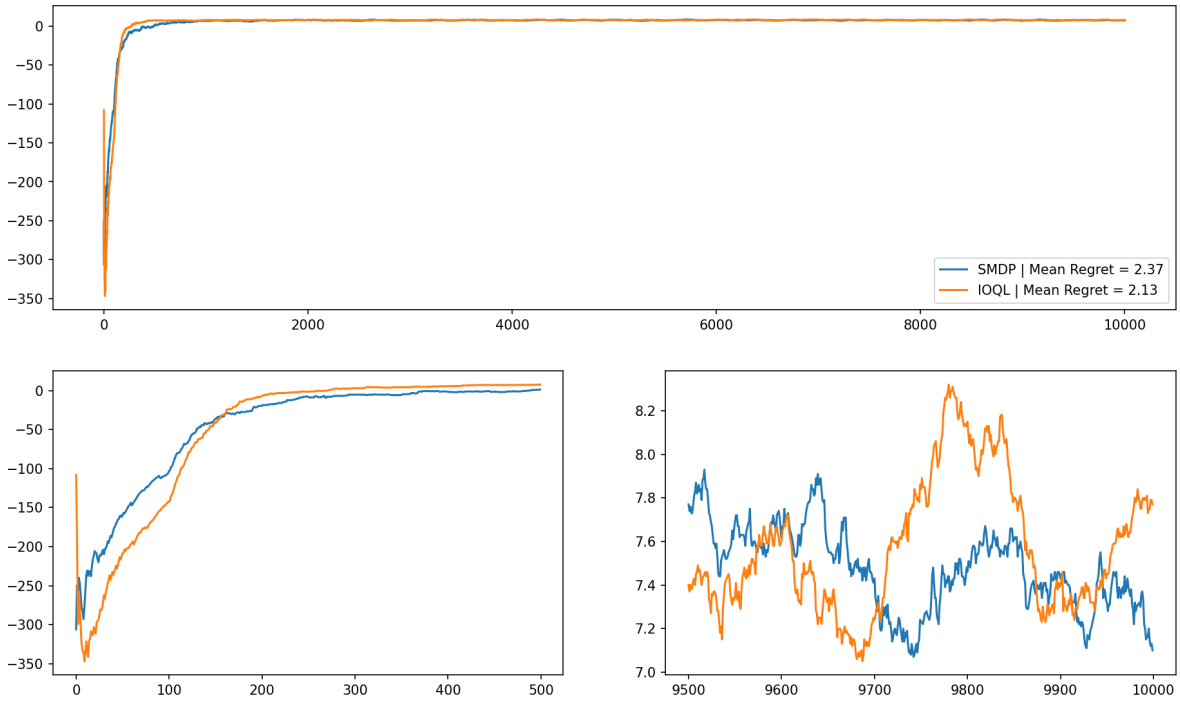

Figure 2: SMDP vs. IOQL

3.2.2 Q-value visualization

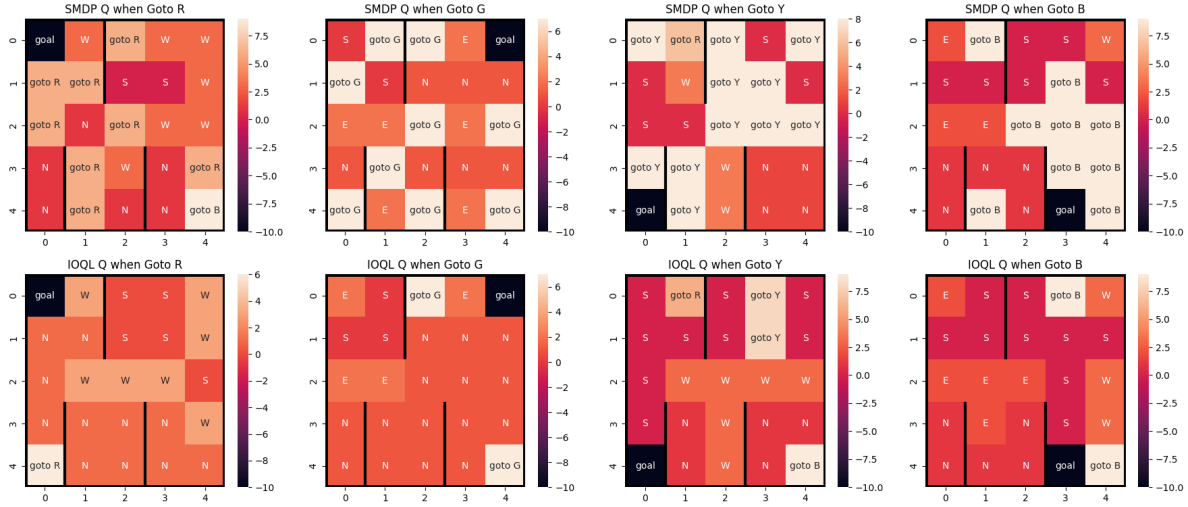

Figure 3: SMDP vs. IOQL

It can be observed that the IOQL policy chooses one-step options more often than the SMDP policy. This can be due to the fact that one-step options / primitive actions are updated more frequently due to intra-option updation during option progression, over SMDP where frequency of updations of one-step vs. macro options is roughly the same.

3.3 One-Step + Mutually-Exclusive Options

3.3.1 Plot

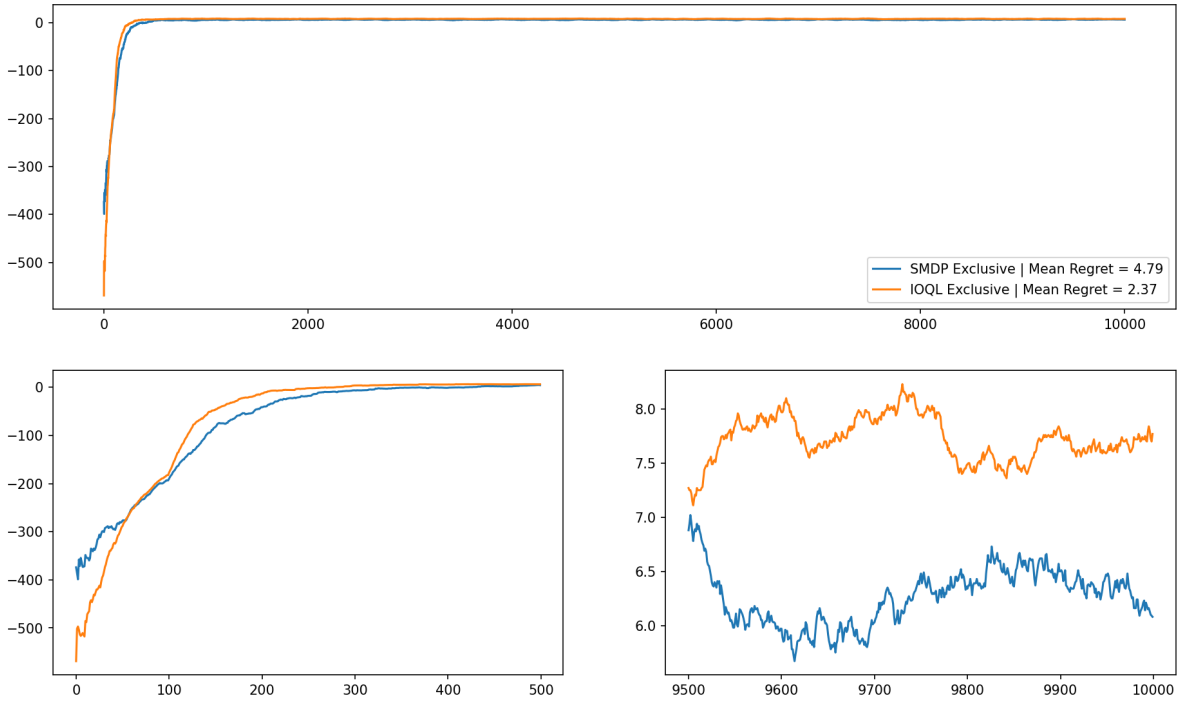

Figure 4: SMDP vs. IOQL (Exclusive)

3.3.2 Q-value visualization

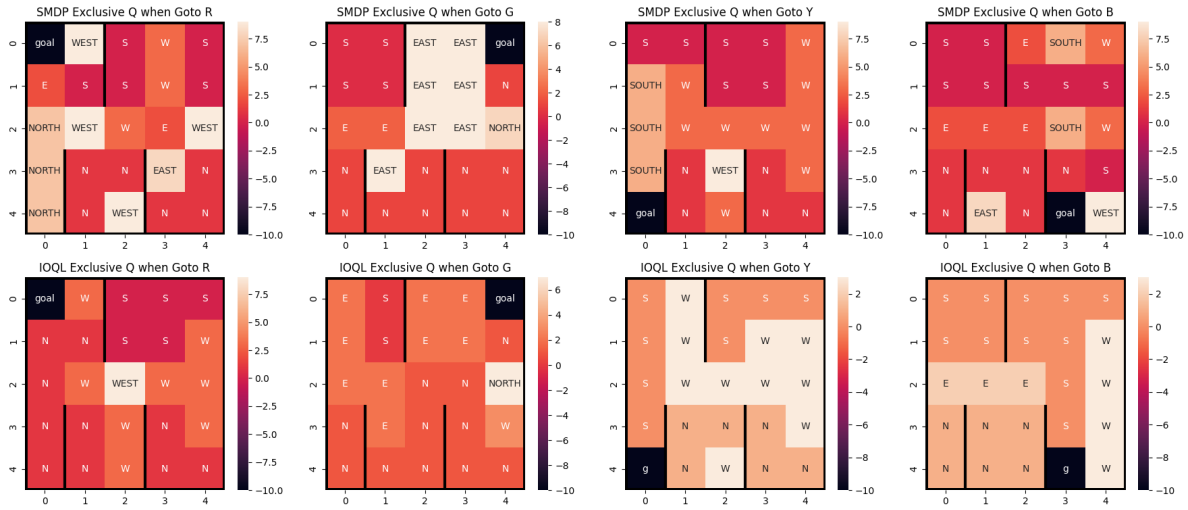

Figure 5: SMDP vs. IOQL (Exclusive)

The learnt policies follow the same behaviour as described in Section 3.2.2.

IOQL: We observe that the advantage IOQL boasts of increased number of updates on consistent policies is significantly reduced here; choosing a macro option results in its own and only the corresponding primitive actions' update. This reduces the sampling efficiency of the model, as indicated by the increase in regret from 2.13 (in Section 3.2.1) to 2.37.

SMDP: The options are not useful / don't fit in naturally to the temporal hierarchical abstraction of the problem. Since SMDP unlke IOQL does not update primitive actions during option execution, and the fact that updation frequency of macro vs one-step options is roughly the same, the algorithm struggles in its pursuit for the optimal policy, as shown by the drastic increase in regret from 2.37 (in Section 3.2.1) to 4.79.

3.4 Comparison between SMDP Q-Learning and Intra Option Q-Learning:

- In SMDP Q-Learning only one Q-learning update is performed at the state where the option was initated, after its termination.
- Intra-Option Q-Learning however is like a relational database; since mutually exclusive and indivisble components (primitive actions) are re-used to form bigger macro options, a state-transition accounts for much more updates (no. of consistent option policies) over SMDP.
- This makes IOQL more sample-efficient than SMDP, as shown by the regrets achieved by the respective algorithms in Section 3.2.1 and Section 3.3.1.

- Therefore IOQL is most suitable for the options framework, which is a composition of primitive actions. The options can be viewed as a user defined heuristic which pushes Q-Learning to traverse through important sections of the state space.
- SMDP Q-Learning's apt use case would rather be in actions that actually have varying times while being atomic; that is, they are not compositions of other actions (options).

4 Github Link

https://github.com/iamviveksrk/RL-PA3