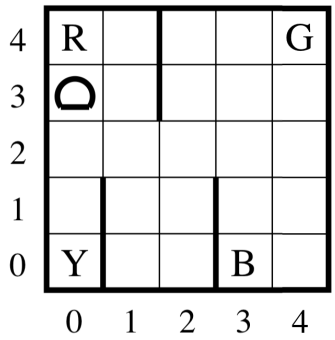
Reinforcement Learning: Solve OpenAI Gym's Taxi-v2 Task

For this coding exercise, you will use OpenAI Gym's Taxi-v2 environment to design an algorithm to teach a taxi agent to navigate a small gridworld. The goal is to adapt all that you've learned in the previous lessons to solve a new environment!



Before proceeding, read the description of the environment in subsection 3.1 of this paper (https://arxiv.org/pdf/cs/9905014.pdf):

Figure 1 shows a 5-by-5 grid world inhabited by a taxi agent. There are four specially-designated locations in this world, marked as R(ed), B(lue), G(reen), and Y(ellow). The taxi problem is episodic. In each episode, the taxi starts in a randomly-chosen square. There is a passenger at one of the four locations (chosen randomly), and that passenger wishes to be transported to one of the four locations (also chosen randomly). The taxi must go to the passenger's location (the "source"), pick up the passenger, go to the destination location (the "destination"), and put down the passenger there. (To keep things uniform, the taxi must pick up and drop off the passenger even if he/she is already located at the destination!) The episode ends when the passenger is deposited at the destination location.

There are six primitive actions in this domain: (a) four navigation actions that move the taxi one square North, South, East, or West, (b) a Pickup action, and (c) a Putdown action. Each action is deterministic. There is a reward of −1 for each action and an additional reward of +20 for successfully delivering the passenger. There is a reward of −10 if the taxi attempts to execute the Putdown or Pickup actions illegally. If a navigation action would cause the taxi to hit a wall, the action is a no-op, and there is only the usual reward of −1.

We seek a policy that maximizes the total reward per episode. There are 500 possible states: 25 squares, 5 locations for the passenger (counting the four starting locations and the taxi), and 4 destinations.

You can verify that the description in the paper matches the OpenAI Gym environment by peeking at the code here (https://github.com/openai/gym/blob/master/gym/envs/toy_text/taxi.py).

While this coding exercise is ungraded, we recommend that you try to attain an average return of at least 9.1 over 100 consecutive trials (best_avg_reward > 9.1).

In [2]:
```
import numpy as np
from collections import defaultdict, deque
import sys
import math
from agent import Agent
from monitor import interact
import gym
```

In [3]:
```
class Agent:

    def __init__(self, nA=6):
        """ Initialize agent.

        Params
        ======
        - nA: number of actions available to the agent
        """
        self.nA = nA
        self.Q = defaultdict(lambda: np.zeros(self.nA))
        self.epsilon = 0.1
        self.policy_s = np.ones(self.nA) * self.epsilon/self.nA
        self.alpha = 0.99
        self.gamma = 0.99

    def epsilon_greedy_probs(self, Q_s, i_episode):
        """ obtains the action probabilities corresponding to epsilon-greedy pol
icy """
        self.epsilon = 1./(4*i_episode)
        policy_s = np.ones(self.nA) * self.epsilon/self.nA
        policy_s[np.argmax(Q_s)] = 1 - self.epsilon + (self.epsilon/self.nA)
        return policy_s

    def select_action(self, state, i_episode):
        """ Given the state, select an action.

        Params
        ======
        - state: the current state of the environment
        - i_episode: the number of the episode

        Returns
        =======
        - action: an integer, compatible with the task's action space
        """
        # get epsilon-greedy action probabilities
        self.policy_s = self.epsilon_greedy_probs(self.Q[state], i_episode)
        # pick action A
        action = np.random.choice(np.arange(self.nA), p=self.policy_s)

        #action = np.random.choice(self.nA)
        return action

    def step(self, state, action, reward, next_state, done):
        """ Update the agent's knowledge, using the most recently sampled tuple.

        Params
        ======
        - state: the previous state of the environment
        - action: the agent's previous choice of action
        - reward: last reward received
        - next_state: the current state of the environment
        - done: whether the episode is complete (True or False)
        """

        # calculate the action-value function estimate using expected SARSA
        if not done:
            self.Q[state][action] += self.alpha*(reward + self.gamma*np.dot(self
.Q[next_state], self.policy_s) - self.Q[state][action])
        else:
            self.Q[state][action] += self.alpha*(reward  - self.Q[state][action]
)
```

In [4]:
```
def interact(env, agent, num_episodes=20000, window=100):
    """ Monitor agent's performance.

    Params
    ======
    - env: instance of OpenAI Gym's Taxi-v1 environment
    - agent: instance of class Agent (see Agent.py for details)
    - num_episodes: number of episodes of agent-environment interaction
    - window: number of episodes to consider when calculating average rewards

    Returns
    =======
    - avg_rewards: deque containing average rewards
    - best_avg_reward: largest value in the avg_rewards deque
    """
    # initialize average rewards
    avg_rewards = deque(maxlen=num_episodes)
    # initialize best average reward
    best_avg_reward = -math.inf
    # initialize monitor for most recent rewards
    samp_rewards = deque(maxlen=window)
    # for each episode
    for i_episode in range(1, num_episodes+1):
        # begin the episode
        state = env.reset()
        # initialize the sampled reward
        samp_reward = 0
        while True:
            # agent selects an action
            action = agent.select_action(state, i_episode)
            # agent performs the selected action
            next_state, reward, done, _ = env.step(action)
            # agent performs internal updates based on sampled experience
            agent.step(state, action, reward, next_state, done)
            # update the sampled reward
            samp_reward += reward
            # update the state (s <- s') to next time step
            state = next_state
            if done:
                # save final sampled reward
                samp_rewards.append(samp_reward)
                break
        if (i_episode >= 100):
            # get average reward from last 100 episodes
            avg_reward = np.mean(samp_rewards)
            # append to deque
            avg_rewards.append(avg_reward)
            # update best average reward
            if avg_reward > best_avg_reward:
                best_avg_reward = avg_reward
        # monitor progress
        print("\rEpisode {}/{} || Best average reward {}".format(i_episode, num_
episodes, best_avg_reward), end="")
        sys.stdout.flush()
        # check if task is solved (according to Udacity's specification)
        if best_avg_reward >= 9.1:
            print('\nEnvironment solved in {} episodes.'.format(i_episode), end=
"")
            break
        if i_episode == num_episodes: print('\n')
    return avg_rewards, best_avg_reward
```

In [5]:
```
# test the performance here
env = gym.make('Taxi-v2')
agent = Agent()
avg_rewards, best_avg_reward = interact(env, agent, 20000)
```

```
Episode 2751/20000 || Best average reward 9.1755
Environment solved in 2751 episodes.
```

In [ ]:
```

```