Stochastic Wind GridWorld King's Moves

Exercise 6.10: King's Move

Based from the plot that I was able to produce, it can be seen that with the fixed epsilon, the performance using SARSA is different from the other two methods. However, it can be seen that once the Variable Epsilon is used, the performances are roughly the same for all the different methods.

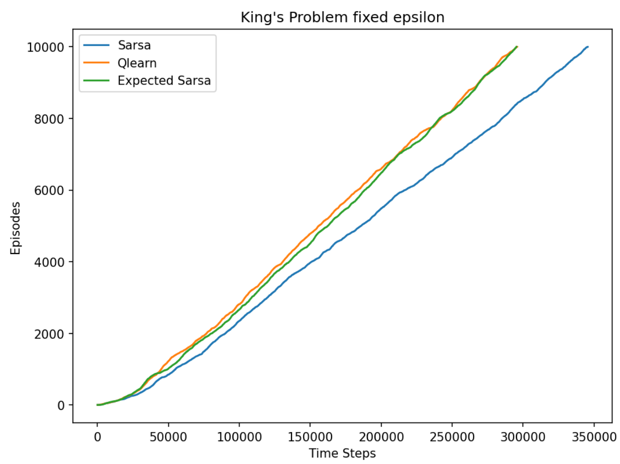

Figure 1: Fixed Epsilon

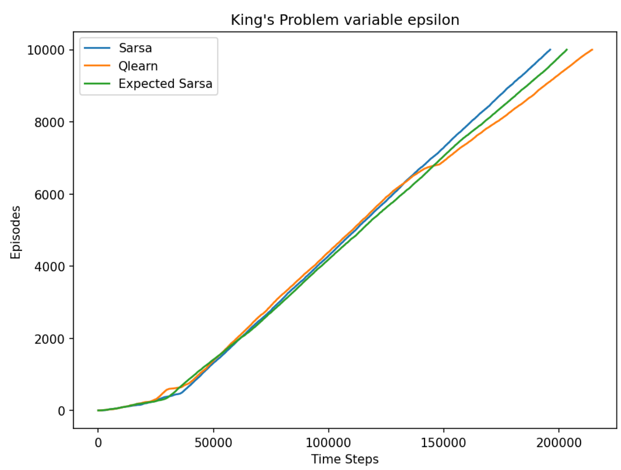

Figure 2: Variable Epsilon

Appendix

Python

```
'''
Justine Serdoncillo
IE 5571 - Dynamic Programming
HW 4 Exercise 6.10
November 6, 2023
'''

"""
Exercise 6.10: Stochastic Wind (programming) Re-solve the windy gridworld task with
 K i n g s moves, assuming that the  e ect  of the wind, if there is any, is stochastic, sometimes
varying by 1 from the mean values given for each column. That is, a third of the time
you move exactly according to these values, as in the previous exercise, but also a third
of the time you move one cell above that, and another third of the time you move one
cell below that. For example, if you are one cell to the right of the goal and you move
left , then one-third of the time you move one cell above the goal, one-third of the time
you move two cells above the goal, and one-third of the time you move to the goal.
"""

import numpy as np
import matplotlib.pyplot as plt
import random
import math

# %%
wind = [0, 0, 0, 1, 1, 1, 2, 2, 1, 0]

class Grid:
    def __init__(self, w, h, wind):
        self.w = w
        self.h = h
        self.wind = wind
        self.actions = list(range(9))
        self.states = [tuple([i, j]) for j in range(self.h) for i in range(self.w)]
        self.starting_state = tuple([0, 3])
        self.terminal_states = [tuple([7, 3])]

    def take_action(self, state, action):
        #locations are x, y
        x = state[0]
        y = state[1]
        win_rand = random.random()
        if win_rand <= 1/3:
            y += wind[x]
        elif win_rand <= 2/3:
            y += wind[x] + 1
        else:
            y += wind[x] - 1
        if action == 0: # up
            x += 0
            y += 1
        if action == 1: # up, right
            x += 1
            y += 1
        if action == 2: # right
            x += 1
            y += 0
        if action == 3: # right, down
            x += 1
            y += -1
        if action == 4: # down
            x += 0
            y += -1
        if action == 5: # left, down
            x += -1
            y += -1
        if action == 6: # left
```

```
            x += -1
            y += 0
        if action == 7: # up, left
            x += -1
            y += 1
        if action == 8: # No move
            x += 0
            y += 0
        if x>= self.w:
            x = self.w - 1
        if y >= self.h:
            y = self.h - 1
        if y < 0:
            y = 0
        if x < 0:
            x = 0
        r = -1
        return tuple([x, y]), r

# %% SARSA
class TDAlg:
    def __init__(self, problem, qs=None, policy=None, eps=.05, gamma=1, alpha=1, max_time=None, special=False):
        self.problem = problem
        if qs is None:
            self.qs = {s: {a: 0 for a in problem.actions} for s in problem.states}
        else:
            self.qs = qs
        if policy is None:
            self.policy = {s: random.choice(problem.actions) for s in problem.states}
        else:
            self.policy = policy
        self.eps = eps
        self.gamma = gamma
        self.alpha = alpha
        self.max_time = max_time
        self.state = None
        self.ep = 1
        self.time = 0
        self.ep_log = []
        self.r_log = []
        self.special = special

    def get_action(self, state):
        if self.special is not False:
            self.eps = self.special/self.ep
        if random.random() < self.eps:
            choice = random.choice(self.problem.actions)
        else:
            choice = self.best_action(state)
        return choice

    def best_action(self, state):
        choices = [a for a in self.problem.actions if self.qs[state][a] == max(self.qs[state].values())]
        best_action = random.choice(choices)
        return best_action

    def run_episode(self):
        if (self.max_time is not None and self.time <= self.max_time) or self.max_time is None:
            old_count = self.time
            cum_reward = self.update()
            self.ep_log.extend([self.ep for _ in range(self.time - old_count)])
            self.ep += 1
            if len(self.r_log) > 0:
                self.r_log.append(self.r_log[-1] + cum_reward)
            else:
                self.r_log.append(cum_reward)
        else:
            pass

```

```python
    def run(self, episodes):
        i = 0
        while i <= episodes:
            self.run_episode()
            i += 1

class Sarsa(TDAlg):
    def update(self):
        cum_reward = 0
        state = self.problem.starting_state
        action = self.get_action(state)
        while state not in self.problem.terminal_states:
            self.time += 1
            new_state, reward = self.problem.take_action(state, action)
            new_action = self.get_action(new_state)
            target = reward + self.gamma*self.qs[new_state][new_action] - self.qs[state][
    action]
            cum_reward += reward
            self.qs[state][action] += self.alpha*target
            action = new_action
            state = new_state
        return cum_reward

# %% Q-Learning

class QLearn(TDAlg):
    def update(self):
        cum_reward = 0
        state = self.problem.starting_state
        while state not in self.problem.terminal_states:
            self.time += 1
            action = self.get_action(state)
            new_state, reward = self.problem.take_action(state, action)
            cum_reward += reward
            best = self.best_action(new_state)
            target = reward + self.gamma*self.qs[new_state][best] - self.qs[state][action]
            self.qs[state][action] += self.alpha*target
            state = new_state
        return cum_reward

    def update_from_model(self, s, a, s_, r):
        best = self.best_action(s_)
        target = r + self.gamma*self.qs[s_][best] - self.qs[s][a]
        self.qs[s][a] += self.alpha*target

# %% Ex-pected SARSA

class ESarsa(TDAlg):
    def update(self):
        cum_reward = 0
        state = self.problem.starting_state
        while state not in self.problem.terminal_states:
            self.time += 1
            action = self.get_action(state)
            new_state, reward = self.problem.take_action(state, action)
            val = reward - self.qs[state][action]
            best = self.best_action(new_state)
            for a in self.problem.actions:
                if a == best:
                    val += (1-self.eps)*self.gamma*self.qs[new_state][a]
                val += (self.eps/len(self.problem.actions))*self.gamma*self.qs[new_state][a]
            target = val
            cum_reward += reward
            self.qs[state][action] += self.alpha*target
            state = new_state
        return cum_reward

# %% Fixed Epsilon
fig, ax = plt.subplots(figsize=(8,6), dpi=150)
ax.set_title("King's Problem fixed epsilon")
ax.set_xlabel("Time Steps")
ax.set_ylabel("Episodes")
```

```
grid = Grid(w=len(wind), h=7, wind=wind)
grid.actions = list(range(8))

sarsaFixed = Sarsa(grid, alpha=0.5, eps=.1)
sarsaFixed.run(10000)
ax.plot(sarsaFixed.ep_log, label='Sarsa')

QFixed = QLearn(grid, alpha=0.5, eps=.1)
QFixed.run(10000)
ax.plot(QFixed.ep_log, label='Qlearn')

EFixed = ESarsa(grid, alpha=0.5, eps=.1)
EFixed.run(10000)
ax.plot(EFixed.ep_log, label='Expected Sarsa')
ax.legend()

# %% Variable Epsilon
fig, ax = plt.subplots(figsize=(8,6), dpi=150)
ax.set_title("King's Problem variable epsilon")
ax.set_xlabel("Time Steps")
ax.set_ylabel("Episodes")

grid = Grid(w=len(wind), h=7, wind=wind)
grid.actions = list(range(8))

sarsaFixed = Sarsa(grid, alpha=0.5, eps=.3, special=.3)
sarsaFixed.run(10000)
ax.plot(sarsaFixed.ep_log, label='Sarsa')

QFixed = QLearn(grid, alpha=0.5, eps=.3, special=.3)
QFixed.run(10000)
ax.plot(QFixed.ep_log, label='Qlearn')

EFixed = ESarsa(grid, alpha=0.5, eps=.3, special=.3)
EFixed.run(10000)
ax.plot(EFixed.ep_log, label='Expected Sarsa')
ax.legend()
```

JUSTINE JOHN A. SERDONCILLO
IE 5571 REINFORCEMENT LEARNING
& DYNAMIC PROGRAMMING

NOVEMBER 6, 2023
HOMEWORK # 4

EXERCISE 5.6 WHAT IS THE EQUATION ANALOGOUS TO

$$V(s) \doteq \frac{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1} G_t}{\sum_{t \in \mathcal{T}(s)} \rho_{t:T(t)-1}}$$

for action values $Q(s,a)$

instead of state values $V(s)$, again given returns generated using b?

now track state-action pairs instead then

$$\boxed{Q(s,a) = \frac{\sum_{t \in \mathcal{T}(s,a)} \rho_{t:T(t)-1} G_t}{\sum_{t \in \mathcal{T}(s,a)} \rho_{t:T(t)-1}}}$$

EXERCISE 5.8 THE RESULTS WITH EX 5.5 & SHOWN IN FIG 5.4 USED A FIRST VISIT MC. CHANGE TO EVERY-VISIT MC, would variance of estimator $= \infty$? why/why not ?

Estimator will still be infinite because reward is at terminal still.
& the visits to the state will still run to infinity expected reward at every state $= 1$

EXERCISE 6.7 DESIGN AN OFF-POLICY VERSION OF THE TD(0) UPDATE THAT CAN BE USED WITH ARBITRARY TARGET POLICY π & COVERING BEHAVIOUR POLICY b;

USING AT EACH STEP THE IMPORTANCE SAMPLING RATIO $\rho_{t:t}$ (5.3)

$$\rho_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)\, p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)\, p(S_{k+1}|S_k, A_k)} = \prod_{k=t}^{t-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$$

$$\rho_{t:t} = \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$$

TD update can be chosen as (6.2)

$$V(S_t) = V(S_t) + \alpha\left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right]$$

Can now make episode is made by b so update becomes

$$\boxed{V_\pi(S_{t+1}) = V_\pi(S_{t+1}) + \alpha\left[\rho_{t:t} R_{t+1} + \rho_{t:t} V(S_{t+1}) - V(S_t)\right]}$$

based on b

EXERCISE 6.8 SHOW THAT AN ACTION-VALUE VERSION OF (6.6)

$G_t - V(S_t) = \sum_{k=t}^{T-1} \gamma^{k-t} \delta_k$ holds for the action-value form of the TD error $\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$, again assuming that the values don't change from step to step.

$$G_t - Q(S_t, A_t) = R_{t+1} + \gamma G_{t+1} - Q(S_t, A_t) + \gamma Q(S_{t+1}, A_{t+1}) - \gamma Q(S_{t+1}, A_{t+1})$$

$$= \delta_t + \gamma [G_{t+1} - Q(S_{t+1}, A_{t+1})]$$

$$= \delta_t + \gamma [\delta_{t+1} + \gamma [G_{t+2} - Q(S_{t+2}, A_{t+2})]]$$

$$= \delta_t + \gamma \delta_{t+1} + \gamma^2 \{ \dots \}$$

$$= \sum_{k=t}^{T-1} \gamma^{k-t} \delta_k$$

$$\boxed{\begin{aligned} G_t - Q(S_t, A_t) &= \sum_{k=t}^{T-1} \gamma^{k-t} \delta_k \\ \delta_t &= R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t) \end{aligned}}$$

EXERCISE 6.12 SUPPOSE ACTION SELECTION IS GREEDY. IS Q-LEARNING THEN EXACTLY THE SAME AS SARSA? WILL THEY MAKE EXACTLY THE SAME ACTION SELECTION & WEIGHT UPDATES?

---

if action selection is greedy then no means for exploration.

IF initialized at same Q for all S, then the two will make the same moves.

However, since initialization is bad to be random, the actions selected will be different & weight updates might be different too. & since greedy then no shot of converging to same solution.