Statistical and Machine Learning (01.113) - HW5 Question 3

Problem 4: Robot vs Snakes

In this problem, we will simulate a robot navigating a room trying to find the exit.

The robot is bleeding and loses 1 hitpoint for every step it takes before it finds the exit.

To make matters worse, there are 2 snakes in the centre of the room. If the robot steps onto a square with a snake, the snake will bite it and cause it to lose 15 hitpoints!

As the robot is terrified of being bitten, it is very nervous and does not strictly respond to commands;

- if told to move in a certain direction, it will only do so half the time.
- One quarter of the time it will instead move to the left of the commanded direction, and
- another quarter of the time it will move to the right of the commanded direction.

Can you help train the robot to find its way out? Begin by installing OpenAI's gym module (`pip install gym`), and include the following code in your script.

In [1]:

```
import numpy as np
import sys
from gym.envs.toy_text import discrete
```

In [2]:

```
UP = 0
RIGHT = 1
DOWN = 2
LEFT = 3

GOAL = 4 # upper right corner
START = 20 # lower left corner
SNAKE1 = 7
SNAKE2 = 17

eps = 0.25
```

In [3]:

```
class Robot_vs_snakes_world(discrete.DiscreteEnv):
    def __init__(self):
        self.shape = [5, 5]

        nS = np.prod(self.shape) # total states
        nA = 4 # total actions

        MAX_X, MAX_Y = self.shape

        P = {}
        grid = np.arange(nS).reshape(self.shape)
        """
        grid:
               [ 0,  1,  2,  3,  4],
               [ 5,  6,  7,  8,  9],
               [10, 11, 12, 13, 14],
               [15, 16, 17, 18, 19],
               [20, 21, 22, 23, 24]
        """
        it = np.nditer(grid, flags=['multi_index'])
        while not it.finished:
            s = it.iterindex
            y, x = it.multi_index

            P[s] = {a: [] for a in range(nA)}
            """{0: [], 1: [], 2: [], 3: []}"""
            is_done = lambda s: s == GOAL # location

            if is_done(s):
                reward = 0. # OK.
            elif s in [SNAKE1, SNAKE2]:
                reward = -15. # OUCH
            else:
                reward = -1. # BLEEDING

            if is_done(s):
                P[s][UP]=[(1.0,s,reward,True)]
                P[s][RIGHT]=[(1.0,s,reward,True)]
                P[s][DOWN]=[(1.0,s,reward,True)]
                P[s][LEFT]=[(1.0,s,reward,True)]
            else:
                # where to move (0 if cant move, else move)
                # (since s in [0, 1 .., 24]
                ns_up = s if y==0 else s-MAX_X
                ns_right = s if x==(MAX_X-1) else s+1
                ns_down = s if y==(MAX_Y-1) else s+MAX_X
                ns_left = s if x==0 else s-1

                # prob, next state , reward, boolean finish
                # reward in [0, -15, -1]
                P[s][UP] = [(1-(2*eps),ns_up,reward,is_done(ns_up)),
                            (eps,ns_right,reward,is_done(ns_right)),
                            (eps,ns_left,reward,is_done(ns_left))]

                P[s][RIGHT]=[(1-(2*eps),ns_right,reward,is_done(ns_right)),
                             (eps,ns_up,reward,is_done(ns_up)),
                             (eps,ns_down,reward,is_done(ns_down))]

                P[s][DOWN] = [(1-(2*eps),ns_down,reward,is_done(ns_down)),
```

```
                                (eps,ns_right,reward,is_done(ns_right)),
                                (eps,ns_left,reward,is_done(ns_left))]

                P[s][LEFT] = [(1-(2*eps) , ns_left , reward , is_done(ns_left))
,
                                ( eps , ns_up , reward , is_done(ns_up)) ,
                                ( eps , ns_down , reward , is_done(ns_down))]

            it.iternext()

        isd = np.zeros(nS)
        isd[START] = 1.
        self.P = P # {}

        super(Robot_vs_snakes_world, self).__init__(nS , nA , P , isd)

    def _render(self):
        grid = np.arange(self.nS).reshape(self.shape)
        it = np.nditer(grid, flags=['multi_index'])

        while not it.finished:
            s = it.iterindex
            y, x = it.multi_index

            if self.s == s:
                output = u" \u001b[30mR "
            elif s == GOAL:
                output = u" \u001b[32mG "
            elif s in [SNAKE1, SNAKE2]:
                output =  u" \u001b[31mS "
#                 output = " S "
            else:
                output = u" \u001b[37mo "

            if x == 0:
                output = output.lstrip()
            if x == self.shape[1] - 1:
                output = output.rstrip()

            sys.stdout.write(output)

            if x == self.shape[1] - 1:
                sys.stdout.write("\n")

            it.iternext()

        sys.stdout.write("\n")
```

You can write `env = Robot_vs_snakes_world()` to start the environment, which is a 5 x 5 grid.

Locations on the grid are numbered starting from 0 on the upper-left, and increasing in a row-wise manner; i.e. the upper-right is numbered 4 (exit/goal state), the lower-left (starting location of robot) is numbered 20, the lower-right corner is numbered 24, and so on. You can access the robot's location at any time using `env.s` and issue a command to move using `env.step(DIR)` where `DIR` is `UP` , `DOWN` , `LEFT` or `RIGHT`.

Finally, `env.p[state][action]` returns a list of tuples each corresponding to a possible next state j and of the form

- (probability of transitioning to state j, j, reward, goal state?).

In [4]:

```
env = Robot_vs_snakes_world()
env._render()
```

```
o  o  o  o  G
o  o  S  o  o
o  o  o  o  o
o  o  S  o  o
R  o  o  o  o
```

In [5]:

```
# env.P[current state][direction to move]
# probability, next state, next state reward , goal state
print(env.P[0])
```

```
{0: [(0.5, 0, -1.0, False), (0.25, 1, -1.0, False), (0.25, 0, -1.0,
False)], 1: [(0.5, 1, -1.0, False), (0.25, 0, -1.0, False), (0.25,
5, -1.0, False)], 2: [(0.5, 5, -1.0, False), (0.25, 1, -1.0, False),
(0.25, 0, -1.0, False)], 3: [(0.5, 0, -1.0, False), (0.25, 0, -1.0,
False), (0.25, 5, -1.0, False)]}
```

Value iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter:

- a small threshold $\theta > 0$ determining accuracy of estimation
- **Initialize** $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(\text{terminal}) = 0$

Loop until $\Delta <$ some threshold θ

- $\Delta \leftarrow 0$
- Loop for each $s \in S$:
- - $v \leftarrow V(s)$
- - $V(s) \leftarrow \max_a \sum_{s',r} p\left(s', r|s, a\right)\left[r + \gamma V\left(s'\right)\right]$
- - $\Delta \leftarrow \max(\Delta, |v - V(s)|)$

Output a deterministic policy $\pi \approx \pi_*$ such that

- $\pi(s) = \arg\max_a \sum_{s',r} p\left(s', r|s, a\right)\left[r + \gamma V\left(s'\right)\right]$

Greedy policy with respect to optimal value function:

$$\pi(s) \approx \arg\max_a \sum_{s',r} p\left(s', r|s, a\right)\left[r + \gamma v_*\left(s'\right)\right] = \arg\max_a q_*(s, a) = \pi_*(s)$$

In [6]:

```
theta = 0.0000001 # small algorithm parameter
discount_factor = 1.

iter_state = range(env.nS)
iter_action = range(env.nA)

def bellman_equation(state, V):
    state_values = []
    for action in iter_action:
        next_info = env.P[state][action]
        components = []
        for next_prob, next_state, reward, _ in next_info:
            components.append(next_prob * (reward + discount_factor * V[next_sta
te]))
        state_values.append(sum(components))
    return max(state_values)

def deterministic_policy(V, policy):
    for state in iter_state:
        state_values = []
        for action in iter_action:
            next_info = env.P[state][action]
            components = []
            for next_prob, next_state, reward, _ in next_info:
                components.append(next_prob * (reward + discount_factor * V[next
_state]))
            state_values.append(sum(components))
        a = np.argmax(state_values)
        policy[state, a] = 1
    return policy

def value_iteration(env):
    # initialise V(s) for all s
    V = np.zeros(env.nS)
    while True:
        delta = 0
        for state in iter_state:
            v_prev = V[state]
            V[state] = bellman_equation(state, V)
            #update delta
            delta = max(delta, np.abs(v_prev - V[state]))
        if delta<theta:
            break

    # Create a deterministic policy using the optimal value function
    policy = np.zeros([env.nS, env.nA])
    policy = deterministic_policy(V, policy)
    return policy, V
```

In [8]:

```
env = Robot_vs_snakes_world()
policy, V = value_iteration(env)

env._render()
while env.s != GOAL:
    DIR = np.argmax(policy[env.s])
    env.step(DIR)
    env._render()
```

```
o  o  o  o  G
o  o  S  o  o
o  o  o  o  o
o  o  S  o  o
R  o  o  o  o

o  o  o  o  G
o  o  S  o  o
o  o  o  o  o
o  o  S  o  o
o  R  o  o  o

o  o  o  o  G
o  o  S  o  o
o  o  o  o  o
o  o  S  o  o
o  o  R  o  o

o  o  o  o  G
o  o  S  o  o
o  o  o  o  o
o  o  S  o  o
o  o  o  R  o

o  o  o  o  G
o  o  S  o  o
o  o  o  o  o
o  o  S  R  o
o  o  o  o  o

o  o  o  o  G
o  o  S  o  o
o  o  o  o  o
o  o  S  o  R
o  o  o  o  o

o  o  o  o  G
o  o  S  o  o
o  o  o  o  R
o  o  S  o  o
o  o  o  o  o

o  o  o  o  G
o  o  S  o  o
o  o  o  o  R
o  o  S  o  o
o  o  o  o  o

o  o  o  o  G
o  o  S  o  o
o  o  o  R  o
o  o  S  o  o
o  o  o  o  o

o  o  o  o  G
o  o  S  o  o
o  o  o  o  R
o  o  S  o  o
o  o  o  o  o

o  o  o  o  G
```

```
o  o  S  o  o
o  o  o  R  o
o  o  S  o  o
o  o  o  o  o

o  o  o  o  G
o  o  S  o  o
o  o  o  o  o
o  o  S  R  o
o  o  o  o  o

o  o  o  o  G
o  o  S  o  o
o  o  o  o  o
o  o  S  o  R
o  o  o  o  o

o  o  o  o  G
o  o  S  o  o
o  o  o  o  o
o  o  S  R  o
o  o  o  o  o

o  o  o  o  G
o  o  S  o  o
o  o  o  o  o
o  o  S  o  R
o  o  o  o  o

o  o  o  o  G
o  o  S  o  o
o  o  o  o  R
o  o  S  o  o
o  o  o  o  o

o  o  o  o  G
o  o  S  o  R
o  o  o  o  o
o  o  S  o  o
o  o  o  o  o

o  o  o  o  G
o  o  S  o  R
o  o  o  o  o
o  o  S  o  o
o  o  o  o  o

o  o  o  o  R
o  o  S  o  o
o  o  o  o  o
o  o  S  o  o
o  o  o  o  o
```

In [12]:

```
dir_map = {0: "U", 1:"R", 2:"D", 3:"L"}

def analyse_policy(policy):
    grid = np.arange(env.nS).reshape(env.shape)
    it = np.nditer(grid, flags=['multi_index'])

    while not it.finished:
        s = it.iterindex
        y, x = it.multi_index

        if s in [SNAKE1, SNAKE2]:
            best_dir = np.argmax(policy[s])
            output = u" \u001b[31m" + f"{dir_map[best_dir] } "
        else:
            best_dir = np.argmax(policy[s])
            output = u" \u001b[30m" + f"{dir_map[best_dir] } "

        if x == 0:
            output = output.lstrip()
        if x == env.shape[1] - 1:
            output = output.rstrip()
        sys.stdout.write(output)
        if x == env.shape[1] - 1:
            sys.stdout.write("\n")
        it.iternext()

    sys.stdout.write("\n")

print("Best Actions")
analyse_policy(policy)
```

```
Best Actions
R  R  U  R  U
U  U  R  R  U
U  U  R  R  U
U  L  R  R  U
R  R  D  R  U
```