K. J. Somaiya College of Engineering, Mumbai-77
(A Constituent College of Somaiya Vidyavihar University)
Department of Computer Engineering

Batch: RL-2	**Roll No.: 16010121129**
Experiment / assignment / tutorial No. 08	
Grade: AA / AB / BB / BC / CC / CD /DD	
Signature of the Staff In-charge with date	

TITLE: 8 To implement n-step learning approach

AIM:

Expected OUTCOME of Experiment: (Mention CO/CO's attained here): CO3

Books/ Journals/ Websites referred:

Richard S. Sutton and Andrew G. Barto, "*Reinforcement Learning:An Introduction",* The MIT Press,Second Edition, 2018

Pre Lab/ Prior Concepts:

n-step reinforcement learning, in which 'n' is the parameter that determines the number of steps that we want to look ahead before updating the Q-function. So for n=1, this is just "normal" TD learning such as Q-learning or SARSA. When n=2, the algorithm looks one step beyond the immediate reward, it looks two steps beyond, etc.

Both Q-learning and SARSA have an n-step version.

An algorithm for doing n-step learning needs to store the rewards and observed states for steps, as well as keep track of which step to update.

Chosen Problem Statement:

Obstacle avoidance in grid world

The obstacle grid is a set of 16 states of 4 rows and 4 columns, where the goal node is marked in green and obstacles are marked in red. The goal of the learning is to avoid the obstacles and find a path which is the shortest distance from the goal node. The starting state can be any state. The actions can be any from left, right, up and down for all states not on the border, only actions that don't take you out-of-bounds are possible for border states.

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

Explain following concepts w.r.t. chosen problem statement:

Policy:

A policy is a mapping from states to actions. In this problem, the policy is taken as a random policy to take any move which takes you forward.

Reward function:

The reward function assigns a reward to each state-action pair. In this problem, the reward function rewards the agent for reaching the goal state ('P') and penalizes the agent for getting stuck in obstacles.

Value function:

The value function estimates the expected return from a state. In this problem, the value function is used to estimate the expected number of steps it will take for the agent to reach the goal state from a given state.

Model of the environment:

The model of environment tracks the agent's current state and the available actions and rewards for each action, and then update the agent's state based on the chosen action. The states and actions are modelled as follows:

```
moves = {
    'A': {
        'down': 'E',
        'right': 'B',
        'moves': ['down', 'right'],
    },

    'D': {
        'down': 'H',
        'left': 'C',
        'moves': ['down', 'left'],
    },

    'M': {
        'up': 'I',
        'right': 'N',
        'moves': ['up', 'right'],
    },

    'P': {
        'up': 'L',
        'left': 'O',
        'moves': ['up', 'left'],
    },

    'B': {
        'left': 'A',
        'right': 'C',
        'down': 'F',
        'moves': ['down', 'right', 'left'],
    },

    'C': {
        'left': 'B',
        'right': 'D',
        'down': 'G',
        'moves': ['down', 'right', 'left'],
```

```
},

'E': {
    'up': 'A',
    'right': 'F',
    'down': 'I',
    'moves': ['down', 'right', 'up'],
},

'I': {
    'up': 'E',
    'right': 'J',
    'down': 'M',
    'moves': ['down', 'right', 'up'],
},

'N': {
    'up': 'J',
    'right': 'O',
    'left': 'M',
    'moves': ['left', 'right', 'up'],
},

'O': {
    'up': 'K',
    'right': 'P',
    'left': 'N',
    'moves': ['left', 'right', 'up'],
},

'L': {
    'up': 'H',
    'down': 'P',
    'left': 'K',
    'moves': ['left', 'down', 'up'],
},

'H': {
    'up': 'D',
    'down': 'L',
    'left': 'G',
    'moves': ['left', 'down', 'up'],
},
```

```
    'F': {
        'up': 'B',
        'down': 'J',
        'left': 'E',
        'right': 'G',
    },

    'G': {
        'up': 'C',
        'down': 'K',
        'left': 'F',
        'right': 'H',
    },

    'J': {
        'up': 'F',
        'down': 'N',
        'left': 'I',
        'right': 'K',
    },

    'K': {
        'up': 'G',
        'down': 'O',
        'left': 'J',
        'right': 'L',
    },
}
```

The rewards for each state is given as:

```
rewards = {
    'A': 1,
    'B': 2,
    'C': -1,
    'D': 1,
    'E': -2,
    'F': 3,
    'G': 1,
    'H': -1,
    'I': 3,
    'J': 4,
    'K': -1,
```

```
    'L': 1,
    'M': 4,
    'N': 5,
    'O': 6,
    'P': 10,
}
```

States:

```
states = [
        'A', 'B', 'C', 'D',
        'E', 'F', 'G', 'H',
        'I', 'J', 'K', 'L',
        'M', 'N', 'O', 'P'
    ]
```

The obstacles and actions are given:

```
obstacles = ['E', 'K', 'C', 'H']
actions = ['left', 'right', 'down', 'up']
```

Program for N-Step Temporal Difference learning technique:

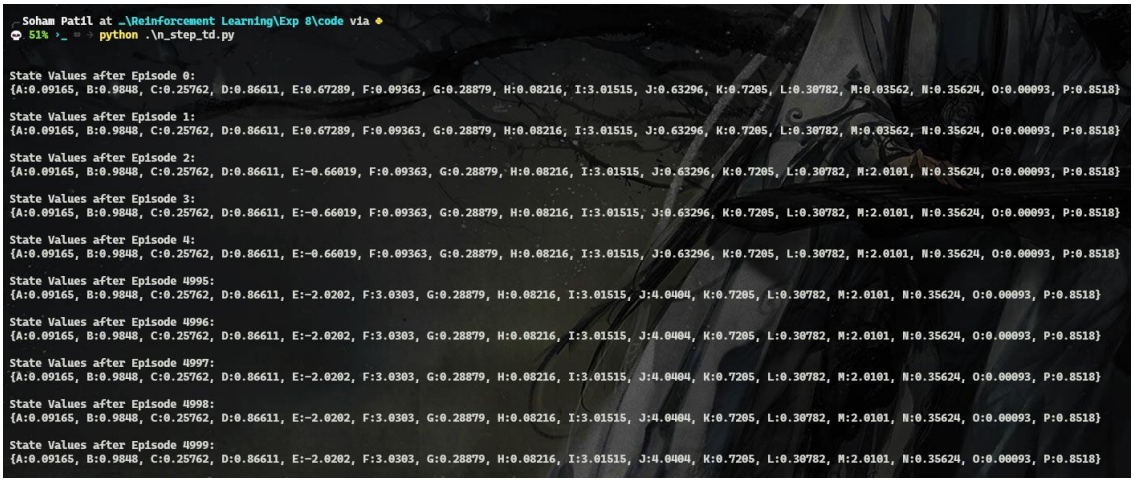
```
import random

obstacles = ['E', 'K', 'C', 'H']
actions = ['left', 'right', 'down', 'up']

states = [
    'A', 'B', 'C', 'D',
    'E', 'F', 'G', 'H',
    'I', 'J', 'K', 'L',
    'M', 'N', 'O', 'P'
]


episodes = [] # format: ['s', 'a', 'st+1', 'st+2', 'r1', 'r2']

def gen_episodes(num_episode: int = 0):
    global episodes
    global obstacles
    global states

    for _ in range(num_episode):

        state_t = states[random.randint(0, 15)]

        try:
            key = random.randint(0, 3)
            state_t1 = moves[state_t][actions[key]]
            state_t2 = moves[state_t][actions[key]]
            action = actions[key]
        except KeyError:
            try:
                key = random.randint(0, 2)
                try:
                    action = moves[state_t]['moves'][key]
                    state_t1 = moves[state_t][action]
                    state_t2 = moves[state_t][action]
                except IndexError:
                    key = 0
                    action = moves[state_t]['moves'][key]
                    state_t1 = moves[state_t][action]
                    state_t2 = moves[state_t][action]
            except KeyError:
```

```
                key = random.randint(0, 1)
                try:
                    action = moves[state_t]['moves'][key]
                    state_t1 = moves[state_t][action]
                    state_t2 = moves[state_t][action]
                except IndexError:
                    key = 0
                    action = moves[state_t]['moves'][key]
                    state_t1 = moves[state_t][action]
                    state_t2 = moves[state_t][action]

        reward = rewards[state_t1]

        #return [state_t, action, state_t1, reward]
        episodes.append([state_t, action, state_t1, reward])

def initialize_policy():
    Q = {}
    for state in states:
        try:
            state_actions = moves[state]['moves']
        except KeyError:
            state_actions = actions
        for action in state_actions:
            Q[state, action] = 0 #random.random()

    return Q

def get_action(state):
    try:
        state_actions = moves[state]['moves']
    except KeyError:
        state_actions = actions

    max_value = -99999
    max_action = ''
    for a in state_actions:
        if Q[state, a] > max_value:
            max_value = Q[state, a]
            max_action = a

    return max_action

import csv
```

```
Q = initialize_policy()

with open('values.csv', 'a', newline='') as file:
    file = csv.writer(file)
    file.writerow(states)

def value_update():
    global episodes

    alpha = 0.5
    discount_factor = 0.1

    values = {}
    for state in states:
        values[state] = random.random()

    n = 2
    env = Environment()

    for _ in range(len(episodes)):
        state_t = episodes[_][0]
        env.reset(state_t)
        T = 999
        time_step = 0

        for t in range(T):
            if time_step == T - 1:
                break

            if t<T:
                action_t = get_action(env.current_state)
                state_t1, reward_t1, done = env.step(action_t)
                if state_t1 == env.goal_state:
                    T = t + 1

            time_step = t - n + 1

            if time_step >= 0:
                return_t = 0

                for i in range(time_step+1, min(time_step+n, T)):
                    action_t = get_action(env.current_state)
                    state_t1, reward_t1, done = env.step(action_t)
```

```
                    return_t += discount_factor**(i-time_step-
1)*reward_t1
                    if time_step + n < T:
                        return_t += discount_factor**n * values[state_t1]

                values[state_t1] = values[state_t1] + alpha*(return_t -
values[state_t1])

        row = list(values.values())

        with open('values.csv', 'a', newline='') as file:
            file = csv.writer(file)
            file.writerow(row)

        if _ in list(range(0, 5)) or _ in list(range(len(episodes)-5,
len(episodes)+1)):
            print(f'State Values after Episode {_}:')
            print(values)

    return values

gen_episodes(1000)
value_update()
```

Output:



All values update per episode in values.txt

Conclusion: Thus, we conclude that N-step TD learning technique can be used to optimize the action value function for a grid world obstacle problem. It showed that the agent is able to learn an optimal policy of the expected rewards from each state, action pair.

Post Lab Descriptive Questions:

How the book keeping required for n-step algorithms can be handled?

To handle the bookkeeping required for n-step algorithms in reinforcement learning, you can follow these steps:

1. Initialize a list or buffer to store the transitions experienced by the agent during each episode.

2. At each time step, record the current state, chosen action, reward received, and next state.

3. After storing a transition in the buffer, check if there are enough previous transitions to form an n-step sequence.

4. If there are enough transitions available (i.e., length of buffer >= n), calculate the discounted return for that sequence using rewards obtained from all future steps within that sequence.

5. Update your value function or policy based on this n-step return instead of just considering immediate rewards at each step.

6. Once an update is made using an n-step return, remove the oldest transition from your buffer until it has fewer than n transitions remaining.

7. Repeat these steps until your training process converges or reaches a predefined stopping criterion.

By following this approach, a sliding window of past experiences is maintained in order to compute accurate estimates of returns over multiple time steps while avoiding excessive storage requirements for long sequences encountered during exploration and training phases in reinforcement learning algorithms with n-steps updates like N-Step TD methods or N-Step Q-Learning variants.

Date: _____ **Signature of faculty in-charge**