CS747: Assignment 2

Aryan Bhosale
210040024
Indian Institute of Technology, Bombay

I. Task 1: Implementation of MDP Planning Algorithms

Problem Statement

Given an MDP, your program must compute the optimal value function V and an optimal policy π^* by applying the algorithm that is specified through the command line. Create a python file called planner.py which accepts the following command-line arguments.*

You are not expected to code up a solver for LP; rather, you can use available solvers as blackboxes. Your effort will be in providing the LP solver the appropriate input based on the MDP, and interpreting its output appropriately. Use the formulation presented in class. We require you to use the Python library PuLP. PuLP is convenient to use directly from Python code: here is a short tutorial and here is a reference. PuLP version 2.4 is included in the requirements.txt file for the environment.

You are expected to write your own code for Value Iteration and Howard's Policy Iteration; you may not use any custom-built libraries that might be available for the purpose. You can use libraries for solving linear equations in the policy evaluation step but must write your own code for policy improvement. Recall that Howard's Policy Iteration switches all improvable states to some improving action; if there are two or more improving actions at a state, you are free to pick anyone.

In this section, we calculate the optimal value function and policy using Linear Programming, Value Iteration and Howard's Policy Iteration. These have been implemented in code. The considerations and theory behind the implementation have been described in the following sections.

A. Code Structure

The code processes input from a file as required by the problem statement. The contents of this file are converted to a dictionary such that all associated information is easily accessible. A Solver class has been written which implements the required algorithm as specified on the command line.

1) Code Description: Please refer to coloured annotations for detailed descriptions.
Variables

Listing 1. Reading the file

```
file_path = args.mdp
#initialize an empty dictionary to store contents of the read file
trnsn_fn = {}
rew_fn = {}

with open(file_path, 'r') as file:
    #reading the file line-by-line
    #the loop which structures the information as required
    for line in file:
        #split at line break
        line_components = line.strip().split()
        keywrd = line_components[0]
        if keywrd == "numStates":
            n_states = int(line_components[1])
        elif keywrd == "numActions":
            n_acts = int(line_components[1])
        elif keywrd == "end":
            sink = line_components[1]
            trnsn_fn = np.zeros((n_states, n_acts, n_states))
            rew_fn = np.zeros((n_states, n_acts, n_states))

        elif keywrd == "transition":
            s, a, s_next, r, prob = map(float, line_components[1:])
            trnsn_fn[int(s),int(a),int(s_next)] = prob
            rew_fn[int(s),int(a),int(s_next)] = r
        elif keywrd == "mdptype":
            mdptype = line_components[1]
        elif keywrd == "discount":
            discount = float(line_components[1])



#the structure that stores all information about the MDP which has been extracted from the file
     and might be required in consequent parts
mdp = {
'n_states': n_states,
'n_acts': n_acts,
'sink': sink,
'transitions': trnsn_fn,
'rewards': rew_fn,
'mdptype': mdptype,
'discount': discount,
}
```

B. Linear Programming

$$\begin{aligned} \text{maximize} \quad & -\sum_{s\in S} V(s) \\ \text{subject to} \quad & V(s) \geq \sum_{s'\in S} T(s,a,s')\left[R(s,a,s') + \gamma V(s')\right] \\ & \forall s \in S, a \in A \end{aligned}$$

where $T(s,a,s')$ is the transition function from state s to state s' through action a, $R(s,a,s')$ is the reward function from state s to state s' through action a and $V(s)$ is the value function from state s Using the Bellman Operator and the Bellman Optimality Operator we have reduced the solution of the above MDP to al linear programming. V* is the unique solution to the above problem. Once we have the optimal value function, we can find one of the optimal policies. We solve the maximization problem of -V as a minimization problem of V.

1) Code Description: Please refer to coloured annotations for detailed description.
Variables

Listing 2. Function Description
```
def lp_solve(self, mdp):
    self.mdp = mdp #extracted using a separate method and then passed as input here
    self.problem = p.LpProblem("MDP_Problem", p.LpMinimize)
    val = np.array(list(p.LpVariable.dicts("val", [i for i in range(self.mdp['n_states'])]).
        values()))
    self.problem += p.lpSum(val) #defining the thing that is to be minimized
    for state in range(self.mdp['n_states']):
        for action in range(self.mdp['n_acts']):
            self.problem += val[state] >= p.lpSum(bellman_rhs(self.mdp, val, stat=state , act=
                action)) #defining all of the constraints

    self.problem.solve(p.apis.PULP_CBC_CMD(msg=0))#function that actually solves the linear
        programming
    #converting the output of the library into usable form
    vals_list = list(map(p.value, val))
    vals_arr = np.array(vals_list)
    sum_vals = np.sum(bellman_rhs_arr(self.mdp, val_arr=vals_arr), axis = -1)

    policy = np.argmax(sum_vals, axis = -1)#calculating one of the optimal policies

    return vals_arr, policy
```

Listing 3. Helper Functions

```
#calculates the rhs of the bellman operator on V for a given state and action
def bellman_rhs(mdp, val, stat, act):
    return mdp['transitions'][stat, act]*(mdp['rewards'][stat, act] + mdp['discount']*val)
```

C. Value Iteration

The iterative update is defined as:
$V_{t+1}(s) \leftarrow \max_a \sum_{s'} T(s,a,s')\left(R(s,a,s') + \gamma \cdot V_t(s')\right)$
where $T(s,a,s')$ is the transition function from state s to state s' through action a, $R(s,a,s')$ is the reward function from state s to state s' through action a and $V_{t+1}(s)$ and $V_t(s)$ are the value functions from state s at time t+1 and t respectively.

Convergence of this method is proved using the Banach Fixed Point Iteration Method and that the Bellman Optimality Operator is a contraction mapping. Thus, we can iteratively find the optimal value function V*.

1) Code Description: Please refer to coloured annotations for detailed description.
Variables

Listing 4. Function Description

```
def val_iter(self, mdp):
    self.mdp = mdp
    val_t = np.zeros(self.mdp['n_states'])
    val_t1 = np.zeros(self.mdp['n_states'])
    n_iters = 0
    #loop that iterates over and over until an acceptable value is reached
    while diff_tol(val_t, val_t1, n_iters) and n_iters<5000000:
        n_iters += 1
        val_t = val_t1
        val_t1 = np.max(np.sum(bellman_rhs_arr(self.mdp, val_arr=val_t), axis = -1), axis = -1)
    #finding one of the optimal policies
    val_pol = np.argmax(np.sum(bellman_rhs_arr(self.mdp, val_arr=val_t1), axis = -1), axis =
        -1)

    return val_t1, val_pol
```

Listing 5. Helper Function

```
def diff_tol(val_t, val_t1, n_iters):
    if n_iters == 0:
        return True

    return np.max(np.abs(val_t1-val_t)) > 1e-9
```

D. Howard's Policy Iteration

Named after Ronald Howard, Howard's Policy Iteration tries to find an optimal policy as opposed to other methods which find an optimal value function. It updates policies by looking for actions that might provide better value estimates. It focuses on Howard's Policy Iteration is a variation of the policy iteration algorithm used to solve Markov Decision Processes (MDPs). Unlike standard policy iteration, Howard's method iteratively refines a policy by considering actions that provide better value estimates within the constraints of the policy. It does this through an iterative process where it focuses on improving actions for states with suboptimal policies. It focuses on states where the current policy performs suboptimally and tries to find actions that can lead to better value estimates. It updates till no improvement can be done on the given policy.

1) Code Description: Please refer to coloured annotations for detailed description.
Variables

Listing 6. Function Description

```
def howrd_iter(self, mdp):
    self.mdp = mdp
    pol_pi = np.zeros(self.mdp['n_states'])
    pol_pi_1 = np.zeros(self.mdp['n_states'])
    val_fn = np.zeros(self.mdp['n_states'])
    #updation loop
    while (1):
        val_fn = find_val_fn(self.mdp, pol=pol_pi)
        sum_vals = np.sum(bellman_rhs_arr(self.mdp, val_arr=val_fn), axis = -1)
        pol_pi_1 = np.argmax(sum_vals, axis = -1)

        if np.array_equal(pol_pi, pol_pi_1):
            break

        else:
            pol_pi = pol_pi_1
    return val_fn, pol_pi_1
```

Reward Updation

Listing 7. Helper Function

```
def find_val_fn(mdp, pol):
    #function which calculates value function given policy
    t = np.zeros((mdp['n_states'], mdp['n_states']))
    r = np.zeros((mdp['n_states'], mdp['n_states']))

    for state in range(mdp['n_states']):
        action = int(pol[state])

        t[state] = mdp['transitions'][state, action]
        r[state] = mdp['rewards'][state, action]

    mul = np.sum(np.multiply(t, r), axis = 1)

    calc_val_fn = np.linalg.solve(np.identity(mdp['n_states'])-mdp['discount'] *t, mul)
    return calc_val_fn
```

II. Task 2: Formulation of an MDP

Problem Statement

You are the manager of a 2-member football team. You want to prepare a strategy that maximises the chances of scoring goals by doing MDP Planning! You are tasked with the problem of Half Field Offense - scoring a goal with two attackers against one defender. The football half pitch is a 4x4 grid with a goal that is 2 units long. You have two players - B1 and B2, with possession of the ball, whose skill levels for different actions are parameterised by p and q in [0, 0.5] and [0.6, 1]. Note that p and q correspond to different actions by these players, and are identical for both players. The opponent has a defender R. You must formulate this as an MDP and score a goal before you lose possession! The game ends when the players score a goal, or lose possession, whichever occurs first.

1) Code Structure: encoder.py formulates the MDP which is then solved by planner.py and represented in the desired form by decoder.py

2) Encoder: It consists primarily of conditional loops which calculate the transition function and probabilities for each of the possible states. The policy being followed by the opponent is also provided in which all possible states have been listed along with the probability of L, R, U, D actions for the opponent. The conditionals can be broken down into many different categories.

- Action Based: Depeding on which it is a movement action, Pass, Shoot or Sink State. The game ends if the opponent takes the ball or a goal is scored
- Possession Based: The probabilities vary depending on whether the player in question in is possession of the ball or not.
- Outcome Based: The state reached as a result of the action varies depending on whether the action(for eg. a pass or a movement) was a success or a failure.
- Based on the Opponents Policy: The state we reach also depends on what action the opponent took. The probabilities for each action also need to be accounted for in the transition function.

It is important to map the states to integers such that they can be interpreted by the consequent code.

3) Planner: This solves the MDP formulated by encoder.py and outputs the solution in its own format. Has been explained in detail in previous sections.

4) Decoder: Maps states back and converts the outputs of planner.py to a more comprehensible form.

Listing 8. Opponent Reader Description

```
#sets up the mapping as required later
opponent_path = args.opponent
#initialize an empty dictionary to store contents of the read file
state = {}
states_list = []
states_mapping_dict = {}
inv_mapping = {}
counter = 0
with open(opponent_path, 'r') as file:
    #reading the file line-by-line
    for line in file:
        #split at line break
        line_components = line.strip().split()
        keywrd = line_components[0]
        if keywrd == "state":
            continue
        else:
            opp_probs = np.zeros(4)
            for i in range(1, 5):
                opp_probs[i-1] = line_components[i]

            this_state = keywrd
            states_list.append(this_state)
            state[this_state] = opp_probs
            states_mapping_dict[this_state] = counter
            inv_mapping[counter] = this_state
            counter += 1
```

Listing 9. Policy Reader Description

```
policy_path = args.value-policy
trnsn_fn = {}
action = {}
count = 0
with open(policy_path, 'r') as file:
    #reading the file line-by-line
    for line in file:
        #split at line break
        count+= 1
        line_components = line.strip().split()
        trnsn_fn[count] = line_components[0]
        action[count] = line_components[1]
```

Listing 10. Printer

```
for j, t in enumerate(trnsn_fn):
    state = str(inv_mapping[j])
    transition = str(t)
    act = str(action[j])

    print(state+" "+act+" "+transition)
```

6) Analysis: As we know, p and q values determine majority of the probabilities calculated. In my opinion, increasing the value of p should decrease the probability of winning since it has an indirect relation with accuracy of movement. On the other hand, increasing the value of q should increase the probability of winning since it has a direct relation with accuracy of passing.