CS 747 : Assignment 2
MDP Planning and the Cricket Game

Shivam Patel, 200070077

October 12, 2022

1 Analysis of Cricket Planning

1.1 Winning Probability vs q_value : Part 1

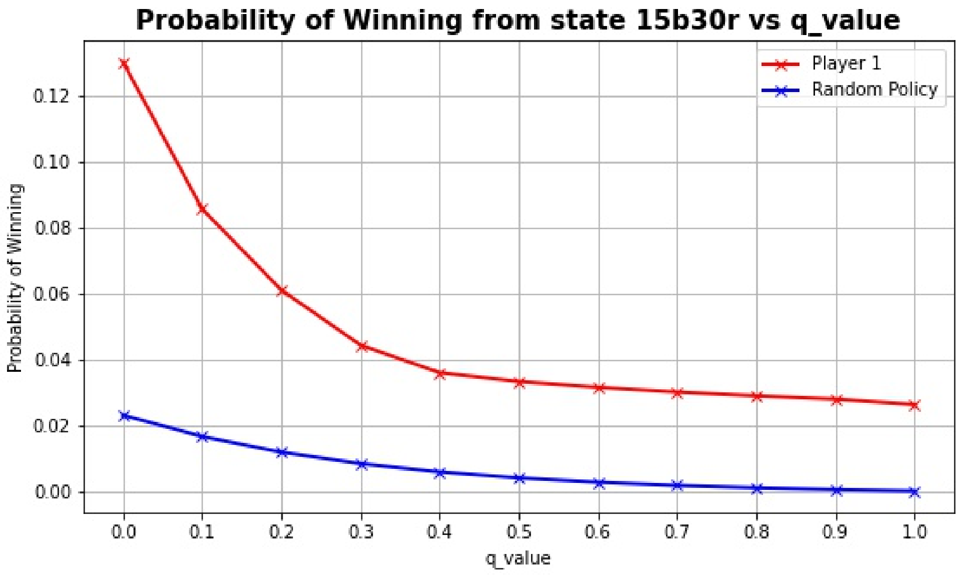

Figure 1: Winning Probability vs q_value, Part1

- In the graph of winning probability vs q_value, we see that as the q_value increases, i.e. the probability of player B getting out at each ball increases, the winning chances decrease.
- This is a monotonically decreasing trend, as there is no case possible where increasing q_value will increase the chances of winning from any state for which player A plays.
- Also, the chances for random policy are always less than the optimal policy. In the degenerate case where q_value=1, the game ends in a loss for all episodes when player B comes to play

1.2 Winning Probability vs Target Runs : Part 2

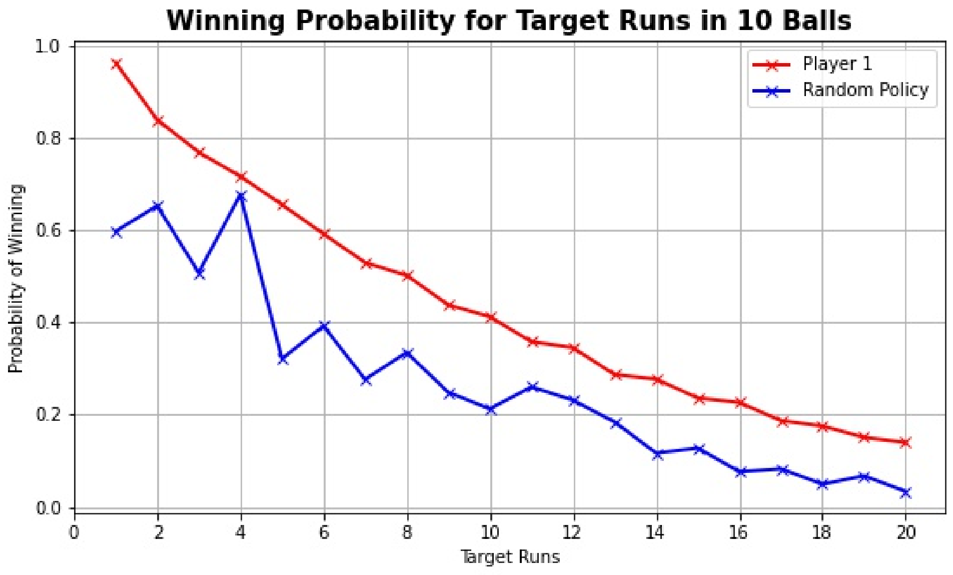

Figure 2: Winning Probability vs Target Runs, Part2

- In the graph of winning probability vs target runs, we see that the chances of winning decrease as the target increases. This is justified as whenever there is an increase in the target within same number of remaining balls, it becomes difficult to achieve it.

- This monotonicity is easily seen for the optimal case. But in the random policy case, we observe that the winning probability is not strictly monotonic, because the policy we are following is non optimal, and may lead to even better chances of winning for a higher target.
- There cannot exist any point on the curve for random policy above the optimal policy, which indicates that random policy is better than the optimal policy, which is a fallacy in itself, as otherwise this random policy would have been the optimal policy, which is not the case here.

1.3 Winning Probability vs Remaining Balls : Part 3

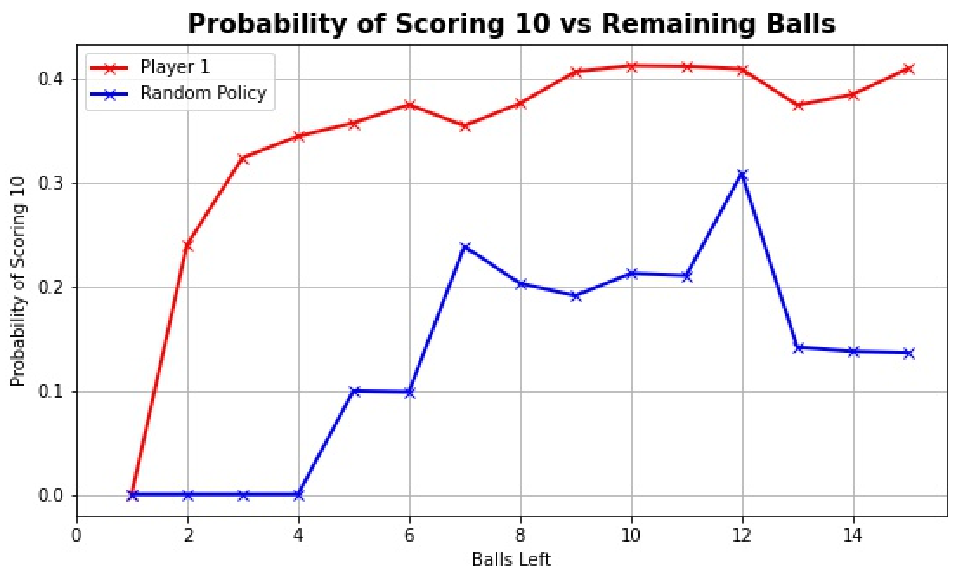

Figure 3: Winning Probability vs Remaining Balls

- In the graph of winning probability vs remaining balls, we find that the line for optimal policy is always above the random policy file (as optimal policy gives the highest probability of winning from any state of bbrr as compared to any other policy).

- An important point to note is that there are dips in the graph for optimal policy at balls 7 and 13, as the strike is likely to change in the very next ball, and the batting goes in the hands of the weak player B, which reduces the chances of winning as compared to the a ball more or less for the same target runs.

2 MDP Planning

2.1 Problem Statement

We aim at implementing various algorithms for MDP planning and evaluation, namely value iteration, linear programming and howard policy iteration. We use a black box linear programming solver PulP.

2.2 Methods

The **Value Iteration, Linear Programming and Howard Iteration** are standard algorithms described in lectures. Thus, an in detail explanation is not required.

2.3 Code Snippets

Value Iteration

```
def value_iteration(mdp_info, dict_mdp):
    ...
    v_t = np.array([0]*num_states)
    v_t1 = np.array([0]*num_states)+0.1
    for end in end_states:
        v_t1[end] = 0

    for i in range(500000):
        if np.max(np.abs(v_t1-v_t)) < 1e-12:
            break
        v_t = v_t1.copy()
        for s1 in set(range(mdp_info[0])).difference(set(end_states)):
            arr = [0]*num_actions # len = num_actions
            for ele in dict_mdp[s1]:
                arr[ele[1]]+=ele[4]*(ele[3]+gamma*v_t[ele[2]])
            v_t1[s1] = max(arr)

    # now, the value function is calculated,
    # we need to take argmax at each state for finding optimal policy
    for s1 in set(range(mdp_info[0])).difference(set(end_states)):
        arr = [0]*mdp_info[1] # len = num_actions
        for ele in dict_mdp[s1]:
            arr[ele[1]]+=ele[4]*(ele[3]+gamma*v_t[ele[2]])
        policy[s1] = arr.index(max(arr))
        # argmax over actions for each state

    return v_t, policy
```

Linear Programming

```
def linear_solver(mdp_info, dict_mdp, dict_pol_eval):
    ...
    prob = LpProblem('MDP-valuefn' ,LpMinimize)
    vars = LpVariable.dicts('state_list',(states))
    prob += (lpSum([vars[s] for s in range(num_states)]) )
    for keys in dict_pol_eval.keys():
        # ele = [s2,r,p]
        prob += lpSum([ele[2]*(ele[1] + gamma*vars[ele[0]]) \\
                for ele in dict_pol_eval[keys]]) <= vars[keys[0]]
    if end_states[0]!=-1:
        for ends in end_states :
            prob += vars[ends]==0
    prob.solve(PULP_CBC_CMD(msg=0))
    # now we have the variables and their optimum value
    x_star=[0]*mdp_info[0]
    for i in range(num_states):
        x_star[i] = vars[i].value()

    policy = [0]*mdp_info[0]
    # now we find the optimal policy
    for s1 in range(mdp_info[0]):
        if s1 in end_states:
            policy[s1]=0
            continue
        arr = [0]*mdp_info[1] # len = num_actions
        for ele in dict_mdp[s1]:
            arr[ele[1]]+=ele[4]*(ele[3]+gamma*x_star[ele[2]])
        policy[s1] = arr.index(max(arr))
        # argmax over actions for each state

    return x_star, policy
```

Howard Policy Iteration

```
def howard_solver(mdp_info, dict_mdp, dict_pol_eval, policy_list):
    # howard solver for finding optimal policy
    num_states = mdp_info[0]
    end_states = mdp_info[2]
    v_current = policy_eval(policy_list, dict_pol_eval, mdp_info)

    d = {i:0 for i in range(num_states)}
    for s1 in set(range(num_states)).difference(set(end_states)):
        d[s1] = get_qa(dict_mdp, mdp_info, dict_pol_eval,\\
                policy_list, v_current, s1)

    new_policy_list = improve_policy(policy_list, d, mdp_info)
    if new_policy_list == policy_list:
        return v_current, policy_list
    else:
        policy_list = new_policy_list
        return howard_solver(mdp_info, dict_mdp, dict_pol_eval, policy_list)
```

Policy Evaluation

```
def policy_eval(policy_list, dict_pol_eval, mdp_info):
    end_states = mdp_info[2]
    gamma = mdp_info[4]
    num_states = mdp_info[0]
    # dict_pol_eval[tuple([s1,ac])].append([s2,r,p])
    # dict_mdp[s1].append([s1,ac,s2,r,p])
    # structure of dictionaries

    a = np.zeros((num_states, num_states))
    # holds coefficients of all v_pi(s)
    b = np.zeros(num_states) # holds the constants in => a*v_pi = b
    for s1 in range(num_states):
        if s1 in end_states:
            b[s1]=0
            a[s1,s1]=1
            # setting zero condition for terminating states
            continue

        for ele in dict_pol_eval[(s1,policy_list[s1])]:
            b[s1] += -ele[1]*ele[2]
            a[s1,ele[0]] += ele[2]*gamma
        a[s1,s1] += -1

    v_pi = np.linalg.solve(a,b)
    return v_pi
```

3 Cricket MDP planning

3.1 Problem Statement

In this real world application of MDPs and planning, we try to implement and formulate an optimal policy for a middle order batter, given the tail ender and the opposing team are a factor of the nature/game environment and out of our control. We formulate an encoder and a decoder, which facilitate the solvers obtained in part1 to predict the correct optimal policy.

3.2 Methods

Here, I have implemented an MDP for 'two' players A and B. There are *num_balls* and *num_runs* given to us through the cricket_states.txt file. There are $num\_balls * num\_runs$ possible states in which player A can be, and take actions as per his choice. But, as player B is a part of the environment, we have to make his outcomes independent of his actions. Thus, for all other $num\_balls * num\_runs$ states for player B, we take all actions with outcomes as [-1,0,1] w.p. [q,(1-q)/2,(1-q)/2], where q is the 'weakness' parameter of player B.

Here is a part of the mdp file -

```
transition 1403a 3 1303a 0 0.025
transition 1403a 3 1302b 0 0.05
transition 1403a 3 1301a 0 0.075
transition 1403a 3 win 1 0.15
transition 1403a 3 win 1 0.5
transition 1403a 4 end 0 0.4
transition 1403a 4 1303a 0 0.05
transition 1403a 4 win 1 0.25
transition 1403a 4 win 1 0.3
transition 1403b 0 end 0 0.9
transition 1403b 0 1303b 0 0.05
transition 1403b 0 1302a 0 0.05
transition 1403b 1 end 0 0.9
```

Such encoded states are then converted to integers mappings, through the following function -

```
def give_state_integer(state, max_balls, max_runs):
    if state=='win':
        return 1
    elif state=='end':
        return 0
    balls = int(state[0:2])
    runs = int(state[2:4])
    player = str(state[-1])
    if balls==max_balls:
        if runs==max_runs:
            return 2
        elif player=='a':
            return (max_runs-runs)*2 + 1
        elif player=='b':
            return (max_runs-runs)*2 + 2
    else:
        if player=='a':
            return 2*max_runs*(max_balls-balls) + (max_runs-runs)*2 + 1
        elif player=='b':
            return 2*max_runs*(max_balls-balls) + (max_runs-runs)*2 + 2
```

After encoding to integers, the mdp looks like this.

```
transition 115 3 175 0 0.025
transition 115 3 178 0 0.05
transition 115 3 179 0 0.075
transition 115 3 1 1 0.15
transition 115 3 1 1 0.5
transition 115 4 0 0 0.4
transition 115 4 175 0 0.05
transition 115 4 1 1 0.25
transition 115 4 1 1 0.3
transition 116 0 0 0 0.9
transition 116 0 176 0 0.05
transition 116 0 177 0 0.05
transition 116 1 0 0 0.9
```

4 References and Acknowledgements

This assignment and code is completely mine, and I have taken no direct code from anywhere.

[1] PulP Linear Solver