CS747-Foundations of Intelligent and Learning Agents

Assignment 2

Pinkesh Raghuvanshi (200050106)

11-10-2022

Contents

Directory Structure

```
|
|__planner.py
|__encoder.py
|__decoder.py
|__report.pdf
```

Task1 - MDP Planning Algorithms

Value Iteration

Approach: I first started with a random policy and then updated the policy using the following equation:

$$V_{t+1}(s) = max_{a \in A} \sum_{s' \in S} T(s, a, s')(R(s, a, s') + \gamma V_t(s')) \tag{1}$$

I stopped when the value functions converged. For convergence, I checked that the norm of vector V(t+1) - V(t) does not exceed a certain threshold.

Linear Programming

Approach: I used Pulp for solving linear programming. The constraints given to the linear program were:

$$V(s) >= \sum_{s' \in S} T(s, a, s')(R(s, a, s') + \gamma V(s')), \forall a \in A, \forall s \in S. \tag{2}$$

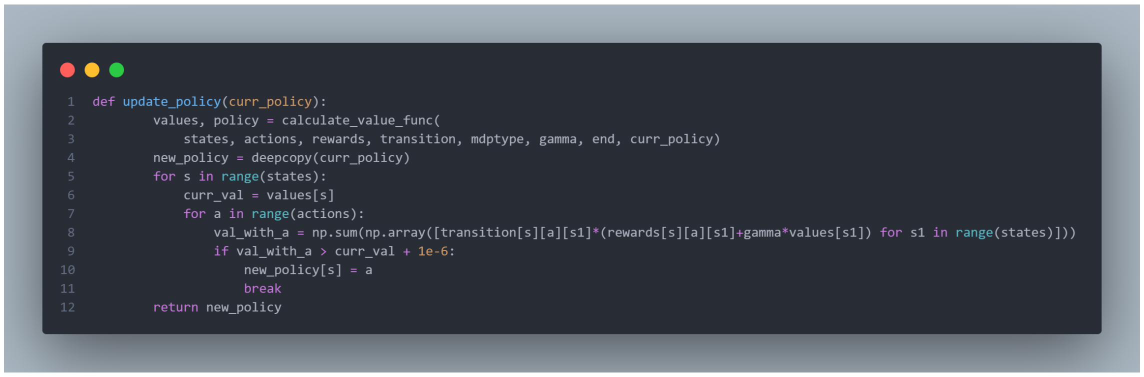
The objective of this linear program was to minimize the objective function:

$$\sum_{s \in S} V(s) \tag{3}$$

We proved in class that the solution to the above linear program gives us the optimal value function. To get the optimal policy from the value function, I checked for each state, what action to take so that the value function matches.

Howard Policy Iteration

Approach: I started with a random policy and updated it every time till no improvement was possible i.e., the improved policy is no better than the current policy. Following code snippets shows how I updated the policy in each iteration.



```
def update_policy(curr_policy):
        values, policy = calculate_value_func(
            states, actions, rewards, transition, mdptype, gamma, end, curr_policy)
        new_policy = deepcopy(curr_policy)
        for s in range(states):
            curr_val = values[s]
            for a in range(actions):
                val_with_a = np.sum(np.array([transition[s][a][s1]*(rewards[s][a][s1]+gamma*values[s1]) for s1 in range(states)]))
                if val_with_a > curr_val + 1e-6:
                    new_policy[s] = a
                    break
        return new_policy
```

Figure 1: Policy Update Function

Task2 - MDP for Cricket Game

Formulation

Let the total number of balls left be B and the total number of runs left be R.

States:
I took a total of 2*B*R + 2 states represented by a five-digit number, where the first two digits represent the number of balls left, followed by two digits for runs left, and the last digit represents the batsman who has the strike, 0 for middle order batsman and 1 for tailender. Apart from these 2*B*R states, I also took two additional states as winning and losing terminal states.

Actions:
There are a total of five actions numbered 0,1,2,3,4 indicating that the batsman tries to score 0,1,2,4,6 runs. For the tailender batsman, all the actions are equivalent in terms of the probability of outcome.

Transitions and Rewards:
I iterated over all (state1, action, outcome) tuple where outcome is [-1,0,1,2,3,4,6] for middle order batsman and [-1,0,1] for tailender. Then I found the state which will be achieved if this outcome is faced from this state. I updated the runs, balls, strike as per the rules of cricket and found the new state, state2. The probability of this transition is as per the parameters given for middle-order batsman and dependent on q for tailender batsman.
Reward for this transition is 1 only if the number of runs required is 0 after this transition, otherwise reward is one. In this case, if the runs required are 1, state2 is the winning state, and if the outcome is -1 or the number of balls left is 0 (and runs required are not 0, then state2 is the losing state.

Gamma:
Gamma for this mdp is 1 since this is an episodic task.

An optimal policy for the above batsman gives the action the middle order batsman should take given the situation of cricket match. Note that since all the actions for tailender batsman are equivalent, they really does not have a choice and therefore the only agent in this setup is the middle order batsman.

Analysis

Plot 1

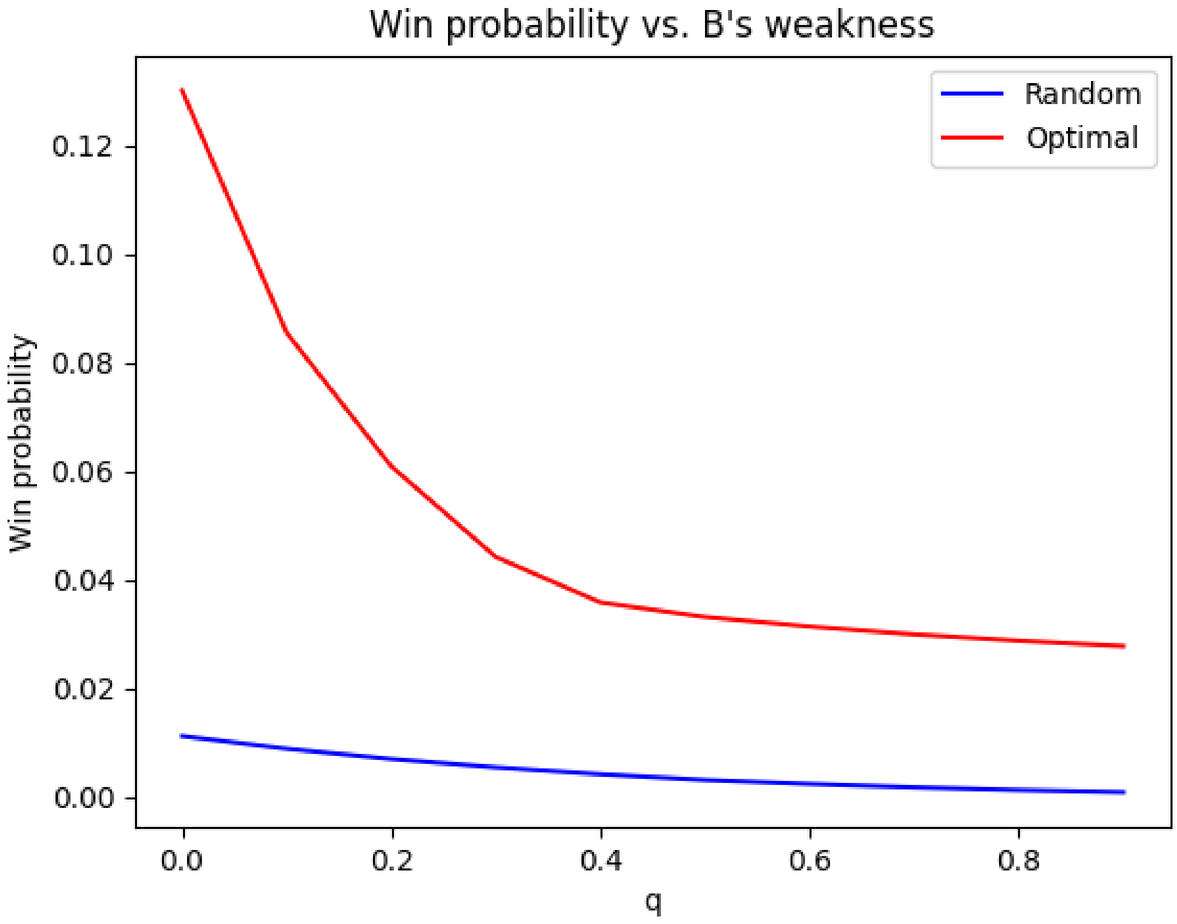


Figure 2:

Observation:
Keeping the number of runs required and the number of balls left constant, the probability of winning decreases as the weakness of player B increases for both the random and optimal policy.
This is obvious, as having a better tailender batsman always helps.

Also, the optimal policy always outperforms the random policy. Also the rate of decrease in winning probability is much larger in optimal policy. This might be because as the weakness of player B increases, A does not have much of a choice. He/She has to keep the strike with themselves and take very minimal risk.

Plot 2

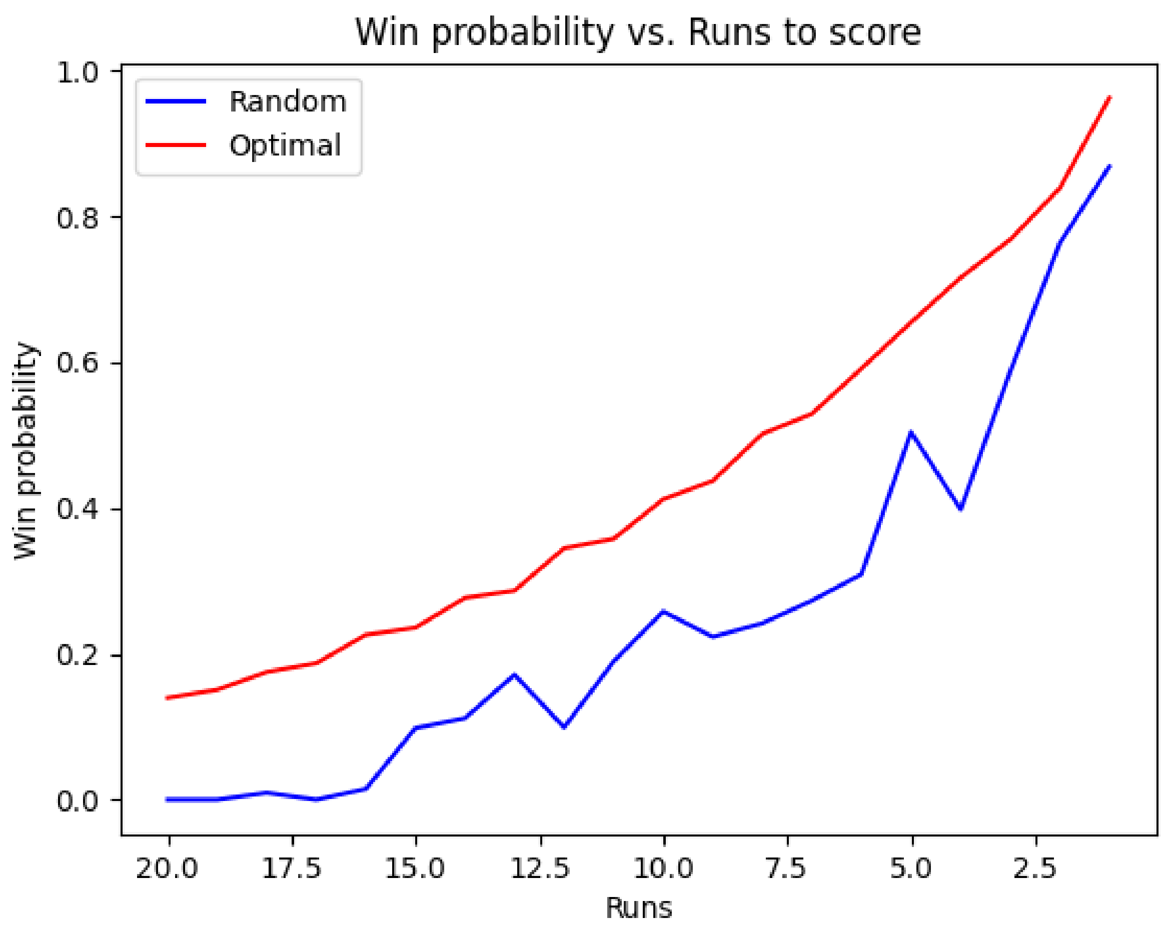


Figure 3:

Observation:
Keeping the number of balls left and the weakness of player B constant, the probability of winning increases as the number of runs required decreases for both the random and optimal policy. This is also obvious because it is easier to score fewer runs than to score more runs.

Here too, the optimal policy outperforms random policy.

Plot 3

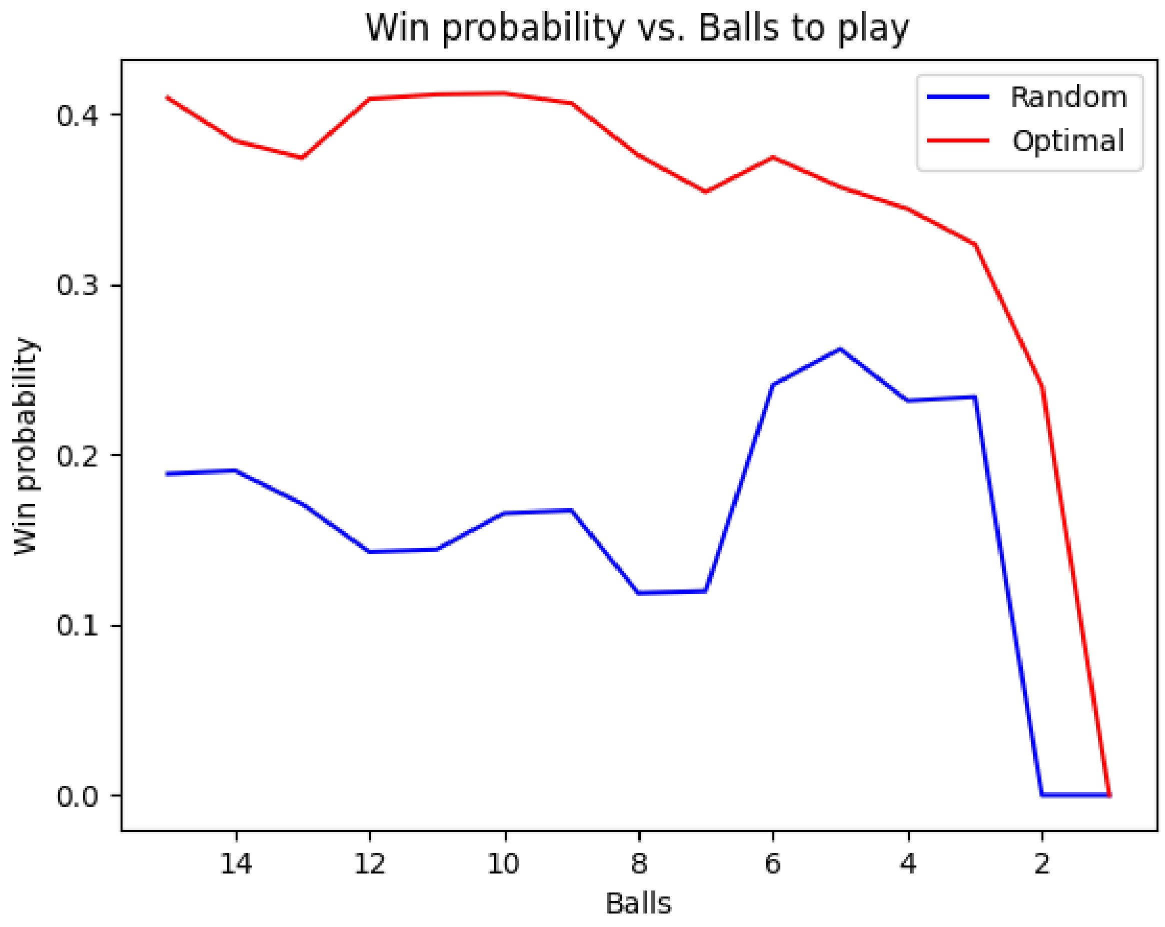


Figure 4:

Observation:
Keeping the number of runs required and the weakness of player B constant, the probability of winning decreases as the number of balls left decreases for both the random and optimal policy. This is also obvious because it is easier to score a certain amount of runs in more balls. This gives the batsman the flexibility to take less risk.

Here too, the optimal policy outperforms random policy.