CS747: Programming Assignment 2

Aayush Rajesh
200070001

1 Task 1

In the implementation of Value Iteration as well as Value Evaluation (which is used in both Howard Policy Iteration as well as Policy Evaluation), we take our initial estimated value function to be 0 for all states. The iteration proceeds until machine precision, which is taken to be when the max norm of the difference between V_t and V_{t+1} is at most 1e-12. That is, successive application of the Bellman operator continues until:

$$\|V_{t+1} - V_t\|_\infty < 10^{-12}$$

For Howard Policy Iteration, we consider improvable actions to be those whose action-value function is greater than the value function, evaluated for the policy at that instant, by at least 1e-6. For all states with at least one improvable action, we switch the policy by choosing one of these actions at random.

In the implementation of Linear Programming, we have used the solver `PULP_CBC_CMD`. The problem is posed with the objective of maximizing the negative of the sum of the value function for all states.

Whether the MDP is continuing or episodic does not make a difference in the implementations. In the case of episodic MDPs, the value function of the terminal states is fixed to 0 by defining their state transition probability density function as:

$$T(s_T, a, s) = 0 \qquad \forall s \in S,\ \forall a \in A$$

The default algorithm is set to be Value Iteration since we can exercise control over its precision (by varying the tolerance we have used in the max norm while defining machine precision). In general, this fact holds for my implementation of Howard Policy Iteration as well.

2 Task 2

2.1 MDP Formulation

The MDP generated by my implementation of `encoder.py` uses the same states as generated by `cricket_states.py`, with the addition of two terminal states. State 0 is a terminal state which denotes losing the game and State 1 is a terminal state which denotes winning the game. Following these two are all the states stored in `cricket_state_list.txt`, with numbering beginning with 2. Therefore, if the states in the text file are stored in a list, the label of a particular state `bbrr` in my

MDP would be 2 more than its corresponding list index.

Each state in my MDP represents the game situation when Player A is on strike (except for the end states, which represent the general game situation at the end). Transitions that involve taking an action and obtaining an outcome that results in Player A retaining strike are straightforward and are expressed as a transition between the current and next state, with transition probabilities corresponding to the action taken and outcome as given in the file containing Player A parameters.

In the case where an action is taken resulting in the outcome of Player A losing strike, we utilize a defined function called `PlayerB`. Let Player A take such an action at the state `curr_state`. The function `PlayerB` generates **all** possible sequences that can be played out by Player B, while calculating the probability of the sequence occurring. The sequence is thus comprised of outcomes where Player B scores 0 runs, 1 run, or gets out. Whenever a sequence ends with Player B losing the strike, winning the game, or getting out (and hence losing the game), a transition is created from `curr_state` to the state of the game upon termination of the sequence. This state is either a win/lose state or a state with Player A on strike. The rewards are assigned accordingly (1 for winning, 0 for anything else), and the probability of the transition is equal to the calculated probability of the sequence panning out the way it did. It is crucial to note that the function `PlayerB` is Markovian as well.

To keep track of transitions, we maintain a dictionary with keys as the tuple $(s, a, s\prime)$, and values $r, p)$. Whenever a transition is created, it is passed to a function called `store`. The function checks if an entry for the transition has already been created in the dictionary. If not, the entry is created, otherwise, the probability of transition is added to the probability of the previous entry. The purpose of this is to take care of situations where different outcomes upon taking the same action still lead to the same state (for example, the outcomes of scoring 1, 2, 3, 4, and 6 lead to a win when there is just 1 run required).

The win state is achieved whenever the required runs become less than or equal to 0. The lost state is achieved whenever the balls remaining becomes zero before the required runs are stored, or a player gets out. The MDP is stored as an episodic MDP, with discount equal to 1, so that the value functions of each state are identical to the win probability.

2.2 Plots

The plot showing the variation in win probability with the weakness of Player B from a given state requiring 30 runs to win off 15 balls is depicted in Figure 1. From the plot, it is evident that our computed optimal policy performs better than the random policy provided. Furthermore, we can observe that as the weakness of Player B increases (approaches 1), our win probability decreases. This is expected, since the other player is more susceptible to getting out off a given ball, and is less likely to keep a hold of their wicket and pass the strike back to Player 1. Nevertheless, the optimal policy still provides a non-zero probability of winning for weakness = 1, while that given by the random policy is negligible.

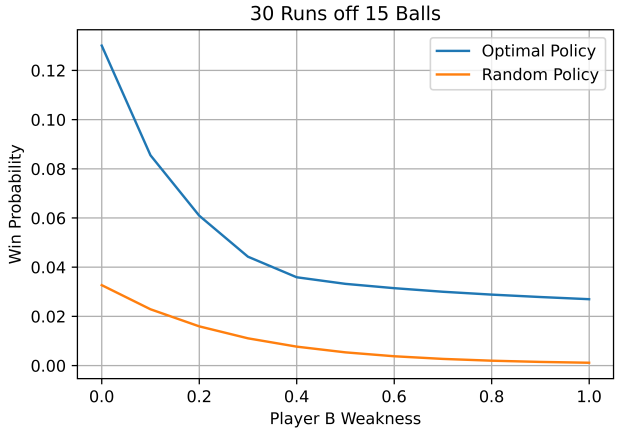

Figure 1: Win Probability vs Player B Weakness

The plot showing the variation in win probability as the target runs are varied is depicted in Figure 2. As expected, the probability of winning increases as the required runs off a fixed number of balls increases, due to more required with a fixed resource (here, the number of balls). When the required runs are just 1, the probability of winning is close to 1. Even here, the policy generated by the code performs better than the random policy. However, while the random policy depicts the overall trend in the decrease of win probability, it has a very jagged variation, due to the randomness in actions taken for a given state.

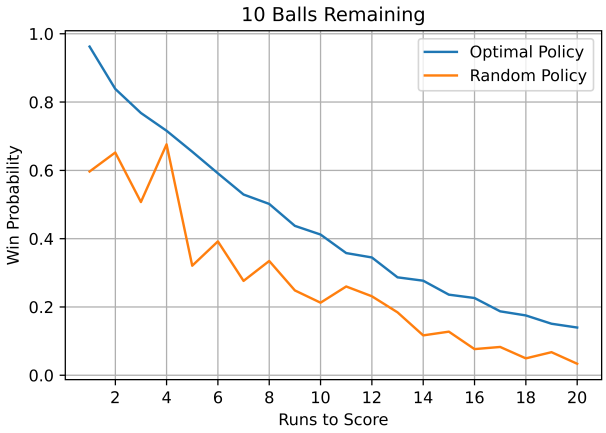

Figure 2: Win Probability vs Required Runs

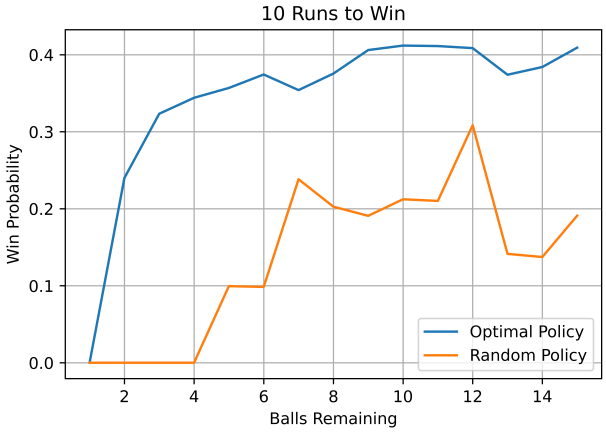

Figure 3: Win Probability vs Balls Remaining

The plot showing the variation in win probability as the number of remaining balls is varied is depicted in Figure 3. We observe that the chance of winning increases as the number of balls to achieve the fixed target is increased. This is expected, as the resources (here, the number of balls) available to achieve a fixed goal are increased. Once again, the optimal policy performs better than the random policy, which has has highly fluctuating win probabilities in this case. When the number of balls remaining is 1, the win probability is 0, as the target of 10 runs is impossible to achieve in the current setting.

An interesting observation is the small dip in probability every time the number of balls is of the form $6k+1$. This can be attributed to the fact that this represents the last ball of each over, and the immediate goal of Player A is to retain strike for the next over, which involves scoring 1 or 3 runs exactly. Furthermore, if Player A fails to retain strike, then the onus is on Player B to hold onto their wicket and pass strike back to Player A, without wasting too many balls and compromising the game situation. All these factors together influence the probability of winning, and hence the chance of winning when 10 runs are required off 13 balls is slightly less than when the same runs are required off 12 balls.