CS 747 : Assignment 2

Sainath Vavilapalli, 200050125

October 2022

Contents

1 Task 1

The code for this section can be found in planner.py. I used vectorized operations for all the three algorithms. Value iteration is the default algorithm for this task. I constructed two matrices of size (A, S, S) named transitionMatrix and rewardMatrix respectively. Here, A ,S denote the number of actions and states respectively. transitionMatrix[a][s_1][s_2] gives the probability of going from state s_1 to state s_2 by taking action a. rewardMatrix[a][s_1][s_2] gives the reward for the above transition.
I defined a function called *getValue* which returns the value for each state given a policy and the above transition and reward Matrices along with the discount.

1.1 Value Iteration

For value iteration, I maintained a predefined threshold, epsilon to be equal to 1e-7. As long as there is an improvement of epsilon or higher in the present value compared to the previous value, we keep on performing the update step.

1.2 Howard's Policy Iteration

I initiliazed the policy with some random values. In each iteration, we calculate the action values for all the states, and see if there are any improvable states. I updated the policy for all the improvable states. The action with maximum value is chosen in this implementation. I used the *getValue* function in each iteration to obtain the values for each state.

1.3 Linear Programming

I used PulP for implementing a Linear Programming based solution. I first defined a set of variables and constraints using PulP and provided the LP with an optimization function. There are a total of nk constraints. There is a constraint for every state action pair. I named the constraints as C_state_action. After solving the LP, I looked at the values of all constraints and found the tight constraints by their values, i.e, I consider constraints with values less than 1e-7 to be tight constraints. We get a tight constraint for each state. Based on the constraint name encoding I previously mentioned, I assigned the optimal action for each state based on the tight constraint for that state. In addition, if the MDP is episodic, I added additonal constriaints to emphasize that all end states have value 0.

1.4 General Observations

I observed that all of the above techniques were returning the correct optimal policy every for all the test cases. However, a higher value of epsilon was needed in the case of value iteration for episodic tasks.

2 Task 2

2.1 Encoding the MDP

2.1.1 States

I created an MDP with 2 states more than twice the given number of states. I have assigned a numerical stateID to all the states, Two states of the MDP are denoted by "W" and "L", indicating Win and Loss states. All the remaining states are of the form bbrrs. Here 'bb' represents the number of balls remaining, 'rr' represents the number of remaining runs to be scored, and 's' represents the current player who has strike. '0' at the end represents player 'A' and '1' at the end represents player 'B'. I have also maintained two separate maps, one for stateID(numerical) to statesID(string) mapping, and one for stateID(string) to stateID(numerical) mapping.

2.1.2 Transitions

I added transitions from every state to every other state for a given action. These are the rules which I encoded into transitions.

- A match is won if the number of runs scored for the current ball is greater than the remaining number of runs needed.
- A match is lost if we run out of balls or if one of the two players gets out.
- Strike changes when the current player scores an even number of runs and the number of balls remaining is a multiple of 6, or when the number of balls remaining is not a multiple of 6 and the number of runs scored is odd.
- All actions for player 'B' are identical. For player B, from any state and for any action, the probabilities of getting out, scoring no run and scoring 1 run are q, (1-q)/2 and (1-q)/2 respectively. Here q was assumed to be 0.25

2.1.3 Rewards and other parameters

I have treated this as an episodic MDP with discount value 1.0. Rewards are 1 only for transitions of the form $(S,\ A,\ W)$, where W represents Winning state, and 0 for all other transitions.

2.2 Calculating the optimal policy

I used planner.py from task 1 to obtain an optimal policy for the MDP formulated above. We obtain the optimal value and optimal action for each state. However, we get the above optimal values for all the states. However, only states of the form 'bbrr0' are relevant for the given problem(i.e, the states given to us initially in cricket_state_list.txt).

2.3 Decoder

planner.py returns a list of optimal values and optimal actions based on numerical state IDs. I used the map I defined above (numerical state ID to string state ID) to get the final output. Again, we consider only states that are relevant to us.

2.4 Plots

Here, I considered the winning probability from a given state to be equal to the value function for that state. Thus, I first generated the optimal policy and then the optimal value for each state. I used those values for obtaining the pplots shown below.

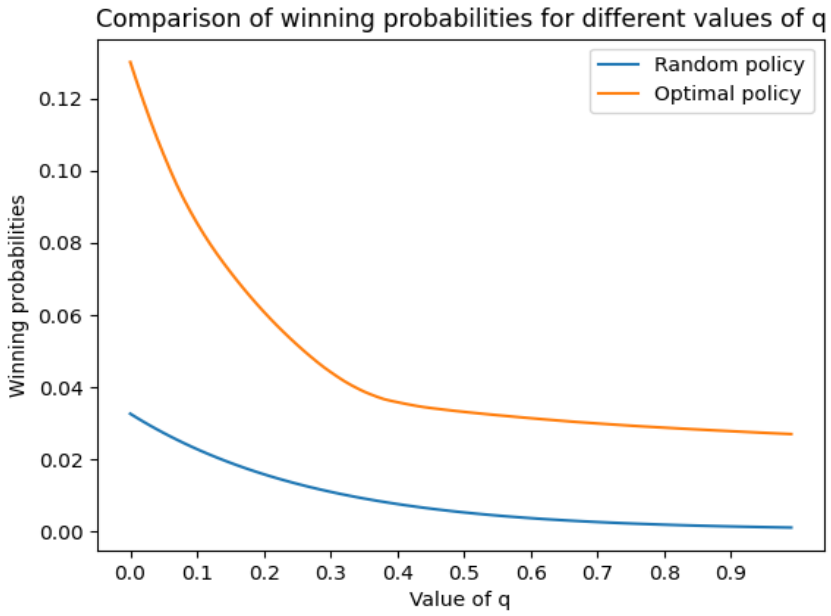

Figure 1: A comparison between the random policy and optimal policy for 100 different values of q

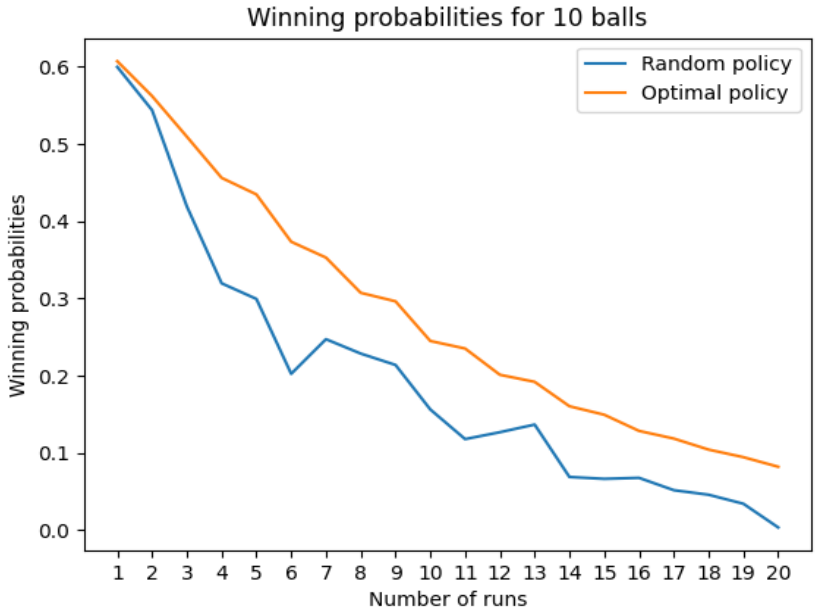

Figure 2: Fixed number of balls to be 10 and q to be 0.25

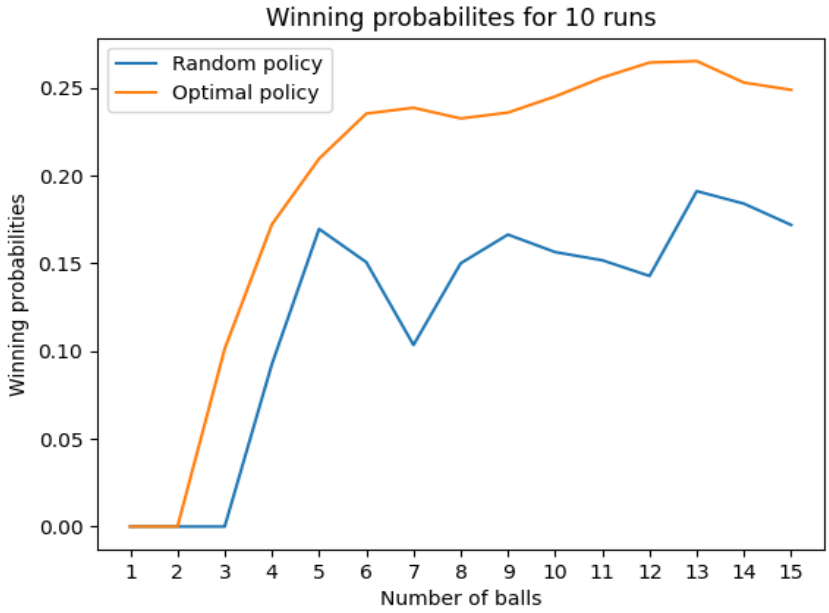

Figure 3: Fixed number of runs to be 10 and q to be 0.25

2.5 Analysis

We can clearly see that the optimal policy performs better than the random policy from the above graphs. Also, the results obtained above match our intuition. As the required number of runs increases for a fixed number of balls remaining, the winning probability decreases. As the number of balls available increases for a fixed number of runs, the winning probability increases.

3 References

- Lecture slides
- https://coin-or.github.io/pulp/