CS747 Assignment 2

Gaurav Misra
200100063

Autumn 2022

Contents

1 Task 1

1.1 Overview

The representation of the MDP in terms of the available data structures is something I struggled with, initially.
My original idea was to use a **HashMap**, or as they are known in Python, **Dictionaries**, in order to represent transitions. The idea was that each *(state, action)* tuple would map to a list of tuples, each containing *(next_state, reward, probability)*.
This caused implementation problems, and moreover, made it impossible to vectorize the code, an aspect that is covered later.
Therefore, I switched my modelling to use three-dimensional **Numpy Arrays** of shape k x n x n
where k is the number of actions and n is the number of states. I made two such arrays, one to store transition probabilities, and another to store reward values.
This model made it very easy to vectorize my code, which was necessary as the implementation of the algorithms with loops was very slow, even on medium-sized MDPs, such as the 50-20 MDP. Task 2 would not have been completed without the vectorized implementation.
For the Policy Evaluation task, the policy is read in as a Numpy Array, this helps vectorize Policy Evaluation as well, which in turn speeds up the code for Howard's Policy Iteration, which uses Policy Evaluation as an integral step.

1.2 Value Iteration

This implementation of Value Iteration uses a random value function to begin with.
The computations involved in Value Iteration are then applied to the vector repeatedly, and the infinity norm of the difference between the current value function and the previous value function is computed. As long as this norm is above a certain threshold (which was empirically determined), the process is repeated.
Fundamental computational step in Value Iteration:

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

Once the norm falls below the set threshold, the corresponding policy (which is also the optimal policy) is computed using a simple argmax, and the value function and the optimal policy are returned and displayed.

1.3 Linear Programming

Linear Programming was by far, the most straightforward to implement.
All one had to do was formulate the MDP as a Linear Programming Problem, and let the LP Solver (in this case, PuLP) do the rest.
The following was implemented:
Optimization Objective:

$$Maximize(-\sum_{s \in S} V(s))$$

Constraints:

$$-V(s) + \sum_{s' \in S} T(s, a, s')(R(s, a, s') + \gamma V_t(s')) \leq 0, \forall s' \in S, a \in A$$

As one can see, the constraint has been rearranged a bit, to convert it into a format suitable for the solver.

1.4 Howard's Policy Iteration

Howard's Policy Iteration was not very tough to implement, either. It was, however, tough to debug once the errors in the value functions started showing up.
To implement HPI, I used my code for policy evaluation, which forms an integral part of HPI. This code is described in the next section.
Essentially, I have started with a random policy and then looped over every state. For each state, I have found out what the improving actions are (using the policy evaluation code) and then stored them in a list.
Once this is done for each and every state, I then update my policy for every improvable state using one of the available improving actions (random selection over the improving actions).
This process is repeated until there are no improving actions available for any state.

1.5 Policy Evaluation

This was a function that was slightly tough to define, as it is used in two different modes. One is when a path to the file containing policies is supplied, and the policy has to be read and then evaluated. The other is when this function is used as part of the Howard's Policy Iteration Algorithm, in which a policy is directly input to the function, and its value function is returned.
Once the intricacies were figured out, however, roughly the same vectorization logic that was applicable to Value Iteration was applied here.
Again, this method uses two variables, one to store the current value function and another to store the previous one. Once the infinity norm of the difference between these two vectors falls below a certain threshold, the value function is returned.
The fundamental computation involved in policy evaluation is the following:

$$V^{\pi}(s) = \sum_{s' \in S} T(s, \pi(s), s')(R(s, \pi(s), s') + \gamma V^{\pi}(s'))$$

2 Task 2

2.1 Encoder

This is the file that took the most time to write, in task 2.
In keeping with the modelling I used for task 1, the approach to task 2 was such that I would attempt to fill the transition matrix first (shape k x n x n) and the reward matrix and then write its contents to *mdpfile*.
The states would be read in from the file *cricket_states_list.txt* would need to be converted into tuples. These would consist of three elements each. The first would be the number of balls, the second would be the number of runs, and the third would be the player currently on strike. Of course, this means that I doubled the number of states in order to craft my MDP.
I even added a few extra states, two for when the game ends in a loss (one when the player is out, and the other when the number of balls goes to zero and the number of runs is till greater than zero, resulting in a loss), and one for a win (number of balls is greater than or equal to zero and the number of runs goes to zero or lower).
Since Numpy Arrays would be used within the planner (and even the encoder), each state would need to be mapped to a unique number. This required the use of Dictionaries. Similarly, choices and outcomes also needed to be mapped to their respective indices, in order to access the probability of an outcome, given a choice.
The only part that caused a bit of a problem was when the time came to handle the edge cases for the players such as over-change, out, win, and so on.
Initially, I had made a small error in this section, in the instance where the batting team would win, the probabilities were being overwritten by my code, instead of being summed up (can be seen in line 168 of *encoder.py*

2.2 Planner

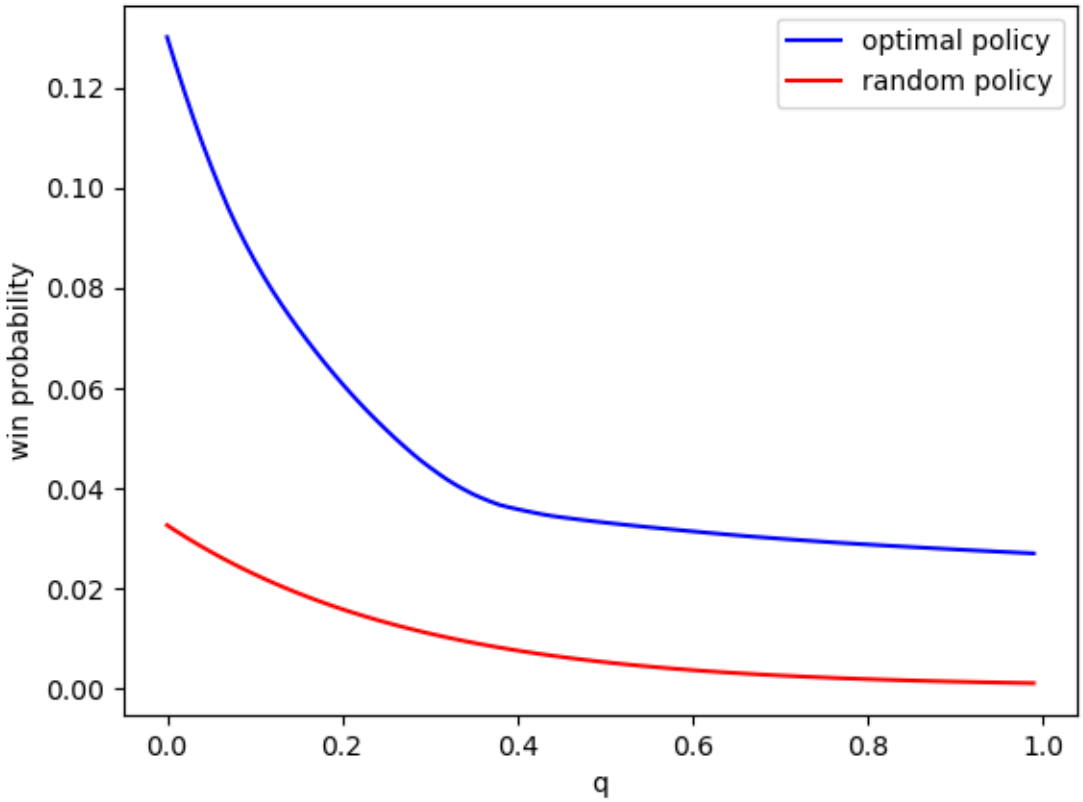
Not much needed to be changed within the file *planner.py*.
The only change that I realized would be needed during task 2 was during the evaluation of the randomly supplied policy. The problem arose because the supplied policy file contained only 150 actions, but the MDP that the EvaluatePolicy function took in would have 303 states. This was corrected by simply creating another function that would modify the input policy so as to have zeros as the actions every time B would be on strike (this makes intuitive sense as B has only one action, really, considering it is essentially a part of the environment as far as the problem statement is concerned and we only need the outcomes of B's actions, not the actions themselves).

2.3 Decoder

The file *decoder.py* was fairly simple to write.
All I had to make sure of was that the mapping of states to indices was the same as in *encoder.py*.
Furthermore, one would need to make sure that only the states which have player A at the strike would have their value functions printed to *policyfile*.

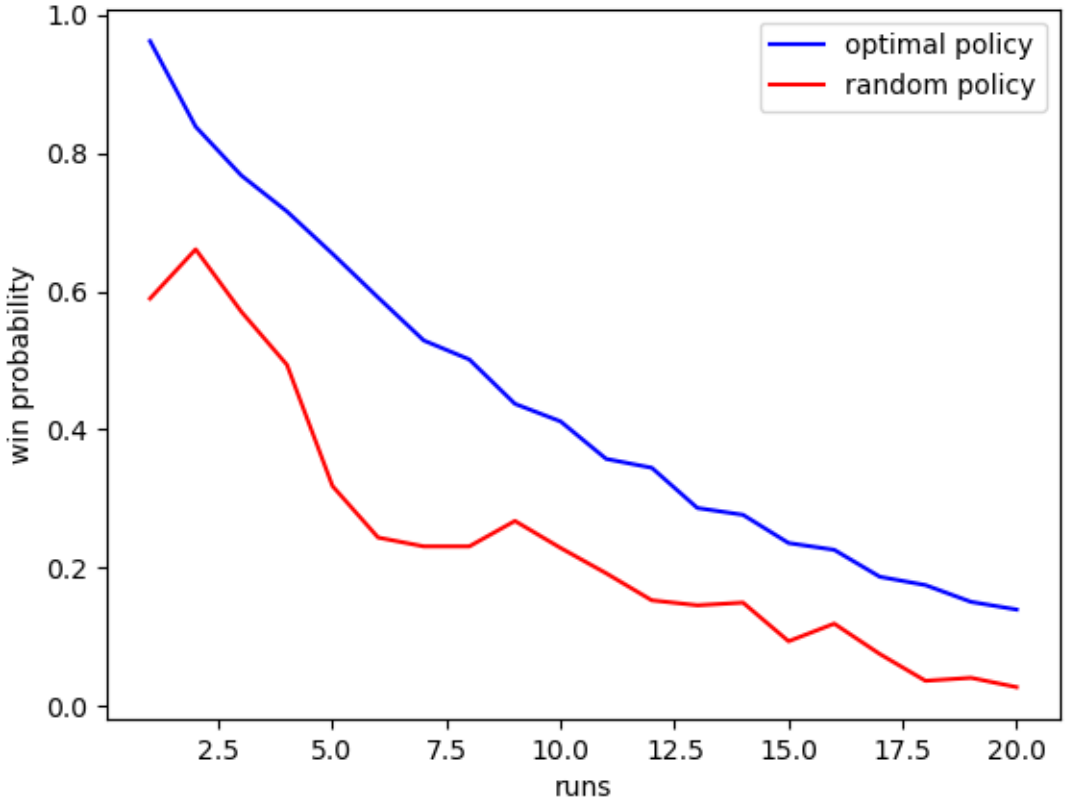
2.4 Graphs

2.4.1 Win Probabilities vs q Values



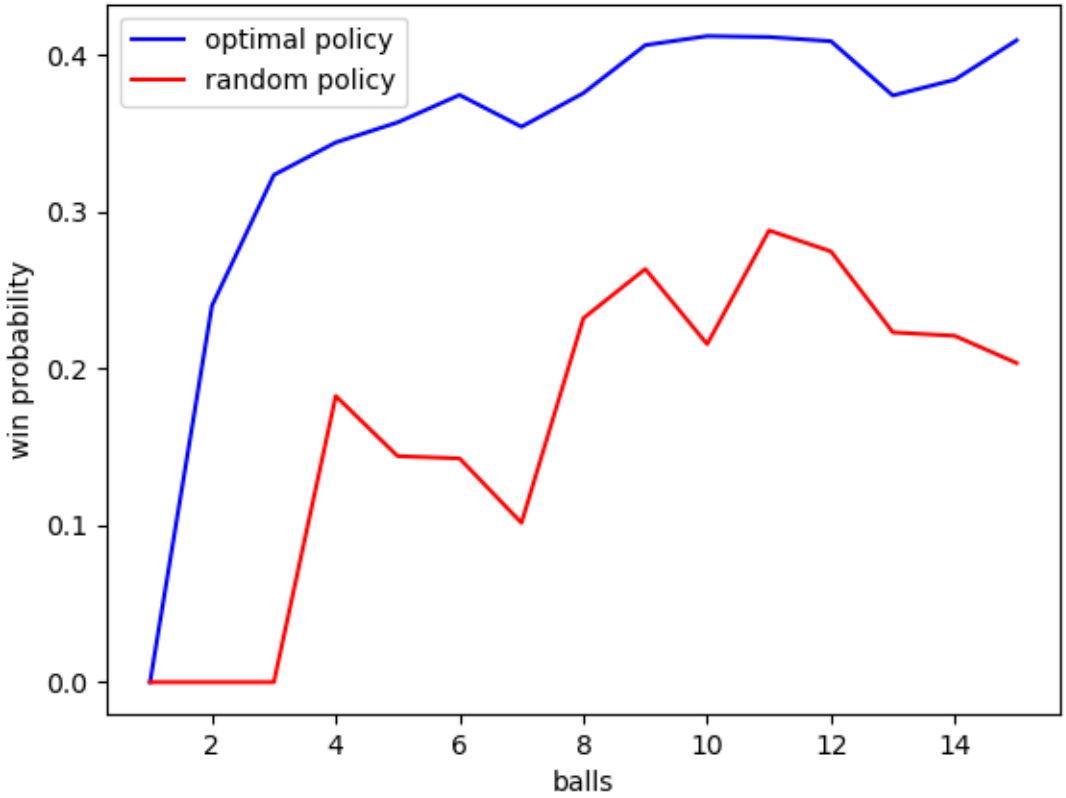
One observation is fairly trivial looking at the above graphs.
As q increases, the probability that player B get out increases. In other words, player B's strength decreases. This would imply that the probability of winning would decrease as q increases, which is what the trend indicates.
Another interesting observation is that the gap between the random policy and the optimal win probabilities for the chosen state decreases as q increases.

2.4.2 Win Probabilities vs Number of Runs



As can be seen, the optimal policy win probability is much smoother as the number of runs changes, as compared to the win probability for the random policy.
The win probability for the random policy actually increases initially, as the number of runs increases. The general trend, however, is that as the number of runs increases, for a given number of balls, the win probability decreases, which makes intuitive sense.

2.4.3 Win Probabilities vs Number of Balls



The general trend is as follows: For a fixed number of runs, as the number of balls is increased, the chances of winning (win probability) increase. This makes sense intuitively.
Again, the graph for the optimal policy is much smoother than that for the random policy.
Another interesting observation is that the win probability for the random policy remains constant at 0 initially. This stems from the fact that the number of runs is fixed at 10, and under the random policy, the player simply does not try to score a higher number of runs in order to reach the target.