PROGRAMMING ASSIGNMENT 2

An assignment report,
submitted to **Prof. Shivaram Kalyanakrishnan**
in the subject of Foundations of Intelligent and Learning Agents

by

Sandarbh Yadav
(22D0374)

INDIAN INSTITUTE OF TECHNOLOGY
BOMBAY

2022

TABLE OF CONTENTS

Task 1

The Value Iteration, Howard's Policy Iteration and Linear Programming algorithms have been implemented. An extra module is also added for evaluating the value function for a given policy. Howard's Policy Iteration is set as default algorithm.

1.1 Value Iteration

Value Iteration is an iterative approach to calculate V^*. It begins with a randomly initialised value function and keeps on applying Bellman optimality operator on it until two successive iterations yield almost same value function. Value iteration is very popular as it is easy to implement. Also, it converges very quickly in practice. [1]

$V_0 \leftarrow$ Arbitrary, element-wise bounded, n-length vector.
$t \leftarrow 0$.
Repeat:
  For $s \in S$**:**
    $V_{t+1}(s) \leftarrow \max_{a \in A} \sum_{s' \in S} T(s, a, s') \left(R(s, a, s') + \gamma V_t(s')\right)$.
    $t \leftarrow t + 1$.
Until $V_t \approx V_{t-1}$ (up to machine precision).

The submitted code implements Value Iteration algorithm in *planner.py*. It uses a tolerance of 1e-9.

1.2 Howard's Policy Iteration

Howard's Policy Iteration is a greedy approach to determine π^*. It begins with a randomly initialised policy and keeps on applying policy improvement on it until it has no more improvable states. In Howard's Policy Iteration, all the improvable states are switched in every policy improvement stage.

$\pi \leftarrow$ Arbitrary policy.
While π has improvable states:
  $\pi' \leftarrow$ PolicyImprovement(π).
  $\pi \leftarrow \pi'$.
Return π.

The submitted code implements Howard's Policy Iteration algorithm in *planner.py*.

1.3 Linear Programming

In Linear Programming approach, the objective function and constraints for LP Solver are setup in such a way that the returned solution is V^*. There are n variables and nk constraints.

$$\text{Maximise} \left(-\sum_{s \in S} V(s) \right)$$
$$\text{subject to}$$
$$V(s) \geq \sum_{s' \in S} T(s, a, s')\{R(s, a, s') + \gamma V(s')\}, \forall s \in S, a \in A.$$

The submitted code implements Linear Programming approach in *planner.py*. PuLP is used for the implementation.

1.4 Policy Evaluation

Policy evaluation is the approach of determining the value function V^π of a given policy π by solving Bellman equations. There are n unknowns and n linear equations. Unique solution is guaranteed if $\gamma < 1$.

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

The submitted code implements Policy Evaluation in *planner.py*.

Task 2

Task 2 has been implemented by introducing dummy states and a dummy action for player B. Also, 2 terminal states are introduced for loss and win. The *encoder.py* encodes these states and actions alongside states and actions of player A and formulates the MDP as *mdpfile*. The *planner.py* takes *mdpfile* as input and determines the optimal policy and optimal value function. The *decoder.py* decodes the output of *planner.py* and outputs the desired *policyfile*. It also ensures that the newly introduced states and actions don't appear in final solution.

The number of states are taken as $2(bb * rr) + 2$, where *bb* denotes the initial balls left and *rr* denotes the initial runs required. A label denoting the player is attached to each state resulting in twice the given states. The two terminal states denoting loss and win are also added. The number of actions are taken as one more than the given actions for player A as a dummy action is introduced for player B. This dummy action is taken only from states corresponding to player B. γ is set as 1 and loss and win states are the terminal states of the episodic MDP.

The transitions from states corresponding to player A are generated according to actions applicable to player A. On the basis of outcome probabilities for each action the next state is determined for such states. The transitions from states corresponding to player B are generated for the dummy action. On the basis of q, the next state is determined for such states.

The submitted code implements above approach and the generated graphs are attached subsequently. These graphs analyse the effect of environment parameters. Each graph contains plots for the optimal policy determined by *planner.py* and the given random policy.

Note: The graphs correspond to player parameters given in *sample-p1.txt*.

2.1 Graph 1

This graph plots the variation of win probability with q which denotes the degree of weakness of player B. For this graph, the state is fixed as (15 balls left, 30 runs required) and q is varied from 0 to 1.

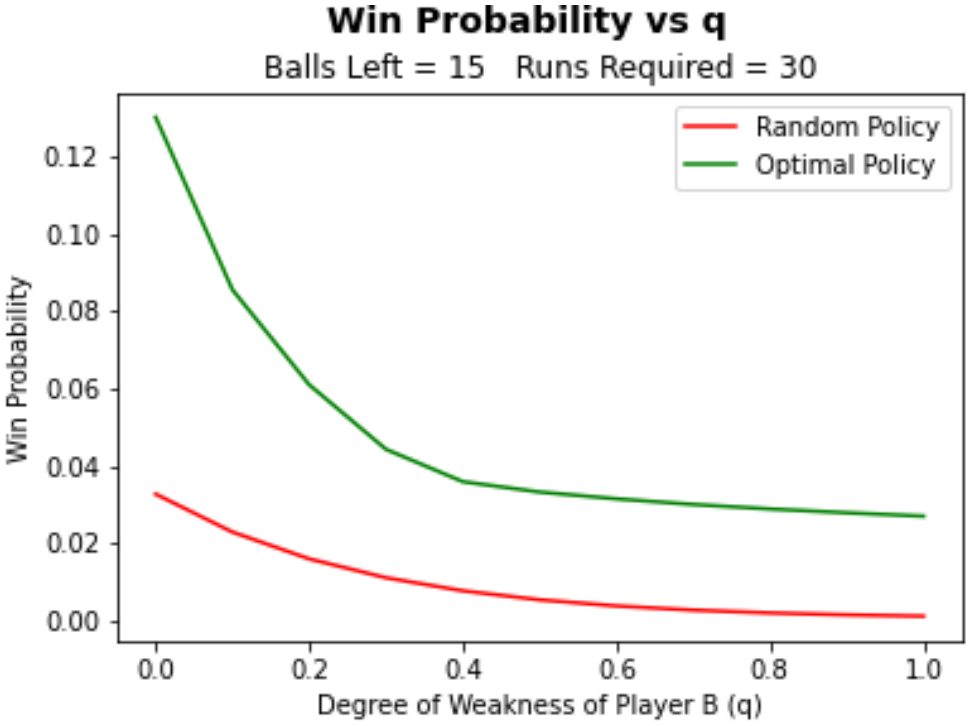

Figure 2.1: Generated graph for variation of win probability with q

2.1.1 Observations

It can be observed from the graph that win probability decreases with increase in q. This is quite intuitive as there is more chance of player B getting out if q increases resulting in lesser win probability. Also, the optimal policy determined by *planner.py* performs better than the given random policy as win probability corresponding to optimal policy is more than win probability corresponding to random policy for every value of q.

2.2 Graph 2

This graph plots the variation of win probability with runs required to win the game. For this graph, balls left are fixed as 10, q is fixed as 0.25 and runs required are varied from 1 to 20.

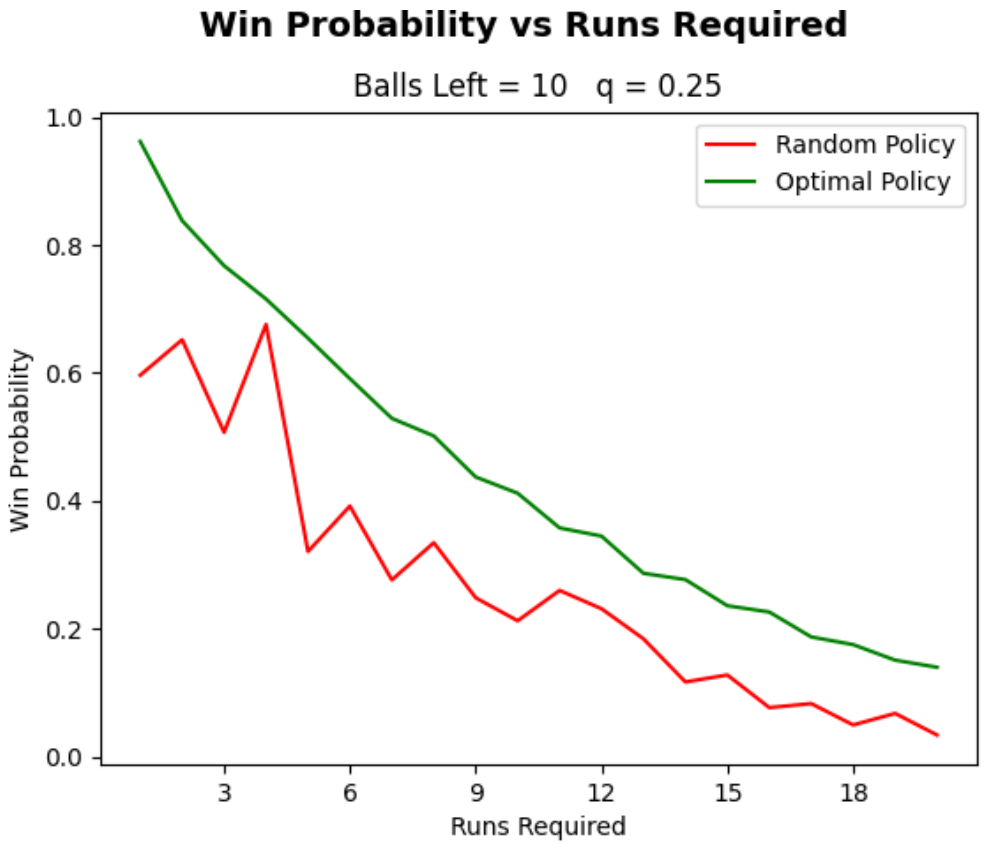

Figure 2.2: Generated graph for variation of win probability with runs required

2.2.1 Observations

It can be observed from the graph that win probability decreases with increase in runs required. This is quite intuitive as it becomes harder to win when runs required are more in the same number of balls. Also, the optimal policy determined by *planner.py* performs better than the given random policy as win probability corresponding to optimal policy is more than win probability corresponding to random policy for every value of runs required.

2.3 Graph 3

This graph plots the variation of win probability with balls left. For this graph, runs required are fixed as 10, q is fixed as 0.25 and balls left are varied from 1 to 15.

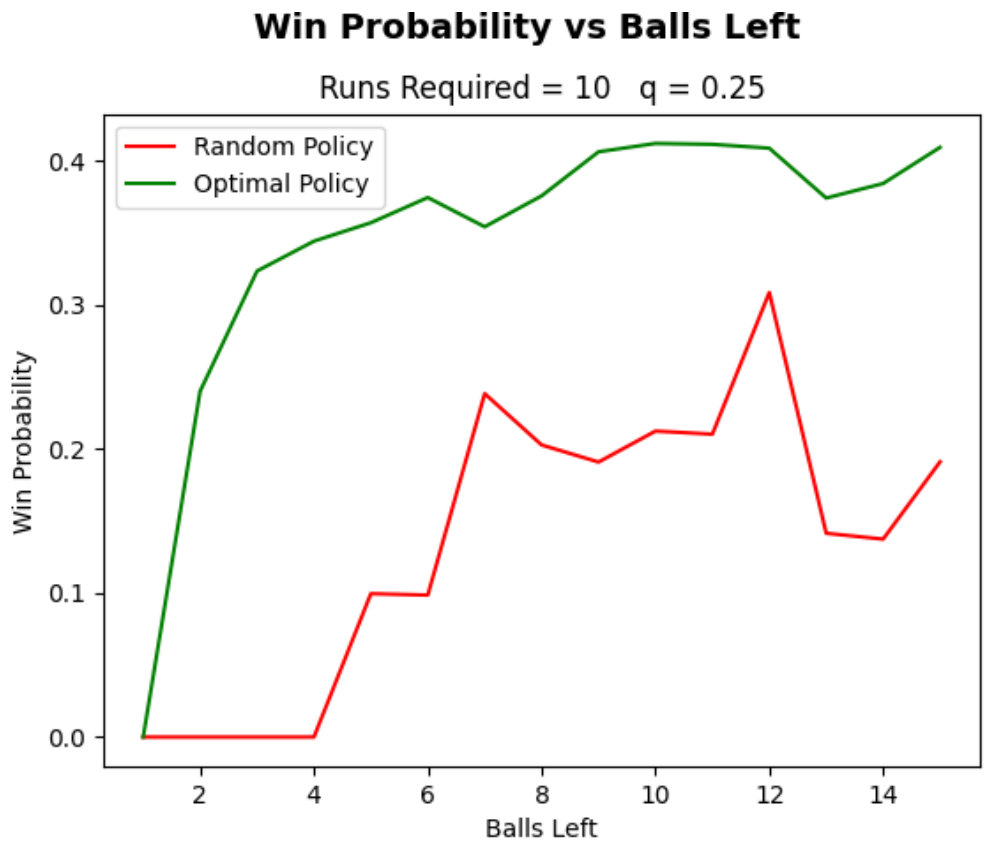

Figure 2.3: Generated graph for variation of win probability with balls left

2.3.1 Observations

It can be observed from the graph that win probability increases with increase in balls left. This is quite intuitive as it becomes easier to win when balls left are more for the same number of runs required. Another observation is that the win probability takes a hit when the over changes. This is also intuitive as there is chance of strike change and weaker batsman coming on strike in next over. Also, the optimal policy determined by *planner.py* performs better than the given random policy as win probability corresponding to optimal policy is more than win probability corresponding to random policy for every value of balls left.

REFERENCES

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.