CS747 Assignment 2

Sarthak Mittal

October 8, 2022

Contents

1 Task 1: MDP Planning

1.1 Design Decisions

We define how we keep track of the variables of the MDP. The states have certain information stored in them, and the MDP stores the global information of all states, actions and transitions.

1.1.1 State

Each state has an associated index `i`. Along with that, for each state, there is an available set of actions `A`.

1.1.2 MDP

The MDP has information about number of states (`ns`), number of actions (`na`), the states map (`S`, from index to state), actions list (`A`, action indices), end states (`e`) and all transitions for each state-action pair (`T`). Additionally, there is a threshold (`thres`) to check for change in the value function (to determine when to stop). The MDP also stores values of discount factor (`g`) and the type (`t`) of the MDP.

1.2 Assumptions

Following the MDP file format as mentioned in the programming assignment. The `planner.py` script takes an optional argument for policy to evaluate instead of finding optimal policy. The solver function returns the value function and policy depending on the input given for policy and algorithm.

1.3 Formats

```
numStates S
numActions A
end ed1 ed2 ... edn
transition s1 ac s2 r p
transition s1 ac s2 r p
. . .
. . .
. . .
transition s1 ac s2 r p
mdptype mdptype
discount gamma
```

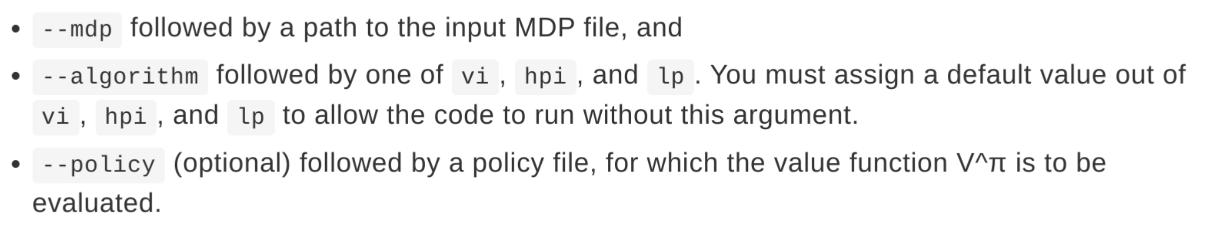
MDP Planning: MDP File Format

```
V*(0)   π*(0)
V*(1)   π*(1)
.
.
.
V*(S - 1)   π*(S - 1)
```

MDP Planning: Output Format

1.4 Algorithms

Depending on the input provided, one of the 4 functions is called.



- `--mdp` followed by a path to the input MDP file, and
- `--algorithm` followed by one of `vi`, `hpi`, and `lp`. You must assign a default value out of `vi`, `hpi`, and `lp` to allow the code to run without this argument.
- `--policy` (optional) followed by a policy file, for which the value function V^π is to be evaluated.

MDP Planning: Planner Arguments

1.4.1 Policy Evaluation

We iterate over the states, get the transitions available from that state, and update the value function of that state. Then we check if the updated value function differs from the previous one by more than the threshold. When we do not get significant change in the value function, it means it has converged to optimal value, and so we stop.

1.4.2 Value Iteration

We iterate over the states, and then iterate over the available actions in a sorted manner while keeping track of the maximum value and the corresponding optimal action. After iterating over actions, we set the value function and policy for the current state. After iterating over states, we check if the updated value function differs from the previous one by more than the threshold. After that, we iterate over all states and assign the first action to states that do not have a valid action assigned.

1.4.3 Howard's Policy Iteration

To start, we first iterate over the states and obtain an initial policy by picking first valid action from sorted available actions. We then evaluate the policy. Then we check optimality of the policy, by iterating over the states, computing teh state-action value `Q(s,a)` over all actions and comparing it with the current value function. If we find a better action, we assign that to the policy. After iterating over all states, we check whether there was a change in the policy. If so, we iterate over policy improvement again (total `10` iterations) otherwise we iterate over all states and assign the first action to states that do not have a valid action assigned, and return the results.

1.4.4 Linear Programming

We formulate the problem as a linear program (minimization type). The constraints include sum of the value function and the value function at state `s` being at least as large as `Q(s,a)` for each valid action. We then invoke the solver, and obtain the value function. Following that, we compute the policy by iterating over the actions, finding `Q(s,a)` and assigning the action with largest value as the policy. After that, we iterate over all states and assign the first action to states that do not have a valid action assigned.

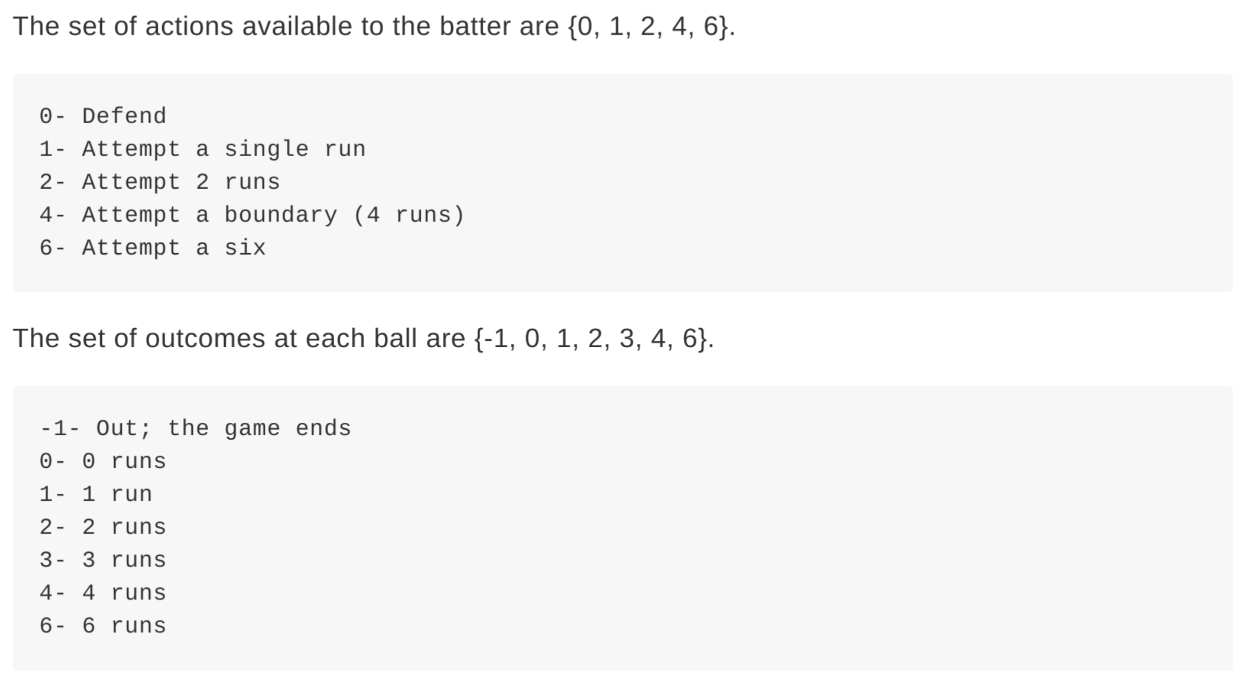
1.5 Observations

The execution time taken for the 3 algorithms roughly follows the relative order `lp` < `hpi` < `vi`.

2 Task 2: Cricket Game

2.1 Formulation

Following the MDP file format as mentioned in the programming assignment. We create double the number of states `S = bb * rr`, taking all states for `A` followed by all for `B`. The game will be solved using a pipeline of `encoder-planner-decoder` scripts.



The set of actions available to the batter are {0, 1, 2, 4, 6}.

```
0- Defend
1- Attempt a single run
2- Attempt 2 runs
4- Attempt a boundary (4 runs)
6- Attempt a six
```

The set of outcomes at each ball are {-1, 0, 1, 2, 3, 4, 6}.

```
-1- Out; the game ends
0- 0 runs
1- 1 run
2- 2 runs
3- 3 runs
4- 4 runs
6- 6 runs
```

Cricket Game: States and Actions

2.2 Encoding

We have two terminal states, one for win and other for game over. We then iterate over the states (`s1`), and over the actions (`a`), and define all possible transitions `T(s1,a,s2,r,p)` for which $p > 0$. Depending on the runs outcome from the action, we find the next state using a function. We also need to keep track of the strike (rotation on odd number of runs or on the over-up ball), so the function that returns the index of the next state and the reward for the transition takes the state mapping also as an input along with current state and runs in that transition. We define these transitions sequentially, first by keeping states with `A` on strike as `s1` and then with `B` on strike. The MDP is assumed to be episodic, so discount factor is set to `1.0`.

2.3 Planning

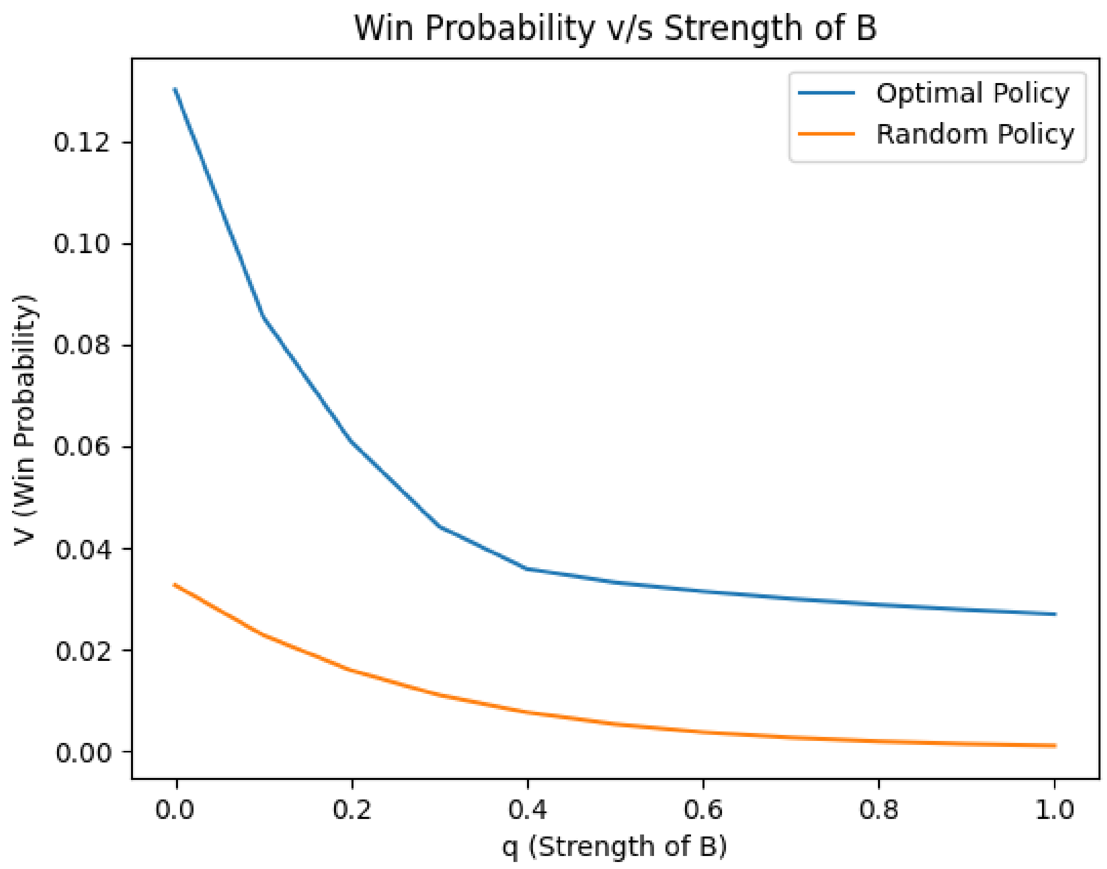
We use the MDP file generated by `encoder.py` as an input for `planner.py`. We use the default algorithm (`lp`) to solve the MDP and find the optimal value function and policy for all the states for both players `A` and `B`. Hence for each state, we have two pairs of values for value function and optimal policy.

2.4 Decoding

Assuming that `A` is on strike initially, we read the states in again, and get the value function and optimal action from the output of `planner.py` and send it as input to `decoder.py` to generate the policy file outout in required format (having state (number of balls and runs remaining), optimal action (runs to score) and associated win probability (value function)).

3 Analysis & Plots

3.1 Plot 1 (v/s Strength of `B`)

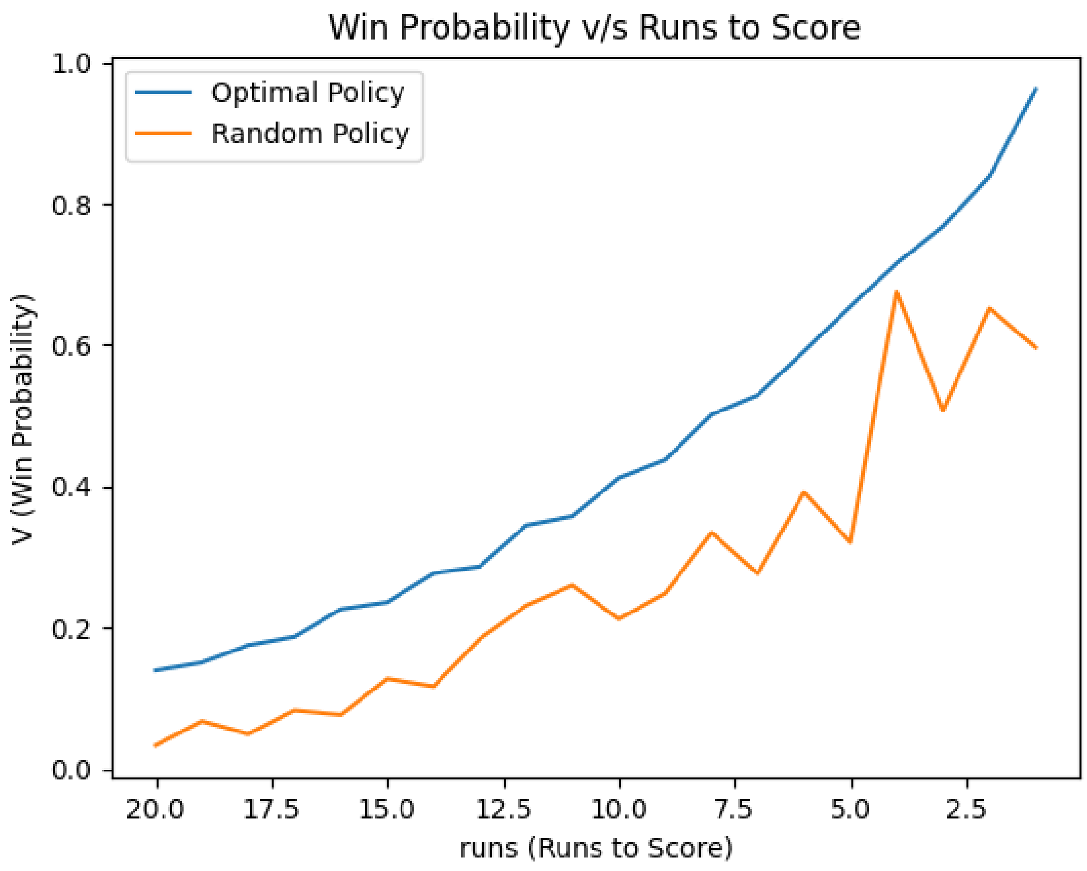


Cricket Game: Win Probability v/s Strength of B

3.2 Inference

With increase in strength of `B` (value of `q`), the chance that `B` gets out rises. Due to this, `A` will try to maintain strike, and so has restricted actions that can be taken to keep chance of winning high. Because chasing `30` runs in `15` balls is a bit tough even in reality, the low probability is justified. The decrease is due to the tail-end player having higher chance of getting out if given the strike, resulting in a loss in all those runs. The random policy performs worse than optimal, but still shows a gradually decreasing trend.

3.3 Plot 2 (v/s Runs to Score)

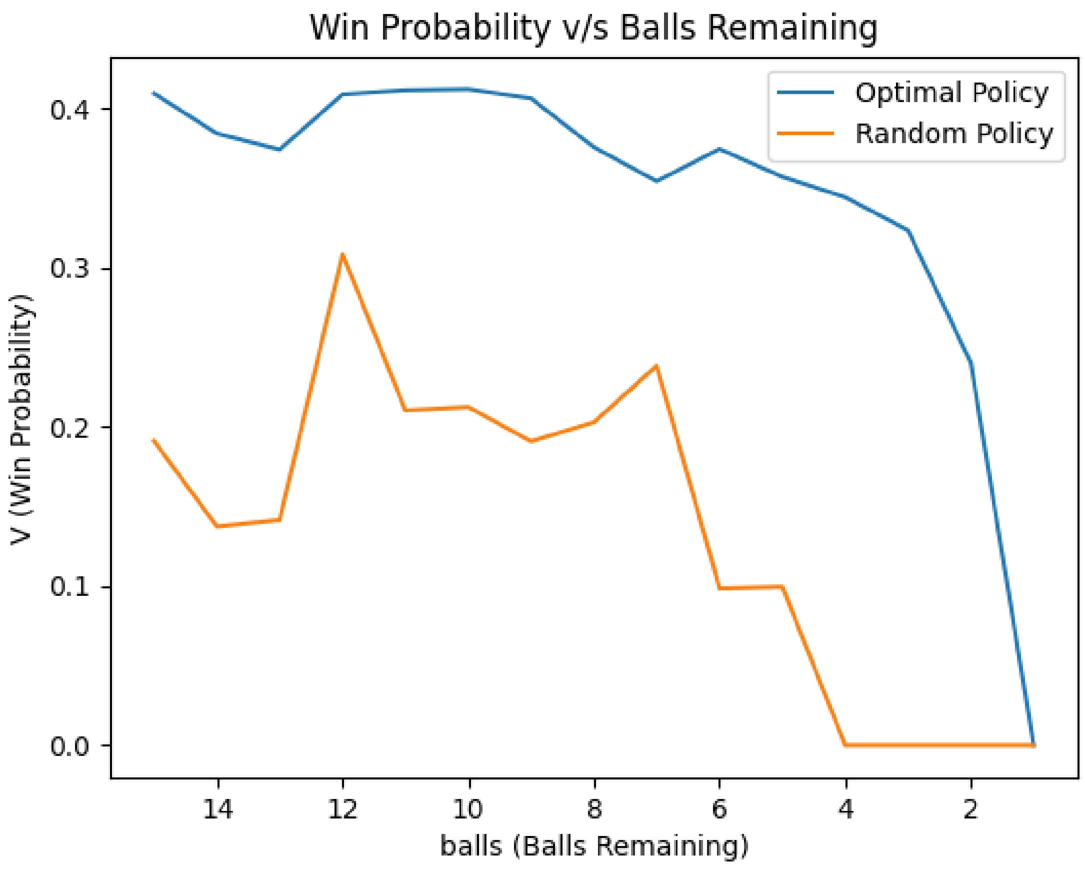


Cricket Game: Win Probability v/s Runs to Score

3.4 Inference

With decrease in runs to score, the chances of winning increase, because total number of balls is fixed (at `10`). Since the targets become relatively easier to achieve, and because `B` has a decent strength (`q = 0.25`), the chance of winning increases gradually (because runs can potentially decrease by more than `1` but balls decrease by `1` each time). The random policy performs worse than optimal, but still shows a gradually increasing trend on average.

3.5 Plot 3 (v/s Balls Remaining)



Cricket Game: Win Probability v/s Balls Remaining

3.6 Inference

With decrease in the balls remaining, the chances of winning decrease, because total number of runs is fixed (at `10`). Since the target becomes relatively harder to achieve, and because `B` has a decent strength (`q = 0.25`), the chances of winning decrease gradually at first but rapidly later (because balls reduce by `1` but runs can potentially decrease by more than `1` in each turn). The subtle drops in the optimal values are at the balls where the over is about to end, and so `A` will try to switch strike deliberately. The random policy performs worse than optimal, but still shows a decreasing trend on average.