CS 747 - Assignment 2

Aditya Sriram (200070004)

October 2022

1 Introduction

In this assignment, we solve the **MDP Planning** problem, which involves finding an optimal policy for the given MDP. Three algorithms are implemented - **Value Iteration**, **Howard's Policy Iteration** and **Linear Programming**. Using our newly designed solver, we aim to find an optimal policy for a batter, chasing a target during the last wicket, in a game of cricket.

2 Part 1: MDP Planning

The MDP instances provided did not conform exactly to the representation discussed in class. State transition probabilities were not directly given. Rather, $p(s', r|s, a) = \mathbb{P}\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$ was given in each line. The function, $p : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathcal{R} \to \mathbb{R}$ is called the *four-argument dynamics function* in Sutton and Barto (2018), where $\mathcal{S}$ is the set of states, $\mathcal{A}$ is the set of actions and $\mathcal{R}$ is the set of rewards. This is a more general representation as it allows for stochastic rewards. We only need to make a small modification to our formulation to account for this stochasticity,

$$V^\pi(s) = \sum_{s',r} p(s', r|s, a) \left(r + \gamma V^\pi(s')\right)$$

Defining, $R(s, a) = \sum_r r \sum_{s'} p(s', r|s, a)$ and $T(s, a, s') = \sum_r p(s', r|s, a)$, we can rewrite the above as,

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

which is the class formulation with the modification that rather than computing $R(s, a, s')$, we are directly computing $R(s, a)$ which is the expected reward on taking action a from state s.

I used two different schemes to store the MDP - `numpy` arrays and the `dict` data structure. `numpy` allows for vectorized code. However, most MDPs possess a sparse transition matrix and reward function (similar to alternate graph representations - adjacency lists and matrices). Dictionaries offer an efficient

representation for MDPs as they only store transitions that occur with non-zero probability. I did not observe a significant speedup for the testcases provided since they were relatively small instances. However, when I tested Value Iteration on a 2500 states, 100 actions MDP, the program ran in under $40s$ using dictionaries in contrast to the vectorization-based approach where the program did not complete execution.

2.1 Value Iteration

2.1.1 Pseudocode

$V_0 \leftarrow$ All zero, n-length vector.
$t \leftarrow 0$
while True **do**
  for $s \in S$ **do**
    $V_{t+1}(s) \leftarrow \max_{a \in \mathcal{A}} \sum_{s',r} p(s', r|s, a) (r + \gamma V_t(s'))$
    $t \leftarrow t + 1$
  end for
  if $V_t \approx V_{t-1}$ **then**
    return V_t
  else
    $V_{t-1} \leftarrow V_t$
  end if
end while

2.1.2 Code

```
def get_policy(S, A, TR, gamma, V):
    pi = np.zeros(S, dtype=int)
    for s1 in TR.keys():
            v_max = -float('inf')
            for a in TR[s1].keys():
                v = 0
                for s2 in TR[s1][a].keys():
                    for r in TR[s1][a][s2].keys():
                        v += TR[s1][a][s2][r] * (r + \
                            gamma * V[s2])

                if v > v_max:
                    v_max = v
                    pi[s1] = a

            V[s1] = v_max

    return pi
```

```
def vi(S, A, TR, gamma):
    V = np.zeros(S)
    V_old = np.zeros(S)
    while(True):
        pi = get_policy(S, A, TR, gamma, V)

        if np.allclose(V, V_old, rtol=1e-11, atol=1e-8):
            return V, pi
        else:
            V_old = V.copy()
```

Although V_0 could be initialised randomly, I chose the all-zero initialisation because this handles episodic tasks by setting the value function of terminal states to be 0. Since there are no outgoing transitions from terminal states, there value function will not be updated by the algorithm as required. Ties were broken by choosing the first occurrence. Convergence was tested using `np.allclose()` with a relative tolerance of 10^{-14} and absolute tolerance of 10^{-11}.

2.2 Howard's Policy Iteration

2.2.1 Pseudocode

$\pi \leftarrow$ Arbitrary policy
while π has improvable states **do**
  $\pi' \leftarrow$ ImproveAllImprovableStates(π)
  $\pi \leftarrow \pi'$
end while

2.2.2 Code

```
def hpi(S, A, TR, gamma):
    pi = np.random.randint(0, A, size=S)
    T, R = get_matrix(S, A, TR)
    while(True):
        v_pi = policy_eval(S, A, T, R, gamma, pi)
        pi_improved = get_policy(S, A, TR, gamma, v_pi)
        if (pi == pi_improved).all():
            return v_pi, pi
        else:
            pi = pi_improved
```

Ties between different improving actions were broken by choosing the improving action with the lowest index. Convergence was tested by comparing actions in all states and checking if they are identical.

2.3 Linear Programming

$$\min\left(\sum_{s\in S} V(s)\right) \text{ subject to } V(s) \geq \sum_{s',r} p(s',r|s,a)\,(r+\gamma V(s'))\,\forall s,a$$

The linear program was formulated and solved using Python's **PuLP** package.

2.3.1 Code

```
def lp(S, A, TR, gamma):
V_star = np.zeros(S)

# Create the LP problem
prob = LpProblem("Primal", LpMinimize)

lpVariables = []
for i in range(S):
    variable = LpVariable(f"V_{i}")
    lpVariables.append(variable)

# Objective
prob += lpSum(lpVariables)

# Constraints
for s1 in range(S):
    if s1 in TR.keys():
        for a in TR[s1].keys():
            constraint = 0
            for s2 in TR[s1][a].keys():
                for r in TR[s1][a][s2].keys():
                    constraint += TR[s1][a][s2][r] * (r + \
                        gamma * lpVariables[s2])

            prob += lpVariables[s1] >= constraint

    else:
        prob += lpVariables[s1] == 0

prob.solve(PULP_CBC_CMD(msg=0))

for s in TR.keys():
    V_star[s] = lpVariables[s].varValue

pi_star = get_policy(S, A, TR, gamma, V_star)
return V_star, pi_star
```

2.4 Policy Evaluation

Policy Evaluation involves solving the following set of $|\mathcal{S}|$ linear equations,

$$V^{\pi}(s) = \sum_{s',r} p(s', r|s, a) \left(r + \gamma V^{\pi}(s')\right) \forall s \in \mathcal{S}$$

For continuing tasks, $\gamma < 1$ which ensures that the linear system has a unique solution. However, for episodic tasks, $\gamma = 1$ which might result in a system with non-zero nullity. Both these cases can be handled using `np.linalg.pinv` which uses the Moore-Penrose inverse in case the inverse is not well-defined.

2.4.1 Code

```
def policy_eval(S, A, T, R, gamma, policy):
    V_pi = np.linalg.pinv(np.eye(S)-gamma*T[np.arange(S),policy,:])\
        @ R[np.arange(S),policy]
    return V_pi
```

3 Part 2: Cricket - The Last Wicket

3.1 Problem

There are two batters A and B. We want to find the optimal policy for batter A, assuming that B, along with the rest of the game dynamics is part of the environment. A is a middle-order batter at the last wicket, while B is a tail-ender. B can only get out, score 0 runs or score 1 run. Player B can be described by a parameter, q which is his degree of weakness. The team needs to chase down O runs in T balls. The state can be encoded as bbrr, where bb is the number of balls left and rr is the number of runs to reach the target.

Batter A has 5 actions - defend, attempt a single run, attempt two runs, attempt a boundary and attempt a six. The outcome of any action is - game over, 0 runs, 1 run, 2 runs, 3 runs, 4 runs or 6 runs. These outcomes are defined by a set of probabilities that will decide the state transition probabilities.

The strike rotates at the beginning of each over. For 1 and 3 runs scored, the strike changes. If both the over changes and 1 or 3 runs are scored, the strike remains the same. The over changes at every ball divisible by 6.

The player/agent is given a reward of 1, whenever he chases down the target. Every other outcome (draw or lose) earns a reward of 0. This implies that the value function represents the probability of winning the game from that state.

3.2 Encoding

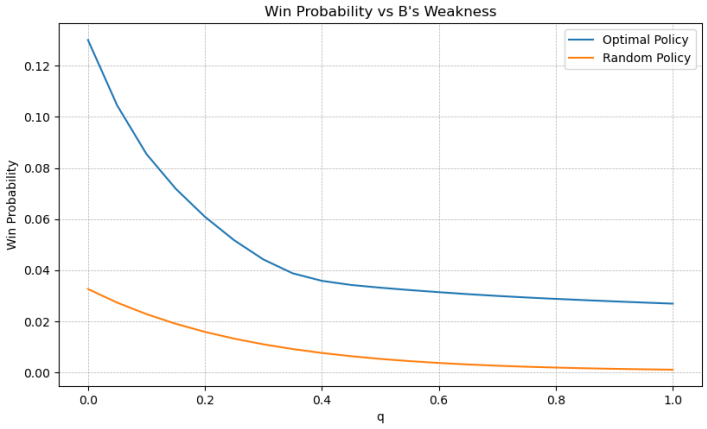
We have already been given the set of states $\mathcal{S}$, set of actions $\mathcal{A}$ and reward function. O lies between 1 and 15 and T lies between 1 and 30. I added terminal states that correspond to either O being 0 orT being 0. Although I could have been merged all these states into one, allowing for multiple end states made the

state encoding easy. Since this is an episodic task, we can choose a discount factor, $\gamma = 1$. Once we find the transition probabilities, we have specified the entire MDP.

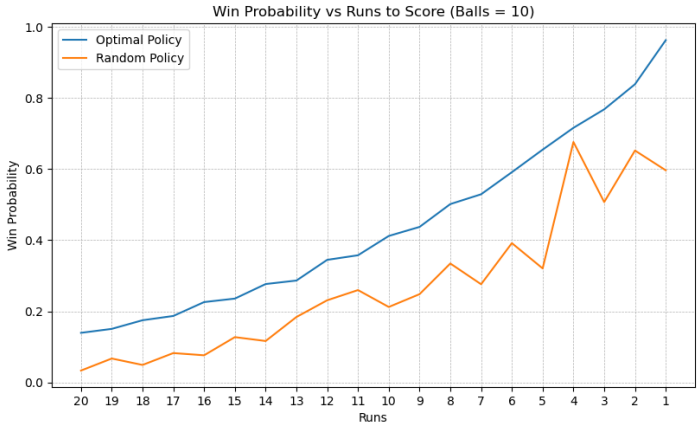
3.2.1 Transition Probabilities

The transition probabilities can be found through a case-by-case analysis for each of batter A's actions and the resultant outcome. However, this is also affected by strike rotation. Once player B gets the strike, he/she can waste an arbitrary number of balls by defending till the end of the over, and then taking a single. The task is simplified by recognising that B's behaviour is Markovian and all these 'extra' transitions can be printed using recursion.

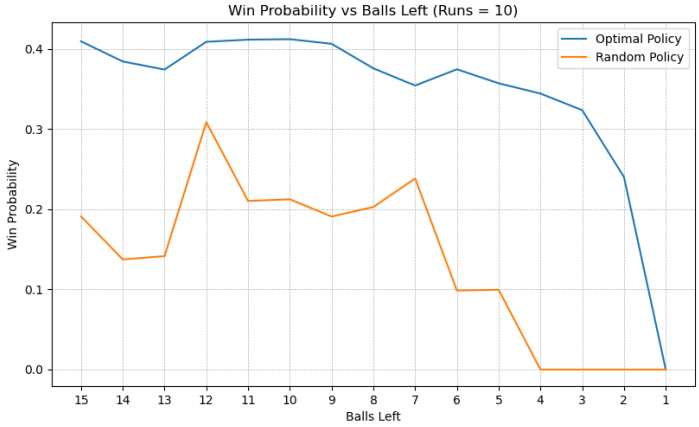
3.3 Analysis



From the plot of win probability vs batter's B weakness, we observe that the agent is more likely to lose when B is weaker. This is in line with our intuition. We can also observe that as q increases, the win probability becomes less sensitive to changes in q.



Fixing the number of balls left to 10, the win probability increases monotonically as the runs decrease from 20 to 1. This is also consistent with our intuition - player A and B have more opportunities to win when they have to chase a lower target.



Fixing the target to 10, the win probability decreases as the numbers of balls left decrease from 15 to 1. There are sudden drops in win probability when the over is just about to get over, i.e. when the number of balls left equal 7 and 13. This is counter-intuitive because in this state, it is optimal for A to retain the strike, i.e. take a single run.

Finally, in all three plots, we observe that the optimal policy outperforms the random policy.