Rishi Agarwal **CS-747 Autumn’21 Assignment-2** **170010047**

Task 1:

<u>Data Structure for storing the MDP</u>

Rather than a 3x3 transition and reward matrix which stores every possible (s,a,s’) combination, a different approach is ideated and implemented for storing the MDP.

This is much more efficient in terms of the space required for those MDPs in which most (s,a,s’) transitions do not happen, i.e., when most entries in the T(s,a,s’) and R(s,a,s’) matrix are zeros.

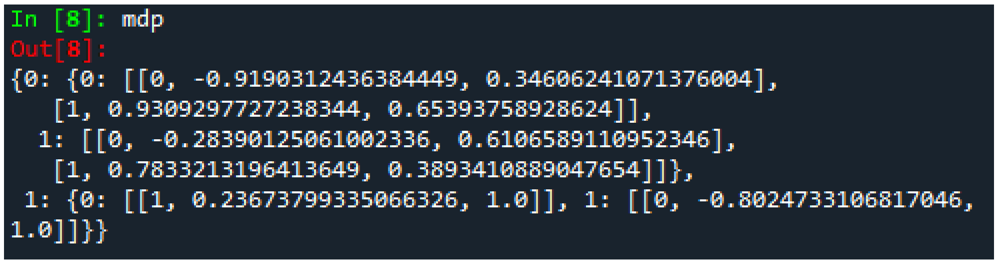
The following is the explanation of the new data structure (say mdp) implemented in the algorithm,

-> We store a 2-level dictionary whose elements are 2-dimensional lists.
The first level of the dictionary is indexed by the state-number; it represents the s in (s,a,s’).

->This mdp[s], is another dictionary which has the possible action numbers ‘a’ as their keys, and 2-dimensional lists as their values.

-> This two-dimensional list is of the shape |s’|x3 and has elements of the form
   [s’, reward, probability]

Example: *continuing-mdp-2-2.txt*



```
In [8]: mdp
Out[8]:
{0: {0: [[0, -0.9190312436384449, 0.34606241071376004],
   [1, 0.9309297727238344, 0.65393758928624]],
  1: [[0, -0.28390125061002336, 0.6106589110952346],
   [1, 0.7833213196413649, 0.3893410889047654]]},
 1: {0: [[1, 0.23673799335066326, 1.0]], 1: [[0, -0.8024733106817046,
1.0]]}}
```

<u>Value Iteration algorithm</u>

The value iteration algorithm implemented in the planner.in file is a variant of the standard value iteration algorithm discussed in class. In the standard value iteration algorithm discussed in the class, the values of all the states are updated in one go during every iteration. It is helpful for this standard implementation when the transition and reward are stored in the matrix form of the data structure. This way of doing the value iteration is ideal and can be done directly without writing much code.

However, in my implementation, a variant of the value iteration algorithm(as was given in a quiz) is used where we update the value of one state at a time during each iteration while keeping the value of all other states fixed. This is much more suited to the type of data structure which we are using for storing our mdp; it also converges much faster as compared to the standard value iteration algorithm.

The algorithm is implemented in such a way that if, for a state, two or more actions give the same value of Q(s,a), then the numerically smaller value of ‘a’ is stored.

Hence, in summary, in the implementation of value iteration, during every iteration, the value of only a single state is updated while keeping the value of all other states fixed.
The outer loop is stopped when the change in the value of ALL the states during two consecutive iterations is less than a certain fixed threshold.

Howard Policy iteration
An array which stores the action number for every state is initialised with zeros. During each pass, for every state, that action is selected which maximises Q(s,a), and the array of actions is updated accordingly. The algorithm is implemented in such a way that if, for a state, two or more actions give the same value of Q(s,a), then the numerically smaller value of 'a' is stored. At the end of the iteration, when the actions are updated for all the states, the value function corresponding to the new policy(obtained at the end of the iteration) is evaluated in a function called update_v.
The Policy Iteration algorithm is stopped when the action numbers do not change for ALL the states in two consecutive iterations, implying that the optimal value function and policies have been reached.

Linear programming method
The LP variables are the value of all the states. The LP problem is to minimize the sum of the value function of all the states, i.e.,
min [$V(s_1)+V(s_2)...V(s_n)$]   where n is the number of states in the MDP.
The declaration of constraints of the LP becomes very efficient because of the data structure that is used in storing the MDP, i.e., for each state, we can add only those actions and subsequent states' expressions to the constraints which are non-zero. This is done by traversing the data structure (which is storing the MDP), and adding the constraint for all the *non-zero* (s,a,s') combinations.
The LP problem is solved by using the PULP_CBC_CMD solver available in the docker.

Task 2:

Creating the encoder.py file
(Please look at the code when reading this section in order to understand the approach better)

The number of states for the MDP is assumed to be the total number of states given in the states' file of the player, plus 2 more states. That is, we create two fictitious states one representing draw/loss and the other representing win. Such a formulation helps in encoding the MDP.

When encoding the MDP for player 1, the state file of player 1 is passed to the script along with the fixed policy file of player 2.
For all the states of Player 1, and all possible actions (replacing '0' with '1'), we do the following,
1) If an entry in the policy file of player 2 exists after Player 1's action, this implies that Player 1's action didn't end the game. Since an entry exists in the Player 2 policy file, player 2 acts and for every possible action of Player 2 (non-zero probability given in the policy file), it is checked whether the subsequent state exists in the states' file of Player 1.

a) If it exists, it implies the game has not ended during this transition and a 'transition' line is printed.

b) If this new state doesn't exist in the States' file of Player 1, then this implies that the game ended right after Player 2's move, implying that Player 1 has won, hence, a 'transition' line is printed with reward 1 and s' being the fictitious 'win state' that we created.

2) If after Player 1's action, an entry couldn't be found in Player 2's policy file, this means that the game has ended right after Player 1's move, implying that either Player 1 has lost or the game has ended in a draw. Either way, a 'transition' line is printed with reward '0' and s' being the fictitious 'draw/lose state' that we created.

When encoding the MDP for player 2, the state file of player 1 is passed to the script along with the fixed policy file of player 1.
For all the states of Player 2, and all possible actions (replacing '0' with '2'), we do the following,
1) If an entry in the policy file of player 1 exists after Player 2's action, this implies that Player 2's action didn't end the game. Since an entry exists in the Player 1 policy file, player 1 acts and for every possible action of Player 1 (non-zero probability given in the policy file), it is checked whether the subsequent state exists in the states' file of Player 2.

a) If it exists, it implies the game has not ended during this transition and a 'transition' line is printed.

b) If this new state doesn't exist in the States' file of Player 2, then this implies that the game ended right after Player 1's move, implying that either Player 1 lost or the game ended in a draw, hence, a 'transition' line is printed with reward 1 or 0 after checking whether the Player 1 lost or the game ended in a draw.

2) If after Player 2's action, an entry couldn't be found in Player 1's policy file, this means that the game has ended right after Player 2's move, implying that Player 2 has lost. A 'transition' line is printed with reward '0' and s' being the fictitious 'draw/lose state' that we created.

Creating the decoder.py file
The file containing the optimal policy(and values), i.e. the optimal_policy_file, is passed to the decoder script along with the states' file of the player. After printing the player id in a new player_policy_file, for every line in the optimal_policy_file, a line is added to the player_policy_file containing the state (eg: 121210200) and the one-hot encoded optimal policy for the corresponding state (eg: 0 0 0 0 0 1 0 0 0) that the player should take.

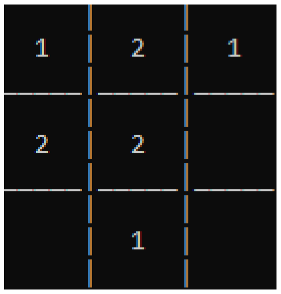
Task 3:

The following points are important to note which helps in proving the convergence in task3,
-> Since the game is won by making the other player complete a row, column or diagonal, a player who does an action cannot win instantly after doing their action.
-> Let us call **'premature end'** the end of the game when it finishes but there are still empty spaces on the board (i.e., there are 0s remaining).
-> A draw is achieved only when the board is filled completely(no premature end), or in other words, a premature end always results in one player winning.

-> If a game ends in a draw, then Player 1 will always do the final move(the 9th move) as he is the one who starts the game (the 1st move).
-> When the board has been completely filled, the game has ended. Since Player 1 did the final move, either the game is a draw or it is Player 2 who wins (if Player 1 draws a pattern).

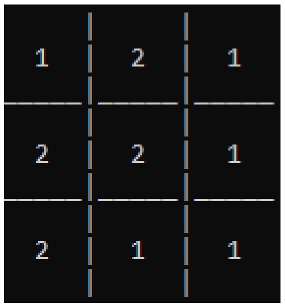
Hence, in summary, it can be said that when Player 2 is the one who ends the game, then it is Player 1 who wins. However, if Player 1 ends the game, and the game has reached a premature end, it is Player 2 who wins, whereas if the board has been filled, then the game is either a draw or Player 2 wins (if Player 1 ends up making a pattern).

Consider the policy file for player 2, obtained during the first iteration, (policyfile2_0.txt from task3.py output). When the following state is reached,



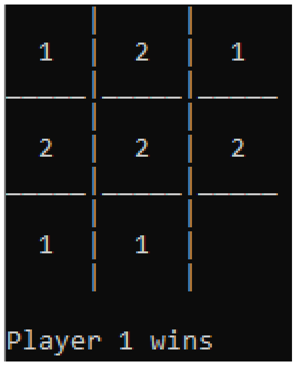
```
      |      |
  1   |  2   |  1
______|______|______
      |      |
  2   |  2   |
______|______|______
      |      |
      |  1   |
      |      |
```

If we put a ‘1’ at the 6th/9th position, Player 2 would intelligently put a 2 at the 7th position and make us put a ‘1’ at the 9th/6th position making us lose the game.



```
      |      |
  1   |  2   |  1
______|______|______
      |      |
  2   |  2   |  1
______|______|______
      |      |
  2   |  1   |  1
      |      |
```

Important: However, if we put a 1 at the 7th position, Player 2 prematurely ends the game,

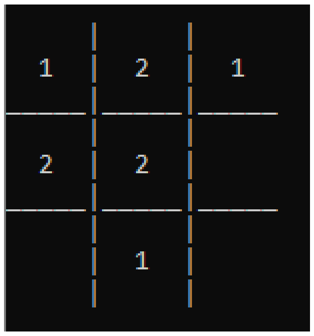


```
      |      |
  1   |  2   |  1
______|______|______
      |      |
  2   |  2   |  2
______|______|______
      |      |
  1   |  1   |
      |      |

Player 1 wins
```

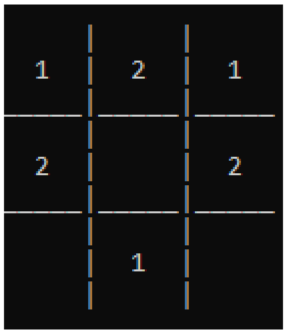
Now the question arises, why would Player 2 prematurely end the game by putting a ‘2’ at the 6th position when it could have put a ‘2’ at the 9th position which would eventually lead to a draw? This is because, speaking from the point of view of solving the MDP, Player 2 has no incentive to differentiate between a draw or a loss. Reward-wise, both the actions yield the same ultimate reward (Q(s,a,) strictly speaking). Hence, as implemented in the planner.py file, that action is chosen which is smaller in numerical value when both give the same Q(s,a). So Player 2 ends up putting a ‘2’ at the 6th position.

The next time(i.e. the next iteration) when Player 2 is getting its MDP solved, in order to improve the policy and choose a <u>possible</u> action which is different/better than the action he took earlier (from Q(s,a) point of view) which results in a better final reward, the Player 2 tries to make improvements one/few steps earlier if it leads to a better value. That is, in policyfile2_1.txt (which is right after policyfile2_0.txt described above), instead of



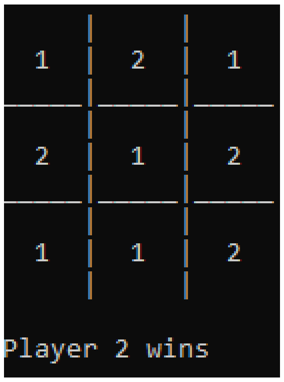
```
      |      |
  1   |  2   |  1
______|______|______
      |      |
  2   |  2   |
______|______|______
      |      |
      |  1   |
      |      |
```

the Player 2 does this



```
      |      |
  1   |  2   |  1
______|______|______
      |      |
  2   |      |  2
______|______|______
      |      |
      |  1   |
      |      |
```

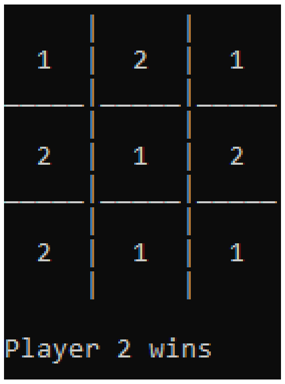
because this new action would **definitely lead Player 2 to win.**



```
      |      |
  1   |  2   |  1
______|______|______
      |      |
  2   |  1   |  2
______|______|______
      |      |
  1   |  1   |  2
      |      |

Player 2 wins
```

or



```
      |      |
  1   |  2   |  1
______|______|______
      |      |
  2   |  1   |  2
______|______|______
      |      |
  2   |  1   |  1
      |      |

Player 2 wins
```

<u>**Conclusion: When getting its MDP solved, a Player would choose an action if it ultimately would lead to a better result (reward-wise speaking).**</u>

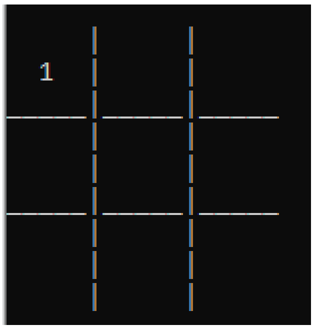
Hence, <u>it can be extrapolated</u> and said that while calculating the policies and realizing progressively better players, ultimately, in convergence, the game would *tend to* go on as much as possible (tending to fill the board) without a player doing a ‘premature end’. The ‘premature end’ would happen only from those states, from where there is no possible improvement in the value function irrespective of the action chosen (as described above).

As observed and tested using the task3.py script, the policies always converge when initialised with any policy for any player. The result is that we get two policy files corresponding to two mathematically(reward-wise) maximally-intelligent players. In other words, we get the policy files corresponding to two maximal possible intelligent players which have policies to play the game to their advantage as optimally as mathematically possible.

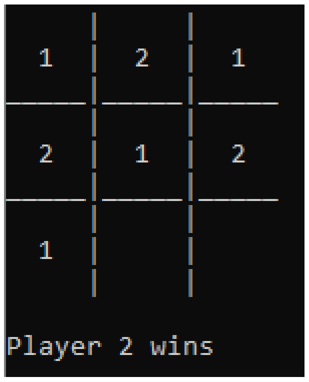
The result, i.e., the convergence of two maximally-intelligent players playing the given game of Anti-Tic-Tac-Toe is that ***the game ends with Player 1 never winning the game.***
The reason can be explained by using all the points that have been concluded so far, which are as follows,
-> If Player 1 wins the game, it implies that player 2 has ended the game. Since Player 2 ending the game always means a 'premature end', it implies that there is room for improvement. Hence, during the next improvement phase of Player 2, it would improve its gameplay by choosing different actions.
-> Similarly for Player 2 winning the game, if Player 1 ends the game prematurely, it would choose better actions in its next improvement phase, (only if a better reward can be achieved).

So, two optimal, intelligent players playing this game of Anti-Tic-Tac-Toe, ends in Player 1 always 'not winning'.
One important point is that the game could either end up in a draw or Player 1 losing, it does not matter to Player 1 from a reward perspective, but for Player 2, it does matter. Therefore, the game could either end in a draw or Player 1 losing, which purely depends on the implementation details of planner.py
Reward-wise it doesn't matter to Player 1. In my implementation of planner.py, for the same action-value function, it would choose the action number which is smalller. The final result is: **Player 1 cannot win.**

For the optimal policies policyfile1_10.txt and policyfile2_10.txt, **in the automated gameplay**,

Player 1 starts the game like this 

```
      |      |
  1   |      |
______|______|______
      |      |
      |      |
______|______|______
      |      |
      |      |
```

and ends like 

```
      |      |
  1   |  2   |  1
______|______|______
      |      |
  2   |  1   |  2
______|______|______
      |      |
  1   |      |
      |      |

Player 2 wins
```

.
Why? Because when the convergence is reached, Player 1 cannot win at the end against an intelligent player 2, hence, losing or drawing is the same from a reward POV. **Hence, it just selects the first action, it does not matter to him, he can't win from an intelligent Player 2.**

Conclusion: The game ends with Player 1 always 'not winning' the game.