Dynamic Programming and Reinforcement Learning Assignment 3

Submitted by:
Abhishek Subramanian Iyer - 2724035
Laréb Fatima Ahmad - 2737338

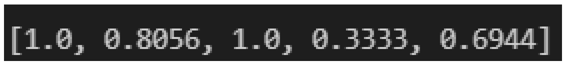
INTRODUCTION

In this report, we will be describing the solutions we have achieved for playing a Tic Tac Toe game from restricted conditions using Monte-Carlo Tree Search (MCTS) algorithm(s).
This report includes the following :

1 A OPPONENT IS PURELY RANDOM

We start off by creating a monte carlo tree with nodes, in which there are five starting branches (one for each action). The tree is created by exploring the results of all possible actions and thus expanding the tree, this is done in a recursive manner. The nodes contain the value 1, -1 or 0 depending on if the given branch of actions results in a win, loss or draw.

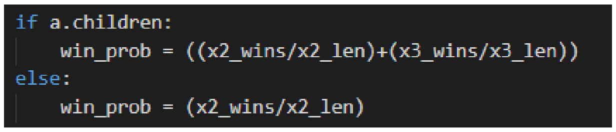
Given the optimal policy, the respective win probabilities of the five possible actions are



```
[1.0, 0.8056, 1.0, 0.3333, 0.6944]
```

Figure 2: The winning probabilities

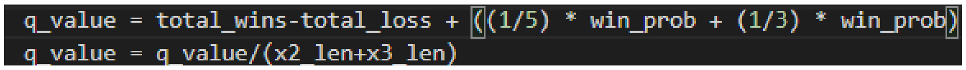
If the player starts with action 0 or action 2, they are guaranteed to win. This is deduced by counting the winning nodes when traversing a given action's branch, and dividing that number with the number of total nodes in the branch. The code in figure 2 shows how, where the variables x2_len and x3_len represent the number of nodes in level 2 and 3 of the tree.



```
if a.children:
    win_prob = ((x2_wins/x2_len)+(x3_wins/x3_len))
else:
    win_prob = (x2_wins/x2_len)
```

Figure 3: Calculating win-probabilities

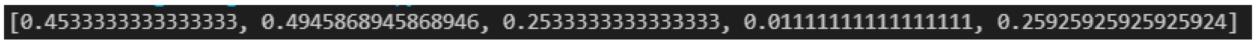
$$Q(x,a) = \frac{\sum_{x'} \left(r + \gamma \max_{a' \in \mathcal{A}} Q(x',a')\right)}{n(x,a)}.$$



```
q_value = total_wins-total_loss + ((1/5) * win_prob + (1/3) * win_prob)
q_value = q_value/(x2_len+x3_len)
```

Figure 4: Q-value formula and how it is implemented in the solution

The q-values for the given actions result in:



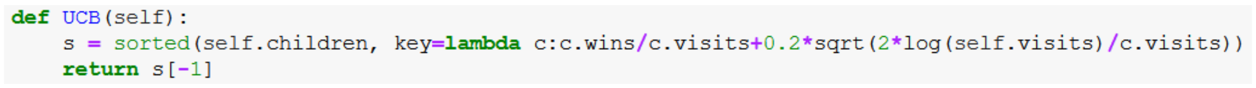
```
[0.4533333333333333, 0.4945868945868946, 0.2533333333333333, 0.01111111111111111, 0.25925925925925924]
```

Figure 5: The q-values

given the formula in figure 2. In the formula, **r is the total number of wins subtracted by the total number of losses**. Gamma is 1 because we are not looking for discounting as Tic Tac Toe is a small game. Then probability of taking any action from the players side is considered. This is ⅕ **for the first action, ⅓ for the second action, then 1 for the last one**. These probabilities are multiplied by the winning probability for the given action, and the values are added together. All this is divided by the total number of nodes in a given branch.
Recursion is not considered as this solution looks at the entire branch of a given action in one go.

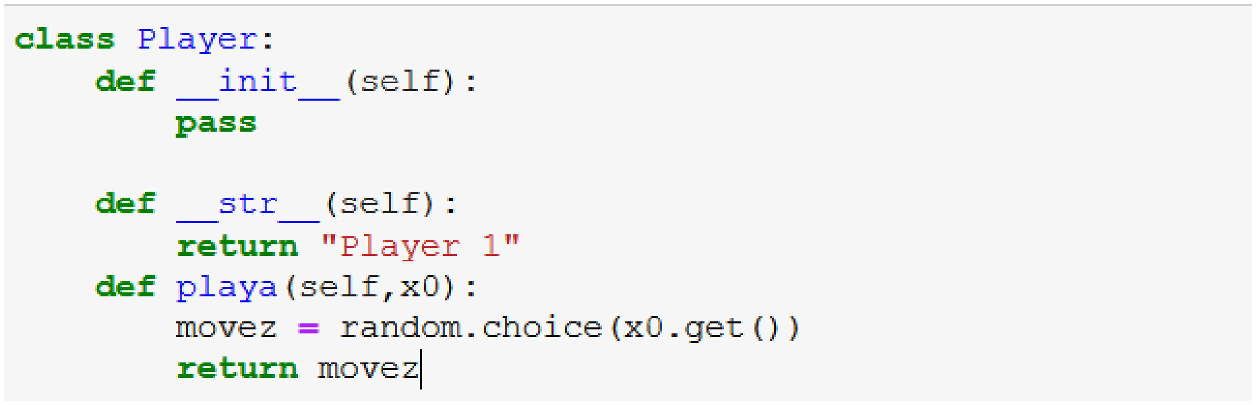
1 B OPPONENT PLAYS OPTIMALLY USING MCTS UCT

When we play against an agent that is implemented using MCTS UCT method, after the agent samples the tree, it selects the child which returns the highest UCB value. I



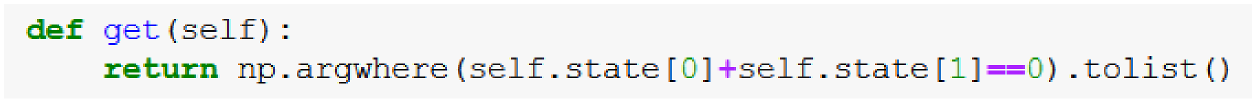
```
def UCB(self):
    s = sorted(self.children, key=lambda c:c.wins/c.visits+0.2*sqrt(2*log(self.visits)/c.visits))
    return s[-1]
```

Using the above formula we get all the children nodes and sort them in ascending order according to their UCB values and select the node with the highest value (the last one).
The random moves are defined by the function playa in the class Player. In the above formula we have chosen exploration parameter= √ 2.



```
class Player:
    def __init__(self):
        pass

    def __str__(self):
        return "Player 1"
    def playa(self,x0):
        movez = random.choice(x0.get())
        return movez
```

Here, x0 is the current state and we make a random choice and call the get function which fetches all the remaining actions that can be taken



```
def get(self):
    return np.argwhere(self.state[0]+self.state[1]==0).tolist()
```

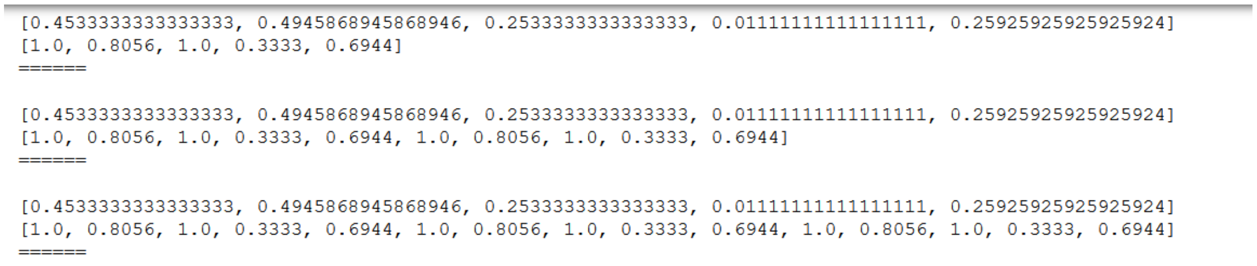
The UCT method is defined in the program using a function called Agent. Where agent simply adds nodes to the tree and runs a for loop for a number of iterations and for every iteration agent sets the root node and copies the tic tac toe board. Then the agent checks if the children are not empty, the agent then calls the UCB function and gets the best child and then places ‘O’ on the board.

The win probabilities for the actions are

```
win probability for action :  1  is  1.0
win probability for action :  2  is  0.5
win probability for action :  3  is  0.6666666666666666
win probability for action :  4  is  0.75
win probability for action :  5  is  0.8
```

1 C CONVERGENCE

When we consider the 1 A, we find that for n number of iterations the win probabilities stay the same when we are playing against a random opponent.



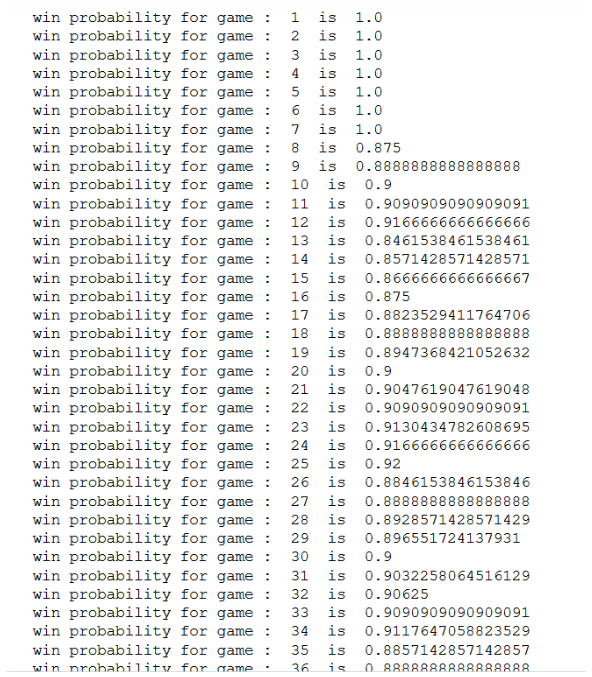
```
[0.4533333333333333, 0.4945868945868946, 0.2533333333333333, 0.01111111111111111, 0.2592592592592592 4]
[1.0, 0.8056, 1.0, 0.3333, 0.6944]
======

[0.4533333333333333, 0.4945868945868946, 0.2533333333333333, 0.01111111111111111, 0.2592592592592592 4]
[1.0, 0.8056, 1.0, 0.3333, 0.6944, 1.0, 0.8056, 1.0, 0.3333, 0.6944]
======

[0.4533333333333333, 0.4945868945868946, 0.2533333333333333, 0.01111111111111111, 0.2592592592592592 4]
[1.0, 0.8056, 1.0, 0.3333, 0.6944, 1.0, 0.8056, 1.0, 0.3333, 0.6944, 1.0, 0.8056, 1.0, 0.3333, 0.6944]
======
```

We keep adding the probabilities to the same list and as it can be seen, the probabilities just keep repeating.

However when we take 1 B, with exploration parameter √2 , when we play 10 games the accuracy looks a bit like this

```
win probability for game :  1  is  1.0
win probability for game :  2  is  1.0
win probability for game :  3  is  1.0
win probability for game :  4  is  0.75
win probability for game :  5  is  0.8
win probability for game :  6  is  0.8333333333333334
win probability for game :  7  is  0.8571428571428571
win probability for game :  8  is  0.875
win probability for game :  9  is  0.8888888888888888
win probability for game :  10  is  0.9
```

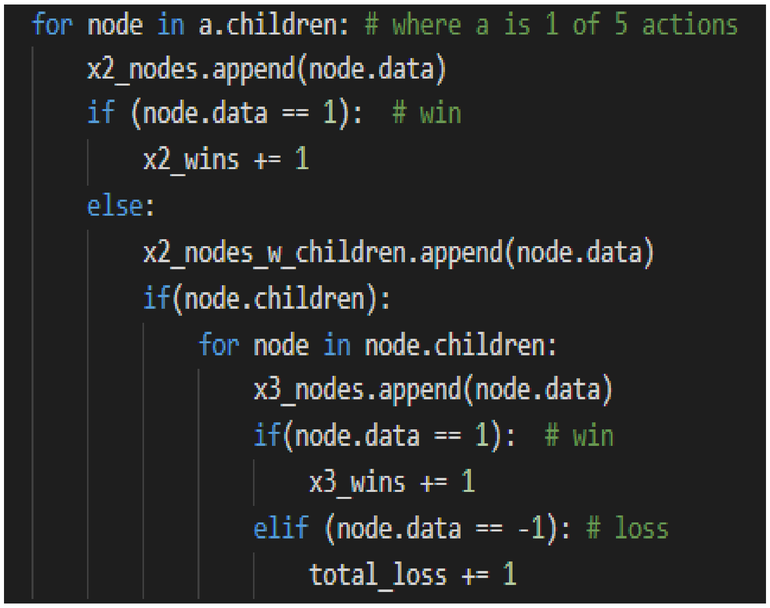
We can see the accuracy falling after game 3 and this is where it starts to converge. When we play a 100 games the convergence looks a bit like



```
win probability for game :  1  is  1.0
win probability for game :  2  is  1.0
win probability for game :  3  is  1.0
win probability for game :  4  is  1.0
win probability for game :  5  is  1.0
win probability for game :  6  is  1.0
win probability for game :  7  is  1.0
win probability for game :  8  is  0.875
win probability for game :  9  is  0.8888888888888888
win probability for game :  10  is  0.9
win probability for game :  11  is  0.9090909090909091
win probability for game :  12  is  0.9166666666666666
win probability for game :  13  is  0.8461538461538461
win probability for game :  14  is  0.8571428571428571
win probability for game :  15  is  0.8666666666666667
win probability for game :  16  is  0.875
win probability for game :  17  is  0.8823529411764706
win probability for game :  18  is  0.8888888888888888
win probability for game :  19  is  0.8947368421052632
win probability for game :  20  is  0.9
win probability for game :  21  is  0.9047619047619048
win probability for game :  22  is  0.9090909090909091
win probability for game :  23  is  0.9130434782608695
win probability for game :  24  is  0.9166666666666666
win probability for game :  25  is  0.92
win probability for game :  26  is  0.8846153846153846
win probability for game :  27  is  0.8888888888888888
win probability for game :  28  is  0.8928571428571429
win probability for game :  29  is  0.896551724137931
win probability for game :  30  is  0.9
win probability for game :  31  is  0.9032258064516129
win probability for game :  32  is  0.90625
win probability for game :  33  is  0.9090909090909091
win probability for game :  34  is  0.9117647058823529
win probability for game :  35  is  0.8857142857142857
win probability for game :  36  is  0.8888888888888888
```

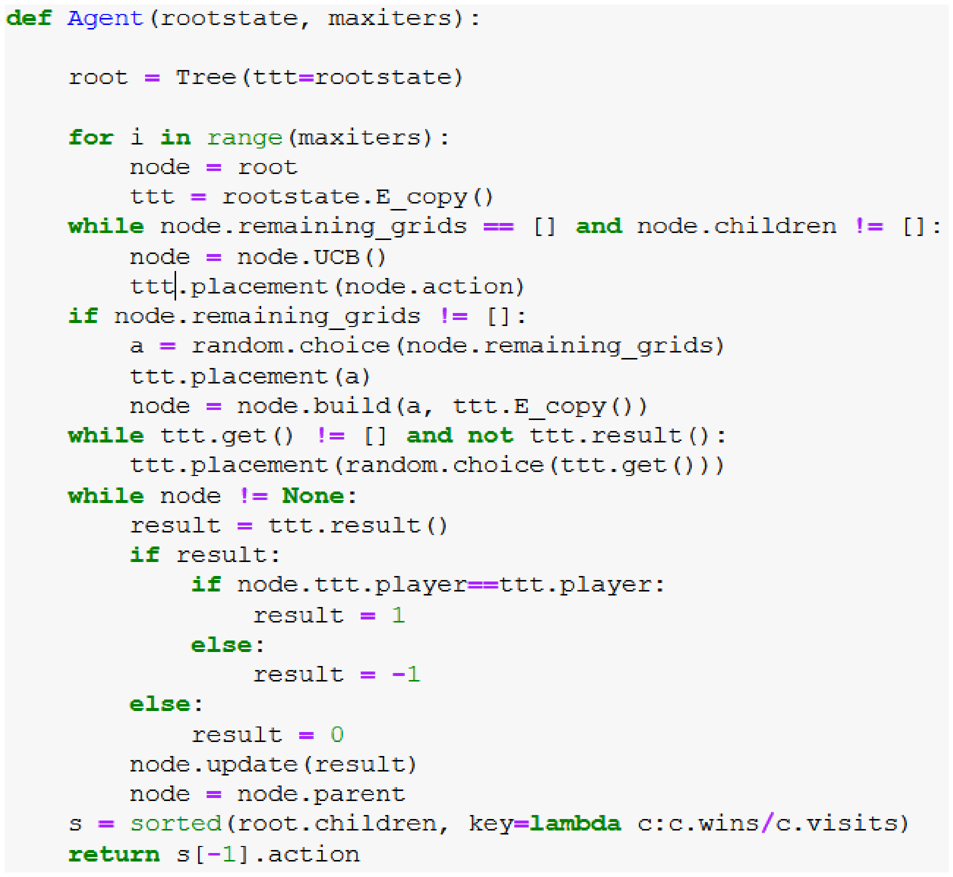
Here, the convergence starts at game 8. So for 1 B convergence occurs when we pay less amounts of game and when we play a lot of games, the convergence occurs relatively at a later stage

2 IMAGES -



```
for node in a.children: # where a is 1 of 5 actions
    x2_nodes.append(node.data)
    if (node.data == 1):  # win
        x2_wins += 1
    else:
        x2_nodes_w_children.append(node.data)
        if(node.children):
            for node in node.children:
                x3_nodes.append(node.data)
                if(node.data == 1):  # win
                    x3_wins += 1
                elif (node.data == -1): # loss
                    total_loss += 1
```

Image 1: Tree traversal code



```
def Agent(rootstate, maxiters):

    root = Tree(ttt=rootstate)

    for i in range(maxiters):
        node = root
        ttt = rootstate.E_copy()
    while node.remaining_grids == [] and node.children != []:
        node = node.UCB()
        ttt.placement(node.action)
    if node.remaining_grids != []:
        a = random.choice(node.remaining_grids)
        ttt.placement(a)
        node = node.build(a, ttt.E_copy())
    while ttt.get() != [] and not ttt.result():
        ttt.placement(random.choice(ttt.get()))
    while node != None:
        result = ttt.result()
        if result:
            if node.ttt.player==ttt.player:
                result = 1
            else:
                result = -1
        else:
            result = 0
        node.update(result)
        node = node.parent
    s = sorted(root.children, key=lambda c:c.wins/c.visits)
    return s[-1].action
```

Image 2: UCT code