Group Name: Tic Tac Toe

Members: Hamna Jamil, Rohit Kumar, Lama Imam

Section: DSA L-1

Instructor: Ayaz-ul-Hassan Khan

Report Breakdown:

Min Max algorithm intro and analysis:

- Minimax is a recursive algorithm which is used to choose an optimal move for a player assuming that the opponent is also playing optimally.

- In this algorithm two players play the game, one is called maximizer and other is called minimizer.
- Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit.
- The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion

Pseudo-code for MinMax Algorithm:

```
function minimax(node, depth, maximizingPlayer) is
    if depth ==0 or node is a terminal node then
        return static evaluation of node

    if MaximizingPlayer then      // for Maximizer Player
        maxEva= -infinity
        for each child of node do
            eva= minimax(child, depth-1, false)
            maxEva= max(maxEva,eva)        //gives Maximum of the values
        return maxEva

    else                          // for Minimizer player
        minEva= +infinity
        for each child of node do
            eva= minimax(child, depth-1, true)
            minEva= min(minEva, eva)        //gives minimum of the values
        return minEva
```

Minmax Algorithm:

```
function minimax(board, depth, isMaximizingPlayer):

  if current board state is a terminal state :
    return value of the board

  if isMaximizingPlayer :
    bestVal = -INFINITY
    for each position in board from 0 to 9 :
      value = minimax(board, depth+1, false)
      bestVal = max( bestVal, value)
    return bestVal

  else :
    bestVal = +INFINITY
    for each position in board from 0 to 9 :
      value = minimax(board, depth+1, true)
      bestVal = min( bestVal, value)
    return bestVal
```

Reference: https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-3-tic-tac-toe-ai-finding-optimal-move/?ref=rp

How min max works in tic tac toe:

This initial state of the board is as follows:

X	**O**	

e(p) = 6 - 5 = 1
Board position of 3x3 matrix with 0 and X as shown above is initial state of the board
we then put 0's or X's in vacant positions alternatively, X is the maximizer and O is the minimizer.
we then perform a Terminal test, which determines if game is over
we form a Utility function which is:
e(p) = (No. of complete rows, columns or diagonals are still open for player) – (No. of complete rows, columns or diagonals are still open for opponent)

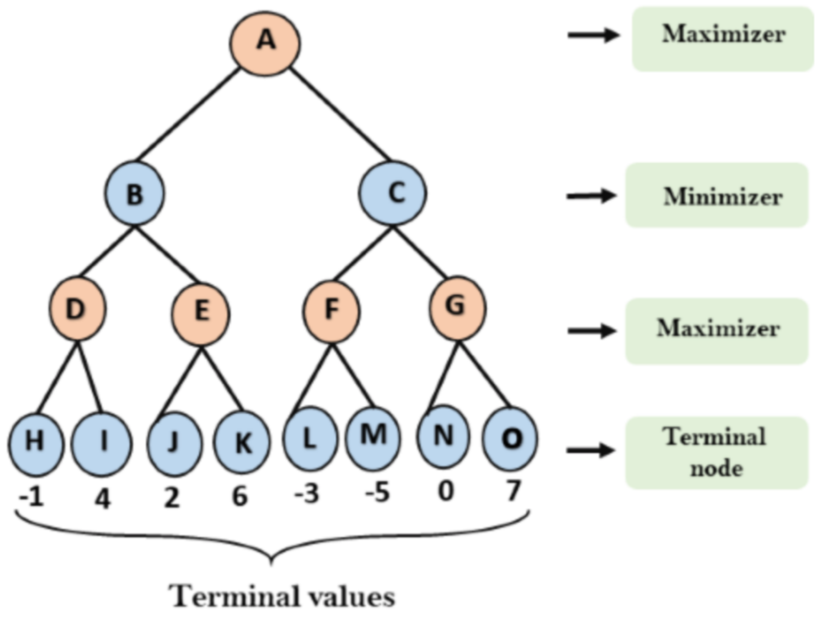
Now we generate the game tree;
The we apply the utility function to each terminal state to get its value
We then use these values to determine the utility of the nodes one level higher up in the search tree;

- From bottom to top
- For a max level, select the maximum value of its successors
- For a min level, select the minimum value of its successors

Finally, from root node select the move which leads to highest value

Dry run of the above explained algorithm:

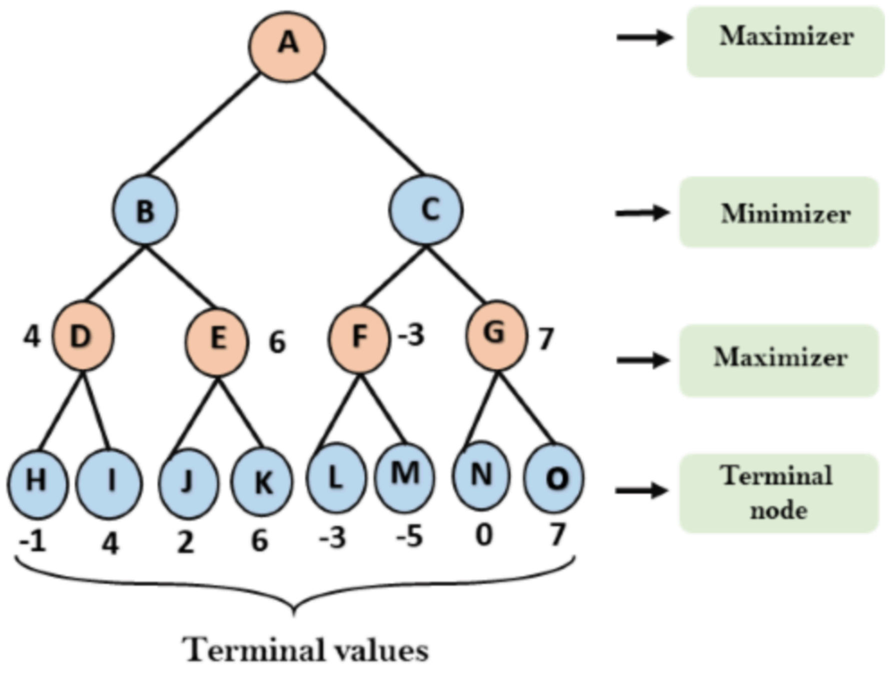
1: In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the tree below lets take A is the initial state of the tree. Suppose maximize, player 1, takes first turn which has worst-case initial value =- infinity, and minimizer, player 2, will take next turn which has worst-case initial value = +infinity.



2: To find the maximum among all possible values, we find the utilities value for the Maximizer, its initial value is -infinity, so we will compare each value in terminal state with initial value of Maximizer and determine the higher nodes values.

- For node D max(-1,- -∞) => max(-1,4)= 4
- For Node E max(2, -∞) => max(2, 6)= 6
- For Node F max(-3, -∞) => max(-3,-5) = -3
- For node G max(0, -∞) = max(0, 7) = 7

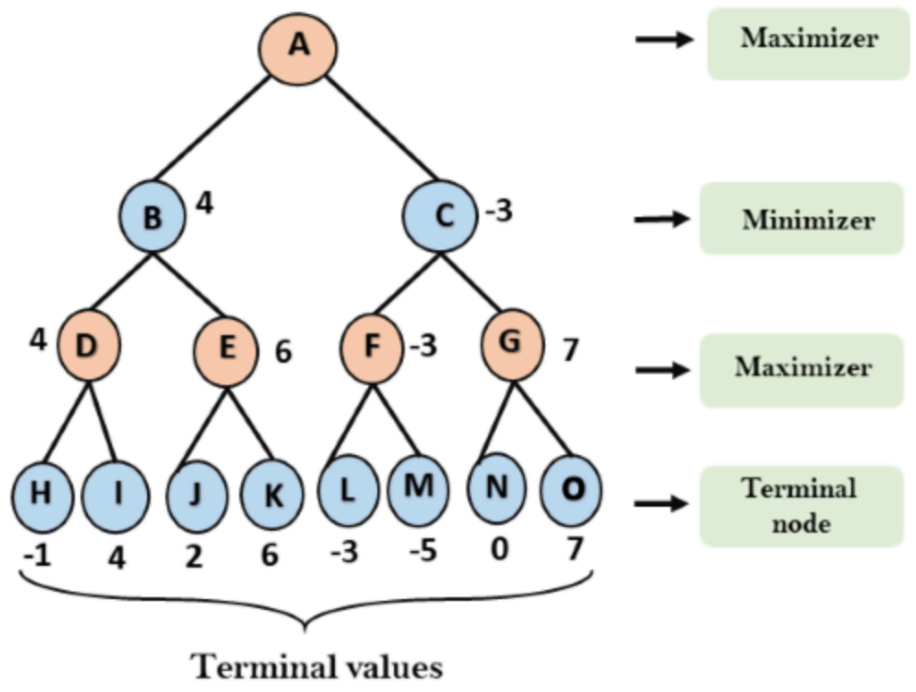
Game tree now looks like:



3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.

- For node B= min(4,6) = 4
- For node C= min (-3, 7) = -3

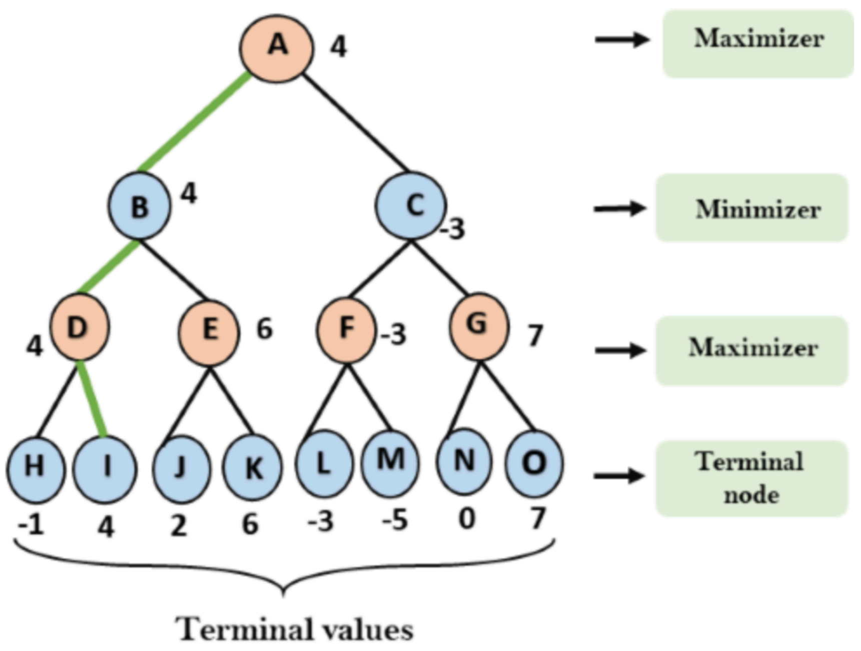
Game tree now looks like:



4: we go back to Maximizer's turn again, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 levels, hence we reach immediately to the root node, but in real life games, there will be more than 4 levels.

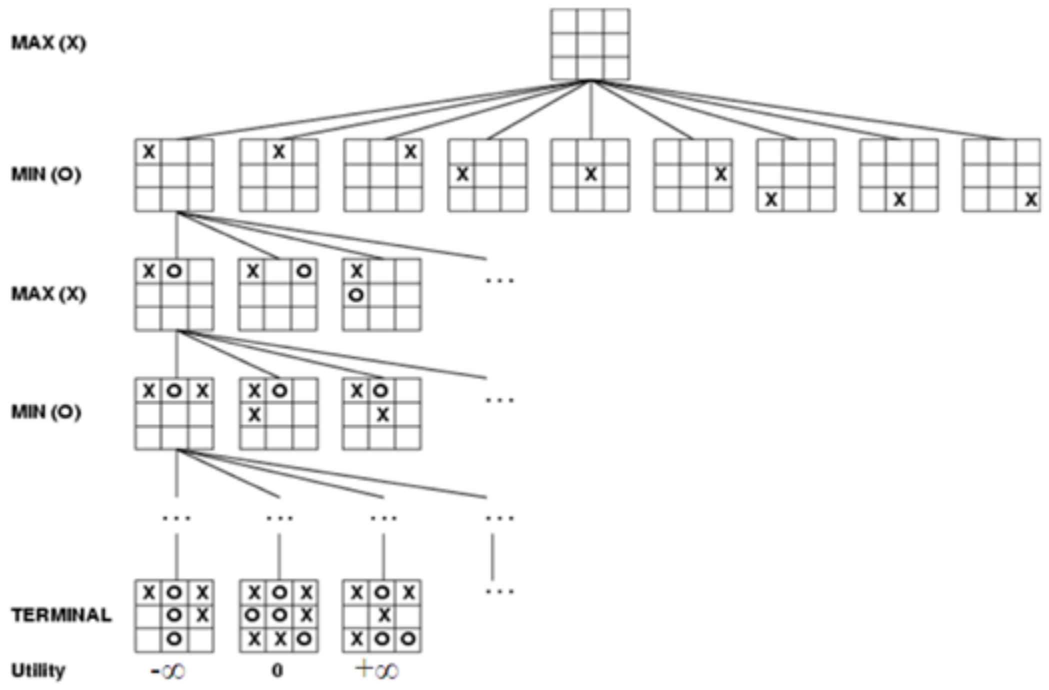
- For node A max(4, -3)= 4

The final game tree, fully parsed now looks like:



Reference: https://www.javatpoint.com/mini-max-algorithm-in-ai

This is how the overall game tree generation and max and min function calls for Tic Tac toe look:



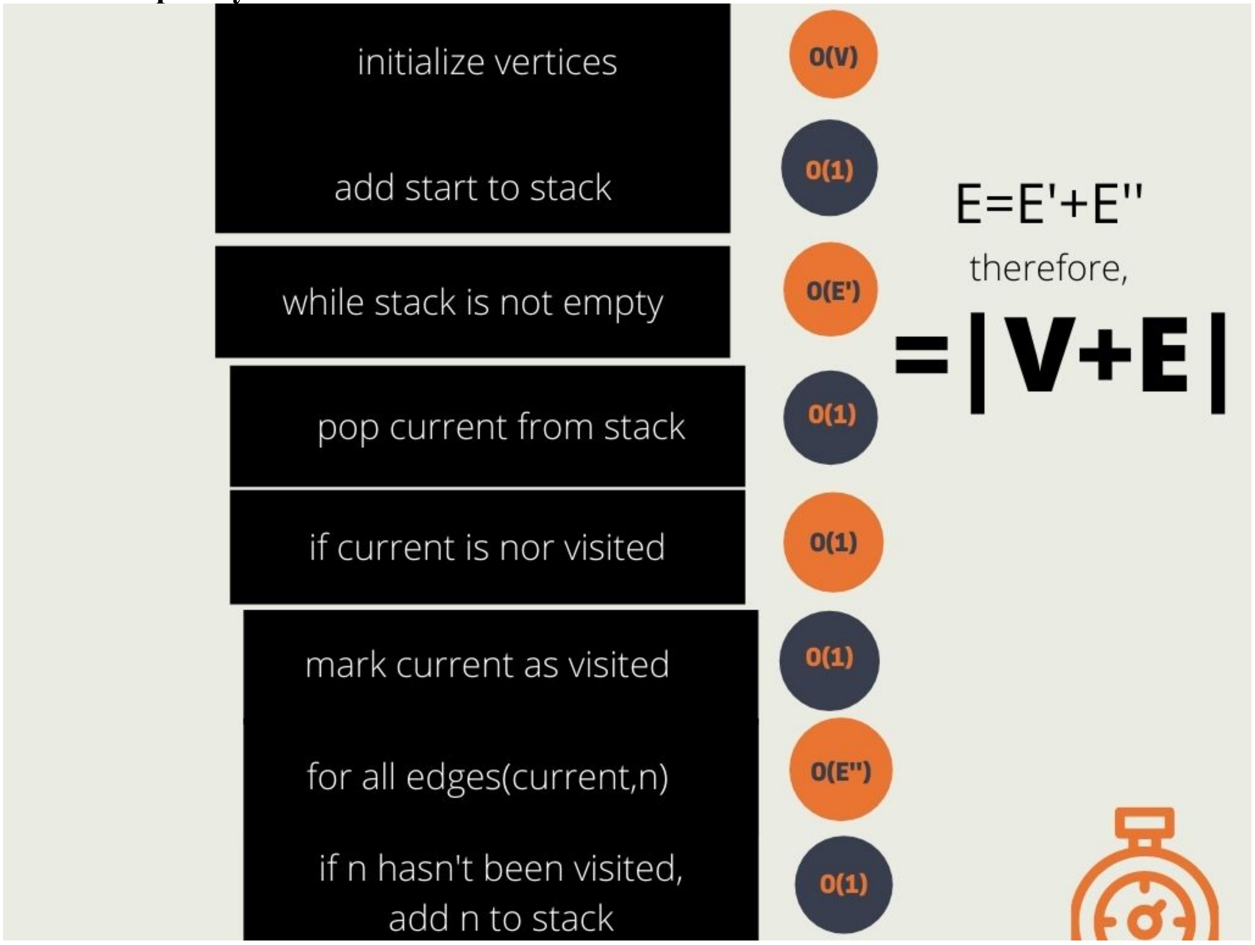
Time and space complexity of the algorithm;

As it performs DFS for the game-tree, the time complexity of the Min-Max algorithm is O(V+E), where V are the vertices of the game-tree, and E are the edges. Space complexity of the Minimax algorithm is also similar to DFS which is O(h) where h is the maximum depth of the tree.

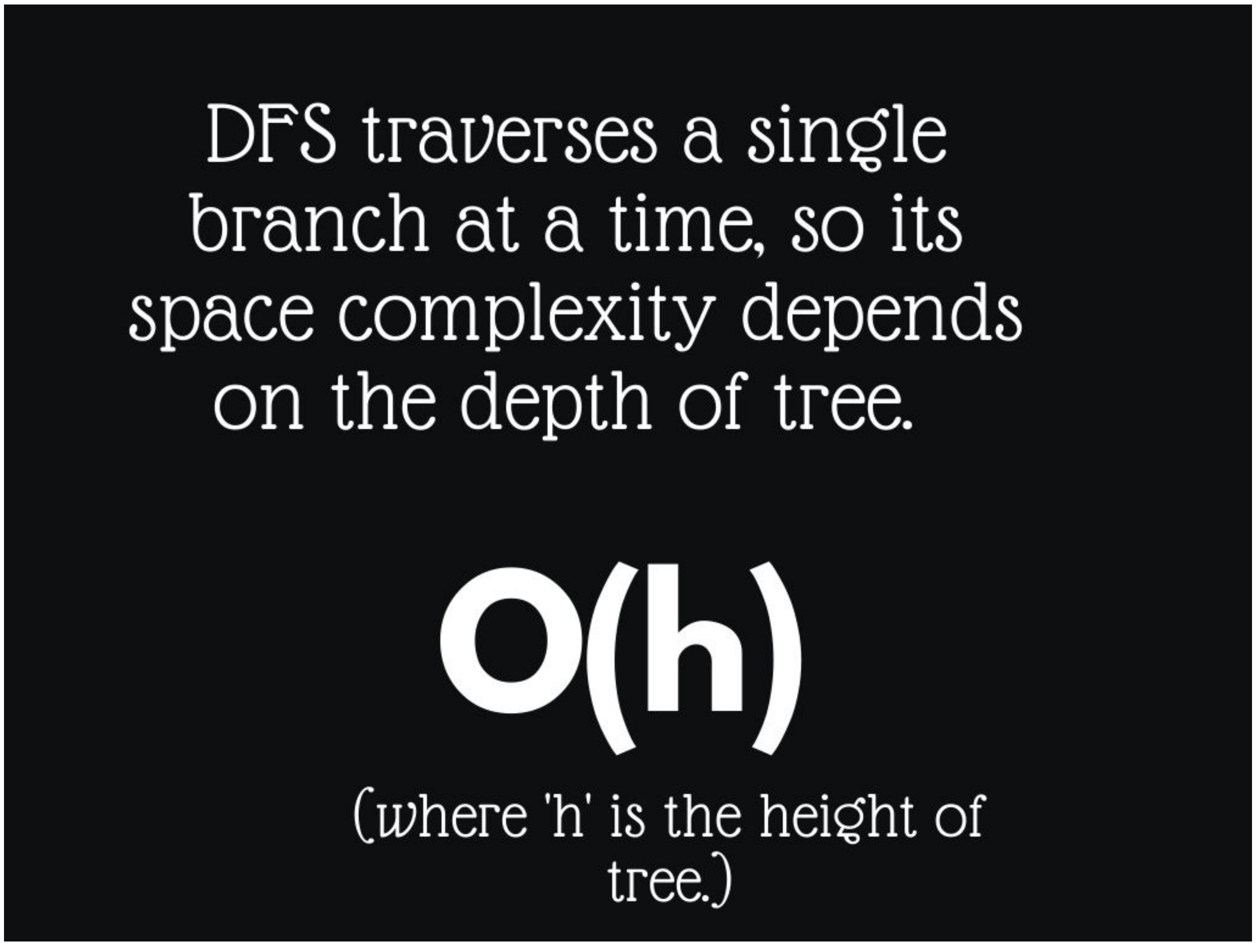
Derivation of the above-mentioned time complexity of minmax algorithm:

Since we use DFS to parse the game tree, DFS time complexity is minmax's time complexity. Below is the derivation for DFS time complexity:

Time Complexity:



Space Complexity:



DFS traverses a single branch at a time, so its space complexity depends on the depth of tree.

$O(h)$

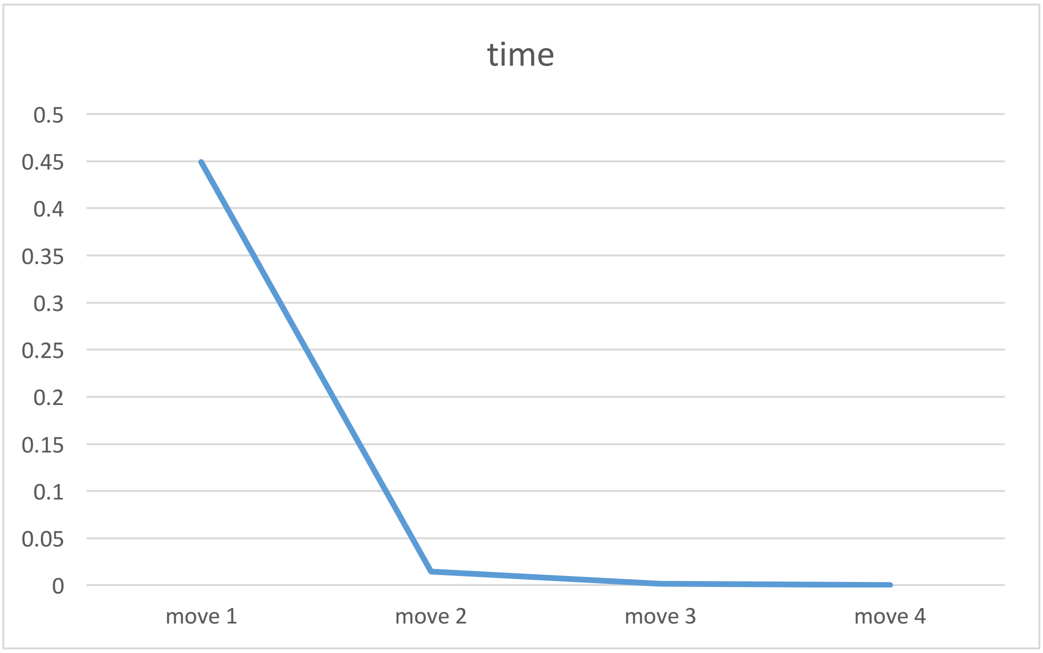
(where 'h' is the height of tree.)

Graphical runtime analysis of Minmax algorithm:

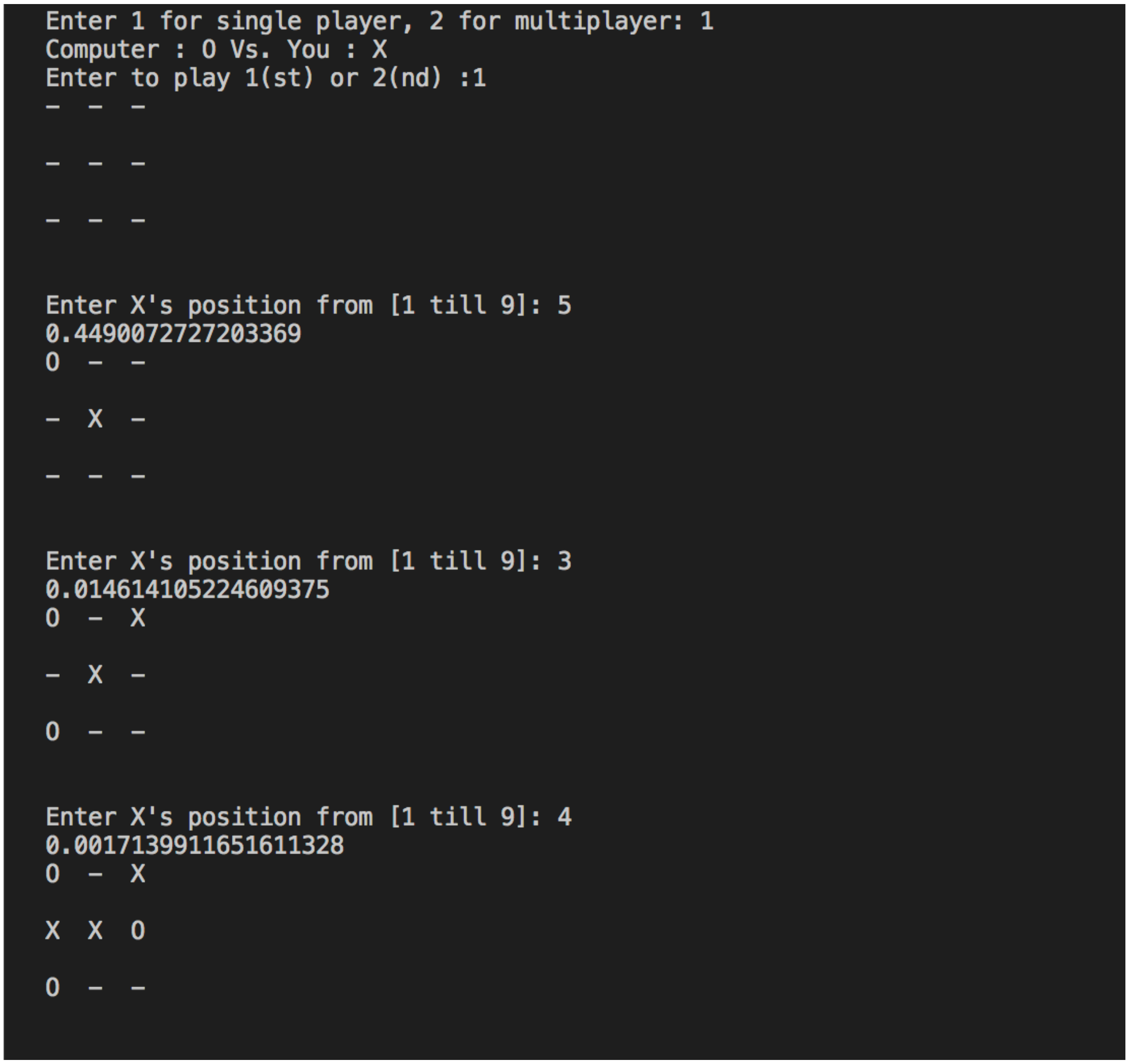
We calculated runtimes for how long it takes for the computer to implement minmax algorithm and make all it's moves**:**

Case-1:

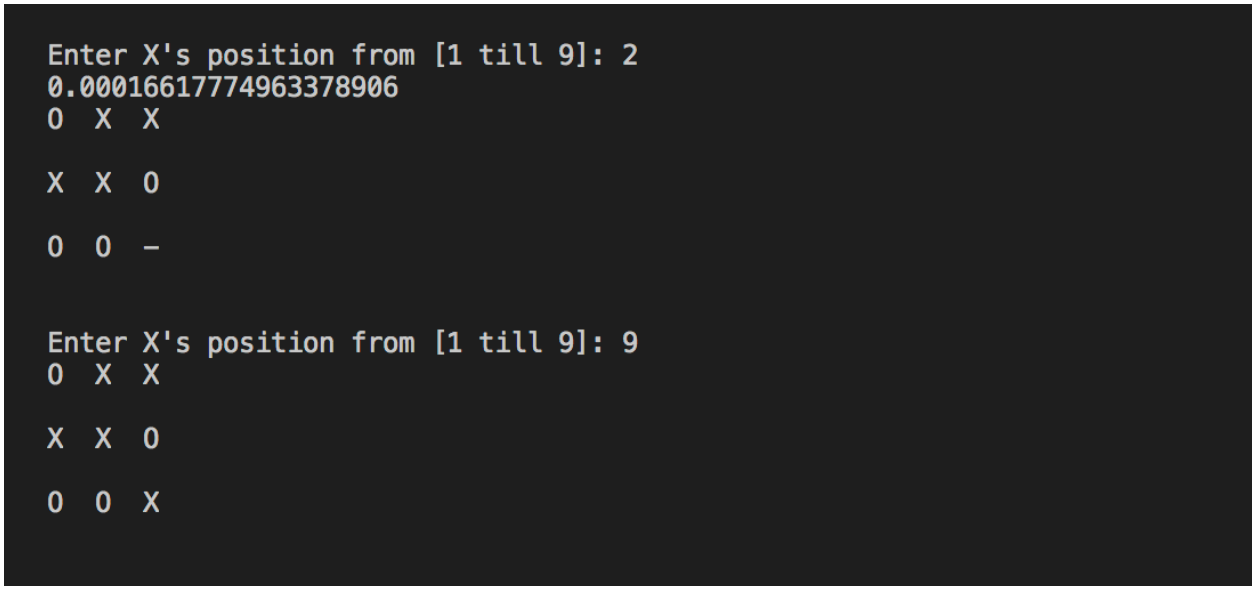
game was a draw
player went first



Screenshots of terminal window



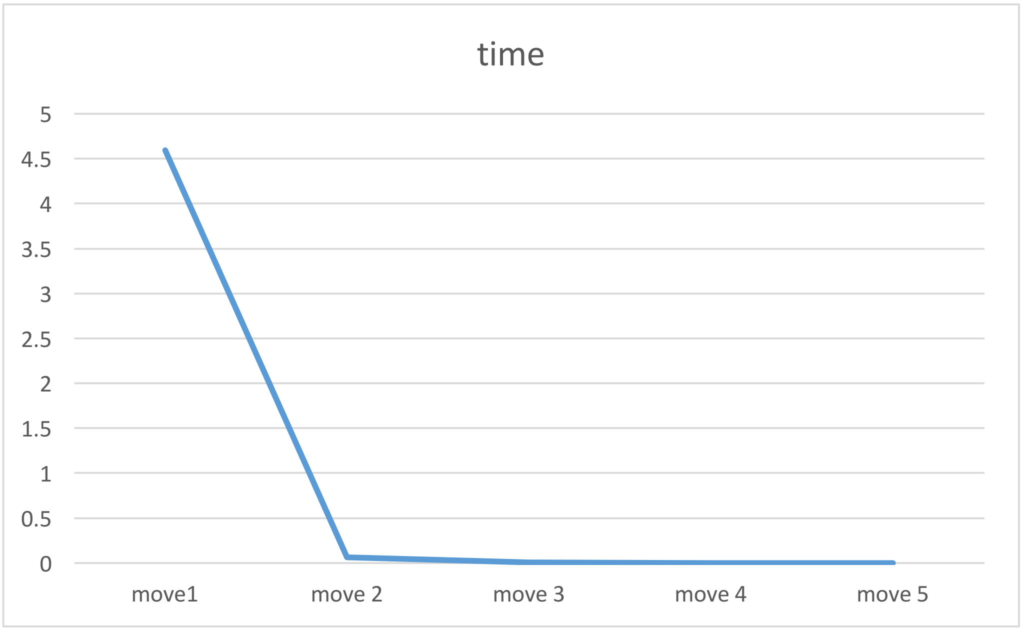
```
Enter 1 for single player, 2 for multiplayer: 1
Computer : O Vs. You : X
Enter to play 1(st) or 2(nd) :1
-  -  -

-  -  -

-  -  -


Enter X's position from [1 till 9]: 5
0.4490072727203369
O  -  -

-  X  -

-  -  -


Enter X's position from [1 till 9]: 3
0.014614105224609375
O  -  X

-  X  -

O  -  -


Enter X's position from [1 till 9]: 4
0.0017139911651611328
O  -  X

X  X  O

O  -  -
```



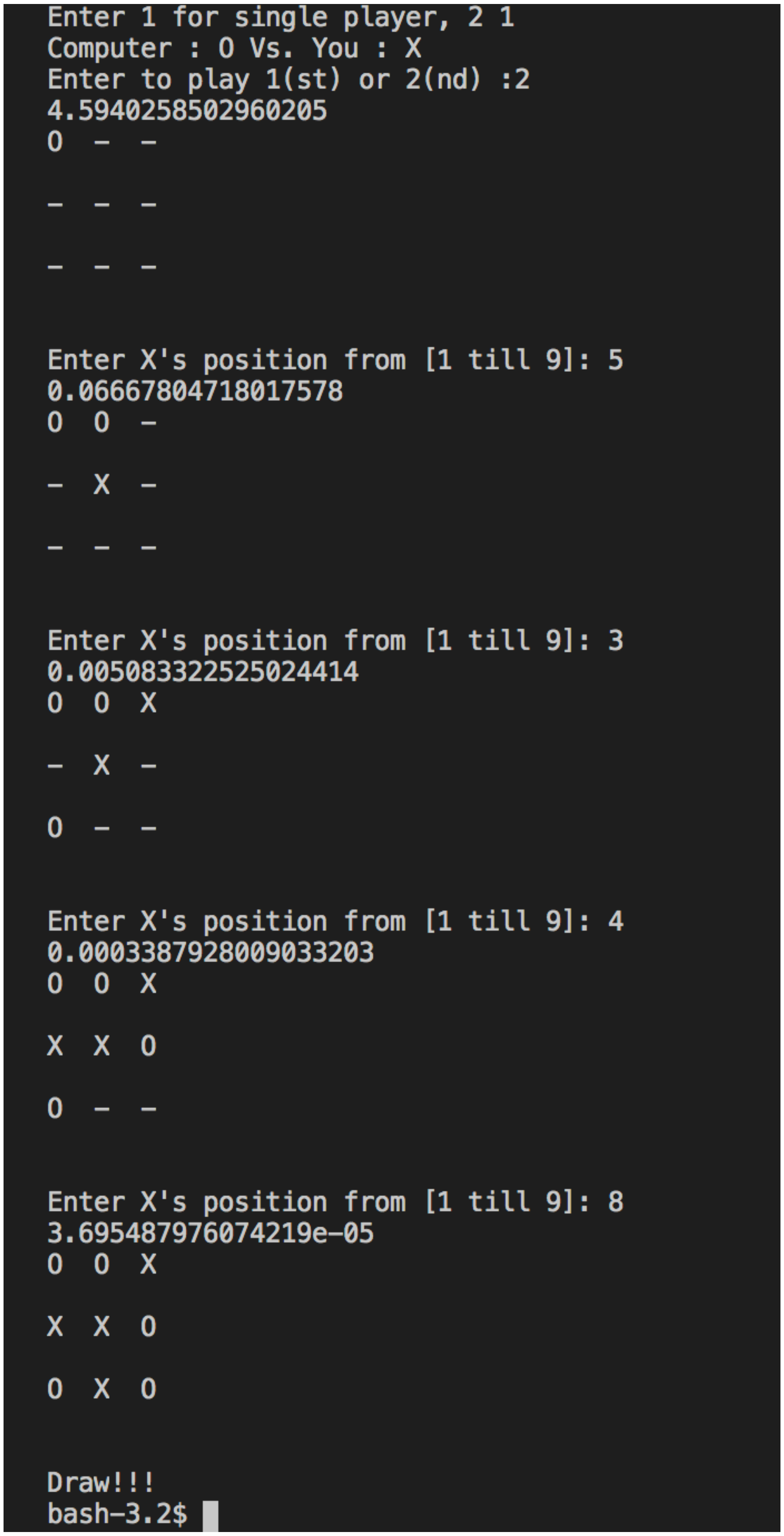
```
Enter X's position from [1 till 9]: 2
0.00016617774963378906
O  X  X

X  X  O

O  O  -


Enter X's position from [1 till 9]: 9
O  X  X

X  X  O

O  O  X
```

Case-2:

Player went second
game was a draw



Screenshot of terminal window showing runtimes:



```
Enter 1 for single player, 2 1
Computer : O Vs. You : X
Enter to play 1(st) or 2(nd) :2
4.5940258502960205
O  -  -

-  -  -

-  -  -


Enter X's position from [1 till 9]: 5
0.06667804718017578
O  O  -

-  X  -

-  -  -


Enter X's position from [1 till 9]: 3
0.005083322525024414
O  O  X

-  X  -

O  -  -


Enter X's position from [1 till 9]: 4
0.0003387928009033203
O  O  X

X  X  O

O  -  -


Enter X's position from [1 till 9]: 8
3.695487976074219e-05
O  O  X

X  X  O

O  X  O


Draw!!!
bash-3.2$
```

Analysis of above experiment:

As seen in the pictures above we took two cases in consideration in which two different conditions were implied. For case 1 as seen we have taken player 1 to make the first move and match was a draw in the end. By noticing their run time, we have seen that as the number of moves made increases time decreases which means with greater number of choices to make minmax algorithm takes more time, however when the number of choices is reduced such as less positions are available as no of moves mad increases then the runtime exponentially declines.

Similarly, to confirm the results second case which is case 2 was taken where player goes second and the game was draw in the end. In this case again the time taken by minmax algorithm was getting reduced with each move, although in this game time reduced drastically from first move made my computer to the second one since computer went first and it had all available positions it took much longer to makes its first move, 4.5 seconds compared with 0.4 seconds it took after player had made the first move.

After taking two different cases we could come up with final conclusion

that minmax algorithm takes lower time when there are lesser choices to make, it will take less time with every move in this tic tac toe game as the number of choices to make gets reduced by 1, every time the player gets a chance to play. This result also makes us aware some short comings that we will be discussing of how minmax algorithm will not be a good option with larger number of choices it’s time complexity will become worst case and runtime will be too much

Code:

We've divided the code into four main sections.

- The first part contributes in building the board as well as updating it after each player's turn.
- The second part is where the minimax algorithm comes in. depending on the mode the game is in, this part interacts with the player to enter their choice. If the game is in single player mode, the minimax algorithm helps the computer to play efficiently.
- Third part is where the board gets analyzed after each player has made its move. It basically checks for any winning combinations, so if any of the players has won, it returns accordingly. If not, the game continues.
- The last part is the main function which prompts the user to select between the two modes. It also lets the player choose if they want to go first. It calls all other functions as well at each step.

Code breakdown:

The foremost function of our constructs the 3x3 board. Initially it returns empty boxes represented by '_' underscores. As the players provide their inputs; it updates them accordingly. It also displays the board in terminal after each player's turn.

```
def basic_board(board): # The function is designed to form the board design for the game
   for a in range(9): # Loop iterates for 9 times from 0 till 9, as we have 9 positions in Tic Tac Toe game
      if((a>0) and (a%3)==0):
         print("\n")
      if(board[a]==0):
         print("- ",end=" ")
      if (board[a]==1):
         print("O ",end=" ")
      if(board[a]==-1):
         print("X ",end=" ")
   print("\n\n")
```

These two functions prompt the user to enter a position where they would want to place 'X'/'O'. It also generates a warning in case any of the users enter the wrong position. Player2 function is helpful when the game is in multiplayer mode. In single player mode, player1 is our user and computer acts as player2 for which we have a separate function to handle computer's moves.

```
def player1(board):
   move =input("Enter X's position from [1 till 9]: ") # The statement seek input of player 1 for positions 1 to 9
```

```
    move=int(move)  # In this statement the input is converted to insteger from string
    if(board[move-1]!=0):
        print("Please enter right move, else it will be dismissed !!")
        exit(0)
    board[move-1]=-1

# now we will take second player inputs in below function
def player2(board):
    move =input("Enter O's position from [1 till 9]: ") # The statement seek input of
player 2 for positions 1 to 9
    move =int(move) # In this statement the input is converted to insteger from string
    if(board[move-1]!=0):
        print("Please enter right move, else it will be dismissed !!")
        exit(0)
    board[move-1]=1
```

When the computer is playing, we utilize minmax algorithm. Computer_move function calls board and minmax function in it, in sequence. When its done the function again calls board function to update the value on the board as well as display it in the terminal.

In this part of our code where we implement minmax algorithm which performs DFS methods to change positions on the board as well as to form the combinations which are then compared with winning combinations to announce the winner (if any) or draw the game.

The codes check every move in all the iterations to perform the DFS as discussed above in minmax algorithm descriptions.

```
#our function implementing the minmax algorithm
def minmax(board,player):
#analyse board is a helper function that parses board for entered player combinations.
It is used to get current player position, then the algorithm then decides which position
for other to select to beat the player
    a=combinations(board)#change name of function when helper function designed
below:
    if(a!=0):
        return (a*player)
    final_position=-1
    value_moved=-2
    #0 to 9 since there are only 9 boxes for each mark to be placed.
    for i in range(0,9):
        if(board[i]==0):
            board[i]=player
            final_score=-minmax(board,(player*-1))
            if(final_score>value_moved):
                value_moved=final_score
                final_position=i
            board[i]=0
#if player 1 turn, final position=-1, return zero else return value of position to be used
by computer to make the best possible move according to the algorithm.

    if(final_position==-1):
        return 0
    return value_moved

#In this function computer makes its move utilising minmax helper function above to
implememnt minmax algorithm
def computer_move(board):
    final_position=-1
    value_moved=-2
    for i in range(0,9):
        if board[i]==0:
            board[i]=1
            #player=-1 implies running minmax algorithm for comoputer's turn.
            final_score=-minmax(board, -1)
            board[i]=0
            if(final_score>value_moved):
                value_moved=final_score
                final_position=i
#board postition used by computer while its computer's turn
    board[final_position]=1
```

This function analyzes the baord to check for any winning combination. We have pre-defined all the combinations that can lead one player to win.

```
#This function checks any possible winning combination and ruturns 0 if no
combination is found.
def combinations(board):
    c=[[0,1,2],[3,4,5],[6,7,8],[0,3,6],[1,4,7],[2,5,8],[0,4,8],[2,4,6]]

    for i in range(0,8):
        if(board[c[i][0]] != 0 and
           board[c[i][0]] == board[c[i][1]] and
        board[c[i][0]] == board[c[i][2]]):
            return board[c[i][2]]
    return 0
```

Main function interacts with user to choose single player or multiplayer mode. It evaluates each move calling helper functions defined above. The helper functions such as basic_board function to create the board and make changes in it with every move, combinations functions to compare the combinations of users as well as minmax algorithm to perform the DFS. If a winning combination is found, it prints the final result and exits the game.

```
def main():
    mode=input("Enter 1 for single player, 2 for multiplayer: ") #for single player or multiplayer mode
    mode=int(mode)
    #The broad is considered in the form of a single dimentional array.
    #One player moves 1 and other move -1.
    board=[0,0,0,0,0,0,0,0,0]
    if mode==1:
        print("Computer : O Vs. You : X")
        player= input("Enter to play 1(st) or 2(nd) :") #lets us choose which player will go first
        player = int(player)
        for i in range (0,9):
            if(combinations(board)!=0):
                break
            if((i+player)%2==0):
                computer_move(board)
            else:
                basic_board(board)
                player1(board)
    else:
        for i in range (0,9):
            if(combinations(board)!=0):
                break
            if((i)%2==0):
                basic_board(board)
                player1(board)
            else:
                basic_board(board)
                player2(board)


    x=combinations(board)
    if(x==0):
        basic_board(board)
        print("Draw!!!")
    if(x==-1):
        basic_board(board)
        print("X Wins!!! O Loose !!!")
    if(x==1):
        basic_board(board)
        print("X Loose!!! O Wins !!!!")

#---------------#
main()
```

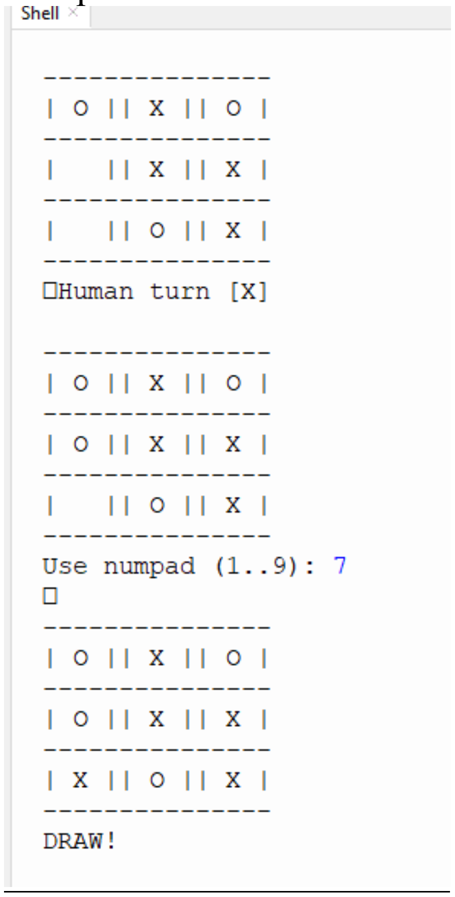
Experimental Analysis:

1: Comparison with already existing min max code:

To test the proper working of our code, we compared the output of our code with another working code which also uses minmax algorithm. We only tested single player mode because that is the part which uses minmax algorithm. *
We set the following assumptions before testing:

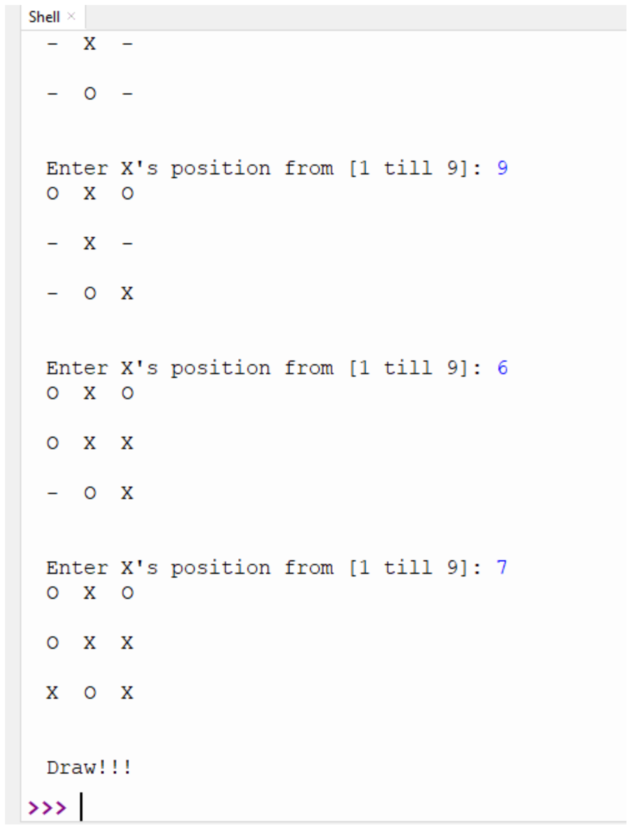
1. 'O' is computer and 'X' is player.
2. Input from player will follow the same sequence in both codes.

<u>Case-1:</u>
Comparison code



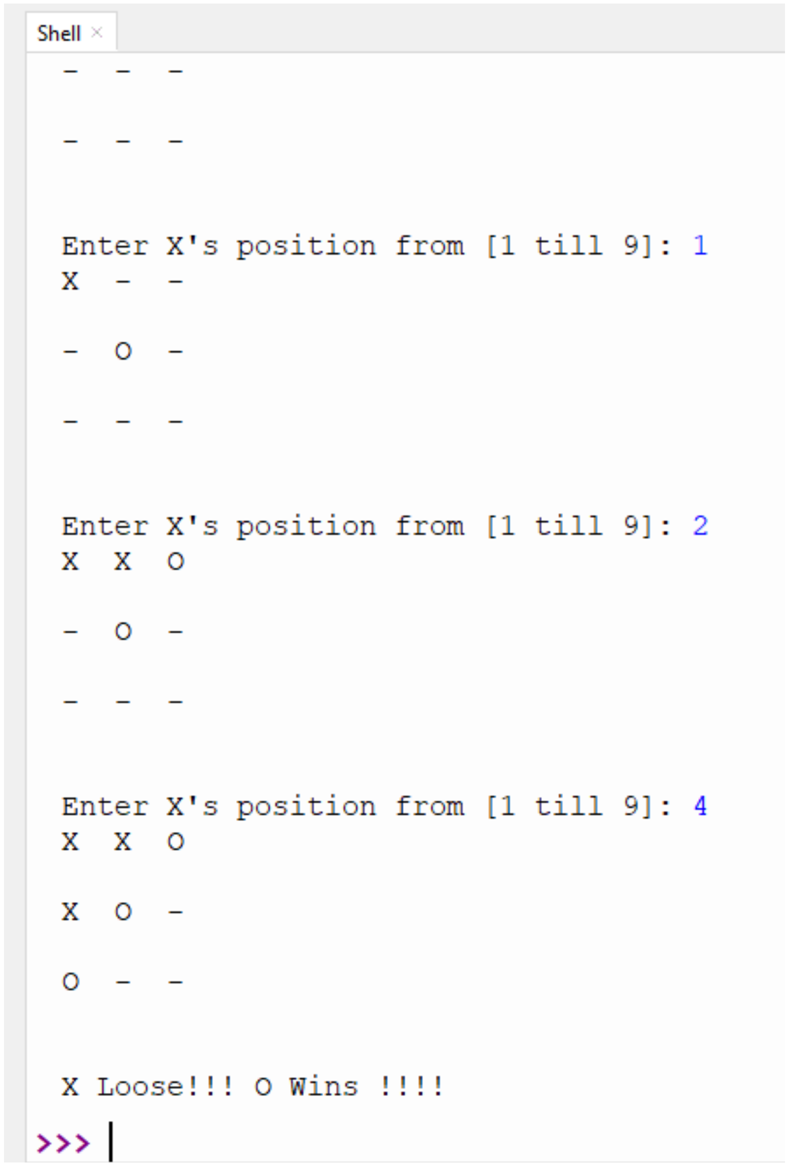
```
---------------
| O || X || O |
---------------
|   || X || X |
---------------
|   || O || X |
---------------
□Human turn [X]

---------------
| O || X || O |
---------------
| O || X || X |
---------------
|   || O || X |
---------------
Use numpad (1..9): 7
□
---------------
| O || X || O |
---------------
| O || X || X |
---------------
| X || O || X |
---------------
DRAW!
```

Our code


Shell

```
-  X  -

-  O  -


Enter X's position from [1 till 9]: 9
O  X  O

-  X  -

-  O  X


Enter X's position from [1 till 9]: 6
O  X  O

O  X  X

-  O  X


Enter X's position from [1 till 9]: 7
O  X  O

O  X  X

X  O  X


Draw!!!
>>> 
```

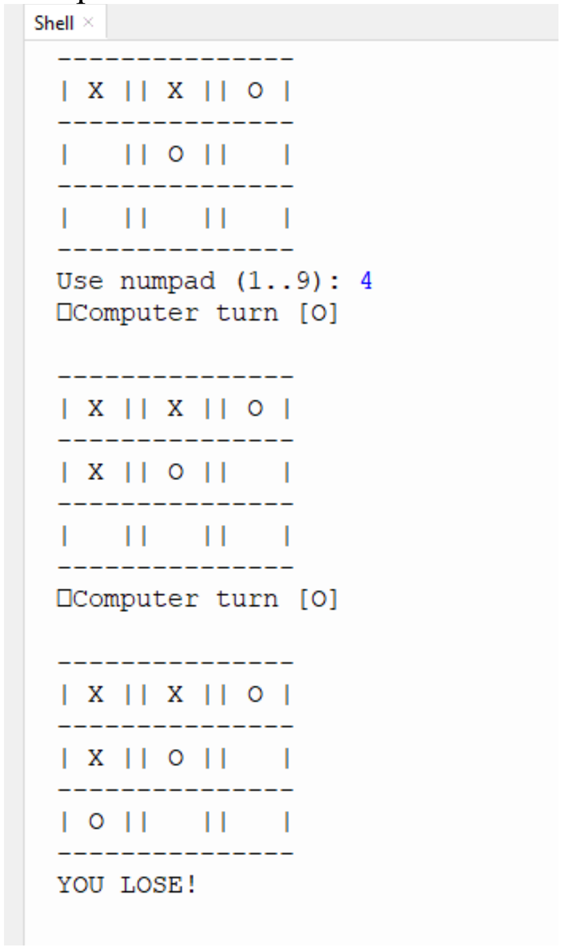
Case-2:
Our code



Shell

```
 -  -  -

 -  -  -


 Enter X's position from [1 till 9]: 1
 X  -  -

 -  O  -

 -  -  -


 Enter X's position from [1 till 9]: 2
 X  X  O

 -  O  -

 -  -  -


 Enter X's position from [1 till 9]: 4
 X  X  O

 X  O  -

 O  -  -


 X Loose!!! O Wins !!!!
>>> 
```

Comparison code



```
---------------
| X || X || O |
---------------
|   || O ||   |
---------------
|   ||   ||   |
---------------
Use numpad (1..9): 4
Computer turn [O]

---------------
| X || X || O |
---------------
| X || O ||   |
---------------
|   ||   ||   |
---------------
Computer turn [O]

---------------
| X || X || O |
---------------
| X || O ||   |
---------------
| O ||   ||   |
---------------
YOU LOSE!
```

In both cases, we played with similar strategies and it is evident that both codes produce the same outputs which proves the validity of our code.

Reference for comparison code:

https://github.com/Cledersonbc/tic-tac-toe
minimax/blob/master/py_version/minimax.py

2: Trying an experiment, try to win against algorithm:

So there's this commonplace trick known to always win the tic tac toe.

Step 1: You make the first move and place your x on the 5th position**.**

Step 2: After your opponents turn you place your x on the 3rd position, diagonally upwards and right from first position**.**

Step 3: Now your opponent will try and block you from making a winning combination by place their O on the 7th position so we don't make a diagonal**.**

Step 4: We next place our X on the 6th position now our opponent has only one turn but we have two winning combinations set out.

Step 5: if opponent doesn't place his O on the 4th position then our winning combination becomes (4,5,6).

Step 6: If they choose to block the 4th position and are unable to place their O on the 9th position then our winning combination becomes (3,6,9) either way I can win.

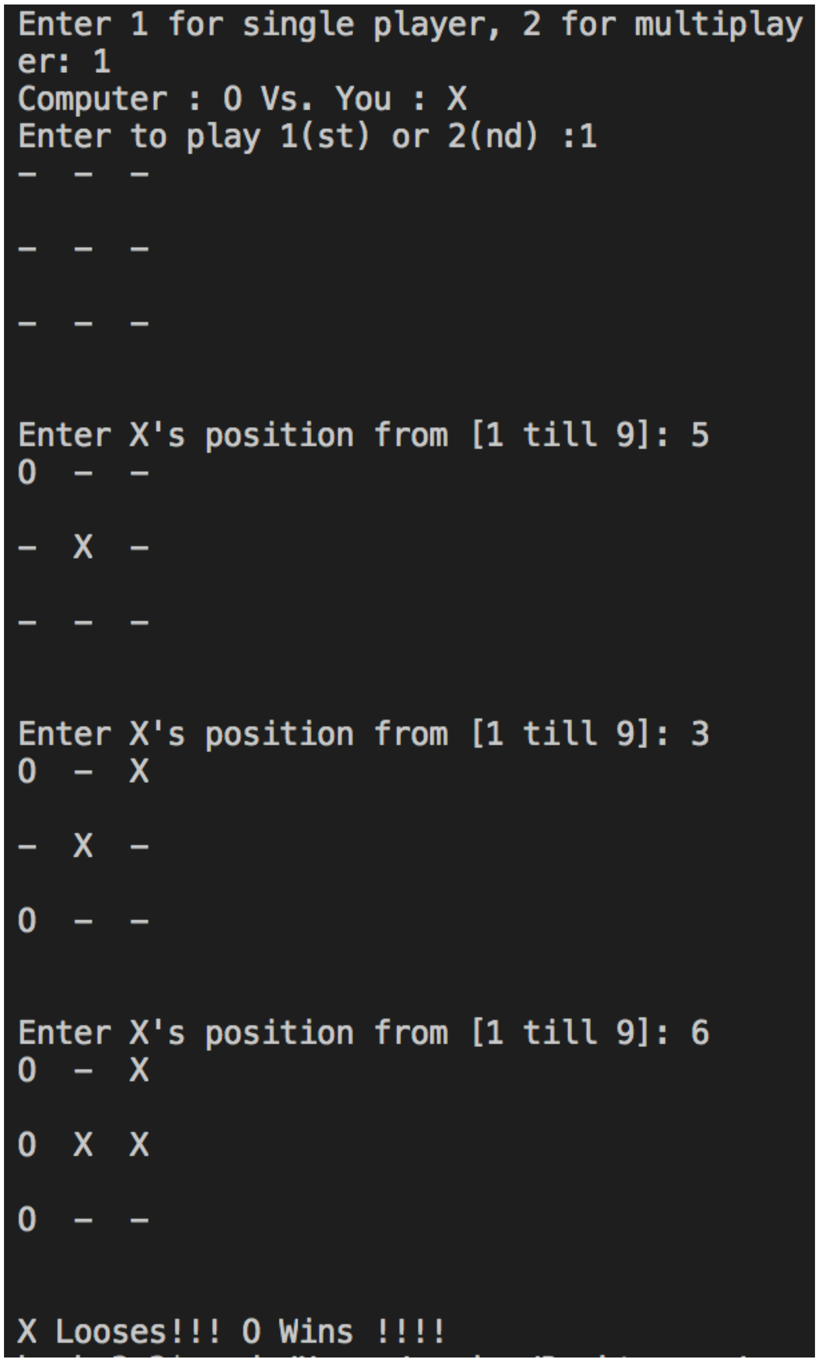
Or any other sequence of such steps

I will first try this trick against the algorithm in our games one player mode:

Snippets of game terminal:

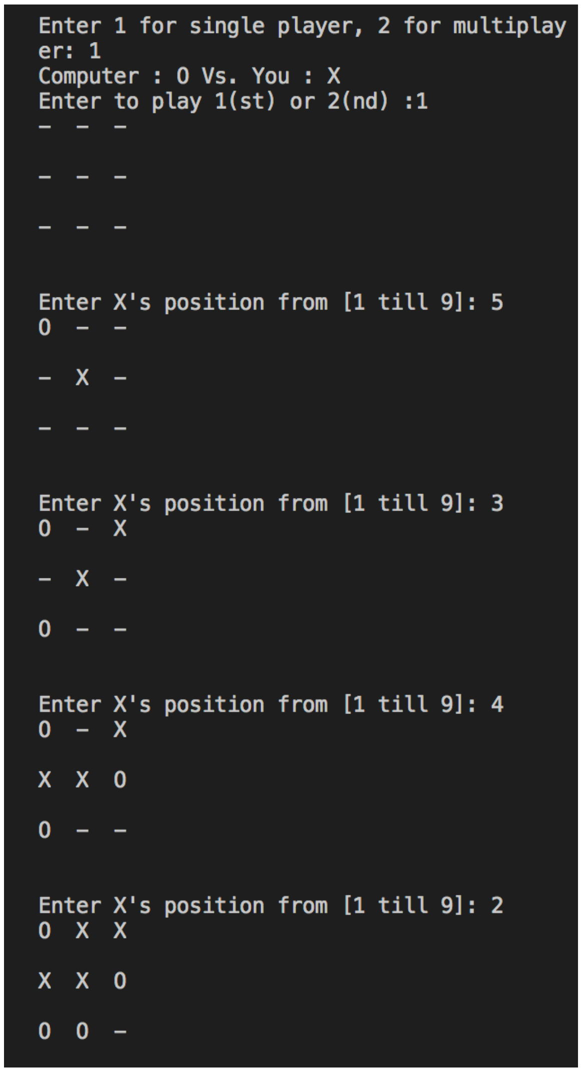
Case 1:

If I followed steps exactly:

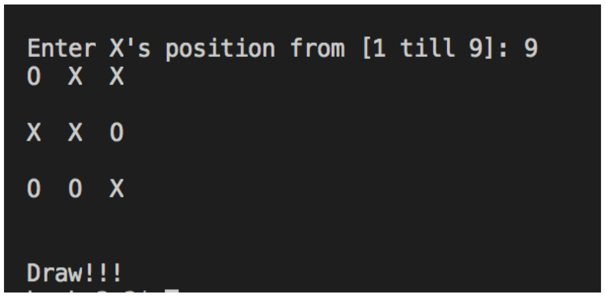


```
Enter 1 for single player, 2 for multiplay
er: 1
Computer : O Vs. You : X
Enter to play 1(st) or 2(nd) :1
-  -  -

-  -  -

-  -  -


Enter X's position from [1 till 9]: 5
O  -  -

-  X  -

-  -  -


Enter X's position from [1 till 9]: 3
O  -  X

-  X  -

O  -  -


Enter X's position from [1 till 9]: 6
O  -  X

O  X  X

O  -  -


X Looses!!! O Wins !!!!
```

Case 2: I deviated a bit from the steps to block the algorithm from winning but still ended up in a draw:



```
Enter 1 for single player, 2 for multiplay
er: 1
Computer : O Vs. You : X
Enter to play 1(st) or 2(nd) :1
-  -  -

-  -  -

-  -  -


Enter X's position from [1 till 9]: 5
O  -  -

-  X  -

-  -  -


Enter X's position from [1 till 9]: 3
O  -  X

-  X  -

O  -  -


Enter X's position from [1 till 9]: 4
O  -  X

X  X  O

O  -  -


Enter X's position from [1 till 9]: 2
O  X  X

X  X  O

O  O  -
```



```
Enter X's position from [1 till 9]: 9
O  X  X

X  X  O

O  O  X


Draw!!!
```

Result and analysis:

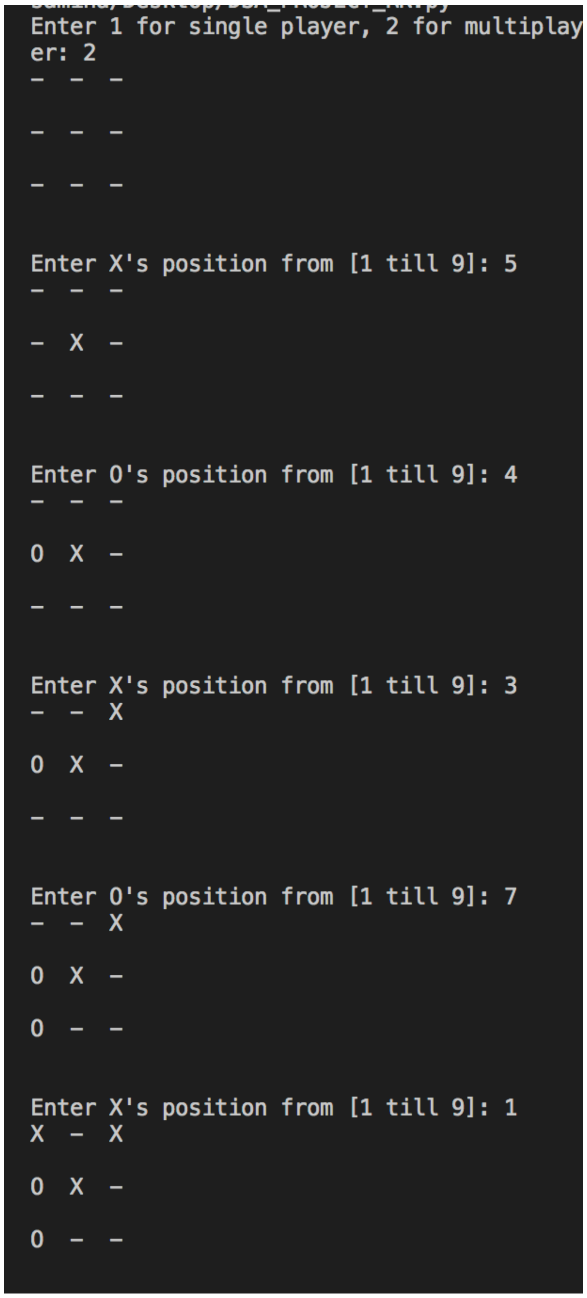
The algorithm truly is unbeatable, this trick couldn’t beat it uses the algorithm to always make the best possible combinations no matter what.

Trying same experiment in 2 player mode.

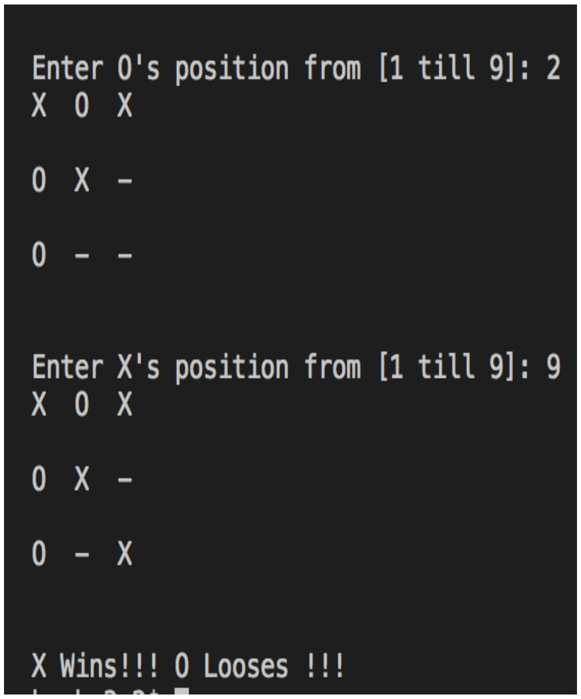
Case 3:

We follow the same steps as mentioned above:

Snippets of game terminal:



```
Enter 1 for single player, 2 for multiplay
er: 2
-  -  -

-  -  -

-  -  -


Enter X's position from [1 till 9]: 5
-  -  -

-  X  -

-  -  -


Enter O's position from [1 till 9]: 4
-  -  -

O  X  -

-  -  -


Enter X's position from [1 till 9]: 3
-  -  X

O  X  -

-  -  -


Enter O's position from [1 till 9]: 7
-  -  X

O  X  -

O  -  -


Enter X's position from [1 till 9]: 1
X  -  X

O  X  -

O  -  -
```



```
Enter O's position from [1 till 9]: 2
X  O  X

O  X  -

O  -  -


Enter X's position from [1 till 9]: 9
X  O  X

O  X  -

O  -  X


X Wins!!! O Looses !!!
```

Result and analysis:

I was able to successfully two winning combinations at the exact same time rendering the other player helpless in beating me using this trick as show above.

Computer move function algorithm:

Our computer move function calls minmax function and uses minmax algorithm to makes its move against the player using the algorithm written below:

```
Function computer_move(board);
    Fianl_position = -1
    Value_moved = -2
    For each position in board from (0 to 9):
        If board[position] == 0;
            Board[position] = 1
            Final score = -minmax (board, -1)
            Board[position] = 0
            If final score is greater than value moved;
                Value moved = final score
                Final_position = Board[position]
    Board[final_position] = 1
```

Procedures and Resources:

The project is a Tic Tac Toe game and for that we first needed the game board and for that we designed a board function where a simple game board was generated.

Soon after completion of the board we needed two players and, in our code,, we have both functionalities such as one player option in which the second player will be computer itself as well as we have two player options where two different players are able to play the game.

The function named as ***player1*** is formulated to take inputs from player1 and if the two-player option is chosen then the function named as ***player2*** is defined to take inputs from the second player. However, if there is a single player game chosen then function named as ***computer move*** is used to take computers input randomly.

After these functions we have designed a ***Minmax*** function for the computer role to play its part, in this function Minmax algorithm is used to play the game.

We have designed functions named as ***combinations*** in which results or combinations are compared to the players. This function holds the winning pairs or condition or draw conditions and returns the result based on input given by players if player falls in winning pairs then player wins and vice versa for player2 and computer.

The final function which is named ***Main***, in this function we are calling all the above functions which all together will generate results and return them on the screen.

Our ***resources*** in this game formulation have been several such as Multidimensional arrays, use of stacks, dictionaries, strings and integers as well as very important we are using tree traversal technique with help of minmax algorithm.

Conclusion

In conclusion we can say that this project is based on learning process and have been able to achieve its task of making fun and interactive python coded Tic Tac Toe game with help of several resources such as arrays, stacks, dictionaries and algorithms like Minmax and Tree traversal techniques. We can conclude that this project successfully generates the game which can be played by either single player or multiplayer on screen.

Shortcomings

The project was done potentially with some analysis that can help in further improving this game formation. According to our Analysis the use of Minmax algorithm is very helpful for games like Tic Tac Toe because it has very small number of choices and options. It has a small board which limits the places and minmax algorithm is perfect fit for it.

However, we will face difficulty in handling games which are very complex and have various numbers of choices and options to consider. If we take example of games like chess which is bit complex as compared to Tic Tac Toe then there will be issues faced if Minmax algorithm is used however there is alternative for everything and we can say that Alpha - Beta pruning can be beneficial such games including Tic Tac Toe.

Contributions:

Lama Imam:

I designed Minmax algorithm implementation part of the code using DFS methods to traverse the game tree of the game. My implementation was based on the pseudocode that I wrote and algorithm mentioned in this report as well as the function for the computer's turn implementing the algorithm. In the presentation I made the slides for Minmax algorithm introduction and analysis and defined that in more detail in the subsequent report, along with a dry run of the algorithm as well as the 3 experiments in experimental analysis portion. I edited and compiled the final version of this report and the presentation as well. I calculated runtimes for our computer move function as well write graphical analysis.

Hamna Jamil:

I designed the function that generated and stored all winning combinations that could exist on the board. My function compared winning combinations with user inputs and returned if the game was a draw or which player won. I also generated the main function of our code which called all helper functions previously generated to implement the game of Tic Tac Toe in its entirety. The Code explanation and breakdown part in this report have been contributed by me as well as the comparison of our code with an already existing code for a game of tic tac toe that implements minmax algorithm in the experimental analysis section. I also added the derivation of the time complexity for min max algorithm. I plotted graphs for runtimes calculated as well.

Rohit Kumar:

I initialised the game board in the first part of our code and designed functions for Player1 and Player2 inputs depending on which game mode the user chooses. I used the implementation of multi-dimensional arrays to store user inputs to be compared with any winning combinations later in the code. I also contributed the conclusion, procedure and resources part of this report and the same part in the presentation as well. I also compared the functions we generated in our code with the minmax algorithm pseudo code to present clearly which function implements which part of the algorithm.