UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Department of Electrical, Electronic and Computer Engineering

EAI 320 - Intelligent systems

EAI 320 - Practical Assignment 4 Report

Author:
Mohamed Ameen Omar

Student number:
u16055323

March 23, 2018

Contents

List of Figures

1 Introduction

During the fourth practical of EAI 320, students were required to solve an *Adversarial search problem*. An *Adversarial search problem* is a search problem wherein the agent is found to be in a competitive environment where the goals of each agent are in conflict, this type of problem is also known as *games* [1].

In AI the most common games are found to be, what is termed as *zero-sum games*. This means that these types of problems consist of agents having a fully observable environment in which each agent act alternatively and the utility value for either agent is equal and opposite to it's opponent.

The search tree for a basic zero-sum game is too large to search by conventional means and therefore Computer scientists have devised numerous search strategies to solve *zero-sum games* specifically. One search strategy is the Min-Max tree with alpha-beta pruning search.

This type of tree makes use of nodes called MIN and MAX nodes. The MIN nodes are states wherein the opponent seeks make an action to minimize our agent's utility value(thereby maximizing it's own utility value) and MAX nodes are states wherein our agent seeks to maximize it's utility value. A higher utility value associated with a particular agents translates to an action that would be most beneficial to that agent.

The problem posed to students was of the well known *Tic-Tac-Toe* game. Students were tasked to implement the game itself, the Min-Max search tree as well as the Alpha-Beta search for the tree.

2 Problem Definition

Students were required to implement the popular game called Tic-Tac-Toe. Tic-Tac-Toe, also known as noughts and crosses, is a *zero-sum, two player game*, where each player is assigned either an "X" or an "O". Each player places it's "X" or "O" alternatively on a 3x3 grid.

The game can either end in a win, loss or draw. The outcome is determined by whichever player succeeds in placing three of it's marks in a horizontal, vertical or diagonal row first. If all cells in the grid have been filled and no player has won, the game ends in a draw. For this game there are approximately 362880 possible games and 19683 possible board layouts or unique states.

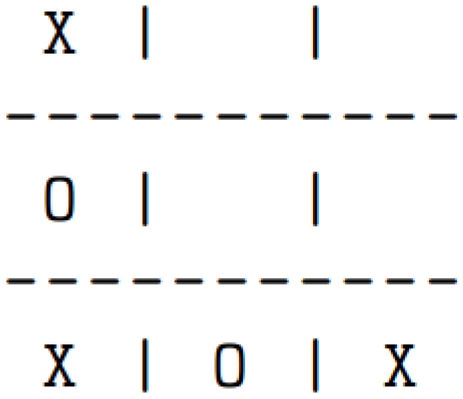
2.1 Tic-Tac-Toe Game

The game was to be implemented in it's own game class which included all member functions and member variables needed for the functionality of the game. The program needed to print out the current state of the game (The 3x3 grid), allow the user to input the position wherein it wanted to play and specified the action of the AI who would be the opponent of the user. The game outputted whether the user won, the AI won or the game ended in a draw. The Utility value associated with the outcome of a game is given as follows: 1 for a win, -1 for a loss and 0 for a draw.

2.2 Alpha-Beta search

In order to allow the AI to choose an "intelligent" action for a specific state, a Alpha-Beta Search needed to be implemented. The alpha-beta search needed to be flexible enough to be applied to any generic game that it was run on. The Min-Max tree needed to be computed for a specific state in order for the search to be applied. The search relied on the Utility value of a specific action and given state. The Utility value is a pay-off function which represents a numeric value associated with the outcome for a game.
The Alpha-Beta search with pruning traverses the Min-Max tree and associates the Utility value with a given state. The search returns the action that maximizes the utility for a given state. The pruning aspect comes in where not every node in the tree is evaluated. The search disregards nodes wherein a better Utility is found by another path along a node based on whether the node is a Min or Max node. This search was to be used as guidance for the AI when playing against the user.

Students were also given the game state shown in 1. The Alpha-Beta search was to be conducted on the state and return the best possible action for Player O in order to maximize it's chances of a positive (Win) outcome.



```
 X | |
-----------
 O | |
-----------
 X | O | X
```

Figure 1: Game state given in practical specification

3 Implementation and Methodology

3.1 Game Class

The Tic-Tac-Toe game class included member functions such as:

1. playGame() - To start or continue a game.
2. checkGameState() - To evaluate whether the current state is a Win, Loss, Draw or Unfinished after a move has been made.
3. printBoard() - To print the current state of the game to the console.
4. makeMove() - to make a move specified by the user.
5. getAIUtility() - return the Utility value associated with the current game state.
6. getActions() - returns a list with all possible valid moves that can be made by the current player.

The constructor of the class could either take in another game object, another game state or no argument. The constructor would then based off of the inclusion of these argument appropriately construct the new game object.

If no argument is passed in, the constructor would prompt the user whether it wanted to play against AI or not. In the case of the user specifying it wanted to play against the AI, it would also be asked if it wanted to play first or let the AI play first. Thereafter based on the input from the user the game will start executing the *playGame()* function. The AI makes a move using the *makeAIMove()* function. The function builds the Min-Max tree based off of the current game state and using the Alpha-beta search, it finds the optimal action for the AI to make. The functionality of the user being able to play against the AI was required in Question 3 of the practical specification. The output of the user playing against the AI can be seen in figures 2 to 7.

If the user opts not to play against the AI, the program prompts the user for the names of the players. Player 1 is always assigned the mark *"X"* and player 2 is always assigned the mark *"O"*. If the users do not enter a name for either player at the prompt, the program defaults to assigning each player a name based off of their mark. The console output for a user opting not to play against the AI can be seen in figure 13.

Although not required to complete the practical this functionality of a "PVP" mode was included to increase the robustness of the game class and make it as generic as possible. Input validation was also included in order to ensure that valid moves, valid names and valid options were inputted by the user to ensure proper functionality and execution of the program. The input validation output can be seen in figure 8 and figure 11.
The program source code for the game class can be found in Appendix A, figures 19 to 27.

3.2 Min-Max Tree

Node Class

A node class was created in order to aid in the construction of the Min-Max tree for a particular game state. The constructor takes in a game class and stores a deep copy of the game state. It includes a list of references to the children of the current node, which is the possible actions that can be taken from the current game state. A node object also includes a reference to the parent of the node in order to allow for backtracking up the tree once the tree has been constructed. The node class contains on a single member function called *addChild()*, which as the name suggests takes in a node and add the node as a child to the current node. The source code can be found in Appendix A, figure 28.

3.3 The AlphaBeta Tree class

The *AlphaBetaTree* class represents the Min-Max tree for a game state of interest. The class makes use of the *node* class to represent a game state in the tree. The root node represents the current game state for which we are looking to determining the appropriate action to make. The *buildTree()* is a recursive function that begins with the root of the tree and adds the children (possible moves from a game state) recursively. In order to determine the children of a node, the *getActions()* member function of the game class is used to retrieve all possible valid actions that can be take from a game state. The *buildTree()* member function checks during the tree creation whether the current node is a Min or Max node and based off of this, it assigns a utility value to the node using the *getAIUtility()* member function of the game class. The details of the utility value and what it represents can be found in the Problem Definition section. The recursive *buildTree()* member function terminates once a terminal node (a node wherein the game has ended) has been reached. The source code can be found in Appendix A, figures 29 to 31.

3.4 Alpha-Beta Search

The Alpha-Beta search is conducted within the *AlphaBetaTree* class as a member function called *AlphaBetaSearch()*. The function makes use of two other member functions called *AlphaBetaMax()* and *AlphaBetaMin()*.

The *AlphaBetaMax()* member function searches the tree recursively in order to find the action with the best or highest utility value. A high utility value means the best move from the point of view of our agent.

The *AlphaBetaMin()* member function searches the tree recursively in order to find the action for a Min node that would maximize it's utility value and in effect minimize our agent's utility value.

The Alpha-Beta search makes use of two variables, the *alpha* variable and the *beta* variable. The alpha value represents the highest utility value from the point of view of our agent - thus representing the best action for our agent, and the beta value is the lowest utility value from the point of view of our agent and represents an action that would to a favorable or the most favorable result for our opponent (the Min nodes).

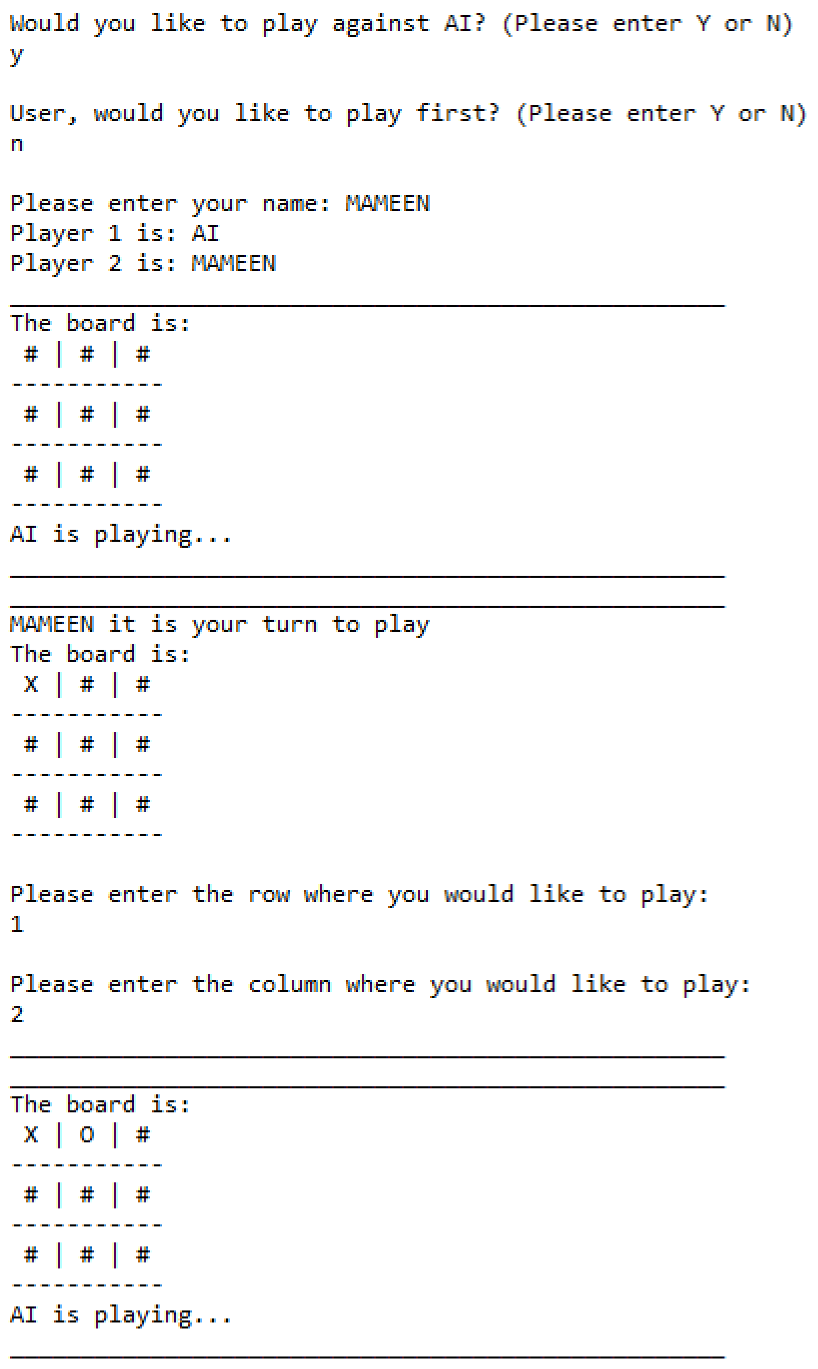
The search makes use of a technique called *Alpha-Bet pruning*, which in effect ignores nodes wherein we know that we have already found a node that has a higher alpha value or a lower beta value and thus, that node's subtree need not be expanded or searched. At the end, the function returns the action or rather the new game state that would result after our agent has made the *"best"* action or move.

The *AlphaBetaSearch()* source code was amended for Question 2 in order to allow for each node being evaluated and the terminal nodes with associated terminal Utility values to be print to the console. The source code can be found in Appendix A, figures 30 and 32. The search looks to maximize the worst-case scenario by assuming that each player or agent will from the current game state onwards will only take actions that would maximize the outcome from it's point of view [1].

The *deepcopy()* function from the *copy* python library was used in order to make a deep copy of game states that need to be duplicated.

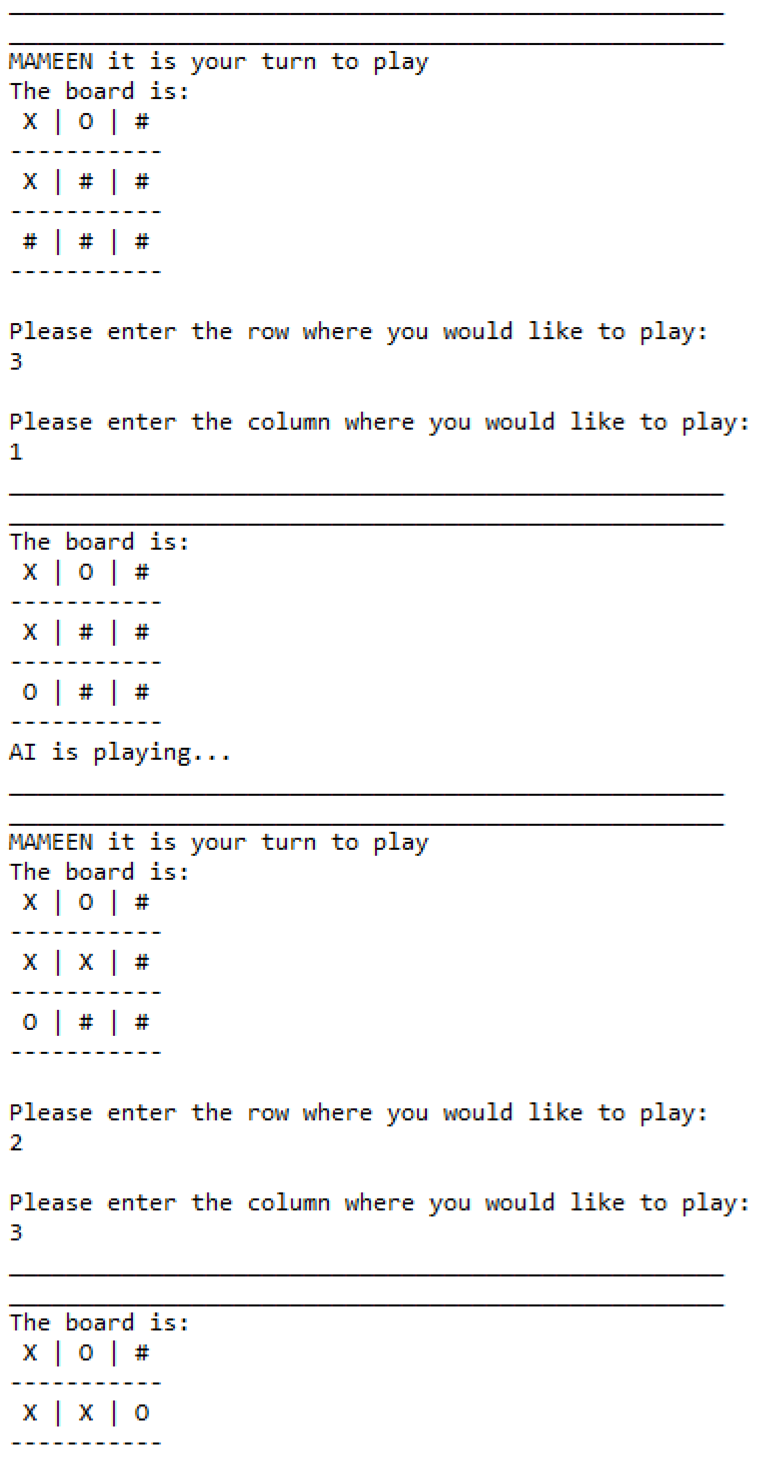
4 Results

Figures 2 to 4 shows the console output for a game wherein the user plays against the AI and the AI plays first.



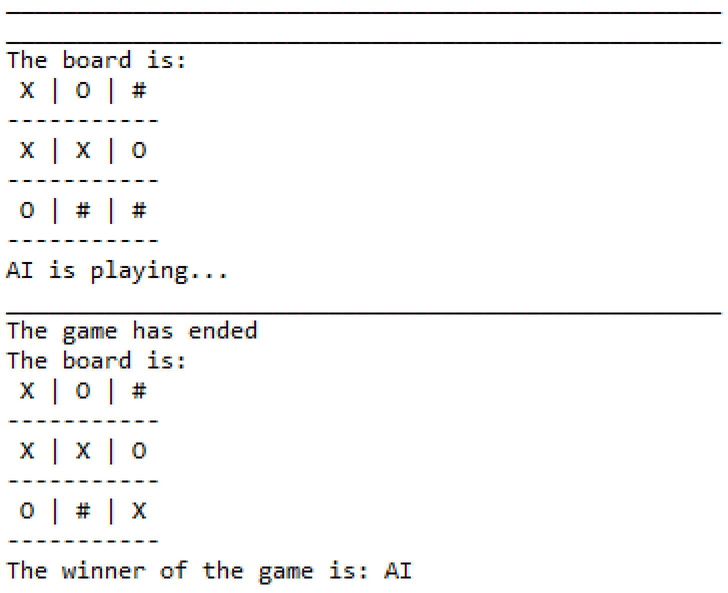
```
Would you like to play against AI? (Please enter Y or N)
y

User, would you like to play first? (Please enter Y or N)
n

Please enter your name: MAMEEN
Player 1 is: AI
Player 2 is: MAMEEN
____________________________________________________
The board is:
 # | # | #
-----------
 # | # | #
-----------
 # | # | #
-----------
AI is playing...
____________________________________________________
____________________________________________________
MAMEEN it is your turn to play
The board is:
 X | # | #
-----------
 # | # | #
-----------
 # | # | #
-----------

Please enter the row where you would like to play:
1

Please enter the column where you would like to play:
2
____________________________________________________
____________________________________________________
The board is:
 X | O | #
-----------
 # | # | #
-----------
 # | # | #
-----------
AI is playing...
____________________________________________________
```

Figure 2: Console output of playing second against AI - 1 of 3



```
___________________________________________________
___________________________________________________
MAMEEN it is your turn to play
The board is:
 X | O | #
-----------
 X | # | #
-----------
 # | # | #
-----------

Please enter the row where you would like to play:
3

Please enter the column where you would like to play:
1
___________________________________________________
___________________________________________________
The board is:
 X | O | #
-----------
 X | # | #
-----------
 O | # | #
-----------
AI is playing...
___________________________________________________
___________________________________________________
MAMEEN it is your turn to play
The board is:
 X | O | #
-----------
 X | X | #
-----------
 O | # | #
-----------

Please enter the row where you would like to play:
2

Please enter the column where you would like to play:
3
___________________________________________________
___________________________________________________
The board is:
 X | O | #
-----------
 X | X | O
-----------
```

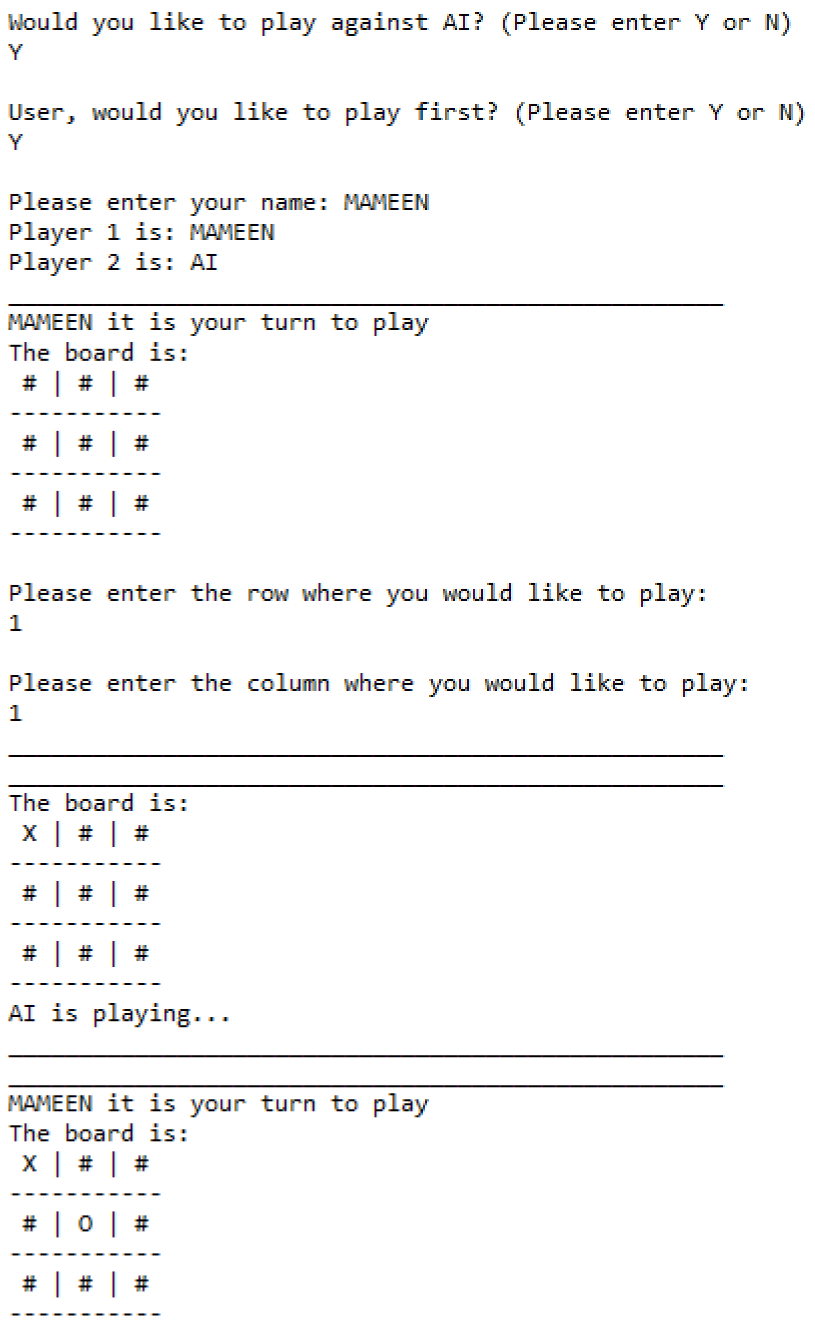
Figure 3: Console output of playing second against AI - 2 of 3



```
The board is:
 X | O | #
-----------
 X | X | O
-----------
 O | # | #
-----------
AI is playing...

The game has ended
The board is:
 X | O | #
-----------
 X | X | O
-----------
 O | # | X
-----------
The winner of the game is: AI
```

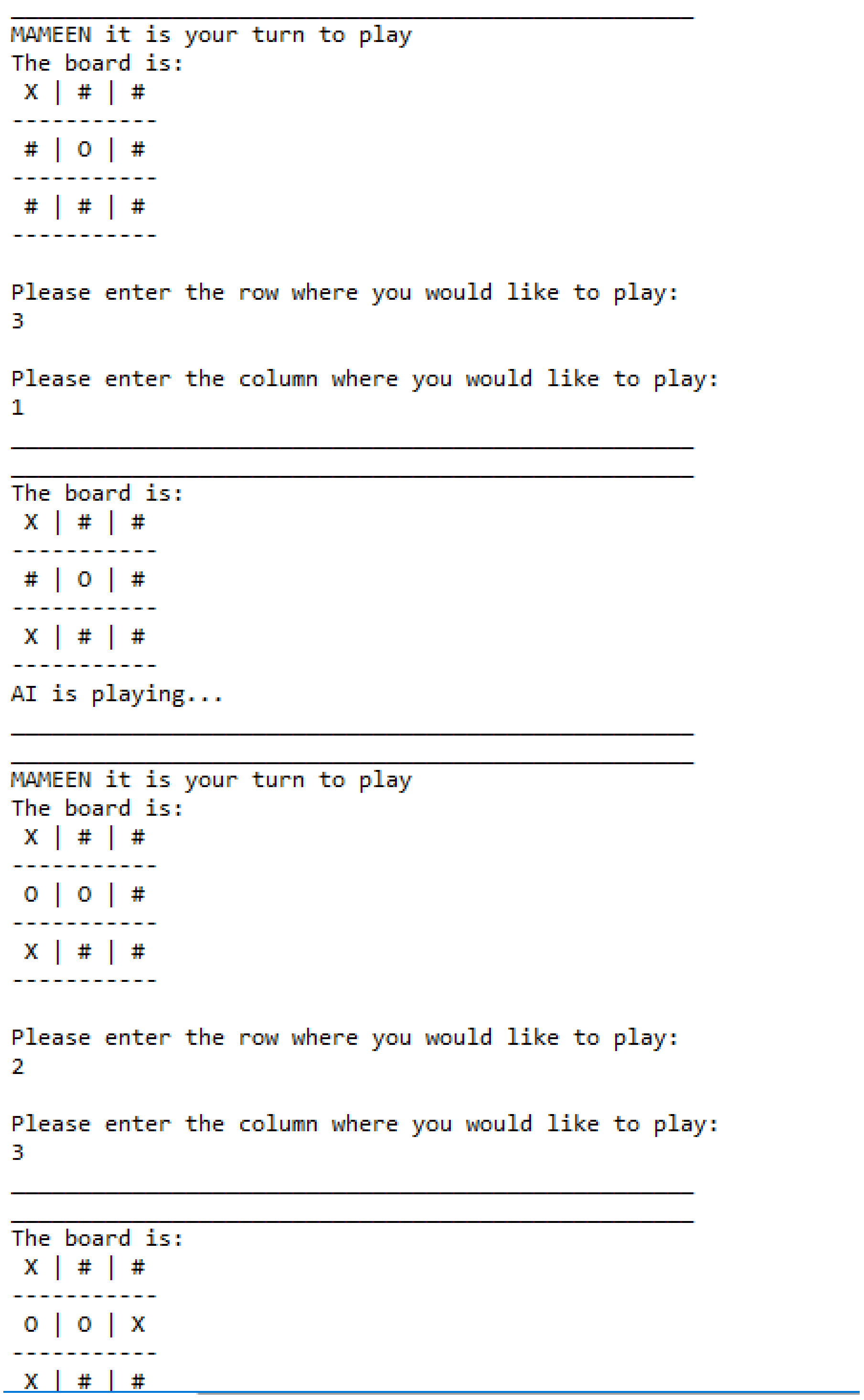
Figure 4: Console output of playing second against AI - 3 of 3

Figures 5 to 7 shows the console output for a game wherein the user plays against the AI and the AI plays second.



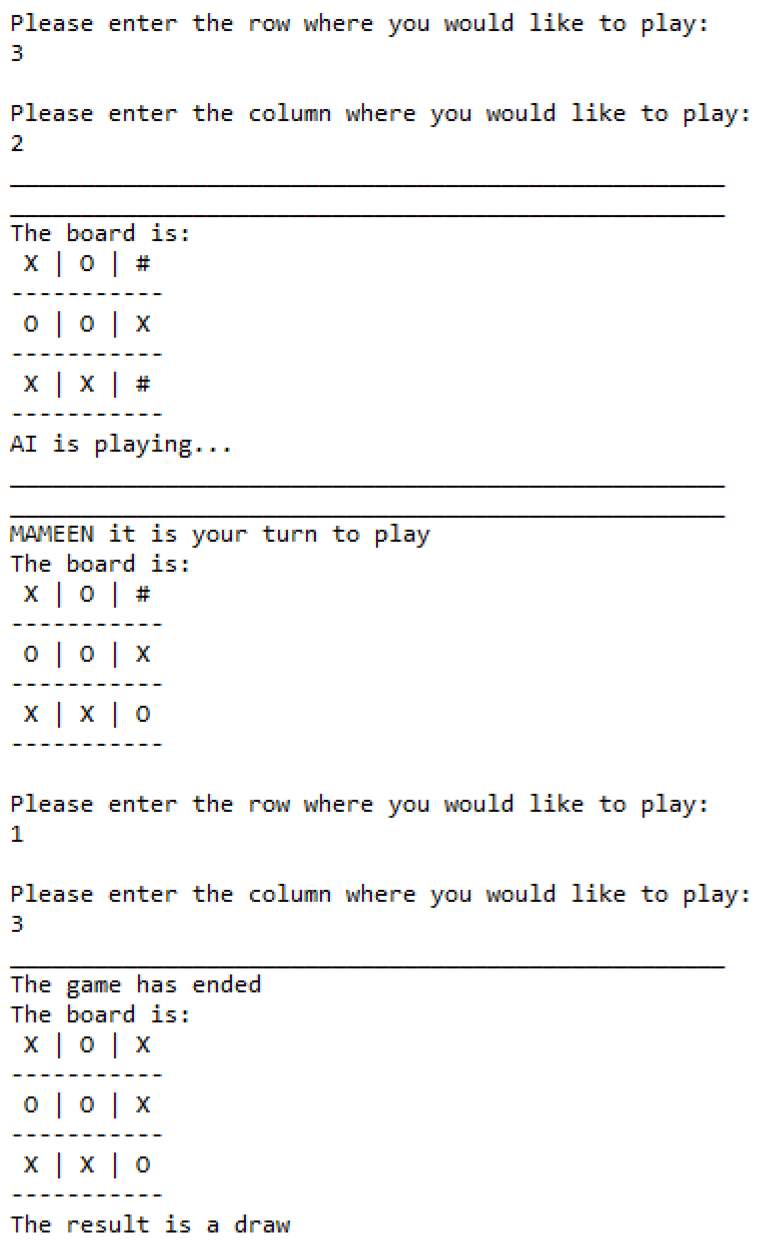
```
Would you like to play against AI? (Please enter Y or N)
Y

User, would you like to play first? (Please enter Y or N)
Y

Please enter your name: MAMEEN
Player 1 is: MAMEEN
Player 2 is: AI
____________________________________________________
MAMEEN it is your turn to play
The board is:
 # | # | #
-----------
 # | # | #
-----------
 # | # | #
-----------

Please enter the row where you would like to play:
1

Please enter the column where you would like to play:
1
____________________________________________________
____________________________________________________
The board is:
 X | # | #
-----------
 # | # | #
-----------
 # | # | #
-----------
AI is playing...
____________________________________________________
____________________________________________________
MAMEEN it is your turn to play
The board is:
 X | # | #
-----------
 # | O | #
-----------
 # | # | #
-----------
```

Figure 5: Console output of playing first against AI - 1 of 3



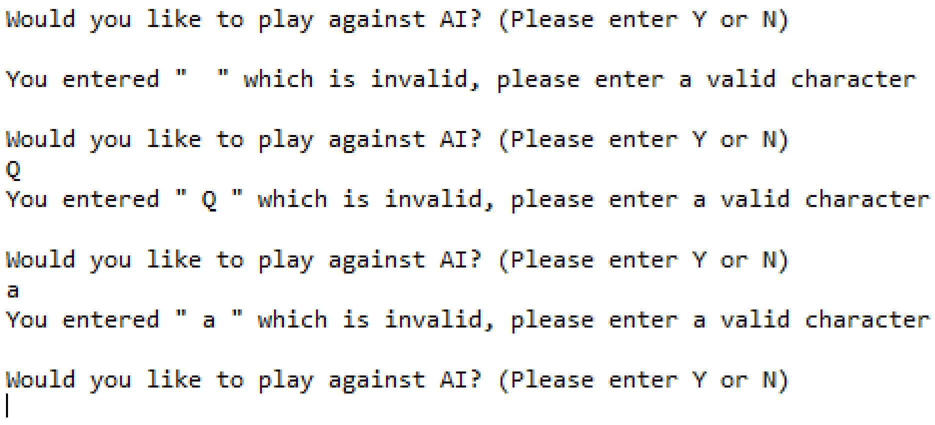
```
______________________________________________________
MAMEEN it is your turn to play
The board is:
 X | # | #
-----------
 # | O | #
-----------
 # | # | #
-----------

Please enter the row where you would like to play:
3

Please enter the column where you would like to play:
1
______________________________________________________
______________________________________________________
The board is:
 X | # | #
-----------
 # | O | #
-----------
 X | # | #
-----------
AI is playing...
______________________________________________________
______________________________________________________
MAMEEN it is your turn to play
The board is:
 X | # | #
-----------
 O | O | #
-----------
 X | # | #
-----------

Please enter the row where you would like to play:
2

Please enter the column where you would like to play:
3
______________________________________________________
______________________________________________________
The board is:
 X | # | #
-----------
 O | O | X
-----------
 X | # | #
```

Figure 6: Console output of playing first against AI - 2 of 3



```
Please enter the row where you would like to play:
3

Please enter the column where you would like to play:
2
___________________________________________________
___________________________________________________
The board is:
 X | O | #
-----------
 O | O | X
-----------
 X | X | #
-----------
AI is playing...
___________________________________________________
___________________________________________________
MAMEEN it is your turn to play
The board is:
 X | O | #
-----------
 O | O | X
-----------
 X | X | O
-----------

Please enter the row where you would like to play:
1

Please enter the column where you would like to play:
3
___________________________________________________
The game has ended
The board is:
 X | O | X
-----------
 O | O | X
-----------
 X | X | O
-----------
The result is a draw
```

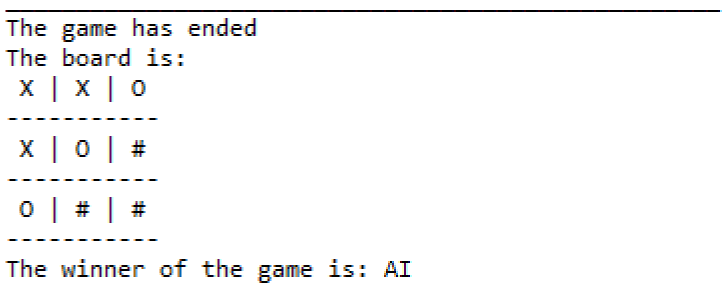
Figure 7: Console output of playing first against AI - 3 of 3



```
Would you like to play against AI? (Please enter Y or N)

You entered "  " which is invalid, please enter a valid character

Would you like to play against AI? (Please enter Y or N)
Q
You entered " Q " which is invalid, please enter a valid character

Would you like to play against AI? (Please enter Y or N)
a
You entered " a " which is invalid, please enter a valid character

Would you like to play against AI? (Please enter Y or N)
|
```

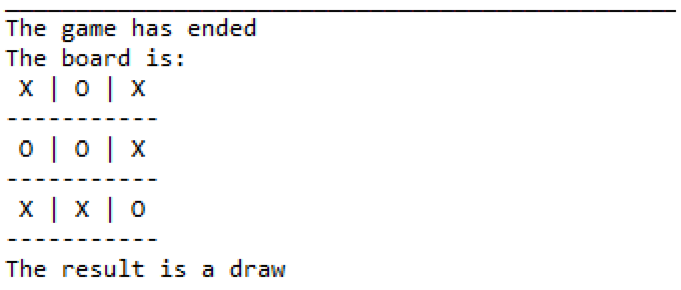
Figure 8: Console output showing input validation for AI prompt

Figure 9 and 10 show example outcomes of games between the user and the AI.



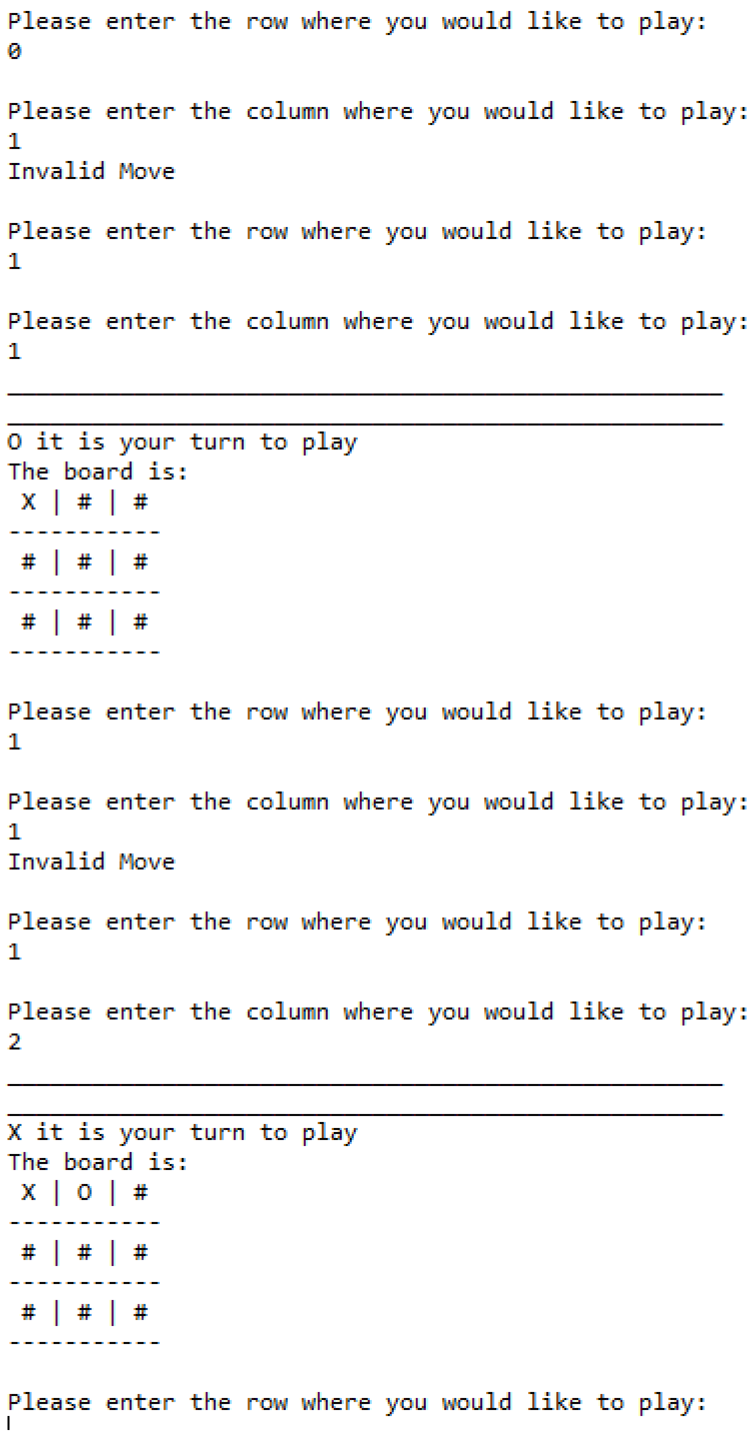
```
______________________________________________________
The game has ended
The board is:
 X | X | O
-----------
 X | O | #
-----------
 O | # | #
-----------
The winner of the game is: AI
```

Figure 9: Console output of a game output of a win for AI



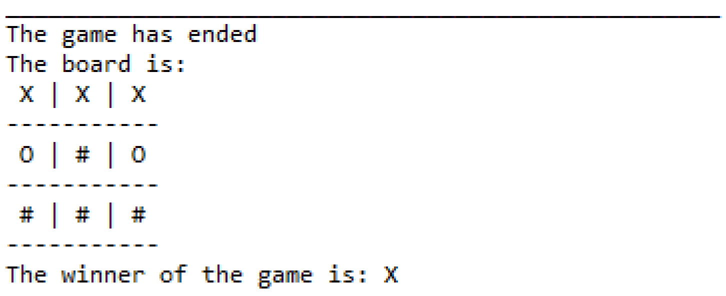
```
______________________________________________________
The game has ended
The board is:
 X | O | X
-----------
 O | O | X
-----------
 X | X | O
-----------
The result is a draw
```

Figure 10: Console output of a game outcome of a draw with AI



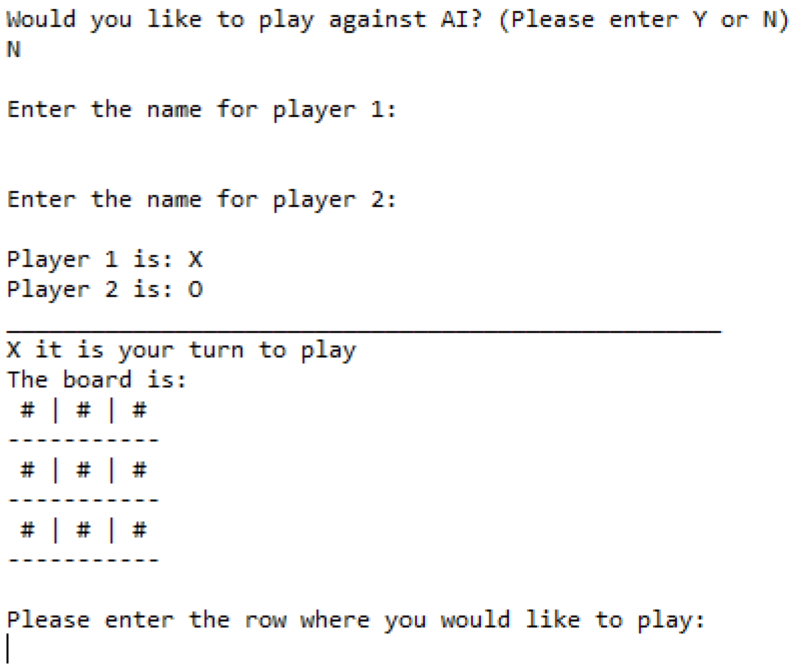
```
Please enter the row where you would like to play:
0

Please enter the column where you would like to play:
1
Invalid Move

Please enter the row where you would like to play:
1

Please enter the column where you would like to play:
1
____________________________________________________
____________________________________________________
O it is your turn to play
The board is:
 X | # | #
-----------
 # | # | #
-----------
 # | # | #
-----------

Please enter the row where you would like to play:
1

Please enter the column where you would like to play:
1
Invalid Move

Please enter the row where you would like to play:
1

Please enter the column where you would like to play:
2
____________________________________________________
____________________________________________________
X it is your turn to play
The board is:
 X | O | #
-----------
 # | # | #
-----------
 # | # | #
-----------

Please enter the row where you would like to play:
```

Figure 11: Input validation for a move



```
The game has ended
The board is:
 X | X | X
-----------
 O | # | O
-----------
 # | # | #
-----------
The winner of the game is: X
```

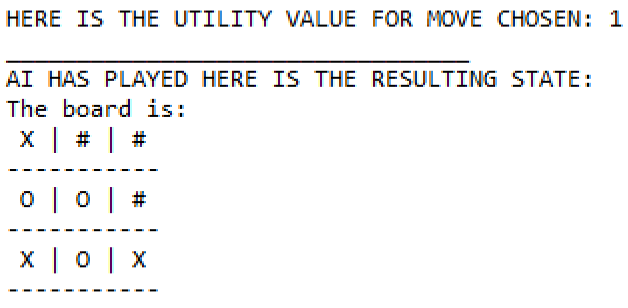
Figure 12: Console output of the user winning against another user



```
Would you like to play against AI? (Please enter Y or N)
N

Enter the name for player 1:


Enter the name for player 2:

Player 1 is: X
Player 2 is: O

X it is your turn to play
The board is:
 # | # | #
-----------
 # | # | #
-----------
 # | # | #
-----------

Please enter the row where you would like to play:
```

Figure 13: Console output showing the user prompt (opting not play AI)

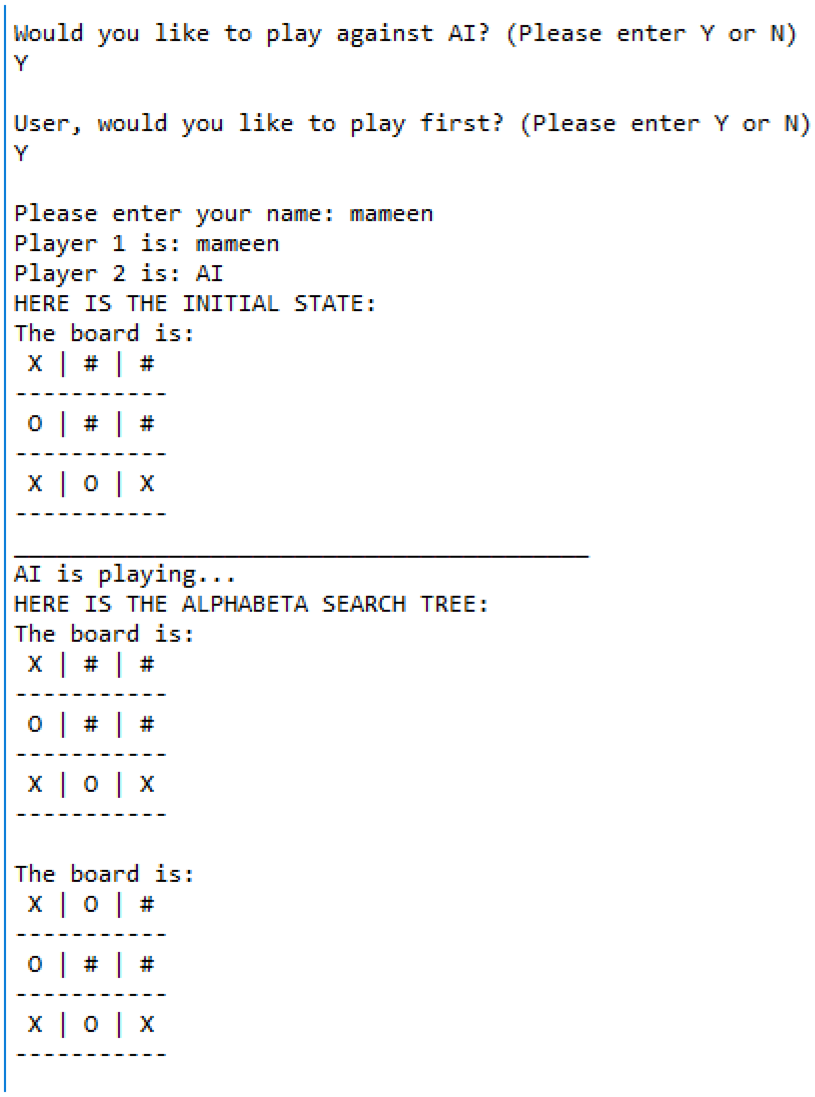
The figures below show the action chosen by the AI for the game state given in figure 1. The figures that follow (figures 15 to 18) shows the output of the *Alpha-Beta search* function as it searches the Min-Max tree in a quest to find the optimal move for our AI in Question 2.



```
HERE IS THE UTILITY VALUE FOR MOVE CHOSEN: 1
_________________________________
AI HAS PLAYED HERE IS THE RESULTING STATE:
The board is:
 X | # | #
-----------
 O | O | #
-----------
 X | O | X
-----------
```

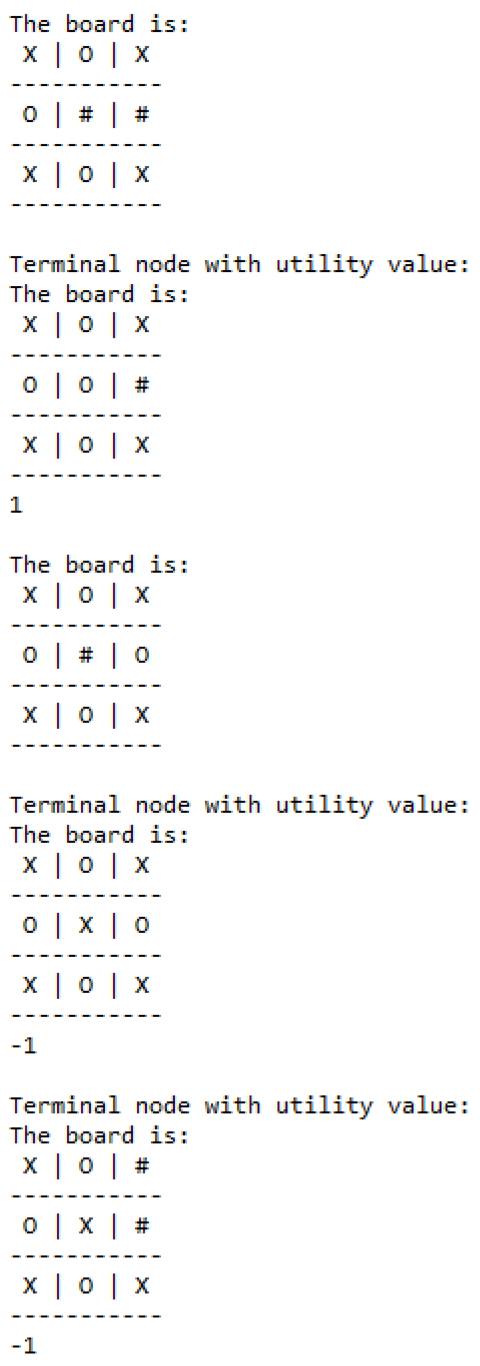
1

Figure 14: Console output of the action chosen by the AI for Question 2 for the game state in figure 1



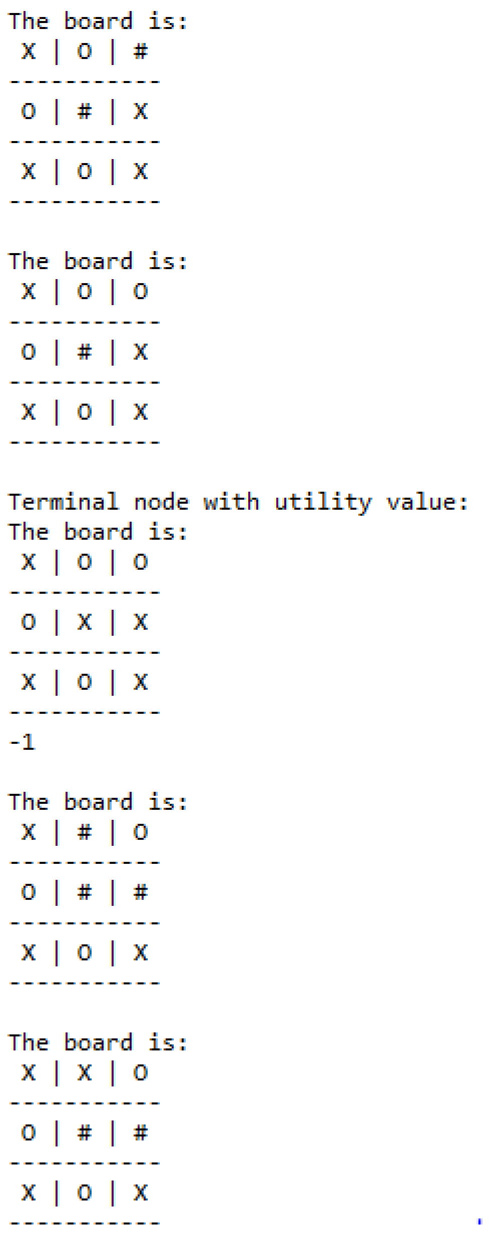
```
Would you like to play against AI? (Please enter Y or N)
Y

User, would you like to play first? (Please enter Y or N)
Y

Please enter your name: mameen
Player 1 is: mameen
Player 2 is: AI
HERE IS THE INITIAL STATE:
The board is:
 X | # | #
-----------
 O | # | #
-----------
 X | O | X
-----------
_________________________________________
AI is playing...
HERE IS THE ALPHABETA SEARCH TREE:
The board is:
 X | # | #
-----------
 O | # | #
-----------
 X | O | X
-----------

The board is:
 X | O | #
-----------
 O | # | #
-----------
 X | O | X
-----------
```

Figure 15: Console output showing the Alpha-Beta Tree and terminal Utilities - 1 of 4



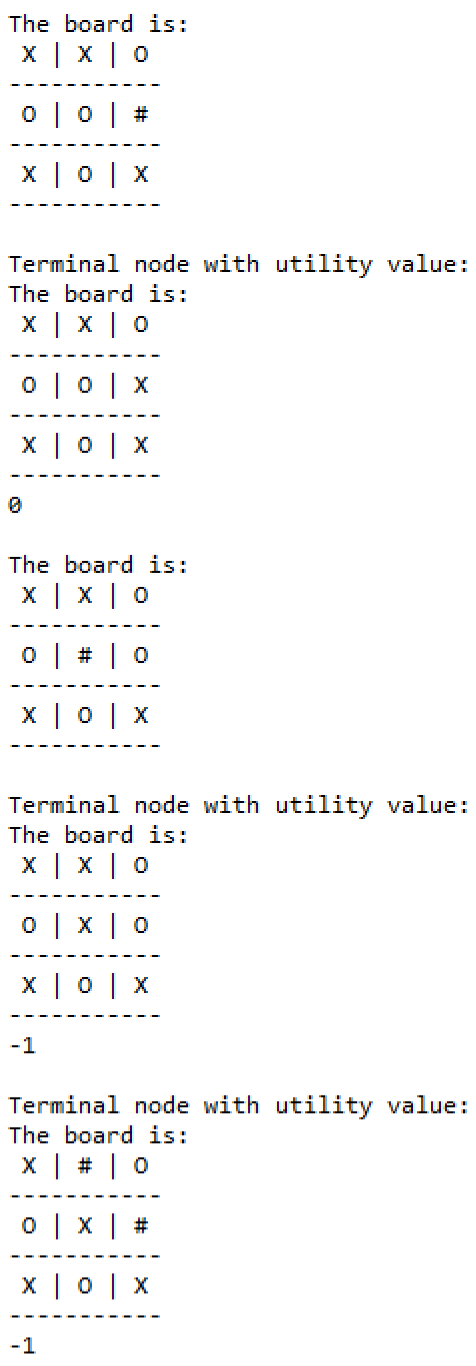
```
The board is:
 X | O | X
-----------
 O | # | #
-----------
 X | O | X
-----------

Terminal node with utility value:
The board is:
 X | O | X
-----------
 O | O | #
-----------
 X | O | X
-----------
1

The board is:
 X | O | X
-----------
 O | # | O
-----------
 X | O | X
-----------

Terminal node with utility value:
The board is:
 X | O | X
-----------
 O | X | O
-----------
 X | O | X
-----------
-1

Terminal node with utility value:
The board is:
 X | O | #
-----------
 O | X | #
-----------
 X | O | X
-----------
-1
```

Figure 16: Console output showing the Alpha-Beta Tree and terminal Utilities - 2 of 4



```
The board is:
 X | O | #
-----------
 O | # | X
-----------
 X | O | X
-----------

The board is:
 X | O | O
-----------
 O | # | X
-----------
 X | O | X
-----------

Terminal node with utility value:
The board is:
 X | O | O
-----------
 O | X | X
-----------
 X | O | X
-----------
-1

The board is:
 X | # | O
-----------
 O | # | #
-----------
 X | O | X
-----------

The board is:
 X | X | O
-----------
 O | # | #
-----------
 X | O | X
-----------
```

Figure 17: Console output showing the Alpha-Beta Tree and terminal Utilities - 3 of 4



```
The board is:
 X | X | O
-----------
 O | O | #
-----------
 X | O | X
-----------

Terminal node with utility value:
The board is:
 X | X | O
-----------
 O | O | X
-----------
 X | O | X
-----------
0

The board is:
 X | X | O
-----------
 O | # | O
-----------
 X | O | X
-----------

Terminal node with utility value:
The board is:
 X | X | O
-----------
 O | X | O
-----------
 X | O | X
-----------
-1

Terminal node with utility value:
The board is:
 X | # | O
-----------
 O | X | #
-----------
 X | O | X
-----------
-1
```

Figure 18: Console output showing the Alpha-Beta Tree and terminal Utilities - 4 of 4

5 Discussion

Figure 1 shows the game state for which we are required to make use of our Alpha-Beta search to compute the next action for player "O". Figure 14 shows the resulting game state after player "O" has compute and made it's move using the implemented Alpha-Beta search. The agent has opted to play at row 2 column 2. This is the correct and best decision for the AI to make, since by making this move, player "O" has stopped player "X" from being able to win the game, since if player "X" played at that position next, he would win with three of his marks in a diagonal row. The AI is *"intelligent"* enough to recognize when the game is in danger of being lost and makes the decision to choose an action that would maximize it's expected outcome, if that outcome is guaranteed to be negative it would delay the outcome as much as possible.

Figures 15 to 18 show the output of the *Alpha-Beta search* function as it searches the Min-Max tree in a quest to find the optimal move for our AI in Question 2. As we can see, not every node in the tree is evaluated and once a node has been found have a better or worse Utility (depending on whether the node is a Min or Max node) the rest of the nodes in the sub-tree are ignored and the search continues to other sub-trees.

The user is able to either play another user, or the AI. The AI makes use of the Alpha-Beta search to compute it's action. Depending on the game state, the time taken for the AI to compute it's action varies dramatically, with some states have an action being made almost instantly whereas others take a noticeable amount of time to be computed. This is due to the size of the search tree that the AI needs to search as the computing power needed to search the tree.

During testing of the AI game-play, I was unable to beat the AI with the best outcome being a draw. No matter what moves I made, the AI would always be able to recover from every disadvantage it had against me and salvage the game to a draw, or turn the game completely and win the game. The Alpha-Beta search is thus extremely successful in returning the action that would result in the best outcome for the agent. It is an extremely appropriate search strategy to be used by agent's in competitive environments or agent's involved in solving a *zero-sum game.*

6 Conclusion

In competitive environments, agent's cannot make use of general search trees or general search strategies in order to find the optimal solution. This is due to the vast amount of memory and processing power that would be needed if these strategies were used. Adversarial search strategies focus on solving *zero-sum game* problems in a competitive environment. One such search strategy is the alpha-beta search strategy conducted on a Min-Max tree.

The classic Tic-Tac-Toe game is a good example of a *zero-sum game* and I have successfully implemented a program to play this game. Extra functionality such as a "PVP" mode so that two users can play against each other and input, move and choice selection, validation was included.

The Min-Max tree was successfully created and the implemented Alpha-Beta pruning search strategy was conducted on the tree in order to find the best possible action for the AI to take in Question 2.

A user is able to play the AI and in most instances the AI is able to make the best move possible in a short period of time. However, it was noticed that if the AI plays first, due to the size of the Min-Max search tree, it takes a noticeable amount of time for the AI to compute it's move.

The Alpha-Beta pruning is successful in eliminating unnecessary nodes from being evaluated, whoever it is not the most optimal search as in "real world problems" the agent would need to make a decision in an instant, whereas in the given problem the agent takes some time to compute it's next action.

7 Appendix A: Python Code

```python
import numpy as np
import copy
import queue
import sys
class game:
    #initializer
    def __init__(self, game = None, board = None):
        if(game == None):
            self.board = np.array([["#","#","#"],
                                   ["#","#","#"],
                                   ["#","#","#"]])
            self.againstAI = False
            self.state = "Unfinished"
            self.player1Name = ""
            self.player2Name = ""
            self.playAiCheck()
            self.setNames()
            self.turn = copy.deepcopy(self.player1Name)
            self.winner = ""
            if(board != None):
                self.board = copy.deepcopy(board)
        else:
            self.againstAI = copy.deepcopy(game.againstAI)
            self.board  = copy.deepcopy(game.board)
            self.state = copy.deepcopy(game.state)
            self.player1Name = copy.deepcopy(game.player1Name)
            self.player2Name = copy.deepcopy(game.player2Name)
            self.turn = copy.deepcopy(game.turn)
            self.winner = copy.deepcopy(game.winner)
```

Figure 19: Program Source Code 1 of 14 [Python]

```python
def playGame(self):
        if(self.againstAI == False):
            while(self.state == "Unfinished"):
                print("----------------------------------------------------")
                print(self.turn, "it is your turn to play")
                self.printBoard()
                move = self.getMoves()
                self.makeMove(move[0],move[1])
                print("----------------------------------------------------")

            print("The game has ended")
            self.printBoard()
            if(self.state == "Draw"):
                print("The result is a draw")
            else:
                print("The winner of the game is:", self.winner)
        else:
            while(self.state == "Unfinished"):
                sys.stdout.flush()
                print("----------------------------------------------------")
                if(self.turn == "AI"):
                    self.printBoard()
                    print("AI is playing...")
                    sys.stdout.flush()
                    self.makeAIMove()
                else:
                    print(self.turn, "it is your turn to play")
                    self.printBoard()
                    move = self.getMoves()
                    self.makeMove(move[0],move[1])
                print("----------------------------------------------------")
            print("The game has ended")
            self.printBoard()
            if(self.state == "Draw"):
                print("The result is a draw")
            else:
                print("The winner of the game is:", self.winner)
```

Figure 20: Program Source Code 2 of 14 [Python]

```python
#checks current state of the game
 def checkGameState(self):
     self.checkX()
     self.checkO()
     if(self.state != "Win"):
         for x in range(0,3):
             for y in range(0,3):
                 if(self.board[x][y] == "#"):
                     self.state = "Unfinished"
                     return
         self.state = "Draw"

 #checks if X won
 def checkX(self):
     #check up left right
     for row in range(0,3):
         if(self.board[row][0] == "X"):
             if(self.board[row][0] == self.board[row][1]):
                 if(self.board[row][1] == self.board[row][2]):
                     self.winner = self.player1Name
                     self.state = "Win"
                     return
     #check up down
     for col in range(0,3):
         if(self.board[0][col] == "X"):
             if(self.board[0][col] == self.board[1][col]):
                 if(self.board[1][col] == self.board[2][col]):
                     self.winner = self.player1Name
                     self.state = "Win"
                     return
```

Figure 21: Program Source Code 3 of 14 [Python]

```python
#check diagonals
        if(self.board[0][0] == "X"):
            if(self.board[0][0] == self.board[1][1]):
                if(self.board[1][1] == self.board[2][2]):
                    self.winner = self.player1Name
                    self.state = "Win"
                    return
        if(self.board[0][2] == "X"):
            if(self.board[0][2] == self.board[1][1]):
                if(self.board[1][1] == self.board[2][0]):
                    self.winner = self.player1Name
                    self.state = "Win"
                    return
    #checks if O won
    def checkO(self):
        #check up left right
        for row in range(0,3):
            if(self.board[row][0] == "O"):
                if(self.board[row][0] == self.board[row][1]):
                    if(self.board[row][1] == self.board[row][2]):
                        self.winner = self.player2Name
                        self.state = "Win"
                        return
        #check up down
        for col in range(0,3):
            if(self.board[0][col] == "O"):
                if(self.board[0][col] == self.board[1][col]):
                    if(self.board[1][col] == self.board[2][col]):
                        self.winner = self.player2Name
                        self.state = "Win"
                        return
```

Figure 22: Program Source Code 4 of 14 [Python]

```python
#check diagonals
        if(self.board[0][0] == "O"):
            if(self.board[0][0] == self.board[1][1]):
                if(self.board[1][1] == self.board[2][2]):
                    self.winner = self.player2Name
                    self.state = "Win"
                    return

        if(self.board[0][2] == "O"):
            if(self.board[0][2] == self.board[1][1]):
                if(self.board[1][1] == self.board[2][0]):
                    self.winner = self.player2Name
                    self.state = "Win"
                    return

    #switches turn of current player
    def switchTurn(self):
        if(self.turn == self.player1Name):
            self.turn = self.player2Name
        else:
            self.turn = self.player1Name

     #adapted from http://inventwithpython.com/chapter10.html
    def printBoard(self):
        print("The board is:")
        for x in range(0,3):
            print('', self.board[x][0],'|',self.board[x][1],'|',self.board[x][2])
            print('-------------')
```

Figure 23: Program Source Code 5 of 14 [Python]

```
#check to see if want to play AI
    def playAiCheck(self):
        againstAi = input("Would you like to play against AI? (Please
    enter Y or N)\n")
        while(againstAi.upper() != "Y" and againstAi.upper() != "N"):
            print("You entered \"",againstAi,"\" which is invalid, please
    enter a valid character")
            againstAi = input("Would you like to play against AI? (Please
    enter Y or N)\n")
        if(againstAi == "Y" or againstAi == "y"):
            self.againstAI = True
            first = input("User, would you like to play first? (Please
    enter Y or N)\n")
            while(first.upper() != "Y" and first.upper() != "N"):
                print("You entered \"",first,"\" which is invalid, please
    enter a valid character")
                first = input("User, would you like to play first? (Please
     enter Y or N)\n")
            if(first.upper() == "Y"):
                 self.player1Name = input("Please enter your name: ")
                 self.player2Name = "AI"
            else:
                self.player1Name = "AI"
                self.player2Name = input("Please enter your name: ")

    #set player names from input
    def setNames(self):
        if(self.againstAI == False):
            self.player1Name = input("Enter the name for player 1: \n")
            self.player2Name = input("Enter the name for player 2: \n")
        if(self.player1Name == ""):
            self.player1Name = "X"
        if (self.player2Name == ""):
            self.player2Name = "O"
```

Figure 24: Program Source Code 6 of 14 [Python]

```
#print Player names
    def printPlayers(self):
        print("Player 1 is:", self.player1Name)
        print("Player 2 is:", self.player2Name)

    #returns current state ie - win, lose, draw, unfinished
    def getState(self):
        print("The current state of the game is:", self.state)

    #makes a move to the specifed row and column
    def makeMove(self,row,col):
        while(row > 3 or col > 3 or row < 1 or col < 1 ):
            print("Invalid Move")
            move = self.getMoves()
            row = move[0]
            col = move[1]
        row = row-1
        col = col-1
        while(self.validateMove(row,col) == False):
            print("Invalid Move")
            move = self.getMoves()
            row = move[0]
            col = move[1]
            row = row-1
            col = col-1
        row = row
        col = col
        if(self.turn == self.player1Name):
            self.board[row][col] = "X"
        else:
            self.board[row][col] = "O"
        self.checkGameState()
        self.switchTurn()
```

Figure 25: Program Source Code 7 of 14 [Python]

```
#validates move to be within range and not occupied
    def validateMove(self,row,col):
        if(self.board[row][col] != "#"):
            return False
        return True
    #gets the row and column from the user
    def getMoves(self):
        row = input("Please enter the row where you would like to play:\n")
        col = input("Please enter the column where you would like to play:\n")
        move = [int(row),int(col)]
        return move

    #utility for the AI
    def getAIUtility(self):
        if(self.state == "Draw"):
            return 0
        if(self.winner == "AI"):
            return 1
        return -1

    def getUserUtility(self):
        if(self.state == "Draw"):
            return 0
        if(self.winner != "AI"):
            return 1
        return -1
```

Figure 26: Program Source Code 8 of 14 [Python]

```python
#returns list of possible game states
    def getActions(self):
        actions = []
        for x in range(0,3):
            for y in range(0,3):
                if(self.board[x][y] == "#"):
                   move = game(self)
                   move.makeMove(x+1,y+1)
                   actions.append(move)
        return actions

    #whether state is terminal or not
    def isTerminal(self):
        return(self.state != "Unfinished")

    def makeAIMove(self):
        if(self.turn == "AI"):
            ABT = AlphaBetaTree(self)
            temp = ABT.AlphaBetaSearch()
            for x in range(0,3):
                for y in range(0,3):
                    if(self.board[x][y] != temp.board[x][y]):
                        self.makeMove(x+1,y+1)
                        return
```

Figure 27: Program Source Code 9 of 14 [Python]

```python
class node:
    def __init__(self, game,parent = None):
        self.game = game
        self.children = []
        self.parent = parent
        self.utility = 0

    def addChild(self,child):
        child.parent = self
        self.children.append(child)
```

Figure 28: Program Source Code 10 of 14 [Python]

```python
class AlphaBetaTree:
    def __init__(self,game1):
        self.root = node(copy.deepcopy(game1))
        self.action = None
        self.buildTree(self.root)

    def buildTree(self,parent = None):
        if(parent == None):
            parent = self.root
        if(parent.game.isTerminal()):
            parent.utility = parent.game.getAIUtility()
            return
        actions = parent.game.getActions()
        for a in actions:
            child = node(a,parent)
            parent.addChild(child)
            self.buildTree(child)
            temp = 55
            for x in parent.children:
                if(parent.game.turn == self.root.game.turn):
                    if(temp < x.utility or temp == 55):
                        temp = x.utility
                else:
                    if(temp > x.utility or temp == 55):
                        temp = x.utility
            parent.utility = temp
```

Figure 29: Program Source Code 11 of 14 [Python]

```
def AlphaBetaSearch(self):
        temp = self.AlphaBetaMax(self.root, -1000,1000)
        for child in self.root.children:
            if(child.utility == temp):
                return child.game

    def AlphaBetaMax(self,node,alpha,beta):
        if(node.game.isTerminal()):
            return node.game.getAIUtility()
        temp = -100
        for action in node.children:
            temp = max(temp,self.AlphaBetaMin(action,alpha,beta))
            if(temp >= beta):
                return temp
            alpha = max(alpha,temp)
        return temp

    def AlphaBetaMin(self,node,alpha,beta):
        if(node.game.isTerminal()):
            return node.game.getAIUtility()
        temp = 100
        for action in node.children:
            temp = min(temp,self.AlphaBetaMax(action,alpha,beta))
            if(temp <= alpha):
                return temp
            beta = min(beta,temp)
        return beta
```

Figure 30: Program Source Code 12 of 14 [Python]

```
def getNumNodes(self):
        count = 0
        temp = queue.Queue()
        temp.put(self.root)
        while(temp.empty() == False):
            node = temp.get()
            if node.game.isTerminal():
                count += 1
            for x in node.children:
                temp.put(x)
        return count

ttt = game()
ttt.printPlayers()
sys.stdout.flush()
ttt.playGame()
```

Figure 31: Program Source Code 13 of 14 [Python]

```python
def AlphaBetaSearch(self):
        temp = self.AlphaBetaMax(self.root, -1000,1000)
        print("HERE IS THE UTILITY VALUE FOR MOVE CHOSEN:", temp)
        for child in self.root.children:
            if(child.utility == temp):
                return child.game

    def AlphaBetaMax(self,node,alpha,beta):
        if(node.game.isTerminal()):
            print("Terminal node with utility value: ")
            node.game.printBoard()
            sys.stdout.flush()
            print(node.game.getAIUtility())
            sys.stdout.flush()
            print()
            sys.stdout.flush()
            return node.game.getAIUtility()
        node.game.printBoard()
        print()
        sys.stdout.flush()
        temp = -100
        for action in node.children:
            temp = max(temp,self.AlphaBetaMin(action,alpha,beta))
            if(temp >= beta):
                return temp
            alpha = max(alpha,temp)
        return temp

    def AlphaBetaMin(self,node,alpha,beta):
        if(node.game.isTerminal()):
            print("Terminal node with utility value: ")
            node.game.printBoard()
            sys.stdout.flush()
            print(node.game.getAIUtility())
            print()
            sys.stdout.flush()
            return node.game.getAIUtility()

        node.game.printBoard()
        print()
        sys.stdout.flush()
        temp = 100
        for action in node.children:
            temp = min(temp,self.AlphaBetaMax(action,alpha,beta))
            if(temp <= alpha):
                return temp
            beta = min(beta,temp)
        return beta
```

Figure 32: Program Source Code 14 of 14 (Amended *AlphaBetaSearch()* member function for Question 2) [Python]

8 Bibliography

[1] S. Russel and P. Norvig, *Artificial intelligence : A modern approach*, Third. Pearson, 2010.