f

SCHOOL OF ENGINEERING AND TECHNOLOGY
BML MUNJAL UNIVERSITY GURGAON

CSE3705:
“Artificial Intelligence”:
Even Semester 2022

Programming Assignment

Tic Tac Toe

Submitted To : Dr. Kushal Shah

Student’s Name	**Enrollment No.**
Ishaan Pandey	200275
Khushi Mahawar	200293
Prachi Bagga	200294
Tanmay Rajeev Khurana	200232
Utsav Sharma	200296

Table of Contents

Abstract

This report provides an overview of how computers are designed to play games. We developed the fundamental minimax method and improved it by reducing the amount of game tree that must be created. To determine the best course of action, we put our algorithms to the test on regular Tic-Tac-Toe. We were able to determine the optimal starting move in Tic-Tac-Toe using our methods while reducing the number of nodes from 255168 to to 8543 in the first step of game tree from.

Acknowledgement

We would like to acknowledge all those without whom this project would not have been successful. Firstly, we would like to thank our Artificial Intelligence Professor, Dr. Kushal Shah whose expertise, patience and understanding added considerably to our capstone project. He made us understand how to successfully complete this project and without him, the project would not have been complete. This project has been a source to learn and bring our theoretical knowledge to the real-life world. So, we would really acknowledge his help and guidance for this project. We would also like to thank our parents who have always been there whenever needed. It wouldn't have been possible without their encouragement and motivation. Once again, thanks to everyone for making this project successful.

***Ishaan Pandey** - Project Report developed, Analysis of the Game*
***Khushi Mahawar** - Explained game through the video, Assisted in Project Report, Research on Alpha Beta Pruning on Tic Tac Toe*
***Prachi Bagga** - Research on the Tic Tac Toe game, Assisted in programming*
***Tanmay Rajeev Khurana** - Project Report developed, Minimax and Alpha Beta Pruning Analysis*
***Utsav Sharma** - Programmed Tic Tac Toe, Research on Alpha Beta Pruning on Tic Tac Toe, Leader of the Group*

Introduction

Mathematicians have been interested in building computers to play board games since computers were conceived. The first computer to beat a human opponent in chess was constructed in 1956. Computers create a tree of possible game options and then work backwards to discover the move that will give the computer the best result. Although computers can analyze board situations fast, it is difficult for a computer to search through the full tree in a game like chess, where there are over 10120 potential board configurations. The issue for the computer then becomes avoiding looking in certain portions of the tree. When playing a game, humans likewise look forward a set number of moves, but experienced players are already aware of certain theories and methods that inform them which sections of the tree to look at. Even the most skilled humans will be defeated as computers become faster in terms of raw processing power and humans become better at programming computers to search more efficiently. In this project, we use Alpha Beta Pruning for reducing the number of nodes in the game tree that the computer must examine in order to find the best possible move. We begin with a simple algorithm called minimax, which searches the entire tree. These algorithms are the foundation of current game-playing computers.

Tic Tac Toe, commonly known as noughts and crosses is a paper and pencil game where two players alternately mark the spaces in a 3 × 3 grid with either an *'O' or 'X'*. The game is won by the player who successfully arranges three of their marks in a row that is either horizontal, vertical, or diagonal. This game is deterministic, with fully observable environments, two agents acting alternately, and utility values that are always equal and opposite at the conclusion of the game.

Problem Statement

A 3x3 grid for the tic tac toe includes 9 cells. The two players, whose respective markings are *'O' and 'X'*, must set their marks one at a time during their turns. Once a mark occupies a cell, it can not be used once more. If the agent can make a row, column, or diagonal totally occupied with their corresponding marks, they have won the game. Once a winning scenario is achieved or all of the cells are occupied, the game is over. The problem is to find an optimal cell to fill the mark with in order to win against the AI. This has to be performed under certain constraints such as a cell cannot be used again, moves shall occur alternatively etc, as per the rules of this 2 player game.

Algorithms Used

Minimax Algorithm

Using the recursive method minimax, a player can select their best move under the assumption that their opponent is likewise playing well. Its goal is to reduce the greatest possible loss. Based on the adversarial search technique, this algorithm. A series of activities leading to a goal state would be the ideal answer to a typical search problem. Instead, MAX finds the contingent strategy in adversarial search, which defines MAX's moves in the initial state, then MAX's moves in the states as a result of every conceivable response by MIN, and continues until the termination condition alternates. Furthermore, when given the option, MAX chooses to move to the state of greatest value while MIN chooses the state of minimal value.

Alpha Beta Pruning

The issue with minimax search is that the number of game states it needs to look at, grows exponentially with tree depth. Because the minimax method calls itself repeatedly until one of the agents wins or the board is full, it is quite time consuming to solve the3X3 grid using the usual minimax approach. Alpha-beta pruning algorithm, which prunes the tree to remove significant portions from consideration, can be used to resolve this issue. It produces the same move as minimax when applied to a conventional minimax tree, but it removes branches that are unable to affect the ultimate choice.

Alpha-beta pruning gets its name from the following parameters:
α= the value of the best choices i.e. Max value so far at any choice point along the path for MAX
β= the value of the best choices i.e. Min value so far at any choice point along the path for MIN

Implementation

We started with building a tic tac toe game for two players i.e. Player A and Player B and allowing them to play against each other. Later, the game was converted into an AI based Tic Tac Toe where the player plays against the computer. The computer or say AI, plays as such that it would never lose, even in any possible condition. Refer Appendix A for python code.

The following function was used to check whether the game has been won or has ended up being a draw. Here, if any row or column or a diagonal turns out to be filled with the same mark, the algorithm sets the winner as the respective player.

```
def checkGameIsOver(board):
    for i in range(3):
        #check horizontals
        if board[i][0] == board[i][1] and board[i][1] == board[i][2]:
            return board[i][0]
        #check verticals
        if board[0][i] == board[1][i] and board[1][i] == board[2][i]:
            return board[0][i]
    #check diagonals
    if board[0][0] == board[1][1] and board[1][1] == board[2][2]:
        return board[0][0]
    if board[0][2] == board[1][1] and board[1][1] == board[2][0]:
        return board[0][2]
    #check draw (board full but no winner)
    for x in range(3):
        for y in range(3):
            if not (board[x][y] == player or board[x][y] == opponent):
                return False
                #meaning that the game is still going
    return "No one"
#means that it was a draw
```

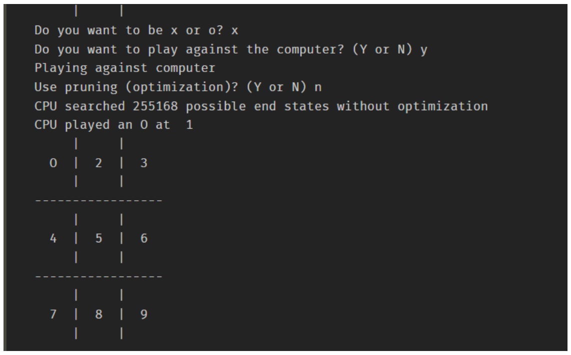
The function fillSquare as mentioned below was used to convert the user mark position i.e from 0-9 to a respective 2d Array position. Modulo function has been used here to find the column number.

```
#index is num 1-9, character is x or o, returns false if another move is already there
def fillSquare(index, character, board):
    row = int((index - 1)/ 3)
    col = (index - 1) % 3
    if board[row][col] == "X" or board[row][col] == "O":
        return False
    board[row][col] = character
    return True
```

The game was initially implemented using the Minimax algorithm. The following function was created for the same. The function creates a copy of the current game board for each available move it can make at that moment. For example, if there are 4 spaces empty and the AI has to make the move, it will create 4 possible copies of the board with 4 different marked positions. And, will then calculate the end state of each possibility by returning a positive value if the winner is the AI else a negative value. This way, an optimized move can be found by the algorithm.

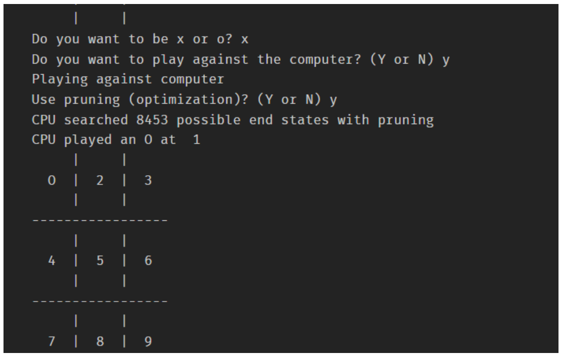
```
def minimax(board, depth, isMaximizer):
    global count #number of times run in one session
    winner = checkGameIsOver(board)
    if not winner == False:
        count += 1
        if winner == player:
            return -10 + depth
        elif winner == opponent:
            return 10 - depth
        else:
            return 0
    if isMaximizer:
        maxEval = -math.inf
        for i in range(countNumEmpty(board)):
            boardCopy = np.copy(board)
            index = fillEmpty(i, opponent, boardCopy)
            eval = minimax(boardCopy, depth + 1, False)
            if (eval > maxEval):
                maxEval = eval
                bestIndex = index
        if depth == 0:
            return bestIndex
        return maxEval
    else:
        minEval = math.inf
        for i in range(countNumEmpty(board)):
            boardCopy = np.copy(board)
            fillEmpty(i, player, boardCopy)
            eval = minimax(boardCopy, depth + 1, True)
            minEval = min(eval, minEval)
        return minEval
```

Later, it was observed that the every time a move was made, the Minimax algorithm searched a large number of possibilities to find an optimal move. For example, fo the first move it searched for 255168 possible moves, which is ridiculous. Thus, alpha beta pruning was implemented on the algorithm to prune unwanted nodes from teh game tree.
After, the alpha beta pruning was implemented, the searched for the first move were cut down to 8543 from 255168. Below, are the images of the game depicting it.



```
     |     |
Do you want to be x or o? x
Do you want to play against the computer? (Y or N) y
Playing against computer
Use pruning (optimization)? (Y or N) n
CPU searched 255168 possible end states without optimization
CPU played an O at  1
     |     |
  O  |  2  |  3
     |     |
-----------------
     |     |
  4  |  5  |  6
     |     |
-----------------
     |     |
  7  |  8  |  9
     |     |
```

Without Alpha Beta Pruning (255168 searches)



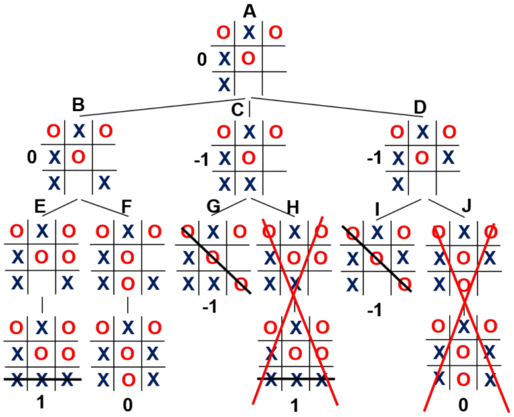
```
     |     |
Do you want to be x or o? x
Do you want to play against the computer? (Y or N) y
Playing against computer
Use pruning (optimization)? (Y or N) y
CPU searched 8453 possible end states with pruning
CPU played an O at  1
     |     |
  O  |  2  |  3
     |     |
-----------------
     |     |
  4  |  5  |  6
     |     |
-----------------
     |     |
  7  |  8  |  9
```

With Alpha Beta Pruning (8543 searches)

Here, is the function for Alpha Beta Pruning on the algorithm.

```
def minimaxAB(board, depth, alpha, beta, isMaximizer):
    global count #number of times run in one session
    winner = checkGameIsOver(board)
    if not winner == False:
        count += 1
        if winner == player:
            return -10 + depth
        elif winner == opponent:
            return 10 - depth
        else:
            return 0
    if isMaximizer:
        maxEval = -math.inf
        for i in range(countNumEmpty(board)):
            boardCopy = np.copy(board)
            index = fillEmpty(i, opponent, boardCopy)
            eval = minimaxAB(boardCopy, depth + 1, alpha, beta, False)
            if (eval > maxEval):
                maxEval = eval
                bestIndex = index
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        if depth == 0:
            return bestIndex
        return maxEval
    else:
        minEval = math.inf
        for i in range(countNumEmpty(board)):
            boardCopy = np.copy(board)
            fillEmpty(i, player, boardCopy)
            eval = minimaxAB(boardCopy, depth + 1, alpha, beta, True)
            minEval = min(eval, minEval)
            beta = min(eval, beta)
            if beta <= alpha:
                break
        return minEval
```

Alpha Beta Pruning Explanation



Consider the tic-tac-toe game tree as shown above. Here, the human is playing ‘*O’s* while the machine is playing *‘X’s*. The game's value is 1 for a victory by the machine, -1 for a victory by a human, and 0 for a tie. It is obvious from the game tree that the AI shall choose the leftmost sub gametree in order to win. Ordinary minimax, however, would still have to go through the entire tree in order to reach that conclusion. The minimum of B's children, which is 0, would be taken first, and that would be contrasted with the minimum of C's children, which is -1. It would then be only able to conclude the maximum of both. But, it can be simply observed that since C’s left child is -1, which guarantees the value of C cannot be greater than -1. It has to be either -1 or less than that, but since C is choosing the least possible value, -1 should be an optimal selection. Thus, the right subtree gets pruned.

Spaces Filled	Without Pruning	With Pruning	Multiple
0	255168	8453	30.186
2	3468	346	10.023
4	94	26	3.615
6	6	4	1.5
8	1	1	1

Table 1

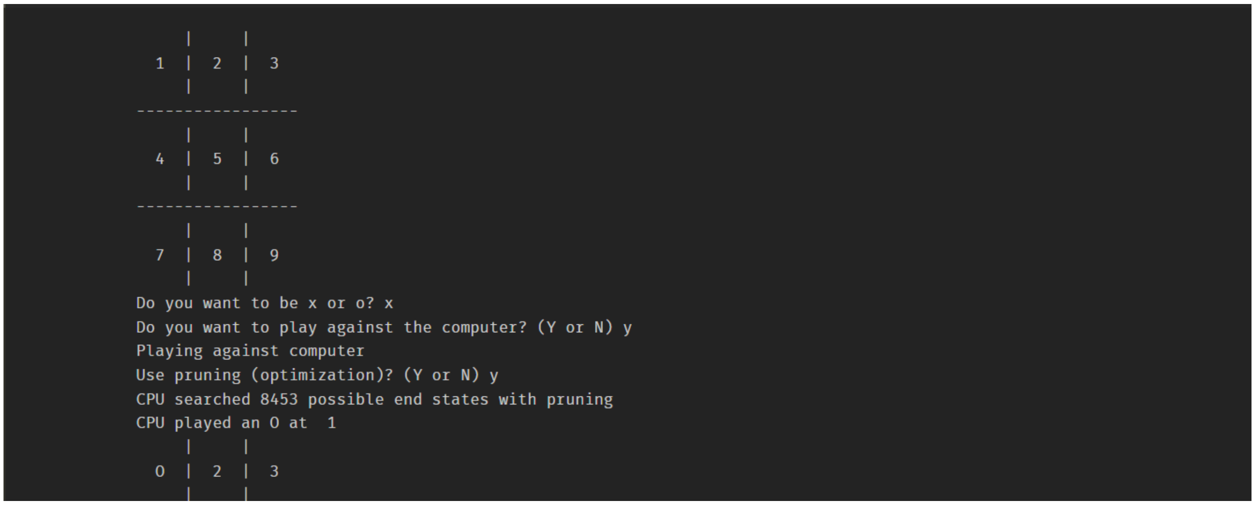
Here, in the game whenever the opponent plays his mark on the board game, the AI searches for all possible end states and finds the optimal move for the algorithm to win. Above is a table describing how pruning played such an important role in the first step where it cut down the searches in the game tree by a multiple of 30.

Here, in our game, the AI is allowed to play the first move every time. It is because we wanted our AI to never lose in the game, and the book by Thomas Bolon - How to never lose at Tic Tac Toe suggested to always pick start from the corner to get more possible chances of winning the game. Also, here, the AI shall only be able to end up in a tie if the opponent plays in the middle, and if the opponent does not play optimally, of course the AI would be smart enough to win.

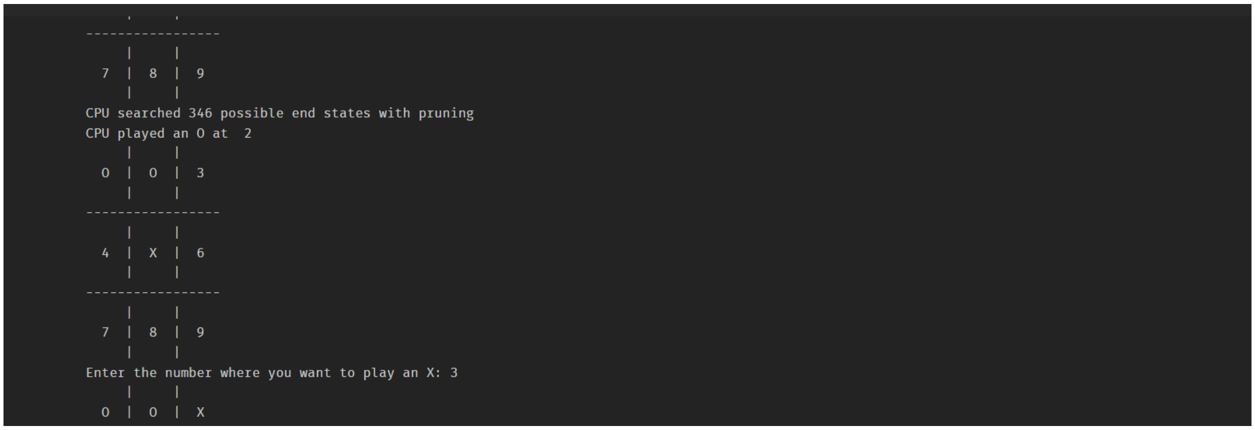
The opponent if is able to track down a strategy using the game tree or his own intelligence to find the best possible move, he shall play optimally in the game, and thus, would be able to end up the game with a draw atleast. If the opponent plays with no proper strategy, he might end up get beated by the AI very easily since AI designs a game tree for every move the opponent plays and chooses its optimal move from the same.

Conclusion

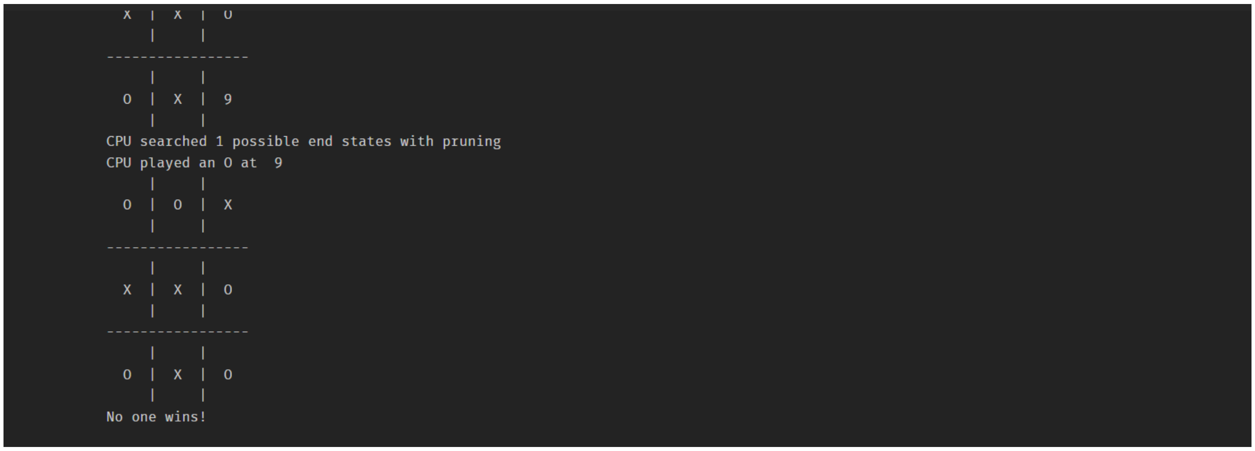
Tic tac Toe has been implemented using two methods in this research, and a comparison between them is investigated. The first algorithm is the mini-max algorithm, while the second is an improved variant of the mini-max algorithm called mini-max with alpha beta pruning. The study found that, for the same level of difficulty, the two algorithms act substantially differently in terms of the time required and the number of iterations carried out, with alpha beta pruning generating the game state much faster and with a lot less iterations than mini-max.



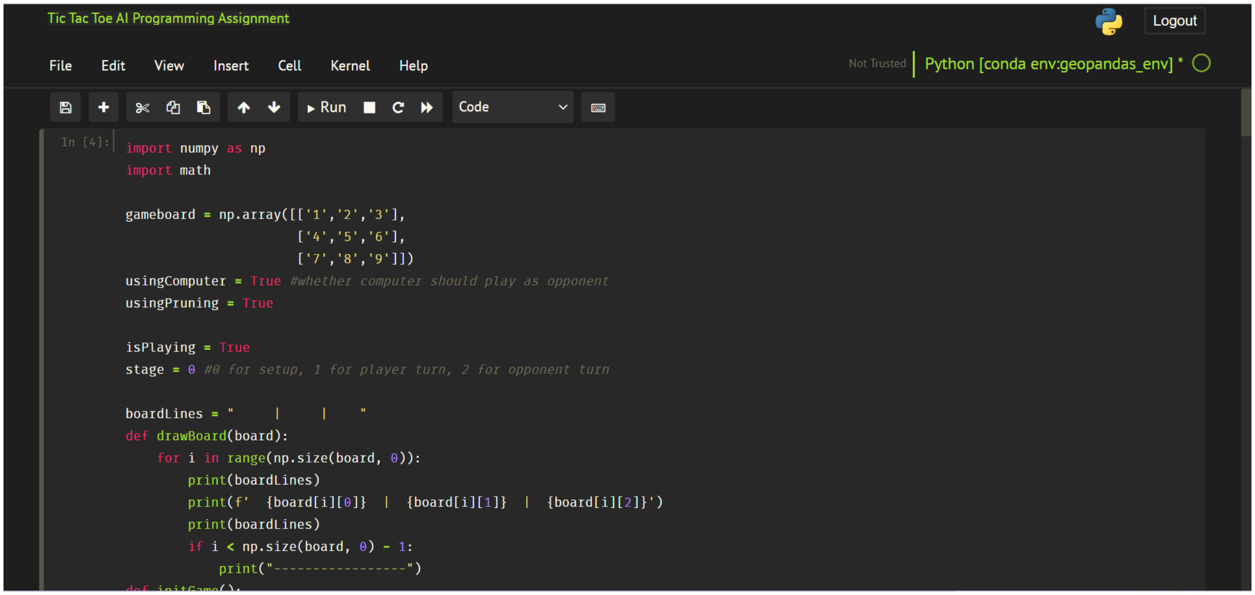
```
     |     |
  1  |  2  |  3
     |     |
-----------------
     |     |
  4  |  5  |  6
     |     |
-----------------
     |     |
  7  |  8  |  9
     |     |
Do you want to be x or o? x
Do you want to play against the computer? (Y or N) y
Playing against computer
Use pruning (optimization)? (Y or N) y
CPU searched 8453 possible end states with pruning
CPU played an O at  1
     |     |
  O  |  2  |  3
     |     |
```



```
-----------------
     |     |
  7  |  8  |  9
     |     |
CPU searched 346 possible end states with pruning
CPU played an O at  2
     |     |
  O  |  O  |  3
     |     |
-----------------
     |     |
  4  |  X  |  6
     |     |
-----------------
     |     |
  7  |  8  |  9
     |     |
Enter the number where you want to play an X: 3
     |     |
  O  |  O  |  X
```



```
  X  |  X  |  O
     |     |
-----------------
     |     |
  O  |  X  |  9
     |     |
CPU searched 1 possible end states with pruning
CPU played an O at  9
     |     |
  O  |  O  |  X
     |     |
-----------------
     |     |
  X  |  X  |  O
     |     |
-----------------
     |     |
  O  |  X  |  O
     |     |
No one wins!
```


Tic Tac Toe AI Programming Assignment
Logout
File
Edit
View
Insert
Cell
Kernel
Help
Not Trusted
Python [conda env:geopandas_env] *
Run
Code
In [4]:

```python
import numpy as np
import math

gameboard = np.array([['1','2','3'],
                      ['4','5','6'],
                      ['7','8','9']])
usingComputer = True #whether computer should play as opponent
usingPruning = True

isPlaying = True
stage = 0 #0 for setup, 1 for player turn, 2 for opponent turn

boardLines = "     |     |     "
def drawBoard(board):
    for i in range(np.size(board, 0)):
        print(boardLines)
        print(f'  {board[i][0]}  |  {board[i][1]}  |  {board[i][2]}')
        print(boardLines)
        if i < np.size(board, 0) - 1:
            print("-----------------")
def initGame():
```

References

1. Bolon T. (2013). How to never lose at Tic-Tac-Toe. BookCountry
2. https://www.whitman.edu/documents/Academics/Mathematics/2019/Felstiner-Guichard.pdf
3. Alpha-Beta Pruning in Mini-Max Algorithm –An Optimized Approach for a Connect-4 Game
4. https://www.academia.edu/42989331/A_Mini_Project_Report_Tic_Tac_Toe_
5. https://www.javatpoint.com/ai-alpha-beta-pruning

Appendix A

```
import numpy as np
import math

gameboard = np.array([['1','2','3'],
                      ['4','5','6'],
                      ['7','8','9']])
usingComputer = True #whether computer should play as opponent
usingPruning = True

isPlaying = True
stage = 0 #0 for setup, 1 for player turn, 2 for opponent turn

boardLines = "     |     |     "
def drawBoard(board):
    for i in range(np.size(board, 0)):
        print(boardLines)
        print(f'  {board[i][0]}  |  {board[i][1]}  |  {board[i][2]}')
        print(boardLines)
        if i < np.size(board, 0) - 1:
            print("-----------------")
def initGame():
    letter = ""
    while not (letter == 'X' or letter == 'O'):
        letter = input("Do you want to be x or o? ").upper()

    global player
    global opponent
    global usingComputer
    global usingPruning
    player = letter
    if player == "O":
        opponent = "X"
    else:
        opponent = "O"
    result = ""
    while not (result == "Y" or result == "N"):
        result = input("Do you want to play against the computer? (Y or N) ").upper()
    usingComputer = (result == "Y")
    print("Playing against " + ("computer" if usingComputer else "another person"))

    if usingComputer:
        result = ""
        while not (result == "Y" or result == "N"):
            result = input("Use pruning (optimization)? (Y or N) ").upper()
        usingPruning = (result == "Y")
#index is num 1-9, character is x or o, returns false if another move is already there
def fillSquare(index, character, board):
    row = int((index - 1)/ 3)
    col = (index - 1) % 3
    if board[row][col] == "X" or board[row][col] == "O":
        return False
    board[row][col] = character
    return True
#fills the nth empty space, returns index of filled square
def fillEmpty(index, character, board):
    count = 0
    for x in range(3):
        for y in range(3):
            if not (board[x][y] == player or board[x][y] == opponent):
                if (count == index):
                    fillSquare(3 * x + y + 1, character, board)
                    return 3 * x + y + 1
                count += 1
def countNumEmpty(board):
    count = 0
    for x in range(3):
        for y in range(3):
            if not (board[x][y] == player or board[x][y] == opponent):
                count += 1
    return count
def playTurn(whichPlayer, board):
    result = False
    while not result:
        num = input(f"Enter the number where you want to play an {whichPlayer}: ")
        if num not in '0 1 2 3 4 5 6 7 8 9'.split():
            continue
        result = fillSquare(int(num), whichPlayer, board)

def computerTurn(board, shouldPrune):
    global count
    count = 0
    if shouldPrune:
        index = minimaxAB(board, 0, -math.inf, math.inf, True)
        print(f"CPU searched {count} possible end states with pruning")
    else:
        index = minimax(board, 0, True)
        print(f"CPU searched {count} possible end states without optimization")
    fillSquare(index, opponent, board)
    print(f"CPU played an {opponent} at ", index)

def minimax(board, depth, isMaximizer):
    global count #number of times run in one session
    winner = checkGameIsOver(board)
    if not winner == False:
        count += 1
        if winner == player:
            return -10 + depth
        elif winner == opponent:
            return 10 - depth
        else:
            return 0
    if isMaximizer:
        maxEval = -math.inf
        for i in range(countNumEmpty(board)):
            boardCopy = np.copy(board)
            index = fillEmpty(i, opponent, boardCopy)
            eval = minimax(boardCopy, depth + 1, False)
            if (eval > maxEval):
                maxEval = eval
                bestIndex = index
        if depth == 0:
            return bestIndex
        return maxEval
    else:
        minEval = math.inf
        for i in range(countNumEmpty(board)):
            boardCopy = np.copy(board)
            fillEmpty(i, player, boardCopy)
            eval = minimax(boardCopy, depth + 1, True)
            minEval = min(eval, minEval)
        return minEval

def minimaxAB(board, depth, alpha, beta, isMaximizer):
    global count #number of times run in one session
    winner = checkGameIsOver(board)
    if not winner == False:
        count += 1
        if winner == player:
            return -10 + depth
        elif winner == opponent:
            return 10 - depth
        else:
            return 0
    if isMaximizer:
        maxEval = -math.inf
        for i in range(countNumEmpty(board)):
            boardCopy = np.copy(board)
            index = fillEmpty(i, opponent, boardCopy)
            eval = minimaxAB(boardCopy, depth + 1, alpha, beta, False)
            if (eval > maxEval):
                maxEval = eval
                bestIndex = index
            alpha = max(alpha, eval)
            if beta <= alpha:
                break
        if depth == 0:
            return bestIndex
        return maxEval
    else:
        minEval = math.inf
        for i in range(countNumEmpty(board)):
            boardCopy = np.copy(board)
            fillEmpty(i, player, boardCopy)
            eval = minimaxAB(boardCopy, depth + 1, alpha, beta, True)
            minEval = min(eval, minEval)
            beta = min(eval, beta)
            if beta <= alpha:
                break
        return minEval

#this is an unintuitive function, should have made one to check if game over, then
#another to check winner
def checkGameIsOver(board):
    for i in range(3):
        #check horizontals
        if board[i][0] == board[i][1] and board[i][1] == board[i][2]:
            return board[i][0]
        #check verticals
        if board[0][i] == board[1][i] and board[1][i] == board[2][i]:
            return board[0][i]
    #check diagonals
    if board[0][0] == board[1][1] and board[1][1] == board[2][2]:
        return board[0][0]
    if board[0][2] == board[1][1] and board[1][1] == board[2][0]:
        return board[0][2]
    #check draw (board full but no winner)
    for x in range(3):
        for y in range(3):
            if not (board[x][y] == player or board[x][y] == opponent):
                return False
                #meaning that the game is still going
    return "No one" #means that it was a draw

drawBoard(gameboard)
while isPlaying:
    global winner
    if stage == 0:
        initGame()
        stage = 2

    elif stage == 1:
        playTurn(player, gameboard)
        drawBoard(gameboard)
        winner = checkGameIsOver(gameboard)
        if not winner == False:
            isPlaying = False
        else:
            stage = 2

    elif stage == 2:
        if usingComputer:
            computerTurn(gameboard, usingPruning)
        else:
            playTurn(opponent, gameboard)
        drawBoard(gameboard)
        winner = checkGameIsOver(gameboard)

        if not winner == False:
            isPlaying = False
        else:
            stage = 1

print(f'{winner} wins!')
```