PROJECT REPORT ON

"Tic Tac Toe" using Minimax Algorithm

Submitted By-

Ishit Choudhary 102003133

Gurleen Kaur 102003138

Submitted to-

Mr. Niyaz Wani

ARTIFICIAL INTELLIGENCE

(UCS 411)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY,

(SEEMED TO BE UNIVERSITY), PATIALA, PUNJAB

INDIA

JAN-JUNE 2022

Table of Contents

DECLARATION

We, the undersigned solemnly declare that the project report on Tic Tac Toe using Minimax Algorithm is based on my own work. I assert the statements made and conclusions drawn are an outcome of my research work. I further certify that I. The work contained in the report is original and has been done by me under the general supervision of my supervisor. The work has not been submitted to any other Institution for any other degree/diploma/certificate in this university or any other University of India or abroad. We have followed the guidelines provided by the university in writing the report.

____

Ishit Choudhary (102003133)

____

Gurleen Kaur (102003138)

INTRODUCTION

Tic-tac-toe also known as noughts and crosses is a paper and pencil game for two players, who take turns marking the spaces in a 3 x 3 grid traditionally. The player who succeeds in placing three of their marks in a horizontal, vertical or diagonal row wins the game. It is a zero-sum of perfect information game. This means that it is deterministic, with fully observable environments in which two agents act alternately and the utility values at the end of the game are always equal and opposite. Because of the simplicity of tic-tac-toe, it is often used as pedagogical tool in artificial intelligence to deal with searching of game trees. The optimal move for this game can be gained by using minimax algorithm, where the opposition between the utility functions makes the situation adversarial, hence requiring adversarial search supported by minimax algorithm with alpha beta pruning concept in artificial intelligence.

OBJECTIVES

1. To develop Artificial intelligence-based tic-tac-toe game for human Vs AI by implementing minimax algorithm.

2. To analyse the complexity of minimax algorithm through 3x3 tic tac toe game.

3. To study (theoretically) alpha-beta pruning concept for improved speed of searching the optimal choice in tic-tac toe game.

4. To study (theoretically) optimizing methods for alpha-beta pruning using heuristic evaluation function.

AGENTS

An agent perceives its environment through sensors and acts on the environment through actuators. Its behaviour is described by its function that maps the percept to action. In tic tac toe there are two agents, the computer system and human. In the AI-based tic-tac toe game the function is mapped using the minimax algorithm alongside alpha beta pruning. The agent can be further described by following factors:

Performance Measure: Number of wins or draws.

Environment: A 3x3 grid with 9 cells respectively, with opponent (human agent). The cell once occupied by a mark cannot be used again. Task environment is deterministic, fully observable, static, multi-agent, discrete, sequential.

Actuators: Display, Keyboard (human agent)

Sensors: The opponent's (human agent) input from keyboard keys

Furthermore, it is a utility-based agent since it uses heuristics concept as utility function that measures its preferences among the states and chooses the action that leads to the best expected utility i.e., winning or tie condition for the computer system agent. The utility function is taken as +1 for winning situation of maximizing agent or AI, -1 for winning situation of minimizing agent or human player and 0 for draw condition.

Agent Environment Description

The basic environment for tic tac toe game agent is a 3x3 grid with 9 cells respectively with opponent as human agent. The cell once occupied by a mark cannot be used again. The agent has access to complete state of environment at any point and can detect all aspects that are relevant to the choice of action making the environment fully observable. Also, the next state of environment can be completely determined by current state and action executed which means environment is deterministic. It is a two-player game with the opponent as human agent, so it is a multi-agent environment. Beside the game environment has a finite number of states, precepts and actions, making the environment static and discrete. Also, the environment is sequential since the current choice of cell could affect all future decisions. Thus, the task environment for the tic tac toe system agent is sequential, deterministic, fully observable, static, discrete and a multi-agent.

PROBLEM SPECIFICATION

Tic tac toe game has 9 cells for a 3x3 grid. The two players with their respective marks as ‘X’ and ‘O’ are required to place their marks in their turns one by one. Once the cell is occupied by a mark it cannot be used again. The game is won if the agent is able to make a row or column or a diagonal occupied completely with their respective marks. The game terminates once the winning situation is gained or the cells are fully occupied. The problem specification for this game is given below:

Problem: Given a 3x3 grid, the agents have to find the optimal cell to fill with respective marks.

Goals: To find the optimal cell to fill with respective marks and in order to win the game, the cell must be filled such that one of the following criteria is satisfied:

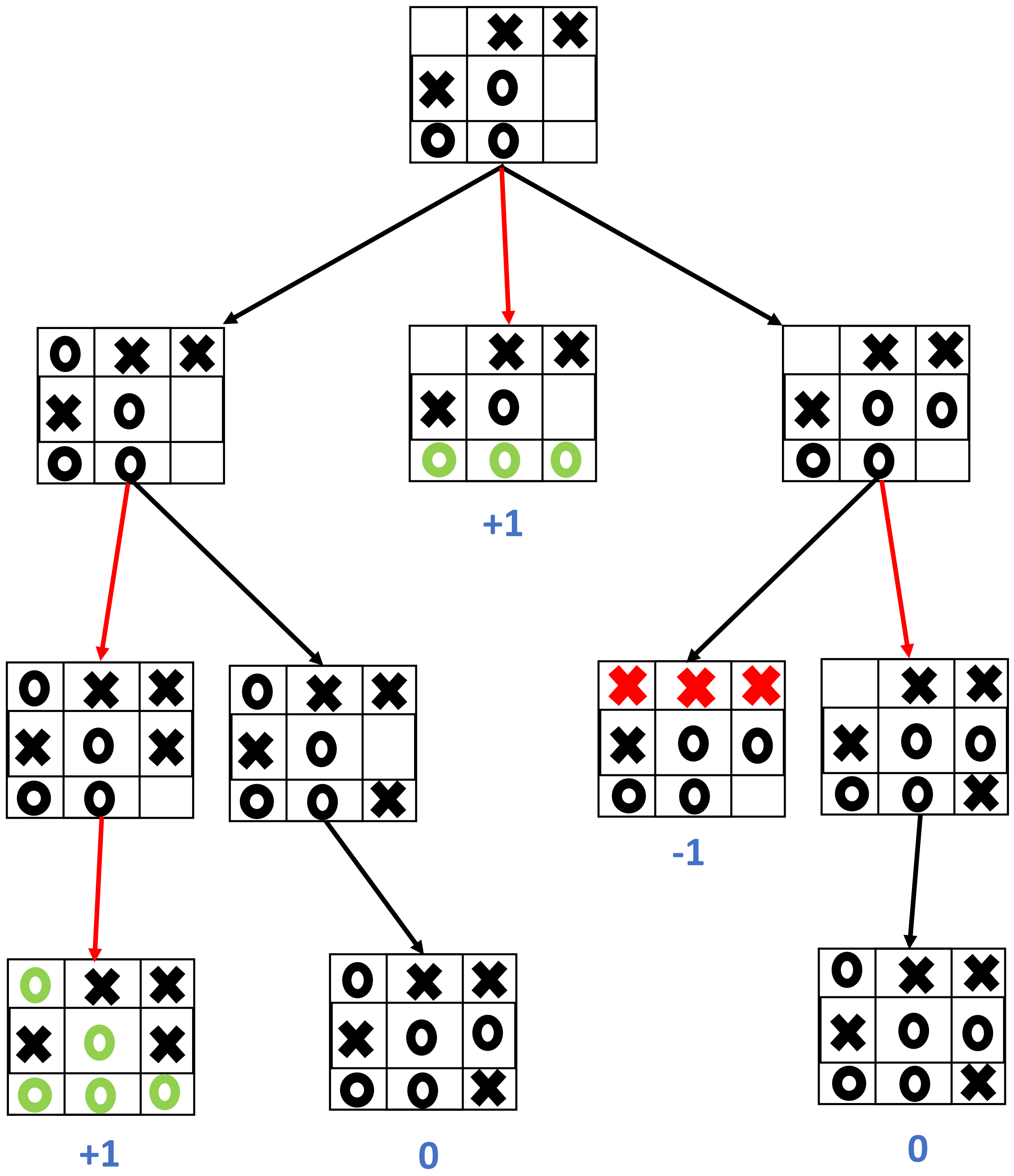
1. A row is completely filled by a mark ‘X’ or ‘O’.
2. A diagonal is completely filled by a mark ‘X’ or ‘O’.
3. A column is completely filled by a mark ‘X’ or ‘O’.

If these criteria are not satisfied by both the agents, the game is terminated with a tie situation.

Constraints:

1. Once the cell is occupied by a mark, it cannot be reused.
2. Agents place the mark alternatively. So, consecutive moves from any agent are not allowed.

STATE SPACE REPRESENATION



ALGORITHM USED

Minimax Algorithm (Applied in project):

Minimax is a recursive algorithm which is used to choose an optimal move for a player assuming that the opponent is also playing optimally. Its objective is to minimize the maximum loss. This algorithm is based on adversarial search technique. In a normal search problem, the optimal solution would be a sequence of actions leading to a goal state. Rather in adversarial search MAX finds the contingent strategy, which specifies the MAX's moves in the initial state, then MAX's moves in the states resulting from every possible response by MIN and continues till the termination condition comes alternately. Furthermore, given a choice, MAX prefers to move to a state of maximum value whereas MIN prefers a state of minimum value.

Alpha-Beta Pruning (Theoretical Analysis Only):

The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree. The minimax algorithm recursively calls itself until any one of the agents wins or the board is full which takes a lot of computation time and makes it impossible to solve the 3X3 grid using standard minimax algorithm. To solve this issue, we can use alpha-beta pruning algorithm which eliminates large parts of the tree from considerations by pruning the tree. When applied to a standard minimax tree, it returns the same move as minimax would, but it prunes away branches that cannot possibly influence the final decision. Basically, this algorithm applies the principle that there is no use expending search time to find out exactly how bad an alternative is if you have a better alternative. Alpha-beta pruning gets its name from the parameters that bound on the backed-up values that appears anywhere along the path:

α= the value of the best choices (i.e., highest value) so far at any choice point along the path for MAX

β= the value of the best choice (i.e., lowest value) so far at any choice point along the path for MIN

OUTCOME ANALYSIS

The minimax algorithm performs a complete depth-first exploration of the game tree. It recursively calls itself until a winning state of any agent is found or the grid is full. If the maximum depth of the tree is m and there are b legal moves at each point, then the time complexity of the minimax algorithm is O (bm). The space complexity is O(bm) for generating all actions at once. For 3X3 board, the possible number of moves is (3 * 3)! = 9! This is quite high and will grow exponentially if we increase number of squares on board.

So, to decrease the computation time, the search can be limited to certain level i.e., reduce the depth of search tree. The maximum depth is taken as 9 in our system. But when reducing the depth of search tree, the output is not optimal as we have cut-off 9 levels of our search tree which is a very large number of nodes. Hence, we have used heuristic evaluation function that approximates the true utility of a state without doing a complete search.

For this problem, the heuristic function used is given by:

E(n) = M(n) – O(n)

where,

M(n) is total of possible winning path of AI

O(n) is total of possible winning path of human player

E(n) is total evaluation for state n

This heuristic gives positive value if AI i.e., maximizing agent has more chance or winning and negative value if the human player i.e., minimizing agent has more chance of winning.

Function Descriptions

main()

Takes input from user for choice of naught or cross and also asks user if they would like to start first or should the AI start first. Calls human_turn and ai_turn functions in a loop until either one wins or all cells on the board are occupied. Calls wins function to check who won and prints corresponding output.

Main method is the driver code which interacts with the user initially and then calls other functions as required.

human_turn (c_choice, h_choice)

This function takes two arguments of c_choice and h_choice. These arguments refer to choice of cross or naught for respective players. This function prints the current state of board and asks for input from the user. It checks if the input is a valid move or not. If not valid, it gives an error message and asks for input again.

The function has a local dictionary called moves which stores all the legal moves possible in the game. It calls the set_move function to update the state according to the input and prints an error message if set_move returns FALSE.

ai_turn (c_choice, h_choice)

Similar to the human_turn function, this function also takes two arguments of c_choice and h_choice. These arguments refer to choice of cross or naught for respective players. This function updates the board according to the result garnered from the minimax function. It updates the state using set_move function and then prints the current state.

This function acts as driver code for AI's turn in the game.

render (state, c_choice, h_choice)

This function prints the current state of the board on the screen. It takes the current state, human notation and AI notation as input parameters.

clean()

This function clears the console. It uses the platform class to determine which platform is the code running on and then uses system commands accordingly.

minimax(state, depth, player)

AI function that chooses the best move for the computer. It takes the current state of the board, node index in the tree and player notation (human or computer) as input parameters. It is a recursive function in which at each recursive call it switches value of player to implement minimax algorithm. Best is a local list with three elements, the best row, the best column, and the best score. Initially best is set as [-1,-1,-infinity] if the player corresponds to computer or it is set as [-1,-1,+infinity] if the player corresponds to human. It generates the entire tree recursively and then backtracks while checking score at each level and storing the best possible route in the best list. This function returns the best list to the calling function.

set_move(x, y, player)

This function sets the move on board. It takes the coordinates of the move along with the player who is doing that move as input parameters. It checks if the entered coordinates are valid and if they are, it updates the current state of the board.

valid_move(x, y)

It takes the coordinates of the new move as input and checks if it is a valid move. A move is valid if it made at an empty cell. It calls function empty_cells to do the same.

empty_cells(state)

This function checks for empty cells. Empty cells are characterised by 0 in out code. It uses class enumerate to return a list of coordinates at which there are empty cells present.

game_over(state)

This function test if the human or computer wins. It takes the current state as input parameter and returns True if either the human or AI wins and returns False otherwise. It calls the wins function to check if either player has won.

wins(state, player)

This function tests if a specific player wins. Winning possibilities are if either player makes a row, column or diagonal. Hence for each player there are 8 winning configurations. All the win configurations are stored in the local list win_state. The function takes current state and player choice of cross or naught as input parameter. It checks if the player choice of cross or naught is present in any of the win configurations or not, if it is then it returns True else it returns False.

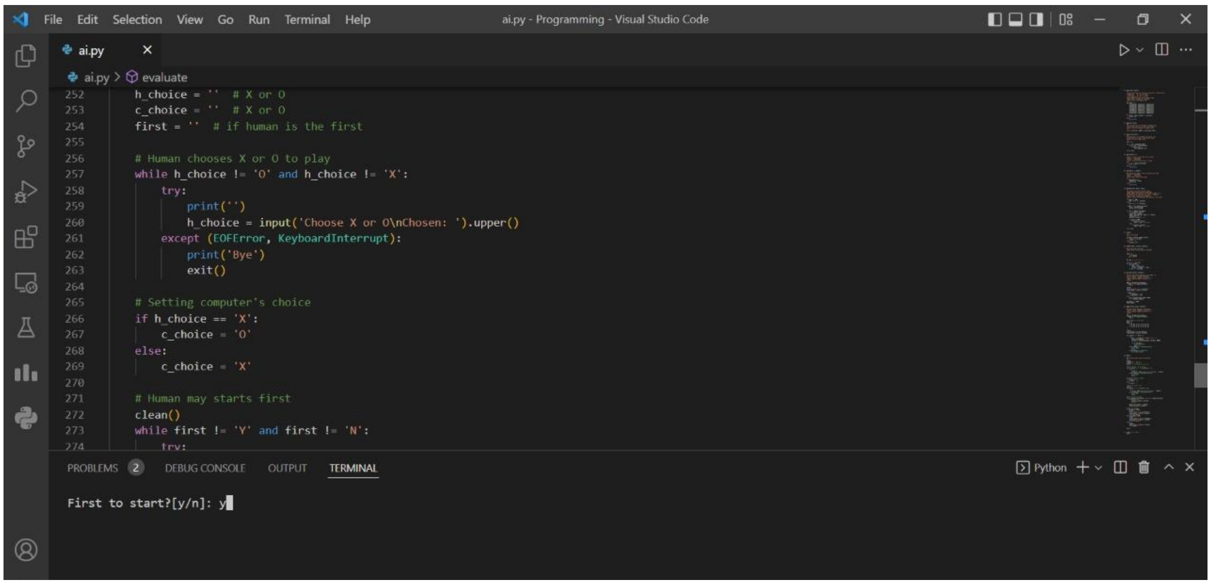
evaluate(state)

Function to evaluate heuristic value of the particular state. The function takes the current state of the board as input parameter and returns +1 if the computer wins, -1 if the human wins and 0 if it is a draw. It calls the win function to check for the win condition.

CONCLUSION

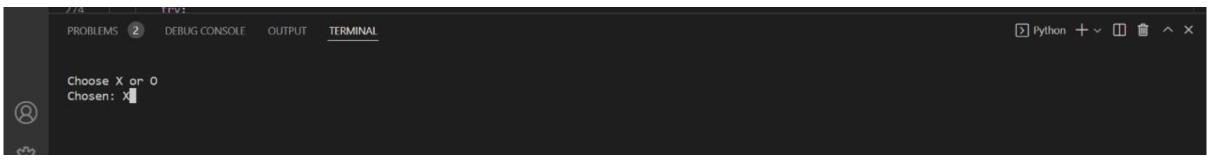
With the basis of minimax algorithm for mathematical analysis and optimizing the utility function using heuristic function, the 3x3 tic tac toe game was developed. We explored that a 3x3 tic tac toe, an adversary search technique game in artificial intelligence, can be developed using these techniques. Increasing the size of this game would create a huge time complexity issue with the same algorithm and techniques, for which other logics must be further researched.

WORKING SCREENSHOTS

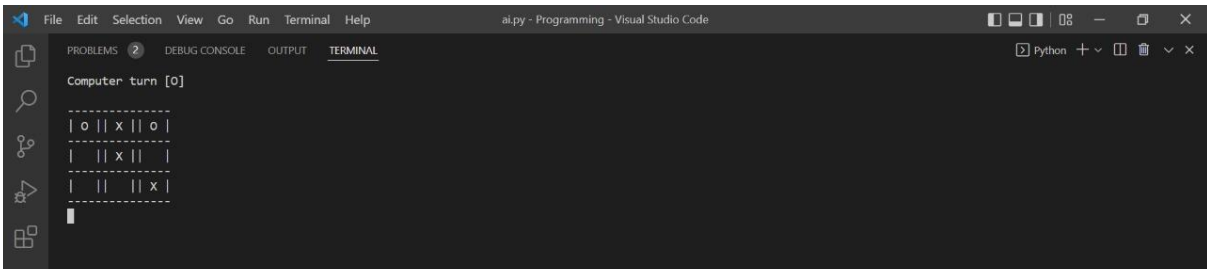


```
    h_choice = ''  # X or O
    c_choice = ''  # X or O
    first = ''  # if human is the first

    # Human chooses X or O to play
    while h_choice != 'O' and h_choice != 'X':
        try:
            print('')
            h_choice = input('Choose X or O\nChosen: ').upper()
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()

    # Setting computer's choice
    if h_choice == 'X':
        c_choice = 'O'
    else:
        c_choice = 'X'

    # Human may starts first
    clean()
    while first != 'Y' and first != 'N':
        try:
```

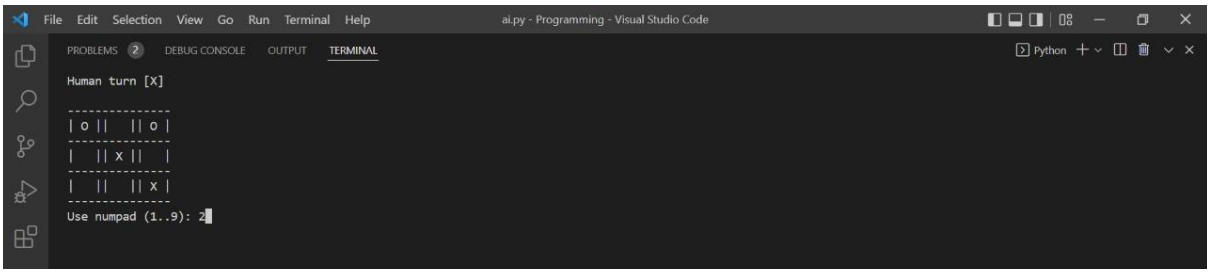
```
First to start?[y/n]: y
```



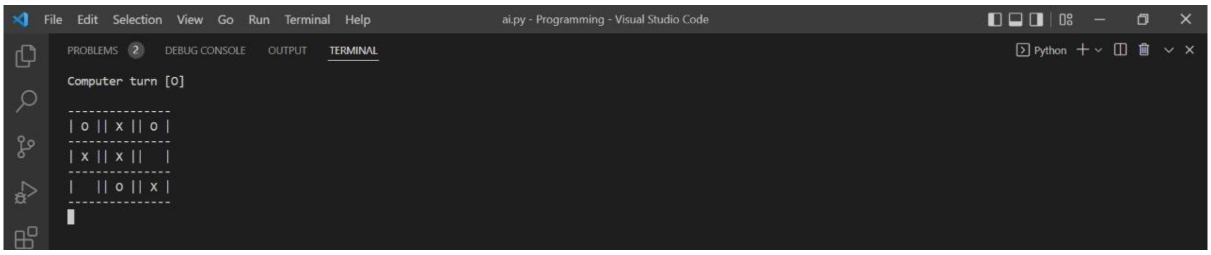
```
Choose X or O
Chosen: X
```



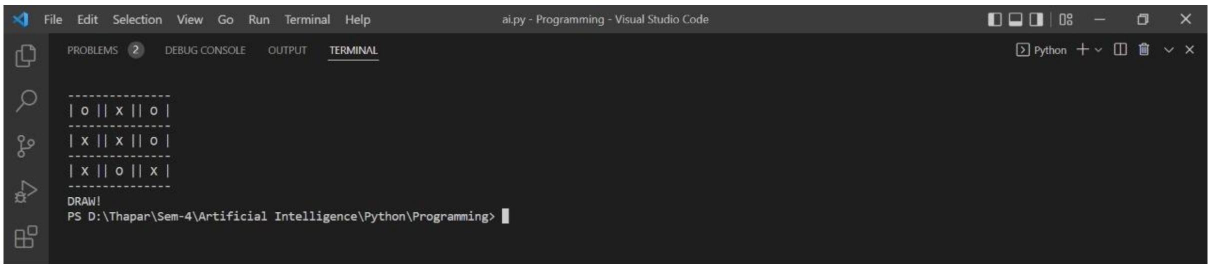
```
Computer turn [O]

---------------
| o || x || o |
---------------
|   || x ||   |
---------------
|   ||   || x |
---------------
```



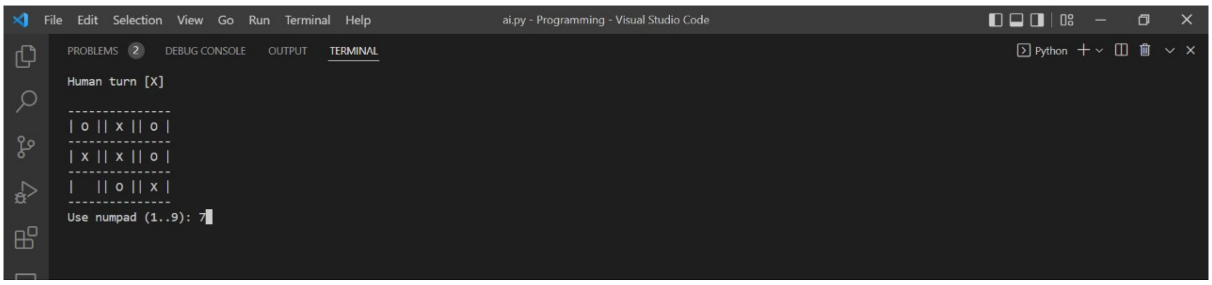
```
Human turn [X]

---------------
| o ||   || o |
---------------
|   || x ||   |
---------------
|   ||   || x |
---------------
Use numpad (1..9): 2
```



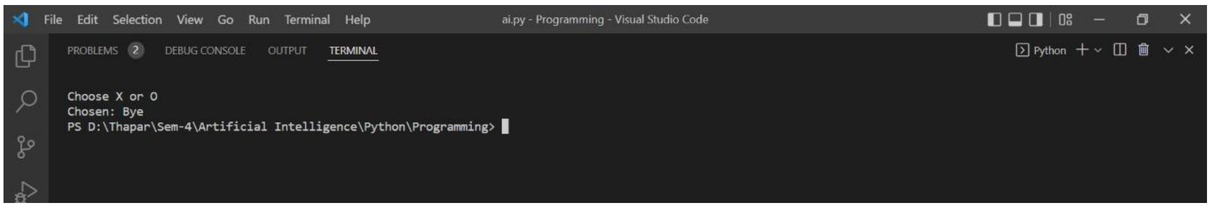
```
Computer turn [O]

---------------
| o || x || o |
---------------
| x || x ||   |
---------------
|   || o || x |
---------------
```


File Edit Selection View Go Run Terminal Help
ai.py - Programming - Visual Studio Code
PROBLEMS 2 DEBUG CONSOLE OUTPUT TERMINAL
Python

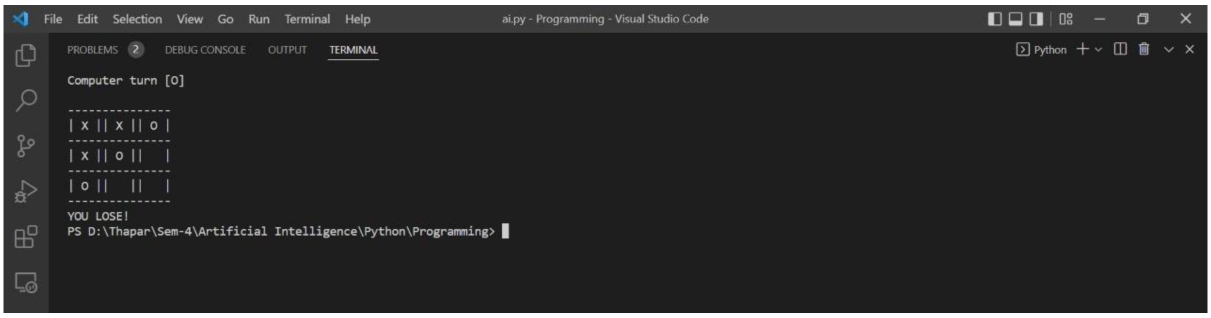
```
---------------
| o || x || o |
---------------
| x || x || o |
---------------
| x || o || x |
---------------
DRAW!
PS D:\Thapar\Sem-4\Artificial Intelligence\Python\Programming>
```


File Edit Selection View Go Run Terminal Help
ai.py - Programming - Visual Studio Code
PROBLEMS 2 DEBUG CONSOLE OUTPUT TERMINAL
Python

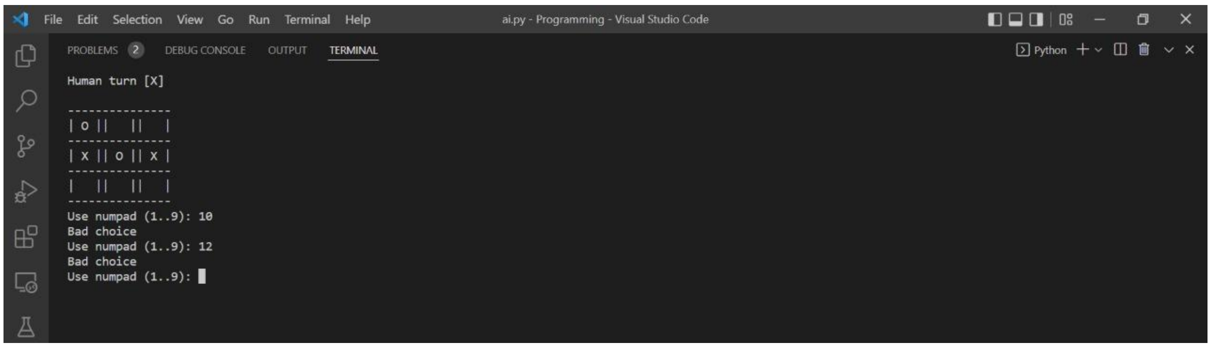
```
Human turn [X]

---------------
| o || x || o |
---------------
| x || x || o |
---------------
|   || o || x |
---------------
Use numpad (1..9): 7
```


File Edit Selection View Go Run Terminal Help
ai.py - Programming - Visual Studio Code
PROBLEMS 2 DEBUG CONSOLE OUTPUT TERMINAL
Python

```
Choose X or O
Chosen: Bye
PS D:\Thapar\Sem-4\Artificial Intelligence\Python\Programming>
```


File Edit Selection View Go Run Terminal Help
ai.py - Programming - Visual Studio Code
PROBLEMS 2 DEBUG CONSOLE OUTPUT TERMINAL
Python

```
Computer turn [O]

---------------
| x || x || o |
---------------
| x || o ||   |
---------------
| o ||   ||   |
---------------
YOU LOSE!
PS D:\Thapar\Sem-4\Artificial Intelligence\Python\Programming>
```


File Edit Selection View Go Run Terminal Help
ai.py - Programming - Visual Studio Code
PROBLEMS 2 DEBUG CONSOLE OUTPUT TERMINAL
Python

```
Human turn [X]

---------------
| o ||   ||   |
---------------
| x || o || x |
---------------
|   ||   ||   |
---------------
Use numpad (1..9): 10
Bad choice
Use numpad (1..9): 12
Bad choice
Use numpad (1..9):
```

Python Code

```
#!/usr/bin/env python3
from math import inf as infinity
import platform
import time
from os import system

"""
An implementation of Minimax AI Algorithm in Tic Tac Toe, using Python.
"""

HUMAN = -1
COMP = +1
board = [
    [0, 0, 0],
    [0, 0, 0],
    [0, 0, 0],
]


def evaluate(state):
    """
    Function to heuristic evaluation of state.
    :param state: the state of the current board
    :return: +1 if the computer wins; -1 if the human wins; 0 draw
    """
    if wins(state, COMP):
        score = +1
    elif wins(state, HUMAN):
        score = -1
    else:
        score = 0

    return score


def wins(state, player):
    """
    This function tests if a specific player wins. Possibilities:
    * Three rows    [X X X] or [O O O]
    * Three cols    [X X X] or [O O O]
    * Two diagonals [X X X] or [O O O]
    :param state: the state of the current board
    :param player: a human or a computer
    :return: True if the player wins
    """
    win_state = [
        [state[0][0], state[0][1], state[0][2]],
        [state[1][0], state[1][1], state[1][2]],
        [state[2][0], state[2][1], state[2][2]],
        [state[0][0], state[1][0], state[2][0]],
        [state[0][1], state[1][1], state[2][1]],
        [state[0][2], state[1][2], state[2][2]],
        [state[0][0], state[1][1], state[2][2]],
        [state[2][0], state[1][1], state[0][2]],
    ]
    if [player, player, player] in win_state:
        return True
    else:
        return False


def game_over(state):
    """
    This function test if the human or computer wins
    :param state: the state of the current board
    :return: True if the human or computer wins
    """
    return wins(state, HUMAN) or wins(state, COMP)


def empty_cells(state):
    """
    Each empty cell will be added into cells' list
    :param state: the state of the current board
    :return: a list of empty cells
    """
    cells = []

    for x, row in enumerate(state):
        for y, cell in enumerate(row):
            if cell == 0:
                cells.append([x, y])

    return cells


def valid_move(x, y):
    """
    A move is valid if the chosen cell is empty
    :param x: X coordinate
    :param y: Y coordinate
    :return: True if the board[x][y] is empty
    """
    if [x, y] in empty_cells(board):
        return True
    else:
        return False


def set_move(x, y, player):
    """
    Set the move on board, if the coordinates are valid
    :param x: X coordinate
    :param y: Y coordinate
    :param player: the current player
    """
    if valid_move(x, y):
        board[x][y] = player
        return True
    else:
        return False


def minimax(state, depth, player):
    """
    AI function that choice the best move
    :param state: current state of the board
    :param depth: node index in the tree (0 <= depth <= 9),
    but never nine in this case (see iaturn() function)
    :param player: an human or a computer
    :return: a list with [the best row, best col, best score]
    """
    if player == COMP:
        best = [-1, -1, -infinity]
    else:
        best = [-1, -1, +infinity]

    if depth == 0 or game_over(state):
        score = evaluate(state)
        return [-1, -1, score]

    for cell in empty_cells(state):
        x, y = cell[0], cell[1]
        state[x][y] = player
        score = minimax(state, depth - 1, -player)
        state[x][y] = 0
        score[0], score[1] = x, y

        if player == COMP:
            if score[2] > best[2]:
                best = score  # max value
        else:
            if score[2] < best[2]:
                best = score  # min value
```

```python
    return best


def clean():
    """
    Clears the console
    """
    os_name = platform.system().lower()
    if 'windows' in os_name:
        system('cls')
    else:
        system('clear')


def render(state, c_choice, h_choice):
    """
    Print the board on console
    :param state: current state of the board
    """

    chars = {
        -1: h_choice,
        +1: c_choice,
        0: ' '
    }
    str_line = '---------------'

    print('\n' + str_line)
    for row in state:
        for cell in row:
            symbol = chars[cell]
            print(f'| {symbol} |', end='')
        print('\n' + str_line)


def ai_turn(c_choice, h_choice):
    """
    It calls the minimax function if the depth < 9,
    else it choices a random coordinate.
    :param c_choice: computer's choice X or O
    :param h_choice: human's choice X or O
    :return:
    """
    depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return

    clean()
    print(f'Computer turn [{c_choice}]')
    render(board, c_choice, h_choice)

    if depth == 9:
        x = choice([0, 1, 2])
        y = choice([0, 1, 2])
    else:
        move = minimax(board, depth, COMP)
        x, y = move[0], move[1]

    set_move(x, y, COMP)
    time.sleep(1)


def human_turn(c_choice, h_choice):
    """
    The Human plays choosing a valid move.
    :param c_choice: computer's choice X or O
    :param h_choice: human's choice X or O
    :return:
    """
    depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return

    # Dictionary of valid moves
    move = -1
    moves = {
        1: [0, 0], 2: [0, 1], 3: [0, 2],
        4: [1, 0], 5: [1, 1], 6: [1, 2],
        7: [2, 0], 8: [2, 1], 9: [2, 2],
    }

    clean()
    print(f'Human turn [{h_choice}]')
    render(board, c_choice, h_choice)

    while move < 1 or move > 9:
        try:
            move = int(input('Use numpad (1..9): '))
            coord = moves[move]
            can_move = set_move(coord[0], coord[1], HUMAN)

            if not can_move:
                print('Bad move')
                move = -1
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')


def main():
    """
    Main function that calls all functions
    """
    clean()
    h_choice = ''  # X or O
    c_choice = ''  # X or O
    first = ''  # if human is the first

    # Human chooses X or O to play
    while h_choice != 'O' and h_choice != 'X':
        try:
            print('')
            h_choice = input('Choose X or O\nChosen: ').upper()
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')

    # Setting computer's choice
    if h_choice == 'X':
        c_choice = 'O'
    else:
        c_choice = 'X'

    # Human may starts first
    clean()
    while first != 'Y' and first != 'N':
        try:
            first = input('First to start?[y/n]: ').upper()
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')

    # Main loop of this game
    while len(empty_cells(board)) > 0 and not game_over(board):
        if first == 'N':
            ai_turn(c_choice, h_choice)
            first = ''

        human_turn(c_choice, h_choice)
        ai_turn(c_choice, h_choice)
    # Game over message
    if wins(board, HUMAN):
        clean()
        print(f'Human turn [{h_choice}]')
        render(board, c_choice, h_choice)
        print('YOU WIN!')
    elif wins(board, COMP):
        clean()
        print(f'Computer turn [{c_choice}]')
        render(board, c_choice, h_choice)
        print('YOU LOSE!')
    else:
        clean()
        render(board, c_choice, h_choice)
        print('DRAW!')

    exit()


if __name__ == '__main__':
    main()
```