Two-Player Zero-Sum Game

*Power by*

Adewale Ajibade **&** Chamiso Sirak

Supervised by

Dr. Darsana Josyula

Department of Computer Science

Bowie State University

December, 2024

ABSTRACT

This project presents the development of a 2-player zero-sum game utilizing a game-playing algorithm to enable competitive interaction between a human user and a computer opponent. The algorithm incorporates heuristic evaluation and decision-making strategies to simulate intelligent gameplay. By leveraging principles of artificial intelligence, such as the minimax algorithm and alpha-beta pruning, the system optimizes game strategies to challenge the human participant while maintaining computational efficiency. This work contributes to the understanding of AI in games and serves as a foundation for exploring strategic decision-making in artificial intelligence systems.

Keywords

* Artificial Intelligence
* Game Playing Algorithm
* Zero-Sum Game
* Minimax Algorithm
* Alpha-Beta Pruning
* Human-Computer Interaction
* Strategic Decision-Making

Problem Statement

The increasing complexity of modern computer games highlights the need for intelligent systems capable of engaging human players in strategic gameplay. This project addresses the challenge of designing and implementing a game-playing algorithm for a 2-player zero-sum game, where a computer competes against a human user. The primary objective is to define a structured framework for the game's initial state, represent its states effectively, and model the transitions between these states through permissible moves. An evaluation function utilizing a heuristic approach will be developed to guide the computer's decision-making process, balancing computational efficiency with strategic depth. The project will explore the inherent difficulties of designing a competitive game-playing algorithm, such as handling game state complexity and achieving optimal gameplay, while providing insights into the lessons learned during the development process.

Explanation: Zero-Sum Game

Tic Tac Toe is a zero-sum game because one players gain (win) is exactly equal to the

other player’s loss. If one player wins, the other loses, and if no one wins, the result is a

draw with no advantage for either side.

Methodology / Rules of the game

1. Rules of the Game

* The game is a classic two-player Tic Tac Toe.
* Each player alternates placing their marker (X or O) on a 3x3 grid.
* The objective is to form a straight line (row, column, or diagonal) of three of the same marker before the opponent.

2. Initial State and Representation

Initial State:

The game begins with an empty 3x3 grid.

Representation:

The board is represented as a list of 9 elements, where each index corresponds to a position (0-8). An empty square is denoted by &#39; &#39;, and players’ moves are denoted by X

or O

3. Possible Transitions/Moves

A player can place their marker in any of the empty squares.

After a move, the game state transitions by updating the board and checking for a win or draw condition.

Moves are valid only in squares that are not already occupied.

Partial Game Tree

A screenshot of a computer

Description automatically generated

Heuristics for Evaluation Function

Winning State: Score +1 if the current player wins.

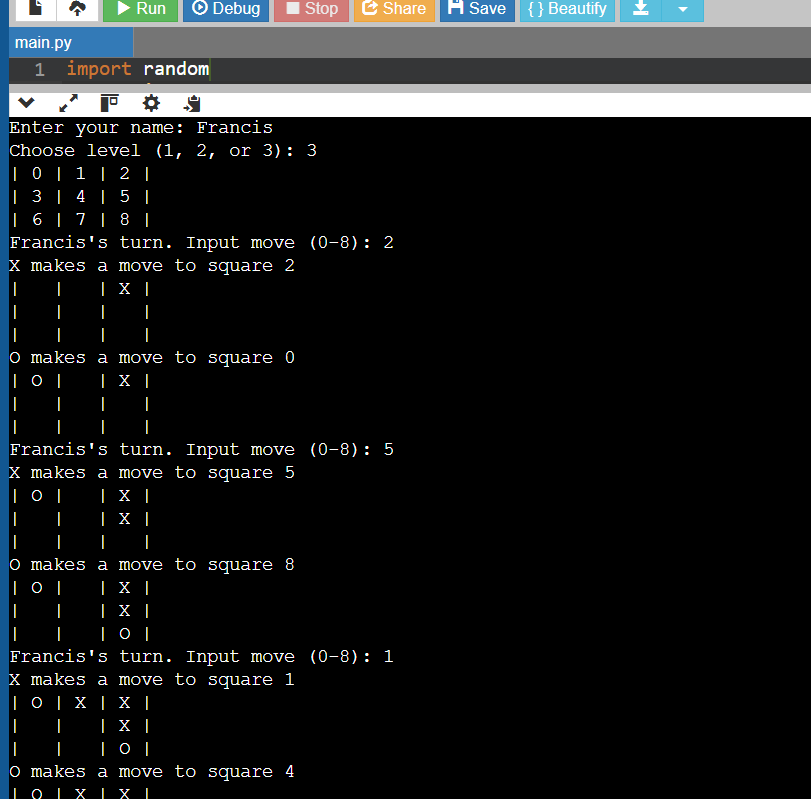
Losing State: Score -1 if the opponent wins.

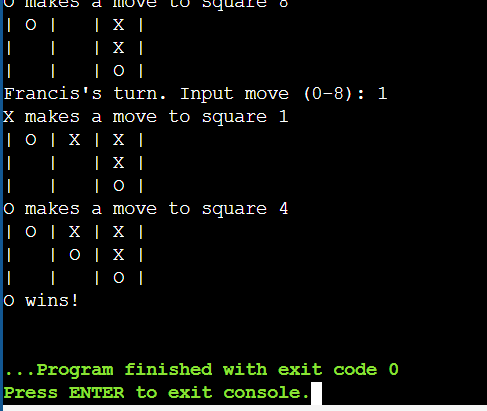
Draw: Score 0.

In-progress States: 0 Evaluate based on the number of two-in-a-row patterns that are open for completion.

Screenshots of the demo

Below screenshot capture the result of the code execution using python programming. The result capture the initial state, the level, and after a few moves, it reveals winning or tying state between the players (human player and computer. At the end of the game the winner emerge with all his marker position either horizontally, vertically or in diagonal





Difficulties Challenges

* To Implement the minimax algorithm with alpha-beta pruning is a challenge
* To balancing performance and depth for the AI is a challenge

Lessons Learned

* Importance of optimizing search algorithms for competitive play is a huge lesson
* The trade-off between AI complexity and responsiveness is another lesson learned

References

1. Eppstein, D. (2004). Games, Puzzles, and Computation.  
  
2. Russell, S., & Norvig, P. (2020). Artificial Intelligence: A Modern Approach (4th Edition).  
  
3. Conway, J. H. (2001).On Numbers and Games.  
  
4. Berlekamp, E. R., Conway, J. H., & Guy, R. K. (1982). Winning Ways for Your Mathematical Plays.  
  
5 Smith, J. M. (1998). Evolution and the Theory of Games.  
  
6 GeeksforGeeks: Tic-Tac-Toe Game Implementation  
  
7 Khan Academy: Game Theory and Tic-Tac-Toe  
  
8 Newell, A., & Simon, H. A. (1956). The Logic Theory Machine: A Complex Information Processing System.  
  
9 Samuel, A. L. (1959). Some Studies in Machine Learning Using the Game of Checkers.  
  
10 Ravi, V., & Meyer, B. (2009). Efficient Algorithms for Tic-Tac-Toe Variants.

#Python Code

import random

# Class to manage the Tic Tac Toe game

class TicTacToe:

    def \_\_init\_\_(self):

        self.board = [' ' for \_ in range(9)]  # Initialize an empty board

        self.current\_winner = None  # Track the winner

    # Print the current board state

    def print\_board(self):

        for row in [self.board[i\*3:(i+1)\*3] for i in range(3)]:

            print('| ' + ' | '.join(row) + ' |')

    # Print a numbered board for reference

    @staticmethod

    def print\_board\_nums():

        number\_board = [[str(i) for i in range(j\*3, (j+1)\*3)] for j in range(3)]

        for row in number\_board:

            print('| ' + ' | '.join(row) + ' |')

    # Return a list of available moves

    def available\_moves(self):

        return [i for i, spot in enumerate(self.board) if spot == ' ']

    # Check if any squares are empty

    def empty\_squares(self):

        return ' ' in self.board

    # Count the number of empty squares

    def num\_empty\_squares(self):

        return self.board.count(' ')

    # Attempt to make a move

    def make\_move(self, square, letter):

        if self.board[square] == ' ':

            self.board[square] = letter

            if self.winner(square, letter):

                self.current\_winner = letter

            return True

        return False

    # Check if the move results in a win

    def winner(self, square, letter):

        row\_ind = square // 3

        row = self.board[row\_ind\*3:(row\_ind+1)\*3]

        if all([spot == letter for spot in row]):

            return True

        col\_ind = square % 3

        column = [self.board[col\_ind+i\*3] for i in range(3)]

        if all([spot == letter for spot in column]):

            return True

        if square % 2 == 0:  # Check diagonals

            diagonal1 = [self.board[i] for i in [0, 4, 8]]

            if all([spot == letter for spot in diagonal1]):

                return True

            diagonal2 = [self.board[i] for i in [2, 4, 6]]

            if all([spot == letter for spot in diagonal2]):

                return True

        return False

# Base player class

class Player:

    def \_\_init\_\_(self, letter, name):

        self.letter = letter

[self.name](http://self.name/) = name

        self.score = 0

    def get\_move(self, game):

        pass

# Human player class

class HumanPlayer(Player):

    def get\_move(self, game):

        valid\_square = False

        val = None

        while not valid\_square:

            square = input([self.name](http://self.name/) + "'s turn. Input move (0-8): ")

            try:

                val = int(square)

                if val not in game.available\_moves():

                    raise ValueError

                valid\_square = True

            except ValueError:

                print('Invalid square. Try again.')

        return val

# Computer player class

class ComputerPlayer(Player):

    def \_\_init\_\_(self, letter, level):

        super().\_\_init\_\_(letter, "Computer")

        self.level = level

    def get\_move(self, game):

        if self.level == 1:

            return random.choice(game.available\_moves())

        else:

            return self.minimax(game, self.letter, depth=2 if self.level == 2 else 4)['position']

    def minimax(self, state, player, depth, alpha=float('-inf'), beta=float('inf')):

        max\_player = self.letter

        other\_player = 'O' if player == 'X' else 'X'

        if state.current\_winner == other\_player:

            return {'position': None, 'score': 1 \* (state.num\_empty\_squares() + 1) if other\_player == max\_player else -1 \* (state.num\_empty\_squares() + 1)}

        elif not state.empty\_squares() or depth == 0:

            return {'position': None, 'score': self.heuristic(state, player)}

        if player == max\_player:

            best = {'position': None, 'score': -float('inf')}

            for move in state.available\_moves():

                state.make\_move(move, player)

                sim\_score = self.minimax(state, other\_player, depth - 1, alpha, beta)

                sim\_score['position'] = move

                best = max(best, sim\_score, key=lambda x: x['score'])

                alpha = max(alpha, best['score'])

                state.board[move] = ' '

                state.current\_winner = None

                if alpha >= beta:

                    break

        else:

            best = {'position': None, 'score': float('inf')}

            for move in state.available\_moves():

                state.make\_move(move, player)

                sim\_score = self.minimax(state, max\_player, depth - 1, alpha, beta)

                sim\_score['position'] = move

                best = min(best, sim\_score, key=lambda x: x['score'])

                beta = min(beta, best['score'])

                state.board[move] = ' '

                state.current\_winner = None

                if alpha >= beta:

                    break

        return best

    def heuristic(self, state, player):

        return 1 if state.current\_winner == player else -1 if state.current\_winner else 0

# Main gameplay logic

def play():

    name = input("Enter your name: ")

    while True:

        level = int(input("Choose level (1, 2, or 3): "))

        if level not in [1, 2, 3]:

            print("Invalid level. Please choose level 1, 2, or 3.")

            continue

        player\_letter = 'X'

        computer\_letter = 'O'

        x\_player = HumanPlayer(player\_letter, name)

        o\_player = ComputerPlayer(computer\_letter, level)

        t = TicTacToe()

        TicTacToe.print\_board\_nums()

        letter = 'X'

        while t.empty\_squares():

            square = (x\_player.get\_move(t) if letter == 'X' else o\_player.get\_move(t))

            if t.make\_move(square, letter):

                print(f'{letter} makes a move to square {square}')

                t.print\_board()

                if t.current\_winner:

                    print(f'{letter} wins!')

                    return

                letter = 'O' if letter == 'X' else 'X'

        print("It's a tie!")

if \_\_name\_\_ == '\_\_main\_\_':

    play()