A MINI PROJECT REPORT

ON

IMPROVISED TIC-TAC-TOE USING MINIMAX AND ALPHA-BETA PRUNING

SEMESTER: IV

SECTION: CSE-A

COURSE: DESIGN AND ANALYSIS OF ALGORITHMS LAB

BY

K. GAUTAM VARMA

(1602-18-733-018)

K KARTHIK

(1602-18-733-024)

UNDER THE GUIDANCE OF

V. PUNNA RAO

ASSISTANT PROFESSOR

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

VASAVI COLLEGE OF ENGINEERING(AUTONOMOUS)

AFFLIATED TO OSMANIA UNIVERSITY

IBRAHIMBAGH

HYDERABAD-500031

ACKNOWLEDGEMENT

We would like to express our heartfelt gratitude to V. Punna Rao, our project guide, for his valuable guidance and constant support, along with his capable instructions and persistent encouragement.

We are grateful to our Head of Department, Dr. T. Adilakshmi, for her steady support and the provision of every resource required for the completion of this project.

We would like to take this opportunity to thank our Principal, Dr. S. V. Ramana, as well as the management of the institute, for having designed an excellent learning atmosphere.

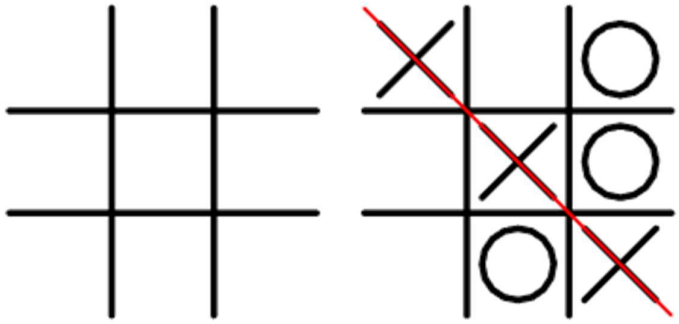
We are thankful to and fortunate enough to get constant encouragement, support and guidance which helped us in successfully completing our project work.

DESIGN AND ANALYSIS OF ALGORITHMS LAB

MINI PROJECT

ABSTRACT

Tic-Tac-Toe game can be played by two players where the square block (3 x 3) can be filled with a cross (X) or a circle (O). The game will toggle between the players by giving the chance for each player to mark their move.



When one of the players make a combination of 3 same markers in a horizontal, vertical or diagonal line the program will display which player has won, whether X or O. In this project, we implement a 3x3 tic-tac-toe game in Java. The game is designed so that user can play tic-tac-toe using Java software.

The program will contain a display function and a select function to place the symbol as well as toggle between the symbols allowing each player a turn to play the game. The program will update after each player makes their move and check for the conditions of game as it goes on.

The game consists of two modes Player vs Player, Player vs AI. The latter mode has 3 modes: easy, medium and difficult. Each mode is implemented differently. The program uses MiniMax Algorithm with Alpha-Beta Pruning to create an improvised TicTacToe.

TABLE OF CONTENTS

LIST OF FIGURES

1. INTRODUCTION

1.1 Mini-Max

Minimax is a kind of backtracking algorithm that is used in decision making and game theory to find the optimal move for a player, assuming that your opponent also plays optimally. It is widely used in two player turn-based games such as Tic-Tac-Toe, Backgammon, Mancala, Chess, etc.

In Minimax the two players are called maximizer and minimizer. The maximizer tries to get the highest score possible while the minimizer tries to do the opposite and get the lowest score possible.

Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

But in the real world when we are creating a program to play Tic-Tac-Toe, Chess, Backgamon, etc. we need to implement a function that calculates the value of the board depending on the placement of pieces on the board. This function is often known as Evaluation Function. It is sometimes also called Heuristic Function.

The evaluation function is unique for every type of game. In this post, evaluation function for the game Tic-Tac-Toe is discussed. The basic idea behind the evaluation function is to give a high value for a board if maximizer‘s turn or a low value for the board if minimizer‘s turn.

2. ANALYSIS AND DESIGN

2.1 Analysis

Let us consider X as the maximizer and O as the minimizer.

Let us build our evaluation function :

1. If X wins on the board we give it a positive value of +10

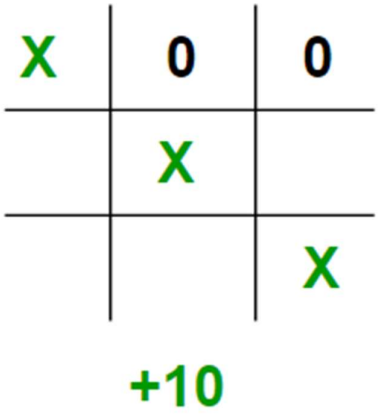


Fig 2.1 X win +10

2. If O wins on the board we give it a negative value of -10

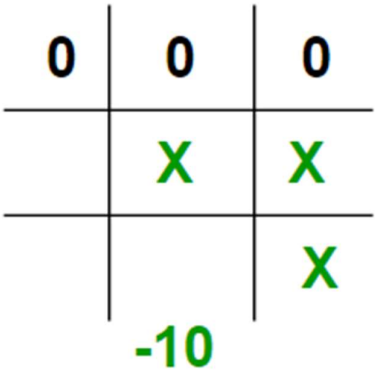


Fig 2.2 O win -10

3. If no one has won or the game results in a draw then we give a value of +0

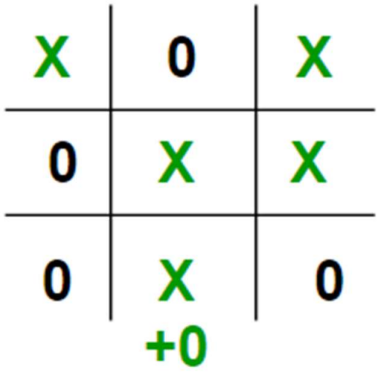


Fig 2.3 Draw Case Score 0

We could have chosen any positive / negative value other than 10. For the sake of simplicity we chose 10 for the sake of simplicity we shall use lower case ‘x’ and lower case ‘o’ to represent the players and an underscore ‘_’ to represent a blank space on the board.

Let us combine what we have learnt so far about minimax and evaluation function to write a proper Tic-Tac-Toe AI (Artificial Intelligence) that plays a perfect game. This AI will consider all possible scenarios and makes the most optimal move.

We shall be introducing a new function called findBestMove(). This function evaluates all the available moves using minimax() and then returns the best move the maximizer can make. The pseudocode is as follows

```
function findBestMove(board):
   bestMove = NULL
   for each move in board :
      if current move is better than bestMove
         bestMove = current move
   return bestMove
```

To check whether or not the current move is better than the best move we take the help of minimax() function which will consider all the possible ways the game can go and returns the best value for that move, assuming the opponent also plays optimally

The code for the maximizer and minimizer in the minimax() function is similar to findBestMove() , the only difference is, instead of returning a move, it will return a value. Here is the pseudocode :

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

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

  if isMaximizingPlayer :
    bestVal = -INFINITY
    for each move in board :
      value = minimax(board, depth+1, false)
      bestVal = max( bestVal, value)
    return bestVal

  else :
    bestVal = +INFINITY
    for each move in board :
      value = minimax(board, depth+1, true)
      bestVal = min( bestVal, value)
    return bestVal
```

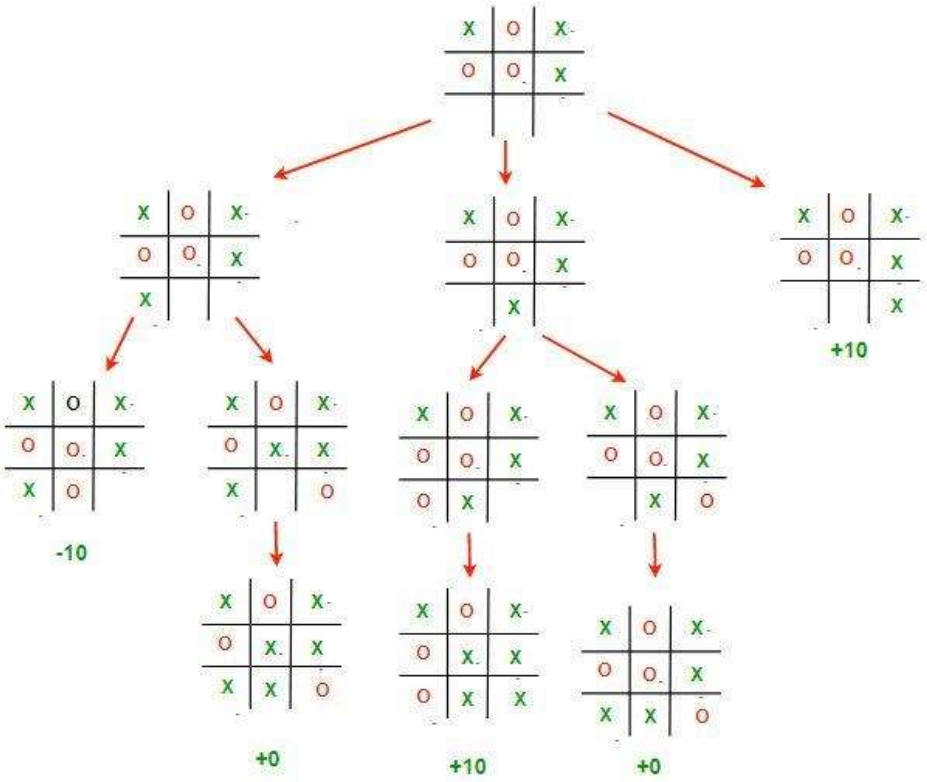


Fig 2.4 Recursion tree Mini-Max Algorithm

This image depicts all the possible paths that the game can take from the root board state. It is often called the Game Tree.

The 3 possible scenarios in the above example are :

Left Move : If X plays [2,0]. Then O will play [2,1] and win the game. The value of this move is -10

Middle Move : If X plays [2,1]. Then O will play [2,2] which draws the game. The value of this move is 0

Right Move : If X plays [2,2]. Then he will win the game. The value of this move is +10;

it is impossible for minimax to compute every possible game state for complex games like Chess. Hence we only compute upto a certain depth and use the evaluation function to calculate the value of the board.

2.2 Alpha-Beta Pruning

Alpha-Beta pruning is not actually a new algorithm, rather an optimization technique for minimax algorithm. It reduces the computation time by a huge factor. This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.

Let's define the parameters alpha and beta.

Alpha is the best value that the maximizer currently can guarantee at that level or above.

Beta is the best value that the minimizer currently can guarantee at that level or above.

```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):

    if node is a leaf node :
        return value of the node

    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each child node :
            value = minimax(node, depth+1, false, alpha, beta)
            bestVal = max( bestVal, value)
            alpha = max( alpha, bestVal)
            if beta <= alpha:
                break
        return bestVal

    else :
        bestVal = +INFINITY
        for each child node :
            value = minimax(node, depth+1, true, alpha, beta)
            bestVal = min( bestVal, value)
```

```
            beta = min( beta, bestVal)
            if beta <= alpha:
                break
        return bestVal
```

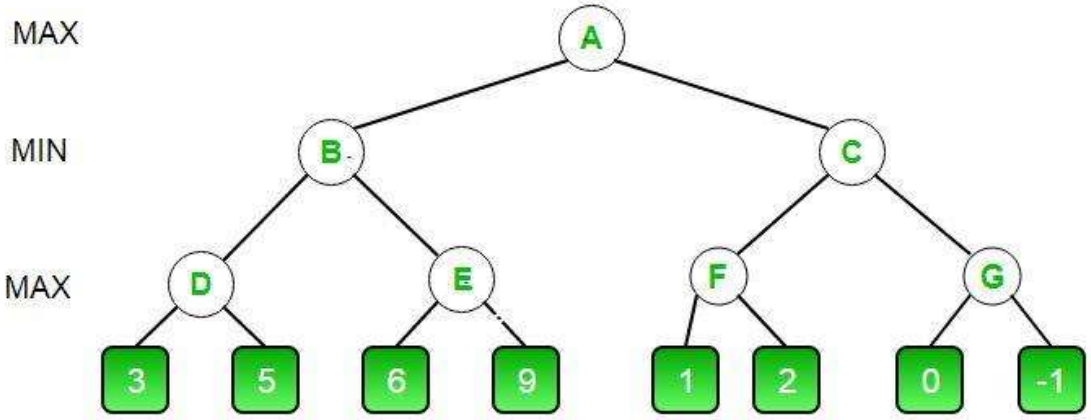


Fig 2.5 Recursion Tree

- The initial call starts from A. The value of alpha here is -INFINITY and the value of beta is +INFINITY. These values are passed down to subsequent nodes in the tree. At A the maximizer must choose max of B and C, so A calls B first
- At B it the minimizer must choose min of D and E and hence calls D first.
- At D, it looks at its left child which is a leaf node. This node returns a value of 3. Now the value of alpha at D is max(-INF, 3) which is 3.
- To decide whether its worth looking at its right node or not, it checks the condition beta<=alpha. This is false since beta = +INF and alpha = 3. So it continues the search.
- D now looks at its right child which returns a value of 5.At D, alpha = max(3, 5) which is 5. Now the value of node D is 5
- D returns a value of 5 to B. At B, beta = min(+INF, 5) which is 5. The minimizer is now guaranteed a value of 5 or lesser. B now calls E to see if he can get a lower value than 5.

- At E the values of alpha and beta is not -INF and +INF but instead -INF and 5 respectively, because the value of beta was changed at B and that is what B passed down to E
- Now E looks at its left child which is 6. At E, alpha = max(-INF, 6) which is 6. Here the condition becomes true. beta is 5 and alpha is 6. So beta<=alpha is true. Hence it breaks and E returns 6 to B
- Note how it did not matter what the value of E's right child is. It could have been +INF or -INF, it still wouldn't matter, We never even had to look at it because the minimizer was guaranteed a value of 5 or lesser. So as soon as the maximizer saw the 6 he knew the minimizer would never come this way because he can get a 5 on the left side of B. This way we dint have to look at that 9 and hence saved computation time.
- E returns a value of 6 to B. At B, beta = min(5, 6) which is 5.The value of node B is also 5

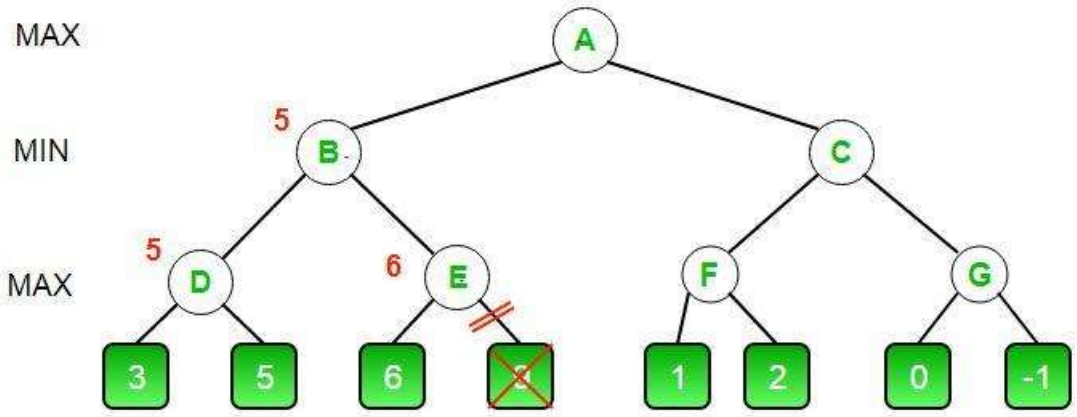


Fig 2.6 Maximizer Minimizer

- **B** returns 5 to **A**. At **A**, alpha = max(-INF, 5) which is 5. Now the maximizer is guaranteed a value of 5 or greater. **A** now calls **C** to see if it can get a higher value than 5.
- At **C**, alpha = 5 and beta = +INF. **C** calls **F**

- At **F**, alpha = 5 and beta = +INF. **F** looks at its left child which is a 1. alpha = max(5, 1) which is still 5.
- **F** looks at its right child which is a 2. Hence the best value of this node is 2. Alpha still remains 5
- **F** returns a value of 2 to **C**. At **C**, beta = min(+INF, 2). The condition beta <= alpha becomes true as beta = 2 and alpha = 5. So it breaks and it does not even have to compute the entire sub-tree of **G**.
- The intuition behind this break off is that, at **C** the minimizer was guaranteed a value of 2 or lesser. But the maximizer was already guaranteed a value of 5 if he choose **B**. So why would the maximizer ever choose **C** and get a value less than 2 ? Again you can see that it did not matter what those last 2 values were. We also saved a lot of computation by skipping a whole sub tree.
- **C** now returns a value of 2 to **A**. Therefore the best value at **A** is max(5, 2) which is a 5.
- Hence the optimal value that the maximizer can get is 5

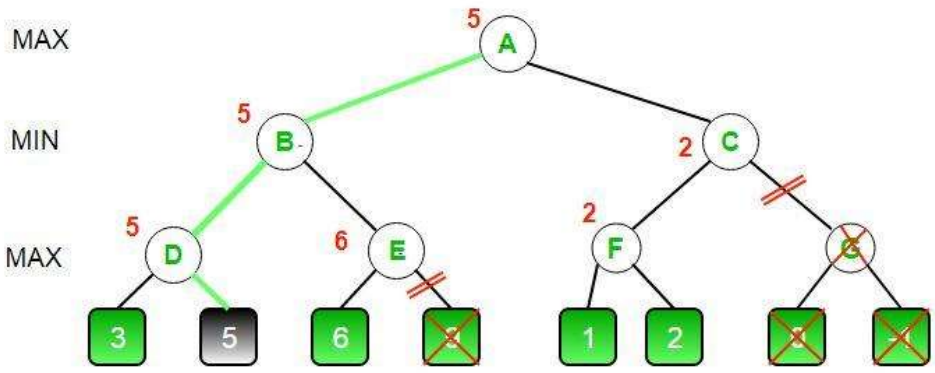


Fig 2.6

Time Complexity of Mini-Max Algorithm

In the worst case, where there is no node to be pruned, the full tree will be examined (or the complete tree up to the cutoff at a depth d). Alpha beta pruning saves, but how much? Notice that in the best case, each node will examine 2b-1 grandchildren to decide on its value. In the worst case, the node would examine b^2 grandchildren. This essentially means that the overall algorithm examined O(b^(d/2)) nodes, the same as a worst-case algorithm whose cutoff is half of d. In practice this is significant.

3. CODE

3.1 Package: ArtificialIntelligence

3.1.1 Filename: Algorithms.java

```
package ArtificialIntelligence;

import TicTacToe.Board;

public class Algorithms {


   private Algorithms() {}

   public static void random (Board board) {
      Random.run(board);
   }
   public static void miniMax (Board board) {
      MiniMax.run(board.getTurn(), board, Double.POSITIVE_INFINITY);
   }

   public static void miniMax (Board board, int ply) {
      MiniMax.run(board.getTurn(), board, ply);
   }


   public static void alphaBetaPruning (Board board) {
      AlphaBetaPruning.run(board.getTurn(), board, Double.POSITIVE_INFINITY);
   }

void alphaBetaPruning (Board board, int ply) {
      AlphaBetaPruning.run(board.getTurn(), board, ply);
   }

   public static void alphaBetaAdvanced (Board board) {
      AlphaBetaAdvanced.run(board.getTurn(), board, Double.POSITIVE_INFINITY);
   }

   public static void alphaBetaAdvanced (Board board, int ply) {
      AlphaBetaAdvanced.run(board.getTurn(), board, ply);
   }

}
```

3.1.2 FileName:AlphaBetaAdvanced.java

```
package ArtificialIntelligence;

import TicTacToe.Board;


class AlphaBetaAdvanced {

   private static double maxPly;


   private AlphaBetaAdvanced() {}

   static void run (Board.State player, Board board, double maxPly) {

      if (maxPly < 1) {
         throw new IllegalArgumentException("Maximum depth must be greater than 0.");
      }

      AlphaBetaAdvanced.maxPly = maxPly;
      alphaBetaPruning(player, board, Double.NEGATIVE_INFINITY,
Double.POSITIVE_INFINITY, 0);
   }


   private static int alphaBetaPruning (Board.State player, Board board, double alpha, double
beta, int currentPly) {
      if (currentPly++ == maxPly || board.isGameOver()) {
         return score(player, board, currentPly);
      }

      if (board.getTurn() == player) {
         return getMax(player, board, alpha, beta, currentPly);
      } else {
         return getMin(player, board, alpha, beta, currentPly);
      }
   }

   /**
    * Play the move with the highest score.
    * @param player       the player that the AI will identify as
    * @param board        the Tic Tac Toe board to play on
    * @param alpha        the alpha value
    * @param beta         the beta value
    * @param currentPly    the current depth
    * @return             the score of the board
    */
   private static int getMax (Board.State player, Board board, double alpha, double beta, int
currentPly) {
      int indexOfBestMove = -1;
```

```
        for (Integer theMove : board.getAvailableMoves()) {

            Board modifiedBoard = board.getDeepCopy();
            modifiedBoard.move(theMove);
            int score = alphaBetaPruning(player, modifiedBoard, alpha, beta, currentPly);

            if (score > alpha) {
                alpha = score;
                indexOfBestMove = theMove;
            }

            if (alpha >= beta) {
                break;
            }
        }

        if (indexOfBestMove != -1) {
            board.move(indexOfBestMove);
        }
        return (int)alpha;
    }

    private static int getMin (Board.State player, Board board, double alpha, double beta, int
currentPly) {
        int indexOfBestMove = -1;

        for (Integer theMove : board.getAvailableMoves()) {

            Board modifiedBoard = board.getDeepCopy();
            modifiedBoard.move(theMove);

            int score = alphaBetaPruning(player, modifiedBoard, alpha, beta, currentPly);

            if (score < beta) {
                beta = score;
                indexOfBestMove = theMove;
            }

            if (alpha >= beta) {
                break;
            }
        }

        if (indexOfBestMove != -1) {
            board.move(indexOfBestMove);
        }
        return (int)beta;
    }
```

```
    private static int score (Board.State player, Board board, int currentPly) {

        if (player == Board.State.Blank) {
            throw new IllegalArgumentException("Player must be X or O.");
        }

        Board.State opponent = (player == Board.State.X) ? Board.State.O : Board.State.X;

        if (board.isGameOver() && board.getWinner() == player) {
            return 10 - currentPly;
        } else if (board.isGameOver() && board.getWinner() == opponent) {
            return -10 + currentPly;
        } else {
            return 0;
        }
    }

}
```

3.1.3 FileName:AlphaBetaPruning.java

```
package ArtificialIntelligence;

import TicTacToe.Board;


class AlphaBetaPruning {

    private static double maxPly;


    private AlphaBetaPruning () {}


    static void run (Board.State player, Board board, double maxPly) {
        if (maxPly < 1) {
            throw new IllegalArgumentException("Maximum depth must be greater than 0.");
        }

        AlphaBetaPruning.maxPly = maxPly;
        alphaBetaPruning(player, board, Double.NEGATIVE_INFINITY,
Double.POSITIVE_INFINITY, 0);
    }


    private static int alphaBetaPruning (Board.State player, Board board, double alpha, double
beta, int currentPly) {
        if (currentPly++ == maxPly || board.isGameOver()) {
            return score(player, board);
```

```
        }

        if (board.getTurn() == player) {
            return getMax(player, board, alpha, beta, currentPly);
        } else {
            return getMin(player, board, alpha, beta, currentPly);
        }
    }

    private static int getMax (Board.State player, Board board, double alpha, double beta, int
currentPly) {
        int indexOfBestMove = -1;

        for (Integer theMove : board.getAvailableMoves()) {

            Board modifiedBoard = board.getDeepCopy();
            modifiedBoard.move(theMove);
            int score = alphaBetaPruning(player, modifiedBoard, alpha, beta, currentPly);

            if (score > alpha) {
                alpha = score;
                indexOfBestMove = theMove;
            }

            // Pruning.
            if (alpha >= beta) {
                break;
            }
        }

        if (indexOfBestMove != -1) {
            board.move(indexOfBestMove);
        }
        return (int)alpha;
    }

    private static int getMin (Board.State player, Board board, double alpha, double beta, int
currentPly) {
        int indexOfBestMove = -1;

        for (Integer theMove : board.getAvailableMoves()) {

            Board modifiedBoard = board.getDeepCopy();
            modifiedBoard.move(theMove);

            int score = alphaBetaPruning(player, modifiedBoard, alpha, beta, currentPly);

            if (score < beta) {
                beta = score;
                indexOfBestMove = theMove;
            }
```

```
            if (alpha >= beta) {
                break;
            }
        }

        if (indexOfBestMove != -1) {
            board.move(indexOfBestMove);
        }
        return (int)beta;
    }


    private static int score (Board.State player, Board board) {
        if (player == Board.State.Blank) {
            throw new IllegalArgumentException("Player must be X or O.");
        }

        Board.State opponent = (player == Board.State.X) ? Board.State.O : Board.State.X;

        if (board.isGameOver() && board.getWinner() == player) {
            return 10;
        } else if (board.isGameOver() && board.getWinner() == opponent) {
            return -10;
        } else {
            return 0;
        }
    }

}
```

3.1.4 FileName:MiniMax.java

```
package ArtificialIntelligence;

import TicTacToe.Board;


class MiniMax {

    private static double maxPly;


    private MiniMax() {}

    static void run (Board.State player, Board board, double maxPly) {
        if (maxPly < 1) {
            throw new IllegalArgumentException("Maximum depth must be greater than 0.");
        }
```

```
    MiniMax.maxPly = maxPly;
    miniMax(player, board, 0);
}


private static int miniMax (Board.State player, Board board, int currentPly) {
    if (currentPly++ == maxPly || board.isGameOver()) {
        return score(player, board);
    }

    if (board.getTurn() == player) {
        return getMax(player, board, currentPly);
    } else {
        return getMin(player, board, currentPly);
    }

}

private static int getMax (Board.State player, Board board, int currentPly) {
    double bestScore = Double.NEGATIVE_INFINITY;
    int indexOfBestMove = -1;

    for (Integer theMove : board.getAvailableMoves()) {

        Board modifiedBoard = board.getDeepCopy();
        modifiedBoard.move(theMove);

        int score = miniMax(player, modifiedBoard, currentPly);

        if (score >= bestScore) {
            bestScore = score;
            indexOfBestMove = theMove;
        }

    }

    board.move(indexOfBestMove);
    return (int)bestScore;
}


private static int getMin (Board.State player, Board board, int currentPly) {
    double bestScore = Double.POSITIVE_INFINITY;
    int indexOfBestMove = -1;

    for (Integer theMove : board.getAvailableMoves()) {

        Board modifiedBoard = board.getDeepCopy();
        modifiedBoard.move(theMove);

        int score = miniMax(player, modifiedBoard, currentPly);
```

```
            if (score <= bestScore) {
                bestScore = score;
                indexOfBestMove = theMove;
            }

        }

        board.move(indexOfBestMove);
        return (int)bestScore;
    }


    private static int score (Board.State player, Board board) {
        if (player == Board.State.Blank) {
            throw new IllegalArgumentException("Player must be X or O.");
        }

        Board.State opponent = (player == Board.State.X) ? Board.State.O : Board.State.X;

        if (board.isGameOver() && board.getWinner() == player) {
            return 10;
        } else if (board.isGameOver() && board.getWinner() == opponent) {
            return -10;
        } else {
            return 0;
        }
    }


}
```

3.1.5 FileName:Random.java

```
package ArtificialIntelligence;

import TicTacToe.Board;


class Random {


    private Random () {}


    static void run (Board board) {
        int[] moves = new int[board.getAvailableMoves().size()];
        int index = 0;
```

```
        for (Integer item : board.getAvailableMoves()) {
            moves[index++] = item;
        }

        int randomMove = moves[new java.util.Random().nextInt(moves.length)];
        board.move(randomMove);
    }

}
```

3.2 Package:TicTacToe

3.2.1 FileName:Board.java

```
package TicTacToe;

import java.util.HashSet;

/**
 * Represents the Tic Tac Toe board.
 */
public class Board {

    static final int BOARD_WIDTH = 3;

    public enum State {Blank, X, O}
    private State[][] board;
    private State playersTurn;
    private State winner;
    private HashSet<Integer> movesAvailable;

    private int moveCount;
    private boolean gameOver;

    /**
     * Construct the Tic Tac Toe board.
     */
    Board() {
        board = new State[BOARD_WIDTH][BOARD_WIDTH];
        movesAvailable = new HashSet<>();
        reset();
    }

    /**
     * Set the cells to be blank and load the available moves (all the moves are
     * available at the start of the game).
     */
    private void initialize () {
```

```
    for (int row = 0; row < BOARD_WIDTH; row++) {
        for (int col = 0; col < BOARD_WIDTH; col++) {
            board[row][col] = State.Blank;
        }
    }

    movesAvailable.clear();

    for (int i = 0; i < BOARD_WIDTH*BOARD_WIDTH; i++) {
        movesAvailable.add(i);
    }
}

/**
 * Restart the game with a new blank board.
 */
void reset () {
    moveCount = 0;
    gameOver = false;
    playersTurn = State.X;
    winner = State.Blank;
    initialize();
}
public boolean move (int index) {
    return move(index% BOARD_WIDTH, index/ BOARD_WIDTH);
}

private boolean move (int x, int y) {

    if (gameOver) {
        throw new IllegalStateException("TicTacToe is over. No moves can be played.");
    }

    if (board[y][x] == State.Blank) {
        board[y][x] = playersTurn;
    } else {
        return false;
    }

    moveCount++;
    movesAvailable.remove(y * BOARD_WIDTH + x);

    // The game is a draw.
    if (moveCount == BOARD_WIDTH * BOARD_WIDTH) {
        winner = State.Blank;
        gameOver = true;
    }

    // Check for a winner.
    checkRow(y);
    checkColumn(x);
```

```
    checkDiagonalFromTopLeft(x, y);
    checkDiagonalFromTopRight(x, y);

    playersTurn = (playersTurn == State.X) ? State.O : State.X;
    return true;
}

public boolean isGameOver () {
    return gameOver;
}

State[][] toArray () {
    return board.clone();
}

public State getTurn () {
    return playersTurn;
}

public State getWinner () {
    if (!gameOver) {
        throw new IllegalStateException("TicTacToe is not over yet.");
    }
    return winner;
}

    public HashSet<Integer> getAvailableMoves () {
    return movesAvailable;
}

private void checkRow (int row) {
    for (int i = 1; i < BOARD_WIDTH; i++) {
        if (board[row][i] != board[row][i-1]) {
            break;
        }
        if (i == BOARD_WIDTH -1) {
            winner = playersTurn;
            gameOver = true;
        }
    }
}

private void checkColumn (int column) {
    for (int i = 1; i < BOARD_WIDTH; i++) {
        if (board[i][column] != board[i-1][column]) {
            break;
        }
        if (i == BOARD_WIDTH -1) {
            winner = playersTurn;
            gameOver = true;
        }
```

```
    }
}


private void checkDiagonalFromTopLeft (int x, int y) {
    if (x == y) {
        for (int i = 1; i < BOARD_WIDTH; i++) {
            if (board[i][i] != board[i-1][i-1]) {
                break;
            }
            if (i == BOARD_WIDTH -1) {
                winner = playersTurn;
                gameOver = true;
            }
        }
    }
}

private void checkDiagonalFromTopRight (int x, int y) {
    if (BOARD_WIDTH -1-x == y) {
        for (int i = 1; i < BOARD_WIDTH; i++) {
            if (board[BOARD_WIDTH -1-i][i] != board[BOARD_WIDTH -i][i-1]) {
                break;
            }
            if (i == BOARD_WIDTH -1) {
                winner = playersTurn;
                gameOver = true;
            }
        }
    }
}
public Board getDeepCopy () {
    Board board             = new Board();

    for (int i = 0; i < board.board.length; i++) {
        board.board[i] = this.board[i].clone();
    }

    board.playersTurn       = this.playersTurn;
    board.winner            = this.winner;
    board.movesAvailable    = new HashSet<>();
    board.movesAvailable.addAll(this.movesAvailable);
    board.moveCount         = this.moveCount;
    board.gameOver          = this.gameOver;
    return board;
}


public String toString () {
    StringBuilder sb = new StringBuilder();
```

```
        for (int y = 0; y < BOARD_WIDTH; y++) {
            for (int x = 0; x < BOARD_WIDTH; x++) {

                if (board[y][x] == State.Blank) {
                    sb.append("-");
                } else {
                    sb.append(board[y][x].name());
                }
                sb.append(" ");

            }
            if (y != BOARD_WIDTH -1) {
                sb.append("\n");
            }
        }

        return new String(sb);
    }

}
```

3.2.2 FileName:Console.java

```
package TicTacToe;

import ArtificialIntelligence.Algorithms;

import java.util.Scanner;


public class Console {

    private Board board;
    private Scanner sc = new Scanner(System.in);


    private Console() {
        board = new Board();
    }

    private void play () {

        System.out.println("Starting a new game.");

        while (true) {
            printGameStatus();
            playMove();
```

```
            if (board.isGameOver()) {
                printWinner();

                if (!tryAgain()) {
                    break;
                }
            }
        }
    }

    /**
     * Handle the move to be played, either by the player or the AI.
     */
    private void playMove () {
        if (board.getTurn() == Board.State.X) {
            getPlayerMove();
        } else {
            Algorithms.alphaBetaAdvanced(board);
        }
    }


    private void printGameStatus () {
        System.out.println("\n" + board + "\n");
        System.out.println(board.getTurn().name() + "'s turn.");
    }


    private void getPlayerMove () {
        System.out.print("Index of move: ");

        int move = sc.nextInt();

        if (move < 0 || move >= Board.BOARD_WIDTH* Board.BOARD_WIDTH) {
            System.out.println("\nInvalid move.");
            System.out.println("\nThe index of the move must be between 0 and "
                    + (Board.BOARD_WIDTH * Board.BOARD_WIDTH - 1) + ", inclusive.");
        } else if (!board.move(move)) {
            System.out.println("\nInvalid move.");
            System.out.println("\nThe selected index must be blank.");
        }
    }


    private void printWinner () {
        Board.State winner = board.getWinner();

        System.out.println("\n" + board + "\n");

        if (winner == Board.State.Blank) {
```

```
            System.out.println("The TicTacToe is a Draw.");
        } else {
            System.out.println("Player " + winner.toString() + " wins!");
        }
    }


    private boolean tryAgain () {
        if (promptTryAgain()) {
            board.reset();
            System.out.println("Started new game.");
            System.out.println("X's turn.");
            return true;
        }

        return false;
    }


    private boolean promptTryAgain () {
        while (true) {
            System.out.print("Would you like to start a new game? (Y/N): ");
            String response = sc.next();
            if (response.equalsIgnoreCase("y")) {
                return true;
            } else if (response.equalsIgnoreCase("n")) {
                return false;
            }
            System.out.println("Invalid input.");
        }
    }

    public static void main(String[] args) {
        Console ticTacToe = new Console();
        ticTacToe.play();
    }

}
```

3.2.3 FileName:Difficulty.java

```
package TicTacToe;

import java.awt.Color;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.GridLayout;
import java.awt.event.*;
```

```
import javax.swing.*;

import TicTacToe.Window.Mode;

public class difficulty extends JFrame implements ActionListener {

	JFrame f;
	JButton b1;
	JButton b2;
	JButton b3;
	public JPanel p1,p2,p3,p4;

	difficulty()
	{
		f=new JFrame();
		f.setSize(600,600);
		f.setResizable(false);
		//f.setLayout(new FlowLayout());
		f.setLayout(new GridLayout(4,1));
		p1=new JPanel();
		p2=new JPanel();
		p3=new JPanel();
		p4=new JPanel();
		JLabel lbl=new JLabel("DIFFICULTY LEVEL");
		//Font font=new Font("Courier",Font.BOLD,24);


		p1.setBackground(Color.DARK_GRAY);
		p2.setBackground(Color.DARK_GRAY);
		p3.setBackground(Color.DARK_GRAY);
		p4.setBackground(Color.DARK_GRAY);
		Font font=new Font("Courier",Font.BOLD,24);
		lbl.setForeground(Color.white);
		lbl.setFont(font);
		p1.add(lbl);
		b1=new JButton("EASY");

		b2=new JButton("MEDIUM");

		b3=new JButton("DIFFICULT");

		//p3.add(lbl);


		b1.setPreferredSize(new Dimension(150,50));
		b1.setBackground(Color.LIGHT_GRAY);
		p2.add(b1);

		b2.setPreferredSize(new Dimension(150,50));
		b2.setBackground(Color.LIGHT_GRAY);
```

```
            p3.add(b2);
            b3.setPreferredSize(new Dimension(150,50));
            b3.setBackground(Color.LIGHT_GRAY);
            p4.add(b3);
            f.add(p1);
            f.add(p2);
            f.add(p3);
            f.add(p4);
            b1.addActionListener(this);
            b2.addActionListener(this);
            b3.addActionListener(this);
            f.setVisible(true);
        }


        public void actionPerformed(ActionEvent e)
        {
            if(e.getSource()==b1) {
                SwingUtilities.invokeLater(() -> new Window(Mode.AI));
                f.dispose();
                Window.flag=1;
                }
            else if(e.getSource()==b2)
            {
                SwingUtilities.invokeLater(() -> new Window(Mode.AI));
                f.dispose();
                Window.flag=2;
            }
            else {
                SwingUtilities.invokeLater(() -> new Window(Mode.AI));
                f.dispose();
                Window.flag=3;

            }

        }

}
```

3.2.4 FileName:IndexPage.java

```
package TicTacToe;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.FlowLayout;
import java.awt.Font;
import java.awt.GridLayout;
```

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.*;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.Toolkit;
import java.io.IOException;



import TicTacToe.Window.Mode;


public class indexPage extends JFrame implements ActionListener {

	public JFrame f;
	public JPanel p1,p2,p3;
	public JButton b1;
	public JButton b2;

	indexPage()
	{
		f=new JFrame("TIC-TAC-TOE");

		f.setSize(600,600);

		p1=new JPanel();
		p2=new JPanel();
		p3=new JPanel();

		p1.setBackground(Color.DARK_GRAY);
		p2.setBackground(Color.DARK_GRAY);
		p3.setBackground(Color.DARK_GRAY);
		Font font=new Font("Courier",Font.BOLD,24);

		JLabel lbl=new JLabel("TIC TAC TOE");
		lbl.setForeground(Color.white);
		lbl.setFont(font);
		p3.add(lbl);
		f.setLayout(new GridLayout(3,1));




		b1=new JButton("Player  Vs  AI");
		b1.setPreferredSize(new Dimension(150,50));
		b1.setBackground(Color.LIGHT_GRAY);
		p1.add(b1);
```

```
            b1.addActionListener(this);
            b2=new JButton("Player Vs Player");
            b2.setPreferredSize(new Dimension(150,50));
            b2.setBackground(Color.LIGHT_GRAY);

            p2.add(b2);
            b2.addActionListener(this);
            f.add(p3);
            f.add(p1);
            f.add(p2);
            f.setResizable(false);
            f.setVisible(true);

        }

        @Override
        public void actionPerformed(ActionEvent e) {

            if(e.getSource()==b1) {
                    new difficulty();
              f.dispose();}
            else
            {
                    SwingUtilities.invokeLater(() -> new Window(Mode.Player));
                    f.dispose();

            }

        }


}
```

3.2.5 FileName:Window.java

```
package TicTacToe;

import javax.imageio.ImageIO;
import javax.swing.*;
import java.awt.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;

import ArtificialIntelligence.*;
```

```
public class Window extends JFrame {

    public static final int WIDTH = 600;
    public static final int HEIGHT = 600;
    public static  int flag;

    public Board board;
    public Panel panel;
    public BufferedImage imageBackground, imageX, imageO;

    public enum Mode {Player, AI}
    public Mode mode;


    public Point[] cells;

    public static final int DISTANCE = 100;


    public Window () {
        this(Mode.AI);
    }


    public Window (Mode mode) {
        this.mode = mode;
        board = new Board();
        loadCells();
        panel = createPanel();
        setWindowProperties();
        loadImages();
    }


    public void loadCells () {
        cells = new Point[9];

        cells[0] = new Point(109, 109);
        cells[1] = new Point(299, 109);
        cells[2] = new Point(489, 109);
        cells[3] = new Point(109, 299);
        cells[4] = new Point(299, 299);
        cells[5] = new Point(489, 299);
        cells[6] = new Point(109, 489);
        cells[7] = new Point(299, 489);
        cells[8] = new Point(489, 489);
    }


    public void setWindowProperties () {
```

```
    setResizable(false);
    pack();
    setTitle("TIC TAC TOE");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
}


public Panel createPanel () {
    Panel panel = new Panel();
    Container cp = getContentPane();
    cp.add(panel);
    panel.setPreferredSize(new Dimension(WIDTH, HEIGHT));
    panel.addMouseListener(new MyMouseAdapter());
    return panel;
}


public void loadImages () {
    imageBackground = getImage("background");
    imageX = getImage("x");
    imageO = getImage("o");
}


public static BufferedImage getImage (String path) {

    BufferedImage image;

    try {
        path = ".." + File.separator + "Assets" + File.separator + path + ".png";
        image = ImageIO.read(Window.class.getResource(path));
    } catch (IOException ex) {
        throw new RuntimeException("Image could not be loaded.");
    }

    return image;
}


public class Panel extends JPanel {

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);
        paintTicTacToe((Graphics2D) g);
    }


    public void paintTicTacToe (Graphics2D g) {
        setProperties(g);
```

```
    paintBoard(g);
    paintWinner(g);
}


public void setProperties (Graphics2D g) {
    g.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
            RenderingHints.VALUE_INTERPOLATION_BILINEAR);
    g.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
    g.drawImage(imageBackground, 0, 0, null);


    g.drawString("", 0, 0);
}


public void paintBoard (Graphics2D g) {
    Board.State[][] boardArray = board.toArray();

    int offset = 20;

    for (int y = 0; y < 3; y++) {
        for (int x = 0; x < 3; x++) {
            if (boardArray[y][x] == Board.State.X) {
                g.drawImage(imageX, offset + 190 * x, offset + 190 * y, null);
            } else if (boardArray[y][x] == Board.State.O) {
                g.drawImage(imageO, offset + 190 * x, offset + 190 * y, null);
            }
        }
    }
}


public void paintWinner (Graphics2D g) {
    if (board.isGameOver()) {
        g.setColor(new Color(255, 255, 255));
        g.setFont(new Font("TimesRoman", Font.PLAIN, 50));

        String s;

        if (board.getWinner() == Board.State.Blank) {
            s = "Draw";
        } else {
            s = board.getWinner() + " Wins!";
        }

        g.drawString(s, 300 - getFontMetrics(g.getFont()).stringWidth(s)/2, 315);

    }
}
```

```
}

public class MyMouseAdapter extends MouseAdapter {
   @Override
   public void mousePressed(MouseEvent e) {
      super.mouseClicked(e);

      if (board.isGameOver()) {
         board.reset();
         panel.repaint();
      } else {
         playMove(e);
      }

   }


   public void playMove (MouseEvent e) {
      int move = getMove(e.getPoint());

      if (!board.isGameOver() && move != -1) {
         boolean validMove = board.move(move);
         if (mode == Mode.AI && validMove && !board.isGameOver()&&  flag==1) {

            Algorithms.random(board);
         }
         if (mode == Mode.AI && validMove && !board.isGameOver()&&  flag==2) {

            Algorithms.miniMax(board);
         }
         if (mode == Mode.AI && validMove && !board.isGameOver()&&  flag==3) {

            Algorithms.alphaBetaAdvanced(board);
         }
         panel.repaint();
      }
   }


   public int getMove (Point point) {
      for (int i = 0; i < cells.length; i++) {
         if (distance(cells[i], point) <= DISTANCE) {
            return i;
         }
      }
      return -1;
   }


   public double distance (Point p1, Point p2) {
      double xDiff = p1.getX() - p2.getX();
```

```
            double yDiff = p1.getY() - p2.getY();

            double xDiffSquared = xDiff*xDiff;
            double yDiffSquared = yDiff*yDiff;

            return Math.sqrt(xDiffSquared+yDiffSquared);
        }
    }

    public static void main(String[] args) {

            new indexPage();
    }

}
```

4. OUTPUT

Fig 4.1 Main Window

- **Player vs AI**
  X-player, O-AI

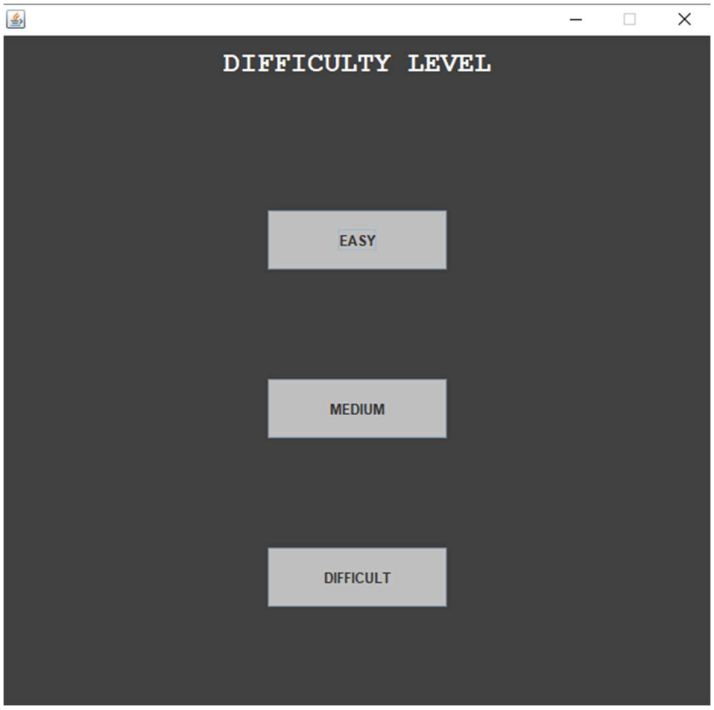


Fig 4.2 Difficulty Level

- **Easy mode**

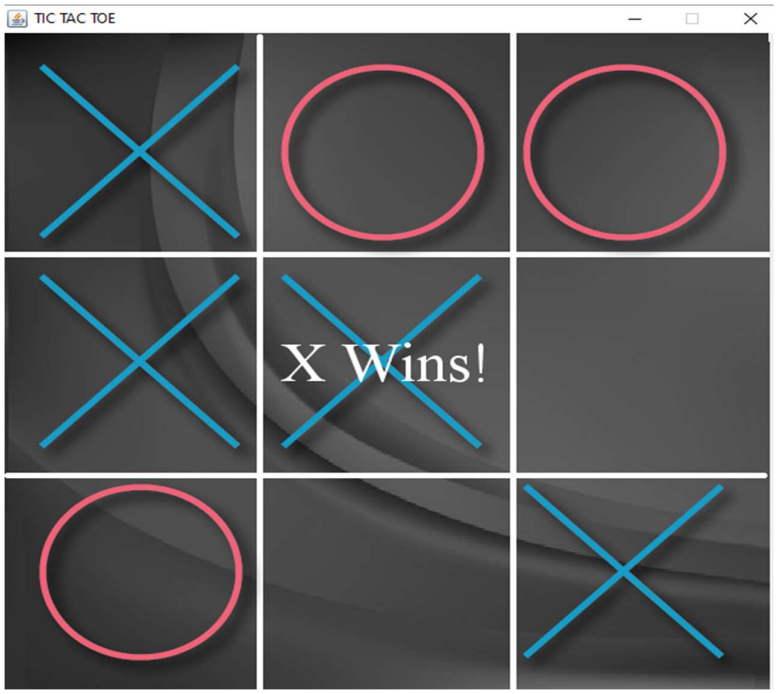


Fig 4.3 Player Wins Case

- **Medium mode**

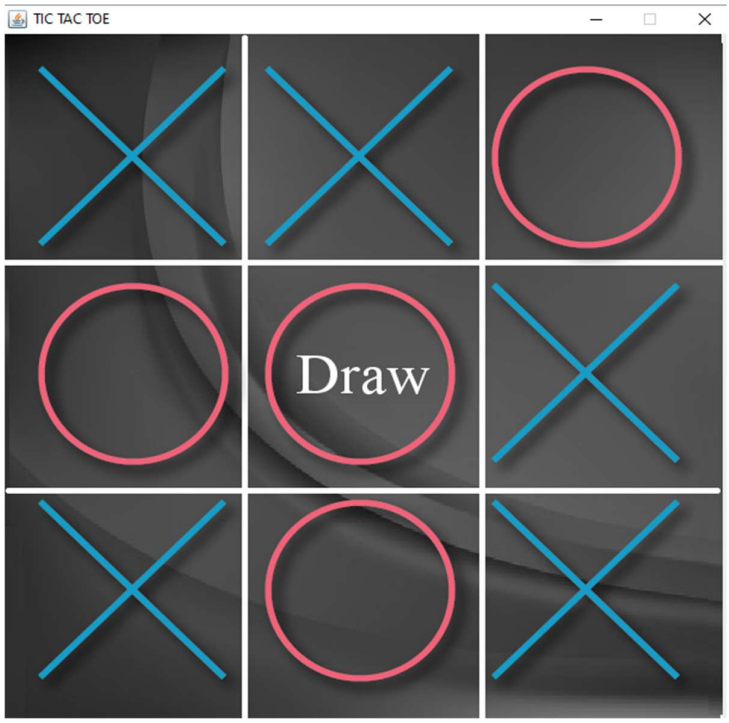


Fig 4.4 Draw Case

Difficult mode

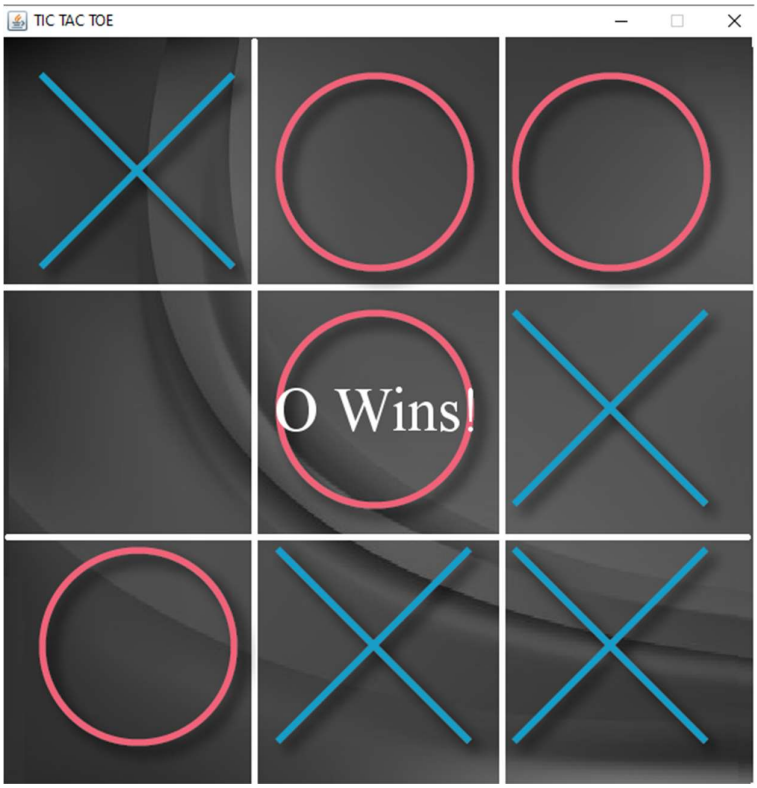


Fig 4.5 Computer Wins Case

Player vs Player
Unpredictable

5. TEST CASES

1. Testing with Random

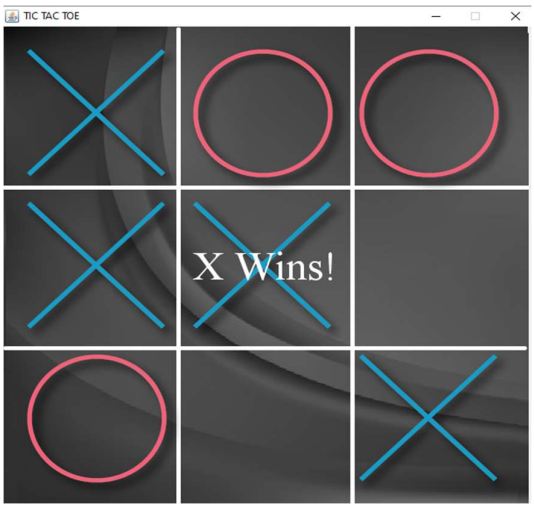


Fig 5.1 70% Chances of Winning

2. Testing with Mini-Max

With Mini-max algorithm chances of winning are reduced to 30%. hence, the game level becomes moderate.

3. Testing with Mini-Max with alpha-beta pruning

Inclusion of alpha-beta pruning with Mini-max algorithm reduces chances of winning to 5%. hence, the game level becomes difficult. So, there are more chances of draw.

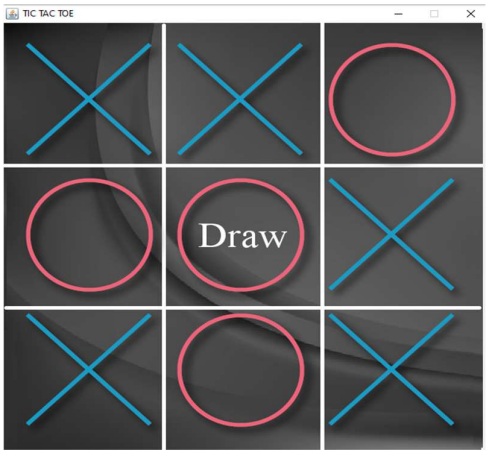


Fig 5.2 More chances for draw.

6. CONCLUSION

Overall program works without any bugs and is able to use Mini-Max Algorithm. We used MiniMax Algorithm along with Alpha-Beta Pruning in a specific case in order to create a perfect unbeatable AI in the difficult mode.

Over main idea is not just to create a game, but to indicate the effectiveness of Unbeatable AI by using the power of Algorithms which has a wide scope in the future.

7. REFERENCES

[1]https://towardsdatascience.com/tic-tac-toe-creating-unbeatable-ai-with-minimax-algorithm-8af9e52c1e7d

[2] https://www.neverstopbuilding.com/blog/minimax

[3] https://www.youtube.com/watch?v=trKjYdBASyQ

[4]https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/

[5] https://www.youtube.com/watch?v=xBXHtz4Gbdo