Tic-Tac-Toe for n-by-n board using Minimax, Alpha Beta and Minimax Cutoff

Dilip Subramaniam, Prajval Bavi

UNCC

Problem Description

Implement a general game-playing agent for two-player deterministic games, using (1) minimax with alpha-beta pruning, and (2) minimax-cutoff (i.e., with cutoff test to replace terminal test and with evaluation function to replace utility function) with alpha-beta pruning. Apply it to the planar 3*3 Tic-Tac-Toe game and extend it to the planar n*n (n > 3 and n is odd) Tic-Tac-Toe game. You need to have a friendly graphical user interface to allow a human user to play the game with your algorithm.

Explanation of tic-tac-toe:

Tic-tac-toe is a type of **Adversarial** game where two players will try and maximize their own utility and try to win the game. **X and O,** who take turns marking the spaces in a 3×3 grid. The player who succeeds in placing three of their marks in a horizontal, vertical, or diagonal row wins the game. The game can be extended to n*n grid (n is odd) and the player who places n of their marks in horizontal, vertical, or diagonal row wins the game.

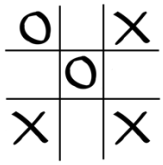


Fig 1. A Game of Tic Tac Toe

Minimax Algorithm

The **minimax algorithm** computes the minimax decision from the current state. It uses a simple recursive computation of the minimax values of each successor state, directly implementing the defining equations. The recursion proceeds all the way down to the leaves of the tree, and then the minimax values are **backed up** through the tree as the recursion unwinds.

The minimax algorithm performs a complete depth-first exploration of the game tree. 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 $O(b^m)$. The space

complexity is O(bm) for an algorithm that generates all actions at once, or O(m) for an algorithm that generates actions one at a time.

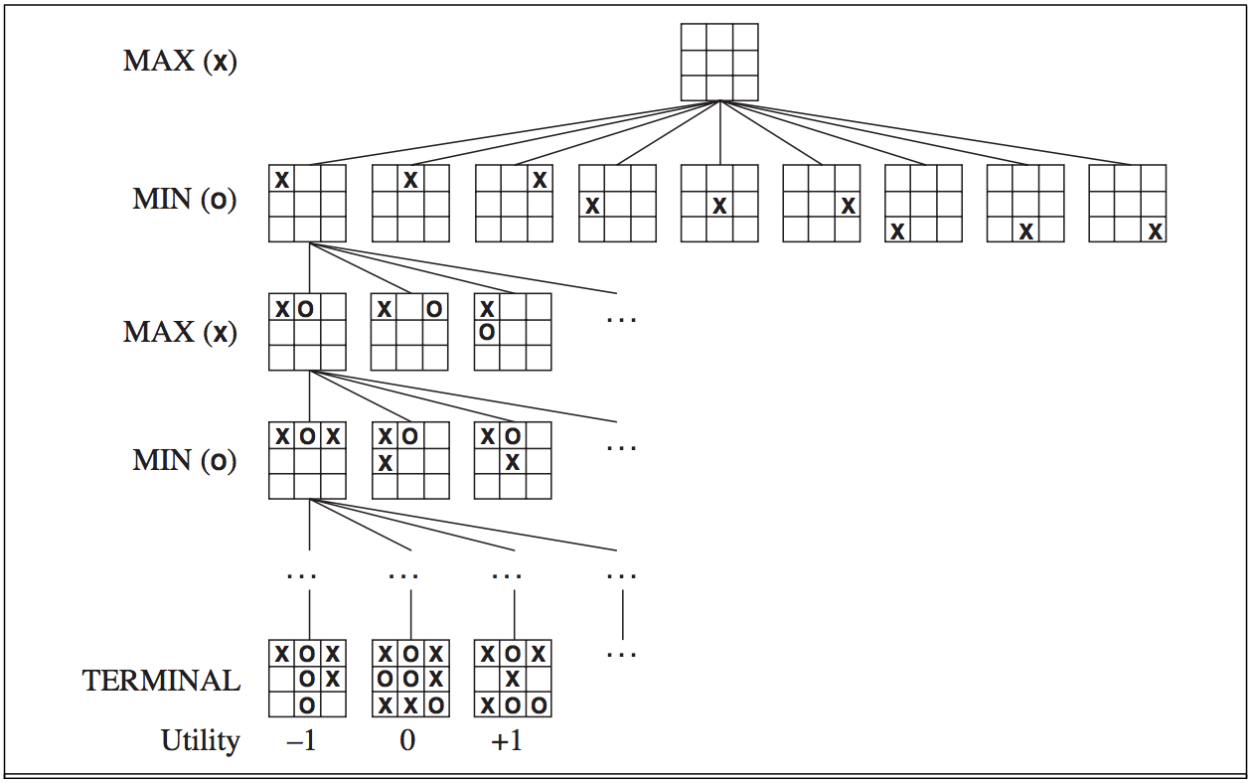


Fig 2: Minimax Illustration

Alpha Beta Pruning

The problem with minimax search is that the number of game states it has to examine is exponential in the depth of the tree.

Alpha–beta pruning, when applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision. Alpha–beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves.

The general principle is: consider a node n somewhere in the such that Player has a choice of moving to that node. If Player has a better choice either at the parent node of n or at any choice point further up, then n *will never be reached in actual play.* So, once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.

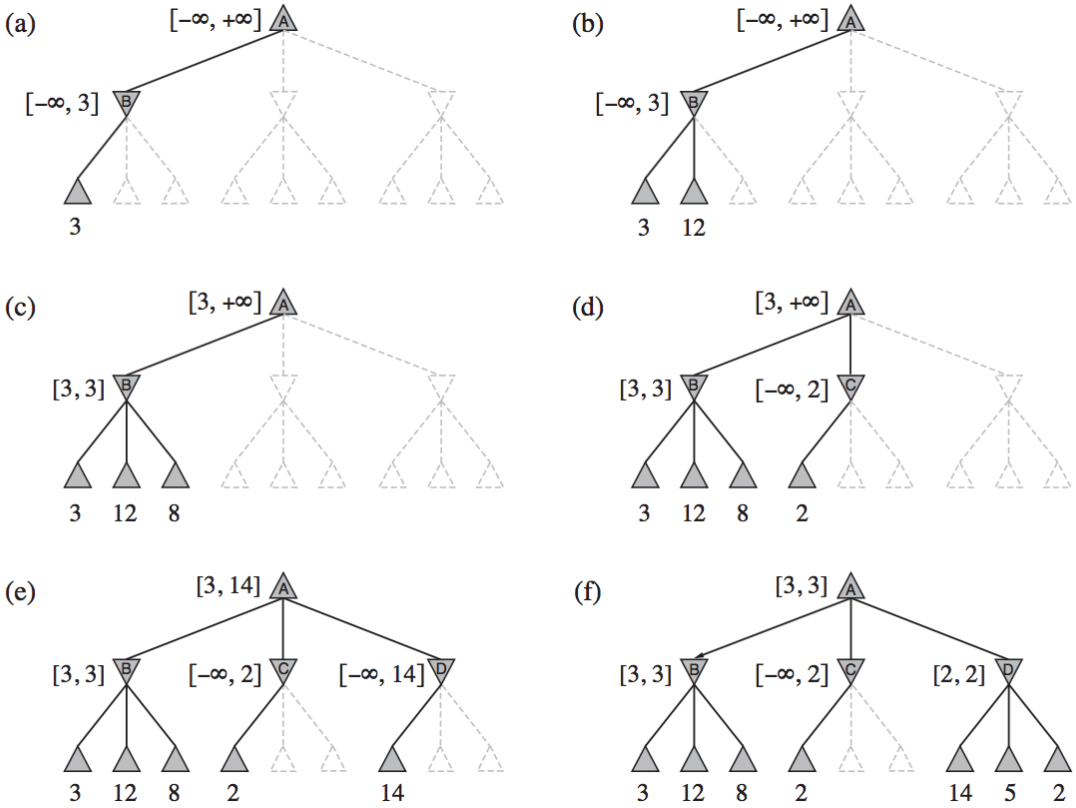


Fig 3: Alpha-Beta Pruning Illustration

Minimax Cutoff:

The next step is to modify Alpha-Beta so that it will call the heuristic evaluation function when it is appropriate to cut off the search. The cutoff is defined by the user so that the computation can be kept in limit and the speed of the agent will be faster in response to the way the other player plays.

The utility function is replaced by a heuristic evaluation function, which estimates the position's utility, and replace the **terminal test** by a **cutoff test** that decides when to apply evaluation function. That gives us the following for heuristic minimax for state s and maximum depth d:

$$\text{H-MINIMAX}(s, d) = \begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s, d) \\ \max_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in Actions(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if PLAYER}(s) = \text{MIN}. \end{cases}$$

Tic-Tac-Toe for n-by-n board using Minimax, Alpha Beta and Minimax Cutoff

Objective of Project

To implement an AI agent using Minimax algorithm to play tic-tac-toe and maximize its utility to win/draw the game. The second stage is to improve upon the implemented Minimax algorithm by using the Alpha Beta pruning so that the speed of the AI agent will increase. This further can be improved by using the Minimax Cutoff with Alpha Beta Pruning which will limit the tree expansion to the depth specified, for the project we have used two heuristics for Minimax Cutoff and different depth for both.

Model/Approach Taken for Project

1. We divided the project into 4 parts:
    a. UI design and implementation
    b. Minimax Algorithm Implementation
    c. Alpha-Beta Pruning Implementation
    d. Minimax Cutoff Implementation
2. To make the project more user friendly we decided to go with using Android Application Development as it already provides a stack of UI and the implementation of the UI won't take more effort and also, we will be able to concentrate implementing the actual algorithm.
3. After the UI design was done and tested, we went ahead with deciding on how to handle the important functions like generate the successors for the given state and designing the utility/evaluation function. Following approaches were agreed upon:
    a. Utility function for Minimax – when the terminal state is reached assign +1000 for win, -1000 for loss and 0 for draw.
    b. Utility function for Alpha Beta – when the terminal state is reached assign +1000-depth for win, -1000-depth for loss and 0 for draw.
    c. Evaluation function for Minimax Cutoff 1 – when the depth of 4 is reached, we will assign a weighted sum of "features:"

        (w1 * feature1) + (w2 * feature2) + ... + (wn * featuren).

        We define X_n as the number of rows, columns, or diagonals with exactly n X's and no O's. Similarly, O_n is the number of rows, columns, or diagonals with just n O's.

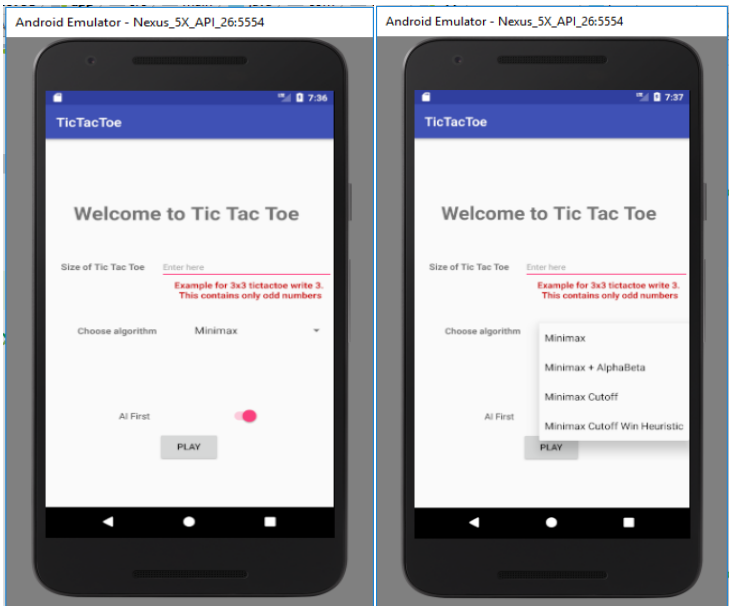
        Example:

        For a 3*3 tic tac toe game, the evaluation function is defined as: $3X_2 + X_1 - (3O_2 + O_1)$.

   d. Evaluation function for Minimax Cutoff 2 – when the depth of 2 is reached, the evaluation function will calculate the difference between the number of winning rows. columns and diagonals for X and winning rows, columns and diagonals for O.
4. Upon the completion of the evaluation/utility functions the actual algorithms were implemented, and the results were tested with various combination.

Testing Results

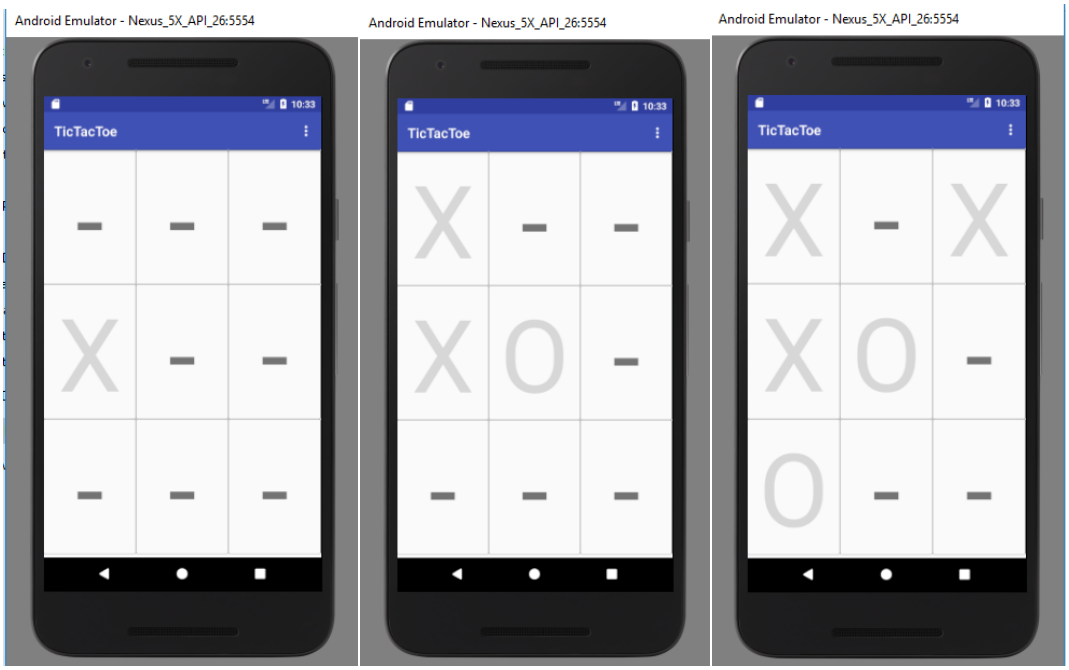
1. MainActivity of the application where user will be able to select the Boardsize of tic-tac-toe, the algorithm it can play against and if the player should play first or the user is going to play first.

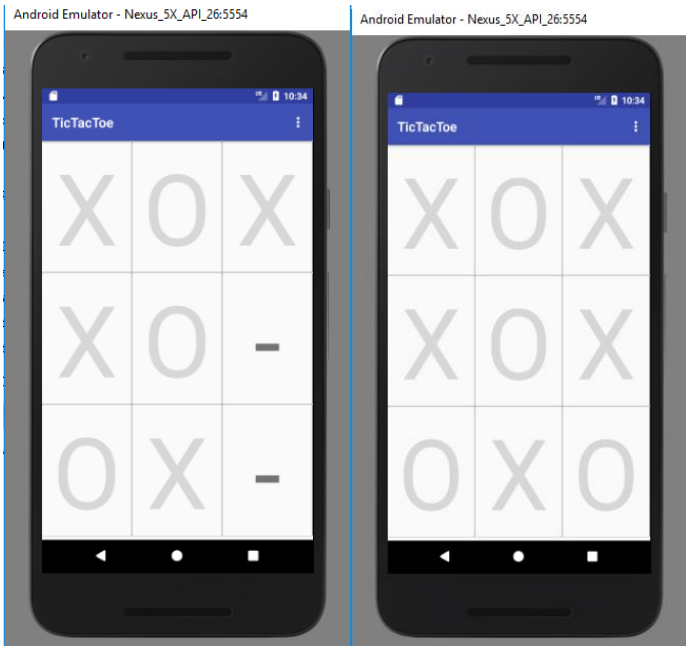


2. Various Boards available for the user to play:

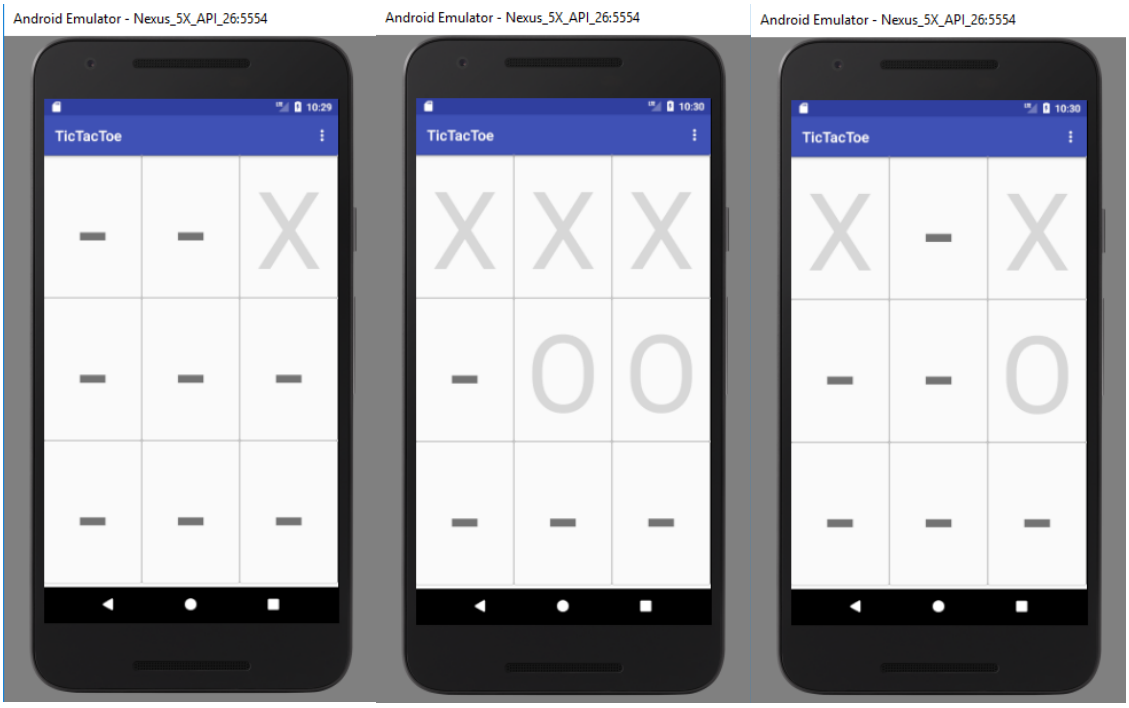
3. Following is the demonstration of when the AI plays first for 3x3 tic-tac-toe and with Minimax Cutoff.

AI and the User play optimally, so the game ends in draw.





Playing against a dumb user, AI wins



Number of States Generated for various combinations:

1. **3*3 Game**

	AI WINS		DRAW	
	Player First (Dumb player)	AI First (Dumb player)	Player First (Optimal Player)	AI First (Optimal Player)
Minimax	60663	7692	60694	8314
Minimax Alpha Beta	9783	1586	9809	1882
Cutoff 1(depth = 4)	2555	1087	2581	1217
Cutoff 2 (depth = 2)	348	200	360	218

2. 5*5 Game

	AI WINS		DRAW	
	Player First (Dumb player)	AI First (Dumb player)	Player First (Optimal Player)	AI First (Optimal Player)
Cutoff 1 (depth = 4)	1027085	420001	131774	11979
Cutoff 2 (depth = 2)	14712	16946	16883	17031

3. 7*7 Game

	AI WINS		DRAW	
	Player First (Dumb player)	AI First (Dumb player)	Player First (Optimal Player)	AI First (Optimal Player)
Cutoff 2 (depth = 2)	202702	154620	229345	234221

Future Improvements

1. Move Ordering

   The effectiveness of alpha–beta pruning is highly dependent on the order in which the states are examined. Sometimes we will not be able to prune any successors of at all because the worst successors (from the point of view of MIN) were generated first. So, evaluating the generated nodes first and finding the best node can help us prune the nodes more effectively.

2. Maintain a Transposition Table:

   We know repeated states in the search tree can cause an exponential increase in search cost. We can use **transposition table**; it is essentially identical to the *explored* list in GRAPH. Using a transposition table can have a dramatic effect, sometimes as much as doubling the reachable search depth in chess. On the other hand, if we are evaluating a million

nodes per second, at some point it is not practical to keep *all* of them in the transposition table. Various strategies have been used to choose which nodes to keep and which to discard.

3. Implement Monte Carlo Algorithm

Division of Work

Dilip –

1. Implementation of minimax algorithm
2. Work on UI
3. Implementation of one evaluation function for minimax cutoff
4. Writing the report

Prajval –

1. Implementation of alpha beta pruning
2. Work on UI
3. Implementation of one evaluation function for minimax cutoff
4. Writing the report

References

Artificial Intelligence – A Modern Approach by Stuart Russel and Peter Norvig

Images are obtained from Google Images.

External Documentation for the Program:

The program consists of 8 Classes:

1. **Board:** The Board Class is used to represent the Nodes of the game tree.
2. **BoardActivity:** This class gets the board size and algorithm from the user and creates the board and call the other classes accordingly.
3. **MainActivity:** This class is the main UI of the application
4. **MenuActionClass:** Adds a Menu to the BoardActivity
5. **Minimax:** Implements the minimax algorithm
6. **MinimaxAlphBeta:** Implements the minimax algorithm along with alpha beta pruning
7. **MinimaxCutoffHeuritsicWin:** Implements minimax along with an evaluation function.
8. **MinimaxCutoff:** Implements minimax with a different evaluation function (hopefully a better one)**.**

Board Class:

The Board class represent board of the tic tac toe game. It has 3 variables:

1. boardSize: Represents the size of the board
2. boardState: Represents the state of board
3. value: Represents the value of the board

This class has the getter and setter methods for the above three variables. It also has following additional methods:

1. checkWinner: This takes an input the board state and the current player and returns true of the current player is winner.
2. convertArray: This method converts the boardState from string to an array.
3. checkX: It returns the number of X's and O's already set in the board
4. printBoardState: Prints the current board state (Used for debugging purpose)
5. printBoardStateMax: Prints the current board state (Used for debugging purpose)
6. printBoardStateMin: Prints the current board state (Used for debugging purpose)
7. findAvaibleIndex: This method returns an array of available index for moves.
8. generateSuccessors: Returns the successor board for the current board state.

9. boardFullCheck: This method returns true if the board is full else returns false.

BoardActivity Class:
This is the class which handles the input from the MainActivity and calls the required algorithms and sets the board to the new state as returned by the algorithm**.**

gameOver: checks if the game is over and shows dialog if the user wants to start the game again.
findBestBoardMinimaxCutoffWinHeuristic: calls the **MinimaxCutoffHeuristicWin**
findBestBoardMinimaxCutoff: calls the **MinimaxCutoff**
findBestBoardMinimaxAlphaBeta: calls the **MinimaxAlphBeta**
findBestBoardMinimax: calls the **Minimax**

MainActivity Class:
This is the main class for UI and this is responsible for the home page of android application. This has three methods:

1. validateInputs: This method checks the board size entered by user and returns true if boardSize is an odd number greater than 1.
2. setOnClickListener: This method starts a new Activity on click of button and this is where the board is displayed with the number of tiles as the user has selected.
3. setOnCheckedChangeListener: These checks if the user has selected to play first or AI will go first. The switch is by default set to ON, so that AI will play the first move and If the switch is OFF then the player will play the first move.

MenuActionClass:
This class is used to add a Menu to the BoardActivity, where the user has the option to restart the game from in between and a whole new game will begin.

Minimax Class:
The minimax class implements the minimax algorithm.
It recursively calls the minimax depending on the current player. if the current player is X, the algorithm will find the max available board and If the current player is O, the algorithm will find the min available board. Before recursively calling minimax, the terminal state check is done at the start.

The terminal checks are checkWinner for player X(AI) and O(User) and whether the board state is full, if either of the conditions gets satisfied then it will return the value as decided if winner is AI(X), assign +1000, if winner is User(O) assign -1000 and if the state is draw assign 0 and return.

MinimaxAlphBeta Class:
This is same as the Minimax class and adds the alpha beta pruning to reduce the number of states generated, so that the whole tree is not necessarily explored every time and only the necessary ones are explored. This will speed up the algorithm and return the result quickly.

MinimaxCutoffHeuristicWin Class:
It is usually impossible to solve games completely. This means we cannot search entire game tree. Instead we must estimate cost of internal nodes. We do this using an evaluation function. Then explore game tree using combination of evaluation function and search.
Evaluation function is used to evaluate the "goodness" of a configuration of the game. Unlike in heuristic search where the evaluation function was a non-negative estimate of the cost from the start node to a goal and passing through the given node, here the evaluation function, also called the static evaluation function estimates board quality in leading to a win for one player.
A simple evaluation function for Tic-Tac-Toe is to count number of lines where X can win and subtract number of lines where O can win. The Value of evaluation function at start of game is zero, because on an empty game board there are 8 possible winning lines for both X and O.
Eval = (number of lines where X can win) – (number of lines where O can win)
In this class, we have used this evaluation function to give a utility value to the board when the board depth reaches 2.
This class has two methods:

1. **minimaxCutoffEval**: This method assigns utility values to the board state based on the evaluation method discussed above.
2. **miniMax**: This method implements the minimax algorithm along with alpha beta pruning.

MinimaxCutoff Class:
This class implements a better evaluation function for the tic tac toe game. It can be specified as a weighted sum of "features:" (w1 * features1) + (w2 * features2) + ... + (wn * featuresn).

We define X_n as the number of rows, columns, or diagonals with exactly n X's and no O's. Similarly, O_n is the number of rows, columns, or diagonals with just n O's.

Example:

For a 3*3 tic tac toe game, the evaluation function is defined as:

Eval $=3X_2 +X_1- 3O_2 - O_1$.

In this class, we have used this evaluation function to give a utility value to the board when the board depth reaches 4. This class has two methods:

1. **minimaxCutoffEval**: This method assigns utility values to the board state based on the evaluation method discussed above.
2. **miniMax**: This method implements the minimax algorithm along with alpha beta pruning.

Source Code:

----------Start of Source Code----------

----------Start of Board.java----------

```
package com.example.tictactoe;

import android.util.Log;

import java.util.ArrayList;
import java.util.Arrays;

/**
 * Created by prajvalb on 11/14/17.
 */

public class Board {
    private int boardSize;
    private String[] boardState;
    private int value;

    //Constructor for new Board
    public Board(int bSize, String[] boardState) {
        this.boardState = boardState;
        this.boardSize = bSize;
    }

    public Board(){

    }

    //Getters and Setters for the board
    public int getBoardSize() {
        return boardSize;
    }

    public void setBoardSize(int boardSize) {
        this.boardSize = boardSize;
    }

    public String[] getBoardState() {
        return boardState;
    }

    public void setBoardState(String[] boardState) {
        this.boardState = boardState;
    }

    public int getValue() {
        return value;
    }
```

```
public void setValue(int value) {
    this.value = value;
}

/**
 *
 * Returns the boolean if the currentPlayer is the winner
 * Checks the rows, columns and diagonals for the winner and return true if the winner is found
 * else returns false
 *
 * @param  state  Board state of the current boardState
 * @param currentPlayer the currentPlayer to check winner for
 * @return boolean
 *
 **/
public boolean checkWinner(String[] state, String currentPlayer){
    int row = 0, col = 0, i = 0;
    String playerCurrent = currentPlayer;
    String [][] a = convertArray(state);
    int n = a.length;
    for ( row = 0; row < n; row++)
    {
        for ( col = 0; col < 1; col++)
        {
            while (a[row][col] == playerCurrent)
            {
                col++;
                i++;
                if (i == n)
                {
                    return true;
                }
            }
            i = 0;
        }
    }

    for ( col = 0; col < n; col++)
    {
        for ( row = 0; row < 1; row++)
        {
            while (a[row][col] == playerCurrent)
            {
                row++;
                i++;
                if (i == n)
                {
                    return true;
                }
            }
            i = 0;
        }
    }
```

```
    for ( col = 0; col < 1; col++)
    {
       for ( row = 0; row < 1; row++)
       {
          while (a[row][col] == playerCurrent)
          {
             row++;
             col++;
             i++;
             if (i == n)
             {
                return true;
             }
          }
          i = 0;
       }
    }

    for ( col = n-1; col > 0+(n-2); col--)
    {
       for ( row = 0; row < 1; row++)
       {
          while (a[row][col] == playerCurrent)
          {
             row++;
             col--;
             i++;
             if (i == n)
             {
                return true;
             }
          }
          i = 0;
       }
    }

    return false;
}

/**
 *
 * Coverts the boardState(represented as array of string into String matrix of
 * boardSize * boardSize for the checkWinner function
 *
 * @param  d  boardState which needs to be converted to matrix
 * @return a  String matrix
 **/
public String[][] convertArray ( String[] d)
{
    String [][] a = new String[BoardActivity.boardSize][BoardActivity.boardSize];

    int k=0;
```

```
        for(int i=0;i<BoardActivity.boardSize;i++){
            for(int j=0;j<BoardActivity.boardSize;j++){
                a[i][j]=d[k++];
            }
        }
        return a;
    }

    /**
     *
     * Returns the boolean true for the boardState is full and if no more moves are avaiable
     * else returns false
     *
     * @param  state  Board state of the current boardState
     * @return boolean
     *
     **/
    public boolean boardFullCheck(String[] state){
        for (int i = 0; i < state.length; i++){
            if (state[i].equals("-"))
                return false;
        }
        return true;
    }

    /**
     *
     * Returns the ArrayList of the Boards for the current boardState and the currentPlayer
     *
     * @param  board  Board state of the current boardState
     * @param currentPlayer the currentPlayer to generate the future moves
     * @return            ArrayList of the Boards i.e the next possible states a player can play
     *
     **/
    ArrayList<Board> generateSuccessors(Board board, String currentPlayer){
        ArrayList<Board> tmpBoard = new ArrayList<Board>();
        Board originalBoard = board;
        Integer[] gloabalIndex = new Integer[BoardActivity.boardSize*BoardActivity.boardSize -
checkX(board.getBoardState())];
        int count = 0;
        String[] tmpState = board.getBoardState();

        int iterateValue = BoardActivity.boardSize*BoardActivity.boardSize - checkX(board.getBoardState());
        for (int i = 0; i < iterateValue; i++) {
            int[] indices = findAvaibleIndex(tmpState, gloabalIndex);
            gloabalIndex[count++] = indices[0];
            String[] maybeState = new String[board.getBoardState().length];
            for (int s = 0; s < maybeState.length; s++)
                maybeState[s] = board.getBoardState()[s];

            maybeState[indices[0]] = currentPlayer;
            tmpState = maybeState;
            Board newBoard = new Board(BoardActivity.boardSize, tmpState);
```

```
            tmpBoard.add(newBoard);
            tmpState = newBoard.getBoardState();
        }
        return tmpBoard;
    }

    /**
     *
     * Returns the avaible index for generating the new Boards
     *
     * @param  mboardState  Board state to generate the avaible index
     * @param globalI globalIndex for which the states are already generated
     * @return the Array of integers where the board states are yet to be generated
     *
     **/
    int[] findAvaibleIndex(String[] mboardState, Integer[] globalI){
        int[] tmpBoardState = new int[mboardState.length];
        int index = 0;

        for (int i = 0; i < mboardState.length; i++){
            if (mboardState[i].equals("-")) {
                if (!(Arrays.asList(globalI).contains(i)))
                    tmpBoardState[index++] = i;


            }
        }
        return tmpBoardState;
    }

    /**
     *
     * Prints the boardState of the current/requested boardState
     *
     * @param  state  current/requested boardState to be printed
     * @return void
     **/
    public void printBoardState(String[] state){
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < state.length; i++){
            sb.append(state[i]);
        }
        Log.d("prb_log", "Printing the board state " + sb.toString());
    }

    /**
     *
     * Prints the boardState of the current/requested boardState
     *
     * @param  state  current/requested boardState to be printed
     * @return void
     **/
    public void printBoardStateMax(String[] state){
```

```
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < state.length; i++){
            sb.append(state[i]);
        }
        Log.d("prb_log", "Printing the board state Max " + sb.toString());
    }

    /**
     *
     * Prints the boardState of the current/requested boardState
     *
     * @param  state  current/requested boardState to be printed
     * @return void
     **/
    public void printBoardStateMin(String[] state){
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < state.length; i++){
            sb.append(state[i]);
        }
        Log.d("prb_log", "Printing the board state Min " + sb.toString());
    }

    /**
     *
     * Returns the count for the number of x's and o's already set in the given board state
     *
     * @param  state  current/requested boardState to be printed
     * @return count for the number os x's and o's
     **/
    public int checkX(String[] state){
        int count = 0;
        for (int i = 0; i < state.length; i++){
            if (state[i]!= null)
                if (state[i].equals("X") || state[i].equals("O"))
                    count++;
        }
        return count;
    }

}
```

----------End of Board.java----------

----------Start of BoardActivity.java----------

```
package com.example.tictactoe;

import android.annotation.SuppressLint;
import android.app.AlertDialog;
import android.content.DialogInterface;
import android.content.Intent;
import android.graphics.drawable.GradientDrawable;
import android.os.Bundle;
```

```
import android.util.Log;
import android.util.TypedValue;
import android.view.Gravity;
import android.view.View;
import android.widget.TableLayout;
import android.widget.TableRow;
import android.widget.TextView;
import android.widget.Toast;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.Random;

public class BoardActivity extends MenuActionClass {
    private TableLayout mainTableLayout;
    public static int boardSize;
    static int drawset = 0;
    static int totalCount;
    AlertDialog.Builder showBuilder;
    AlertDialog showalertDialog;
    String[] boardState;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_board);


        Intent intent = getIntent();
        Bundle bundle = intent.getBundleExtra(MainActivity.BUNDLEVAL);

        Random random = new Random();
        //Gets the value like name of the algorithm and the boardsize from the MainActivity
        //If for some reason the app get crashed and is not able to send the data, this will
        //finish the current activity and it will go to the MainActivity.
        //boardSize - the size of the board, eg: for 3x3 it will be 3
        //boardState - the x and o's of the current board
        //algoName - which algorithm the user has choosen to play against
        if (bundle != null) {
            totalCount = 0;
            String[] algoDetails = bundle.getStringArray(MainActivity.BOARDSIZE);
            boardSize = Integer.parseInt(algoDetails[0]);
            boardState = new String[boardSize*boardSize];
            final String algoName = algoDetails[1];
            int k = 0;
            for (int some = 0; some < boardSize*boardSize; some++){
                boardState[k++] = "-";
            }

            //If the user has selected the AI should play first, then we will generate a random
            //number between 1-(boardSize*boardSize) and the AI will play its first move there
            if (algoDetails[2].equals("1")) {
                int someindex = random.nextInt(boardSize*boardSize);
```

```
            boardState[1] = "X";
        }

        //This will be used to set the layout for the tic tac toe board
        //Setting up the number of rows and columns according to the boardSize the player has
        //selected
        float tableLayoutWeight = (1 / (float)boardSize);
        int textSizeSetting = (450/boardSize);
        mainTableLayout = (TableLayout) findViewById(R.id.mainTableLayout);
        TableRow.LayoutParams layoutParamsTR = new
TableRow.LayoutParams(TableRow.LayoutParams.MATCH_PARENT, 0, tableLayoutWeight);
        GradientDrawable gd = new GradientDrawable();
        gd.setCornerRadius(5);
        gd.setStroke(1, 0xFF000000);
        int count = 1;

        //Actual setup for the rows and columns, each and every cell will be treated as a
        //textview and clicking on the textview will disable the specific textview and you wont
        //to click that textview again and also the textview the AI has player will also be
        //disabled so as to not consider the clicking events on those textview
        for (int i = 0; i < boardSize; i++) {
            TableRow tableRow = new TableRow(BoardActivity.this);
            int sizeDP = (int) TypedValue.applyDimension(TypedValue.COMPLEX_UNIT_DIP, 2,
getResources().getDisplayMetrics());
            layoutParamsTR.setMargins(sizeDP, sizeDP, sizeDP, sizeDP);
            tableRow.setLayoutParams(layoutParamsTR);

            for (int j = 0; j < boardSize; j++) {
                final TextView textView = new TextView(BoardActivity.this);
                TableRow.LayoutParams textLayoutParams = new TableRow.LayoutParams(0,
TableRow.LayoutParams.MATCH_PARENT, tableLayoutWeight);
                textLayoutParams.gravity = Gravity.CENTER_VERTICAL | Gravity.CENTER_HORIZONTAL;
                textView.setLayoutParams(textLayoutParams);
                textView.setTextSize(TypedValue.COMPLEX_UNIT_SP, textSizeSetting);
                textView.setTextAlignment(View.TEXT_ALIGNMENT_CENTER);
                textView.setId(count++);
                //OnClickListener for the TextView is implemented here, It will check which
                //cell has been clicked and will updated the boardState accordingly to the
                //set TextView and it will disable the corresponding TextView
                //Following steps will be done after a click on any TextView from the user
                //1. Update the boardState, and call checkWinner for "O" i.e the user
                //if the checkWinner returns true, then user is the winner and finish the game
                //2. Check if the boardState is full, ie if no addtional moves are available,
                // then the game is a draw and notify the user with the same.
                //3. Call the AI algorithm depending on the choice of algorithm the user has
                //selected and call that specific method to do the further processing
                //4. findBestBoardMinimax*alog* will return a new board state that the AI thinks
                //is the best move, update the boardState accordingly
                //5. Check if the updated boardState is the winner for AI, and update the game
                //status accordingly
                //6. Start the whole process for the new OnClick of a new TextView all over again
                textView.setOnClickListener(new View.OnClickListener() {
                    @SuppressLint("ResourceType")
```

```
        @Override
        public void onClick(View view) {
            ((TextView) view).setText("O");
            ((TextView) view).setEnabled(false);

            for (int i = 0; i < mainTableLayout.getChildCount(); i++){
                TableRow tmpTR = (TableRow) mainTableLayout.getChildAt(i);
                for (int j = 0; j < tmpTR.getChildCount(); j++){
                    boardState[textView.getId()-1] = textView.getText().toString();
                    if (checkWinner(boardState, "O")){
                        Toast.makeText(BoardActivity.this, "You won ? This should not have happened",
Toast.LENGTH_SHORT).show();
                        gameOver();
                    } else if (boardFullCheck(boardState)){
                        Toast.makeText(BoardActivity.this, "Eww! Its a Draw", Toast.LENGTH_SHORT).show();
                        drawset = 1;
                        gameOver();
                    }

                }
            }
            Log.d("prb_log", "Printing the board state after player plays");
            printBoardState(boardState);

            Board board = new Board(boardSize, boardState);
            if (textView.getText().toString().equals("O")) {
                if (algoName.equals("Minimax")) {
                    Log.d("prb_log", "Going for minimax");
                    findBestBoardMinimax(board);
                } else if (algoName.equals("Minimax + AlphaBeta")){
                    Log.d("prb_log", "Going for minimax + AlphaBeta");
                    findBestBoardMinimaxAlphaBeta(board);
                } else if (algoName.equals("Minimax Cutoff")){
                    Log.d("prb_log", "Going for minimaxCutoff");
                    findBestBoardMinimaxCutoff(board);
                } else {
                    Log.d("prb_log", "Going for minimaxCutoff wint number of win ");
                    findBestBoardMinimaxCutoffWinHeuristic(board);
                }

                for (int j = 1; j < (boardSize*boardSize +1); j++){
                    ((TextView)findViewById(j)).setText(boardState[j-1]);
                    if (!boardState[j-1].equals("-"))
                        ((TextView) findViewById(j)).setEnabled(false);

                }

                if (checkWinner(boardState, "X")){
                    Toast.makeText(BoardActivity.this, "AI won !", Toast.LENGTH_SHORT).show();
                    gameOver();
                } else if(boardFullCheck(boardState)){
                    if (drawset == 0) {
                        Toast.makeText(BoardActivity.this, "Eww! Its a Draw", Toast.LENGTH_SHORT).show();
```

```
                        gameOver();
                    }
                }


            }
        }
    });
    textView.setBackground(gd);
    tableRow.addView(textView);

}
mainTableLayout.addView(tableRow);
}
for (int j = 1; j < (boardSize*boardSize + 1); j++) {
    ((TextView) findViewById(j)).setText(boardState[j - 1]);
    if (!boardState[j-1].equals("-"))
        ((TextView) findViewById(j)).setEnabled(false);

}

} else {
    finish();
}
}


/**
 *
 * Prints the boardState of the current/requested boardState
 *
 * @param  state  current/requested boardState to be printed
 * @return void
 **/
public void printBoardState(String[] state){
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < state.length; i++){
        sb.append(state[i]);
    }
    Log.d("prb_log", "Printing the board state " + sb.toString());
}

/**
 *
 * Coverts the boardState(represented as array of string into String matrix of
 * boardSize * boardSize for the checkWinner function
 *
 * @param  d  boardState which needs to be converted to matrix
 * @return a  String matrix
 **/
public String[][] convertArray ( String[] d)
{
    String [][] a = new String[boardSize][boardSize];
```

```
    int k=0;
    for(int i=0;i<boardSize;i++){
        for(int j=0;j<boardSize;j++){
            a[i][j]=d[k++];
        }
    }
    return a;
}


/**
 *
 * Updates the boardState with the play the AI thinks is most suitable according to the
 * algorithm, here the algorithm is simple Minimax, which will generate the whole tree
 * and assign the utility value at the leaf nodes.
 * The values assigned are +1000 for winner of AI, -1000 for winner of user and 0 if draw
 * The values are then backed up to the original list of available generatedBoards and the best
 * of the board is selected.
 *
 * @param  board  Board state of the current boardState
 * @return void
 *
 **/
public void findBestBoardMinimax(Board board){
    ArrayList<Board> someStates = board.generateSuccessors(board, "X");
    int winnerVal = (int) Double.NEGATIVE_INFINITY;
    for (Board mState : someStates) {
        int tmpwinnerVal = Minimax.miniMax(mState,  "O");

        if (tmpwinnerVal  > winnerVal) {
            boardState = mState.getBoardState();
            winnerVal = tmpwinnerVal;
                    printBoardState(boardState);

        }
    }
}

/**
 *
 * Updates the boardState with the play the AI thinks is most suitable according to the
 * algorithm, here the algorithm is simple Minimax with AlpaBeta pruning,
 * which will generate the whole tree and assign the utility value at the leaf nodes.
 * The values assigned are +1000-depth for winner of AI, -1000-depth for winner of user and
 * 0 if draw The values are then backed up to the original list of available generatedBoards
 * and the best of the board is selected.
 *
 * @param  board  Board state of the current boardState
 * @return void
 *
 **/
public void findBestBoardMinimaxAlphaBeta(Board board){
```

```
        ArrayList<Board> someStates = board.generateSuccessors(board, "X");
        int winnerVal = (int) Double.NEGATIVE_INFINITY;
        for (Board mState : someStates) {
            int tmpwinnerVal = MinimaxAlphBeta.miniMax(mState, 0, (int)Double.NEGATIVE_INFINITY,
(int)Double.POSITIVE_INFINITY, "O");
             if (tmpwinnerVal  > winnerVal) {
              boardState = mState.getBoardState();
              winnerVal = tmpwinnerVal;

              printBoardState(boardState);

            }
        }
    }

    /**
     *
     * Updates the boardState with the play the AI thinks is most suitable according to the
     * algorithm, here the algorithm is simple Minimax with Cutoff,
     * this will go to the depth as defined in the algorithm, here the algorithm is going at depth 4
     * When the depth is reached, the evalution function will return the value for that board
     * and these values are backed up to the original list of available generatedBoards
     * Heuristic of Evalution Function:
     * Checks the number of rows, columns and diagonals with only x's and o's
     * evaluationValue = 3 * X2 + 1 * X1 - 3 * O2 + 1 * O1
     * where, X2 - number of rows, columns and diagonals with only 2 number of X's
     *        O2 - number of rows, columns and diagonals with only 2 number of O's
     *        X1 - number of rows, columns and diagonals with only 1 number of X's
     *        O2 - number of rows, columns and diagonals with only 1 number of O's
     *
     * @param  board  Board state of the current boardState
     * @return void
     *
     **/
    public void findBestBoardMinimaxCutoff(Board board){
        ArrayList<Board> someStates = board.generateSuccessors(board, "X");
        int winnerVal = (int) Double.NEGATIVE_INFINITY;
        for (Board mState : someStates) {
            int tmpwinnerVal = MinimaxCutoff.miniMax(mState, 0, (int)Double.NEGATIVE_INFINITY,
(int)Double.POSITIVE_INFINITY, "O");


            if (tmpwinnerVal  > winnerVal) {
                boardState = mState.getBoardState();
                winnerVal = tmpwinnerVal;

                printBoardState(boardState);

            }
        }
    }

    /**
```

```
    *
    * Updates the boardState with the play the AI thinks is most suitable according to the
    * algorithm, here the algorithm is simple Minimax with Cutoff,
    * this will go to the depth as defined in the algorithm, here the algorithm is going at depth 2
    * When the depth is reached, the evalution function will return the value for that board
    * and these values are backed up to the original list of available generatedBoards
    * Heuristic of Evalution Function:
    * Checks the number of rows, columns and diagonals with victories for X and O
    * evaluationValue = winX - winO
    * where, winX - number of rows, columns and diagonals with X can win
    *       winO - number of rows, columns and diagonals with O can win
    *
    * @param  board  Board state of the current boardState
    * @return void
    *
    **/
   public void findBestBoardMinimaxCutoffWinHeuristic(Board board){
      ArrayList<Board> someStates = board.generateSuccessors(board, "X");
      int winnerVal = (int) Double.NEGATIVE_INFINITY;
      for (Board mState : someStates) {
         int tmpwinnerVal = MinimaxCutoffHeuristicWin.miniMax(mState, 0, (int)Double.NEGATIVE_INFINITY,
(int)Double.POSITIVE_INFINITY, "O");

         if (tmpwinnerVal  > winnerVal) {
            boardState = mState.getBoardState();
            winnerVal = tmpwinnerVal;
            printBoardState(boardState);

         }
      }
   }


   /**
    *
    * Returns the boolean if the currentPlayer is the winner
    * Checks the rows, columns and diagonals for the winner and return true if the winner is found
    * else returns false
    *
    * @param  state  Board state of the current boardState
    * @param currentPlayer the currentPlayer to check winner for
    * @return boolean
    *
    **/
   public boolean checkWinner(String[] state, String currentPlayer){
      int row = 0, col = 0, i = 0;
      String playerCurrent = currentPlayer;
      String [][] a = convertArray(state);
      int n = a.length;
      for ( row = 0; row < n; row++)
      {
         for ( col = 0; col < 1; col++)
         {
```

```
            while (a[row][col] == playerCurrent)
            {
                col++;
                i++;
                if (i == n)
                {
                    return true;
                }
            }
            i = 0;
        }
    }

    for ( col = 0; col < n; col++)
    {
        for ( row = 0; row < 1; row++)
        {
            while (a[row][col] == playerCurrent)
            {
                row++;
                i++;
                if (i == n)
                {
                    return true;
                }
            }
            i = 0;
        }
    }

    for ( col = 0; col < 1; col++)
    {
        for ( row = 0; row < 1; row++)
        {
            while (a[row][col] == playerCurrent)
            {
                row++;
                col++;
                i++;
                if (i == n)
                {
                    return true;
                }
            }
            i = 0;
        }
    }

    for ( col = n-1; col > 0+(n-2); col--)
    {
        for ( row = 0; row < 1; row++)
        {
            while (a[row][col] == playerCurrent)
```

```
            {
                row++;
                col--;
                i++;
                if (i == n)
                {
                    return true;
                }
            }
            i = 0;
        }
    }

    return false;
}


/**
 *
 * Displays toasts if the winner is found or if the boardState is full
 * If the user then decides to restart the game, then the new game is spwaned from here
 * with the same boardState as choosen at the start of the MainAcitivity
 *
 * @return void
 *
 **/
public void gameOver(){
    Log.d("prb_log", "Total states generated/explored " + totalCount);
    AlertDialog.Builder builder = new AlertDialog.Builder(this);
    builder.setTitle("Restart Game ?")
            .setPositiveButton("Yes", new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialogInterface, int i) {
                    finish();
                    startActivity(getIntent());
                }
            })
            .setNegativeButton("No", new DialogInterface.OnClickListener() {
                @Override
                public void onClick(DialogInterface dialogInterface, int i) {
                    finish();
                    startActivity(new Intent(BoardActivity.this, MainActivity.class));
                }
            })
            .setCancelable(true);
    AlertDialog alertDialog = builder.create();
    alertDialog.show();
}

/**
 *
 * Returns the boolean true for the boardState is full and if no more moves are avaiable
 * else returns false
```

```
     *
     * @param  state  Board state of the current boardState
     * @return boolean
     *
     **/
    public boolean boardFullCheck(String[] state){
        for (int i = 0; i < state.length; i++){
            if (state[i].equals("-"))
                return false;
        }
        return true;
    }

    @Override
    protected void onStop() {
        super.onStop();
    }
}
```

----------End of BoardActivity.java----------

----------Start of MainActivity.java----------

```
package com.example.tictactoe;

import android.content.Intent;
import android.graphics.Color;
import android.graphics.drawable.GradientDrawable;
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.util.TypedValue;
import android.view.Gravity;
import android.view.View;
import android.widget.ArrayAdapter;
import android.widget.Button;
import android.widget.CompoundButton;
import android.widget.EditText;
import android.widget.LinearLayout;
import android.widget.Spinner;
import android.widget.Switch;
import android.widget.TableLayout;
import android.widget.TableRow;
import android.widget.TextView;
import android.widget.Toast;

public class MainActivity extends AppCompatActivity {
    public final static String BOARDSIZE = "boardsize";
    public final static String BUNDLEVAL = "boardsval";
    int playerTurn;
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
```

```
        playerTurn = 1;
        //Checks if the user has selected to play first or AI will go first.
        //Switch is by default set to ON, so that AI will play the first move.
        //If the switch is OFF then the player will play the first move.
        final EditText editTextGetSize = (EditText) findViewById(R.id.editTextGetSize);
        Switch switchPlayerTurn = (Switch) findViewById(R.id.switchPlayerTurn);

        switchPlayerTurn.setOnCheckedChangeListener(new CompoundButton.OnCheckedChangeListener() {
            @Override
            public void onCheckedChanged(CompoundButton compoundButton, boolean b) {
                if (b)
                    playerTurn = 1;
                else
                    playerTurn = 0;
            }
        });

        //Gives a list of available algorithms the user can choose to play against.
        //These are 1. Minimax, 2. Minimax with Alpha Beta Pruning,
        //3. Minimax Cutoff Heuristic - Checks the number of X's and O's in row, column and diagonal
        //and allocates the value to eval function accordingly
        //4. Minimax Cutoff Heuristic - Checks the number of rows, columns and diagonal where the
        //winner is X or O.
        final Spinner spinnerChooseAlgo = (Spinner) findViewById(R.id.spinnerChooseAlgo);
        ArrayAdapter adapterAlgo = ArrayAdapter.createFromResource(this, R.array.listofAlgo,
android.R.layout.simple_spinner_item);
        adapterAlgo.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item);
        spinnerChooseAlgo.setAdapter(adapterAlgo);


        Button buttonPlay = (Button) findViewById(R.id.buttonPlay);

        //OnClick of Play Button, a new Acitivity will start and this is where the board is
        //displayed with the number of tiles as the user has selected.
        buttonPlay.setOnClickListener(new View.OnClickListener() {
            @Override
            public void onClick(View view) {
                if (validateInputs(editTextGetSize.getText().toString())) {
                    Intent intent = new Intent(MainActivity.this, BoardActivity.class);
                    Bundle bundle = new Bundle();
                    String[] fromMain = {editTextGetSize.getText().toString(),
spinnerChooseAlgo.getSelectedItem().toString(), Integer.toString(playerTurn)};
                    bundle.putStringArray(BOARDSIZE, fromMain);
                    intent.putExtra(BUNDLEVAL, bundle);
                    startActivity(intent);

                } else {
                    Toast.makeText(MainActivity.this, "Please check size of board", Toast.LENGTH_SHORT).show();
                }
            }
        });
    }
```

```
    /**
     * Returns boolean true or false after checking the input from the user for boardsize
     * If the board size is not odd, it will return false else true.
     *
     * @param  getBoardSize  boardSize as entered by the user
     * @return boolean true/false
     **/
    public boolean validateInputs(String getBoardSize) {
        if (!(getBoardSize.equals(""))) {
            if (getBoardSize.matches("\\d+")) {
                int size = Integer.parseInt(getBoardSize);
                if (size % 2 != 0 && size > 1)
                    return true;
            }
        }
        return false;
    }


}
```

----------End of MainActivity.java----------

----------Start of MenuActionClass.java----------

```
package com.example.tictactoe;

import android.app.Activity;
import android.content.Intent;
import android.support.v7.app.AppCompatActivity;
import android.util.Log;
import android.view.Menu;
import android.view.MenuInflater;
import android.view.MenuItem;

/**
 * Created by prajvalb on 11/22/17.
 */

//Adds a Menu to the BoardActivity, where the user has the option to restart the game from
//in between and the whole new game will begin
public class MenuActionClass extends AppCompatActivity {
    @Override
    public boolean onCreateOptionsMenu(Menu menu) {

        MenuInflater menuInflater = getMenuInflater();
        menuInflater.inflate(R.menu.actionbaritem, menu);
        return super.onCreateOptionsMenu(menu);
    }

    @Override
```

```
    public boolean onOptionsItemSelected(MenuItem item) {
        switch (item.getItemId()) {
            case R.id.referesh:
                finish();
                startActivity(getIntent());
                return true;
            default:
                return super.onOptionsItemSelected(item);
        }
    }

}
```

----------End of MenuActionClass.java----------

----------Start of Minimax.java----------

```
package com.example.tictactoe;

import android.util.Log;

import java.util.ArrayList;

/**
 * Created by prajvalb on 11/24/17.
 */

public class Minimax {
    /**
     *
     * This will recursively call minimax depending on the current player,
     * if the current player is X, the algorithm will find the max available board and if
     * the current player is O, the algorithm will find the min avaiable board
     * The terminal state check is done at the start before recursively calling minimax,
     * the terminal checks are checkWinner for player X(AI) and O(User) and if the board state is
     * full, if either of the conditions gets satisfied then it will return the value as decided
     * if winner is AI(X), assign +1000, if winner is User(O) assign -1000 and if the state is
     * draw assign 0 and return
     *
     * @param  board  board for which the minimax will be called recursively
     * @param player  player for which the board state should be generated
     * @return boolean true/false
     **/
    public static int miniMax(Board board, String player){
        BoardActivity.totalCount++;
        if (board.checkWinner(board.getBoardState(), "X")) {
            return 1000;
        } else if (board.checkWinner(board.getBoardState(), "O")) {
            return -1000;
        } else if (board.boardFullCheck(board.getBoardState())){
            return 0;
        }
        else {
```

```
        if (player.equals("X")) {
            int value = (int)Double.NEGATIVE_INFINITY;
            ArrayList<Board> someStates = board.generateSuccessors(board, "X");
            for (Board sBoard : someStates) {
                board.printBoardStateMax(sBoard.getBoardState());
                value = Math.max(value, miniMax(sBoard,  "O"));
}
            return value;
        }

        else  {
            int best = (int)Double.POSITIVE_INFINITY;
            ArrayList<Board> otherStates = board.generateSuccessors(board, "O");
            for (Board oBoard : otherStates) {
                board.printBoardStateMin(oBoard.getBoardState());
                best = Math.min(best, miniMax(oBoard, "X"));
}
            return best;
        }

    }

  }
}
```

----------End of Minimax.java----------
----------Start of MinimaxAlphBeta.java----------

```
package com.example.tictactoe;

import android.util.Log;

import java.util.ArrayList;

/**
 * Created by prajvalb on 11/24/17.
 */

public class MinimaxAlphBeta {
    /**
     *
     * This will recursively call minimax depending on the current player,
     * if the current player is X, the algorithm will find the max available board and if
     * the current player is O, the algorithm will find the min avaiable board
     * The terminal state check is done at the start before recursively calling minimax,
     * the terminal checks are checkWinner for player X(AI) and O(User) and if the board state is
     * full, if either of the conditions gets satisfied then it will return the value as decided
     * if winner is AI(X), assign +1000-depth, if winner is User(O) assign -1000-depth and if the state is
     * draw assign 0 and return.
     * This also implemets Alpha Beta pruinig, so that the whole tree is not necessarily explored
     * every time and only the necessary ones are explored, this will speed up the algorithm
     * and return the result quickly.
     *
     * @param  board  board for which the minimax will be called recursively
```

```
     * @param player  player for which the board state should be generated
     * @return boolean true/false
     **/
    public static int miniMax(Board board, int depth, int alpha, int beta, String player){
        BoardActivity.totalCount++;
        if (board.checkWinner(board.getBoardState(), "X")) {
            return 1000-depth;
        } else if (board.checkWinner(board.getBoardState(), "O")) {
            return -1000-depth;
        } else if (board.boardFullCheck(board.getBoardState())){
            return 0;
        } else {
            int value = (int)Double.NEGATIVE_INFINITY;
            if (player.equals("X")) {
                ArrayList<Board> someStates = board.generateSuccessors(board, "X");
                for (Board sBoard : someStates) {
                    board.printBoardStateMax(sBoard.getBoardState());
                    value = Math.max(value, miniMax(sBoard,depth+1, alpha, beta, "O"));
                    alpha = Math.max(alpha, value);
                    if (beta <= alpha)
                        break;
                }
                return value;
            }

            else  {
                int best = (int)Double.POSITIVE_INFINITY;
                ArrayList<Board> otherStates = board.generateSuccessors(board, "O");
                for (Board oBoard : otherStates) {
                    board.printBoardStateMin(oBoard.getBoardState());
                    best = Math.min(best, miniMax(oBoard, depth+1, alpha, beta,"X"));
                    beta = Math.min(beta, best);
                    if (beta <= alpha)
                        break;
                }
                return best;
            }

        }

    }
}

----------End of MinimaxAlphBeta.java----------
----------Start of MinimaxCutoff.java----------
package com.example.tictactoe;

import android.util.Log;

import java.util.ArrayList;
import java.util.Arrays;

/**
```

```
 * Created by prajvalb on 11/24/17.
 */

public class MinimaxCutoff {
    /**
     *
     * This will recursively call minimax depending on the current player,
     * if the current player is X, the algorithm will find the max available board and if
     * the current player is O, the algorithm will find the min avaiable board
     * Here, the depth is set to 4, so the evalution function is used to give a utility value
     * to the board when the board reaches the depth 4.
     * This also implemets Alpha Beta pruinig, so that the whole tree is not necessarily explored
     * every time and only the necessary ones are explored, this will speed up the algorithm
     * and return the result quickly.
     *
     * @param board board for which the minimax will be called recursively
     * @param player player for which the board state should be generated
     * @return boolean true/false
     **/
    public static int miniMax(Board board, int depth, int alpha, int beta, String player) {
        BoardActivity.totalCount++;
        if (depth == 4)
            return minimaxCutoffEval(board, board.getBoardSize());
        else if (board.checkWinner(board.getBoardState(), "X"))
            return minimaxCutoffEval(board, board.getBoardSize()) + 1000;
        else if (board.checkWinner(board.getBoardState(), "O"))
            return minimaxCutoffEval(board, board.getBoardSize()) - 1000;
        else if (board.boardFullCheck(board.getBoardState()))
            return 0;
        else {
            if (player.equals("X")) {
                int value = (int) Double.NEGATIVE_INFINITY;
                ArrayList<Board> someStates = board.generateSuccessors(board, "X");
                for (Board sBoard : someStates) {
                    board.printBoardStateMax(sBoard.getBoardState());
                    value = Math.max(value, miniMax(sBoard, depth + 1, alpha, beta, "O"));
                    alpha = Math.max(alpha, value);
                    if (beta <= alpha)
                        break;
}
                return value;
            } else {
                int best = (int) Double.POSITIVE_INFINITY;
                ArrayList<Board> otherStates = board.generateSuccessors(board, "O");
                for (Board oBoard : otherStates) {
                    board.printBoardStateMin(oBoard.getBoardState());
                    best = Math.min(best, miniMax(oBoard, depth + 1, alpha, beta, "X"));
                    beta = Math.min(beta, best);
                    if (beta <= alpha)
                        break;


                }
```

```
            return best;
        }

    }


}

public static int minimaxCutoffEval(Board board, int boardSize){
    String[][] tmpState = new String [boardSize][boardSize];
    ArrayList<Integer> XCount = new ArrayList(boardSize+1);
    ArrayList<Integer> OCount = new ArrayList(boardSize+1);
    int[] weights = {1, 3, 5, 10, 20, 40, 80, 160};
    for ( int i = 0;i<boardSize+1;i++)
    {
        XCount.add(0);
        OCount.add(0);
    }

    int k = 0;
    for (int i = 0; i < boardSize; i++){
        for (int j = 0; j < boardSize; j++){
            tmpState[i][j] = board.getBoardState()[k++];
        }
    }

    ArrayList<Integer> totalResult = new ArrayList<>();
    ArrayList<String[]> rowList = new ArrayList<>();
    for (int i = 0; i < boardSize; i++){
        String[] tmpBoard = new String[boardSize];
        for (int j = 0; j < boardSize; j++){
            tmpBoard[j] = tmpState[i][j];
        }
        rowList.add(tmpBoard);
    }
    for (int i = 0; i < boardSize; i++){
        if (Arrays.asList(rowList.get(i)).contains("X") && Arrays.asList(rowList.get(i)).contains("O"))
            totalResult.add(0);
        else if (Arrays.asList(rowList.get(i)).contains("X") && !Arrays.asList(rowList.get(i)).contains("O")){
            int c = java.util.Collections.frequency(Arrays.asList(rowList.get(i)), "X");
            XCount.set(c, XCount.get(c)+1);
        }
        else if (Arrays.asList(rowList.get(i)).contains("O") && !Arrays.asList(rowList.get(i)).contains("X")){
            int d = java.util.Collections.frequency(Arrays.asList(rowList.get(i)), "O");
            OCount.set(d, OCount.get(d)+1);
        }

        else
            totalResult.add(0);
    }

    ArrayList<String[]> columnList = new ArrayList<>();
    for (int i = 0; i < boardSize; i++){
```

```
        String[] tmpBoard = new String[boardSize];
        for (int j = 0; j < boardSize; j++){
            tmpBoard[j] = tmpState[j][i];
        }
        columnList.add(tmpBoard);
    }

    for (int i = 0; i < boardSize; i++){
        if (Arrays.asList(columnList.get(i)).contains("X") && Arrays.asList(columnList.get(i)).contains("O"))
            totalResult.add(0);
        else if (Arrays.asList(columnList.get(i)).contains("X") && !Arrays.asList(columnList.get(i)).contains("O")){
            int e = java.util.Collections.frequency(Arrays.asList(columnList.get(i)), "X");
            XCount.set(e, XCount.get(e)+1);
        }
        else if (Arrays.asList(columnList.get(i)).contains("O") && !Arrays.asList(columnList.get(i)).contains("X")){
            int f = java.util.Collections.frequency(Arrays.asList(columnList.get(i)), "O");
            OCount.set(f, OCount.get(f)+1);
        }
        else
            totalResult.add(0);
    }

    k=0;
    String[] tmpBoard = new String[boardSize];
    ArrayList<String[]> diagonalList = new ArrayList<>();
    for (int i = 0; i < boardSize; i++){
        for (int j = 0; j < boardSize; j++){
            if (i == j)
                tmpBoard[k++] = tmpState[i][j];
        }
    }
    diagonalList.add(tmpBoard);

    k=0;
    String[] tmpBoard1 = new String[boardSize];
    for (int i = 0; i < boardSize; i++){

        for (int j = 0; j < boardSize; j++){
            if (i == boardSize-j-1)
                tmpBoard1[k++] = tmpState[i][j];
        }

    }
    diagonalList.add(tmpBoard1);

    for (int i = 0; i < 2; i++){
        if (Arrays.asList(diagonalList.get(i)).contains("X") && Arrays.asList(diagonalList.get(i)).contains("O"))
            totalResult.add(0);
        else if (Arrays.asList(diagonalList.get(i)).contains("X") && !Arrays.asList(diagonalList.get(i)).contains("O")){
            int g = java.util.Collections.frequency(Arrays.asList(diagonalList.get(i)), "X");
            XCount.set(g, XCount.get(g)+1);
        }
        else if (Arrays.asList(diagonalList.get(i)).contains("O") && !Arrays.asList(diagonalList.get(i)).contains("X")){
```

```
            int h = java.util.Collections.frequency(Arrays.asList(diagonalList.get(i)), "O");
            OCount.set(h, OCount.get(h)+1);
        }
        else
            totalResult.add(0);
    }

    int evalValue =0;
    int sum = 0;
    for ( int i = 1; i <= boardSize-1; i++) {
        sum = sum+weights[i-1]*XCount.get(i);

    }
    evalValue = evalValue +sum;
    int diff = 0;
    for ( int i = 1; i <= boardSize-1; i++) {
        diff = diff+weights[i-1]*OCount.get(i);
    }
    evalValue = evalValue - diff;

    return evalValue;
  }
}
```

----------End of MinimaxCutoff.java----------
----------Start of MinimaxCutoffHeuristicWin.java----------

```
package com.example.tictactoe;

import android.util.Log;

import java.util.ArrayList;
import java.util.Arrays;

/**
 * Created by prajvalb on 11/24/17.
 */

public class MinimaxCutoffHeuristicWin {
    /**
     *
     * This will recursively call minimax depending on the current player,
     * if the current player is X, the algorithm will find the max available board and if
     * the current player is O, the algorithm will find the min avaiable board
     * Here, the depth is set to 4, so the evalution function is used to give a utility value
     * to the board when the board reaches the depth 2.
     * This also implemets Alpha Beta pruinig, so that the whole tree is not necessarily explored
     * every time and only the necessary ones are explored, this will speed up the algorithm
     * and return the result quickly.
     *
     * @param  board  board for which the minimax will be called recursively
     * @param player  player for which the board state should be generated
     * @return boolean true/false
     **/
```

```
    public static int miniMax(Board board, int depth, int alpha, int beta, String player) {
        BoardActivity.totalCount++;
        if (depth == 2)
            return minimaxCutoffEval(board, board.getBoardSize());
        else if (board.checkWinner(board.getBoardState(), "X"))
            return minimaxCutoffEval(board, board.getBoardSize()) + 1000;
        else if (board.checkWinner(board.getBoardState(), "O"))
            return minimaxCutoffEval(board, board.getBoardSize()) - 1000;
        else if (board.boardFullCheck(board.getBoardState()))
            return 0;
        else {
            if (player.equals("X")) {
                int value = (int) Double.NEGATIVE_INFINITY;
                ArrayList<Board> someStates = board.generateSuccessors(board, "X");
                for (Board sBoard : someStates) {
                    board.printBoardStateMax(sBoard.getBoardState());
                    value = Math.max(value, miniMax(sBoard, depth + 1, alpha, beta, "O"));
                    alpha = Math.max(alpha, value);
                    if (beta <= alpha)
                        break;
}
                return value;
            } else {
                int best = (int) Double.POSITIVE_INFINITY;
                ArrayList<Board> otherStates = board.generateSuccessors(board, "O");
                for (Board oBoard : otherStates) {
                    board.printBoardStateMin(oBoard.getBoardState());
                    best = Math.min(best, miniMax(oBoard, depth + 1, alpha, beta, "X"));
                    beta = Math.min(beta, best);
                    if (beta <= alpha)
                        break;


                }
                return best;
            }

        }


    }

    public static int minimaxCutoffEval(Board board, int boardSize){
        String[][] tmpState = new String [boardSize][boardSize];
        int winX = 0, winO = 0;

        int k = 0;
        for (int i = 0; i < boardSize; i++){
            for (int j = 0; j < boardSize; j++){
                tmpState[i][j] = board.getBoardState()[k++];
            }
        }
```

```
ArrayList<Integer> totalResult = new ArrayList<>();
ArrayList<String[]> rowList = new ArrayList<>();
for (int i = 0; i < boardSize; i++){
    String[] tmpBoard = new String[boardSize];
    for (int j = 0; j < boardSize; j++){
        tmpBoard[j] = tmpState[i][j];
    }
    rowList.add(tmpBoard);
}
for (int i = 0; i < boardSize; i++){
    if (Arrays.asList(rowList.get(i)).contains("X") && Arrays.asList(rowList.get(i)).contains("O"))
        totalResult.add(0);
    else if (Arrays.asList(rowList.get(i)).contains("X") && !Arrays.asList(rowList.get(i)).contains("O")){
        winX++;
    }
    else if (Arrays.asList(rowList.get(i)).contains("O") && !Arrays.asList(rowList.get(i)).contains("X")){
        winO++;
    }

    else
        totalResult.add(0);
}

ArrayList<String[]> columnList = new ArrayList<>();
for (int i = 0; i < boardSize; i++){
    String[] tmpBoard = new String[boardSize];
    for (int j = 0; j < boardSize; j++){
        tmpBoard[j] = tmpState[j][i];
    }
    columnList.add(tmpBoard);
}

for (int i = 0; i < boardSize; i++){
    if (Arrays.asList(columnList.get(i)).contains("X") && Arrays.asList(columnList.get(i)).contains("O"))
        totalResult.add(0);
    else if (Arrays.asList(columnList.get(i)).contains("X") && !Arrays.asList(columnList.get(i)).contains("O")){
        winX++;
    }
    else if (Arrays.asList(columnList.get(i)).contains("O") && !Arrays.asList(columnList.get(i)).contains("X")){
        winO++;
    }
    else
        totalResult.add(0);
}

k=0;
String[] tmpBoard = new String[boardSize];
ArrayList<String[]> diagonalList = new ArrayList<>();
for (int i = 0; i < boardSize; i++){
    for (int j = 0; j < boardSize; j++){
        if (i == j)
            tmpBoard[k++] = tmpState[i][j];
    }
```

```
        }
        diagonalList.add(tmpBoard);

        k=0;
        String[] tmpBoard1 = new String[boardSize];
        for (int i = 0; i < boardSize; i++){

            for (int j = 0; j < boardSize; j++){
                if (i == boardSize-j-1)
                    tmpBoard1[k++] = tmpState[i][j];
            }

        }
        diagonalList.add(tmpBoard1);

        for (int i = 0; i < 2; i++){
            if (Arrays.asList(diagonalList.get(i)).contains("X") && Arrays.asList(diagonalList.get(i)).contains("O"))
                totalResult.add(0);
            else if (Arrays.asList(diagonalList.get(i)).contains("X") && !Arrays.asList(diagonalList.get(i)).contains("O")){
                winX++;
            }
            else if (Arrays.asList(diagonalList.get(i)).contains("O") && !Arrays.asList(diagonalList.get(i)).contains("X")){
                winO++;
            }
            else
                totalResult.add(0);
        }

        int evalValue =0;

        evalValue = winX - winO;

        return evalValue;
    }
}
```

----------End of MinimaxCutoffHeuristicWin.java----------
----------End of Source Code----------