CS210 Project 2021
Emily Curran, 20492496, MH 603

Overview of Game

I decided to create a simple text-based game of tic tac toe complete with an ai which plays against the player. The player decides who goes first and how difficult they want to make the ai. When the ai is set to maximum difficulty, the computer is actually impossible to beat and the best the human player can do is to tie with it. This would make for a very boring game and so the player can instead customise the difficulty themselves by introducing some randomness into the game. The player can select a difficulty value from 1-100 which corresponds to a probability of ((difficulty value/2) - 50)/(100) of the computer playing a random position instead of the algorithmically determined best move.

How to Play

The rules for the game are quite simple.

1) The game is played on a 3x3 grid of 9 empty slots .
2) The human plays as 'X' and the computer plays as 'O'.
3) The human and the computer alternate in taking turns placing their symbol, filling a slot.
4) Symbols can only be placed in empty slots.
5) The first player to get 3 of their symbols in a row (vertically, horizontally or diagonally) is the winner.
6) If all the slots on the board are full and no player has successfully placed all of their pieces in a row then the game is over and declared a tie.

To begin, the player will be asked to set the difficulty and then whether they want to go first. When it is their turn they will be asked to enter a number between 1 and 9. Each of these numbers represents a position on the board like above. Once they enter a number, an 'X' will be printed at that position and then the computer will print an 'O' at another empty space on the board. The player will then be asked to enter another number until the somebody wins or until the board is full. The result is then printed on screen and the player is asked if they would like to play again.

Code Structure

The code consists of just a single class 'Game' which contains all of the methods needed in order to run the program. Although the Game class contains many methods, only 2 of them are complex enough to warrant detailed discussion.

Main Method

This acts as the program execution start point. From here all of the other methods are called and the control flow of the program is determined.

Outer Loop

The main method consists of two nested while loops. The outer loop determines whether the player wants to keep the program running. Once this loop terminates, the program prints a thank you message and closes. This loop basically represents

one complete game between players and as such initialises the board on which the game will be played. Then it asks the human if they would like to play first before entering the inner loop.

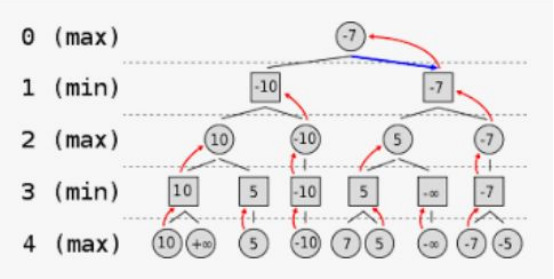
Inner Loop

Each iteration of the inner loop represents one computer move and then one human move. First it places a move for the computer, checks if the game is over, and then prompts the human for their move, places it and checks if the game is over again. The computer move is determined either by a call to either the minimax function or by random chance, the probability of which is determined by the difficulty selected. If at any stage the game appears to be over, the endgame method will be called to announce the winner and ask if the player wants to continue. If the player wants to continue, the inner loop will terminate and a new game will start from the outer loop. If the player does not wish to continue, both loops will terminate and a thank you message will be printed on screen and the program will close.

Minimax Method

This is the method used in determining the optimal move for the cpu. This method is based on the minimax method as used in game theory to form strategy for a two player zero sum game like tic tac toe. Tic tac toe is an excellent example of how to implement this algorithm as the minimax algorithm depends on being able to calculate all possible end states for any given board and tic tac toe is a sufficiently simple game that this is possible.

The algorithm works by recursively simulating play between two players by alternatively placing Xs and Os on the board and checking if the game is over after each placement. If the game is over i.e. the algorithm has reached a base case, this base case is given a utility score, eval, which is calculated as number of turns taken to reach case (depth) - 10 ie. eval = depth -10 when ‘X’ wins or if ‘O’ wins eval = 10 -depth. If the game is a draw the board is given a value of 0.



From there, the algorithm assumes that ‘O’ wants to maximise the final score and that ‘X’ wants to select the move which will return a minimum. This can be used then to calculate the score for one ‘depth’ immediately above each base case, depending on whose turn it is, and then further on up to each previous board until each of the moves immediately available to ‘O’ have an associated utility score. The move with the highest utility score can then be returned and placed down on the board. This process can be represented by a decision tree as in the diagram above (got from https://upload.wikimedia.org/wikipedia/commons/thumb/6/6f/Minimax.svg/600px-Minimax.svg.png).

Other Methods

Some other methods implemented in this program include:

- **printBoard(board):** take in a board as an argument and prints to the console
- **deepCopy(board):** takes a board as input and returns a copy as output.
- **placeMove(board, pos, player, symbol):** takes in four arguments and places symbols on the board depending on their value.

Data Structures and Algorithms Used

As mentioned above I implemented recursion in the minimax algorithm. In order to decide what data type to use to represent a board, I considered several factors. First, that the size of a board tic tac toe board is small and uniform, second that my implementation of the minimax algorithm required no sorting or deleting and finally that my implementation of the minimax algorithm would require thousands of deep copies of boards which requires a lot of insertion. Considering these factors, I decided that an array would be the most efficient data structure to represent the boards. In order to do this I represented each 5x5 2 dimensional array of characters which included both positions for pieces and symbols to draw the board. In addition, I also used 1 dimensional arrays to store all the empty values for a board.

Learning Outcomes

While making this project I learned a lot about methods, recursive functions and control flow. In order to make my code more readable, I had to continuously refactor my code in order to make it more readable. I did this by take large chunks of code which were initially in the main method and creating individual methods which carried out the same operations like the endGame() method.

In addition, I also feel like I have a much better understanding of recursive functions as the minimax algorithm employs them so heavily. I fully understand now how to implement a base case and how that value gets propagated back up to return a final value.

Finally, I think I developed a much better understanding of control flow as I had to very carefully handle how each question posed to the user would affect what order certain methods would be called in.

Appendix

```
import java.util.*;
public class GameF{
        //initialising variables which are accessible everywhere in the class
        public static final Scanner myscanner = new Scanner(System.in);
        public static final Random random = new Random();
        public static boolean keepPlaying = true;

        public static void main(String[] args){
        // main loop which runs the game
        while(keepPlaying == true){
        for(int i=0; i<50; i++){
                System.out.println(); //clear the screen for game
```

```
}
int difficulty = selectDifficulty();

//show index of each position on the board
char [][] sampleBoard = {{'1', '|','2','|', '3'},
                         {'-', '+','-','+', '-'},
                         {'4', '|','5','|', '6'},
                         {'-', '+','-','+', '-'},
                         {'7', '|','8','|', '9'}};

//final board to be played on
char [][] OrigBoard = {{' ', '|',' ','|', ' '},
                       {'-', '+','-','+', '-'},
                       {' ', '|',' ','|', ' '},
                       {'-', '+','-','+', '-'},
                       {' ', '|',' ','|', ' '}};

//human decides who goes first
boolean firstMove = humanFirst(sampleBoard, OrigBoard);

//while loop places moves until game over
while(checkWinner(OrigBoard) == 'f' ){ //'f' means game is ongoing
        cpuMove(difficulty, OrigBoard); //place the move for the cpu
        if(checkWinner(OrigBoard) != 'f'){ //check for winner
        endGame(checkWinner(OrigBoard));
        break;
        }

        //print sampleBoard if this is the first human move
        if(firstMove){
        printBoard(sampleBoard);
        firstMove = false; //remove flag
        }
        humanMove(OrigBoard);//prompt human for move and place it

        //check if the game is over
        if(checkWinner(OrigBoard) != 'f'){ //check for winner
        endGame(checkWinner(OrigBoard));
        break;
        }
}
}
System.out.println();
System.out.println("Thanks for playing!"); //end of game loop
}
public static void printBoard (char[][] board){
//prints the given board to the screen
for(int i = 0; i<board.length; i++){
```

```
;  //fixes bug in how java prints 2d array
for(int j= 0; j<board[i].length; j++){
          System.out.print(board[i][j]);
}
System.out.println();
}
System.out.println();
}
public static char checkWinner(char [][] board){
//checks to see if there are 3 symbols in a row or if the board is full
//if there are three symbols in a row it will return symbol of the winner
//if the game is over and a draw it will return d
//else it will return 'f'
if(board[0][0] == board[0][2] && board[0][2] == board[0][4]
&&  board[0][0]  != ' '){
return board[0][0];
} else if (board[2][0] == board[2][2] && board[2][2] == board[2][4]
&&  board[2][0]  != ' '){
return board[2][0];
} else if (board[4][0] == board[4][2] && board[4][2] == board[4][4]
&&  board[4][0]  != ' ') {
return board[4][0];
} else if (board[0][0] == board[2][0] && board[2][0] == board[4][0]
&&  board[0][0]  != ' ') {
return board[0][0];
} else if (board[0][2] == board[2][2] && board[2][2] == board[4][2]
&&  board[0][2]  != ' ') {
return board[2][2];
} else if (board[0][4] == board[2][4] && board[2][4] == board[4][4]
&&  board[0][4]  != ' ') {
return board[2][4];
} else if (board[0][0] == board[2][2] && board[2][2] == board[4][4]
&&  board[0][0]  != ' ') {
return board[0][0];
} else if (board[0][4] == board[2][2] && board[2][2] == board[4][0]
&&  board[0][4]  != ' ') {
return board[0][4];
} else if(boardFull(board)) {return 'd';}
else{return 'f';}
}
public static int placeMove(char [][] board, int pos,String player,
char symbol){
//places a move into the appropiate spot with the appropiate symbol
//methos calls itself until an input is put into an empty slot
if(pos == 1 && board[0][0] == ' '){
board[0][0] = symbol;
return 1;
}else if(pos == 2 && board[0][2] == ' '){
```

```
board[0][2] = symbol;
return 2;
}else if(pos == 3 && board[0][4] == ' '){
board[0][4] = symbol;
return 3;
}else if(pos == 4 && board[2][0] == ' '){
board[2][0] = symbol;
return 4;
}else if(pos == 5 && board[2][2] == ' '){
board[2][2] = symbol;
return 5;
}else if(pos == 6 && board[2][4] == ' '){
board[2][4] = symbol;
return 6;
}else if(pos == 7 && board[4][0] == ' '){
board[4][0] = symbol;
return 7;
}else if(pos == 8 && board[4][2] == ' '){
board[4][2] = symbol;
return 8;
}else if(pos == 9 && board[4][4] == ' '){
board[4][4] = symbol;
return 9;
} else {
if(player.equals("human")){
        //triggered for invalid human inputs
        //section is called until human inserts into a valid position
        System.out.println("Invalid input. Please try again.");
        pos = myscanner.nextInt();
        return placeMove(board, pos,"human", 'X');
}else{ //impossible that this section gets called
        int next = pos + 1;
        return placeMove(board, next, "cpu", symbol);
}
}
}
public static boolean boardFull(char[][] board){
//function to check if board is full, returns true if full.
boolean isFull = true;
for(int i=1; i<=9;i++){
if(charAtPos(board,i) == ' '){
        isFull = false;
}
}
return isFull;
}
public static int[] arrayEmpty(char [][] board){
//returns an array with the positions of every empty cell. The length
```

```
//of the array is equal to the number of empty spaces.
int count = 0;
for(int i = 1; i <= 9; i++){
if(charAtPos(board, i) == ' '){
        count ++;
}
}
int[] emptySpaces = new int[count];
int position = 0;
for(int i = 1; i <= 9; i++){
if(charAtPos(board, i) == ' '){
        emptySpaces[position]=i;
        position ++;
}
}
return emptySpaces;
}
public static char charAtPos(char[][] board, int pos){
//Returns the character at a given position. Returns 0 for invalid
//input
if(pos == 1){
return board[0][0];
}else if(pos == 2){
return board[0][2];
}else if(pos == 3){
return board[0][4];
}else if(pos == 4){
return board[2][0];
}else if(pos == 5){
return board[2][2];
}else if(pos == 6){
return board[2][4];
}else if(pos == 7){
return board[4][0];
}else if(pos == 8){
return board[4][2];
}else if(pos == 9){
return board[4][4];
}else {return '0';}
}
public static int minimax(char [][] board,int depth,boolean isMaximiser){
//minimax algorithm which recursively searches for the optimal move
//for the selected person for the given state of the board.

/*
Base case for if the game is over. Returns negative score if the
human wins, a positive score for a computer win and zero for a draw.
The maginitude of the score depends on how many moves it took to
```

```
        arrive to end state (depth). This value is back propogated.
        */
        char winner = checkWinner(board);
        if(winner != 'f'){
        if (winner == 'X'){return -10 + depth;}
        else if (winner == 'O'){return 10 - depth;}
        else {return 0;}
        }

        //
        int bestIndex = 0;
        if(isMaximiser){ //cpu is the maximiser
        int maxEval = -1000; //maxEval holds the best score for a move at each depth
        for(int i=0; i < arrayEmpty(board).length; i++){
                char [][] boardCopy = deepCopy(board); //copy the board to simulate potential
moves
                int index = placeMove(boardCopy, arrayEmpty(boardCopy)[i], "cpu",
                'O'); //returns the position for each move placement
                int eval = minimax(boardCopy, depth + 1, false); //recursive call to the
function
                //to function to get score for move
                if(eval>maxEval){ //records the values for position and score of best move
                //at a given depth
                maxEval = eval;
                bestIndex = index;
                }
        }
        if(depth == 0){ //final value method returns when called. This is
                //the positon for the optimal move for the state of the board.
                return bestIndex;
        }else{return maxEval;}  //returns the score to previous call.
        }else{ //human player is the minimiser
        int minEval = 1000;
        for(int i=0; i < arrayEmpty(board).length; i++){
                char [][] boardCopy = deepCopy(board);
                int index = placeMove(boardCopy, arrayEmpty(boardCopy)[i], "cpu",
                'X');
                int eval = minimax(boardCopy, depth + 1, true);
                if(eval<minEval){
                minEval = eval;
                bestIndex = index;
                }
        }
        if(depth == 0){
                return bestIndex;
        }else{return minEval;}
        }
        }
```

```
public static char[][] deepCopy(char[][] board){
//copies the board so that changing objects in the copy of the board
//does not alter objects in the original board
if(board == null){
return null;
}
char[][] boardCopy = new char[board.length][];
for(int i=0; i<board.length; i++){
boardCopy[i] = Arrays.copyOf(board[i], board[i].length);
}
return boardCopy;
}
public static void endGame(char checkWinner){
//prints the winner of the game and invites player to end the game
//or start the main loop all over again
if(checkWinner == 'X'){
System.out.println("Congratulations! X wins!");
System.out.println();
}else if(checkWinner == 'O'){
System.out.println("O wins. Better luck next time!");
System.out.println();
}else if(checkWinner == 'd'){
System.out.println("It's a tie.");
System.out.println();
}
System.out.println("Would you like to play again? (y/n)");
myscanner.nextLine();
String answer = myscanner.nextLine();
if(!answer.equals("y")){
keepPlaying = false;
}
}
public static int selectDifficulty(){
//assigns the difficulty value
System.out.println("Select difficulty 1-100. 1 is the easiest, 100 is the hardest.");
int difficulty = myscanner.nextInt();
System.out.println();
while(difficulty < 1 || difficulty > 100){
        System.out.println("Invalid input. Please select a number between 1 and
100.");
        difficulty = myscanner.nextInt();
        System.out.println();
}
return difficulty;
}
public static boolean humanFirst(char[][] sampleBoard, char[][] OrigBoard){
System.out.println("Would you like to go first?(y/n)");
myscanner.nextLine(); //read in to fix bug
```

```
        String humanFirst = myscanner.nextLine();
        System.out.println();
        boolean firstMove = true; //flag for when to print sampleBoard
        if(humanFirst.equals("y")){
        printBoard(sampleBoard);
        firstMove = false; //remove flag
        System.out.println("Enter your first move (1-9):");
        int humanMove = myscanner.nextInt();
        System.out.println();
        humanMove = placeMove(OrigBoard, humanMove, "human", 'X');
        printBoard(OrigBoard);
        System.out.println("You played " +humanMove+".");
        System.out.println();
        }
        return firstMove;
        }
        public static void cpuMove(int difficulty, char[][] OrigBoard){
        //places the move for the cpu depending on difficulty
        int cpuMove;
        if(random.nextInt(50) + 51 > difficulty){
        //randomly selects whether the computer plays optimal
        //move or random move based on selected difficulty
        int [] possibleMoves = arrayEmpty(OrigBoard);
        int i = random.nextInt(possibleMoves.length);
        cpuMove = possibleMoves[i];
        }else{
        cpuMove = minimax(OrigBoard, 0, true);
        }
        cpuMove = placeMove(OrigBoard, cpuMove, "cpu", 'O'); //place the move for the
computer
        printBoard(OrigBoard);
        System.out.println("Computer played " + cpuMove+".");
        System.out.println();
        }
        public static void humanMove(char[][] OrigBoard){
        //prompt human for the next move and insert it
        System.out.println("Enter the next move (1-9):");
        int humanMove = myscanner.nextInt();
        System.out.println();
        humanMove = placeMove(OrigBoard, humanMove, "human", 'X');
        printBoard(OrigBoard);
        System.out.println("You played " +humanMove+".");
        System.out.println();
        }
}
```