ÖZYEĞİN ÜNİVERSİTESİ

CS 333

Project 1

"Connect 4" Game

by: Selin Köleş (S021541)

2022

Table of Content

Introduction

This project aims to implement a game called “Connect 4” by Hasbro using Java programming language. This report will give us deep insight into the “Connect 4” game explaining, how to play the game, the strategies we can use to win, the implementation of the alpha-beta pruning, and minimax algorithm, and a brief discussion about the outcomes.

“Connect 4” by Hasbro

Connect 4 is a two-player connection game. Initially, each player gets a set of 21 same-colored small discs. Players take turns one by one by dropping the pieces into a seven-column wide and six-row vertically high grid. The player wins, who makes a connection of four pieces horizontally, vertically, or diagonally first.

Fig 1. The figure shows a 6×7 grid “Connect 4” board with yellow and red discs.

Winning Strategy

In this section, the strategies are mentioned which can help a player to increase the possibilities of winning the game by setting traps and at least not letting the opponent win the game by making a draw ;)

Drop in the middle column

If a player gets the chance to play first, then it's a better move to drop the disc in the center column. Since the board has 7 columns, dropping the first disc in the middle column creates five possibilities to make the connection of four discs in a row horizontally, vertically, and diagonally.

Play offensively

It is a good tactic for a player to establish connections with their discs. A player should build rows outwards horizontally or stack discs on top of each other to form vertical connections or make horizontal connections using their discs or the opponent's discs.

Don’t let the opponent connect 4

In order (at least) not to lose the game, it’s a better move to block the opponent’s way to win the game. In this game, we shouldn’t let the opponent make a connection of four discs in a row. So, we should prevent the opponent from getting a connection of three by placing the disc next in line to block in advance. This strategy will not allow the opponent to set up a trap for the player.

Make a “7 trap”

A "7 trap" is a strategy in which the position of discs is configured in such a way that it resembles the number 7. This formation gives a player multiple chances to create a connection of four discs in a row. The following figures explain the “7 trap” in detail.

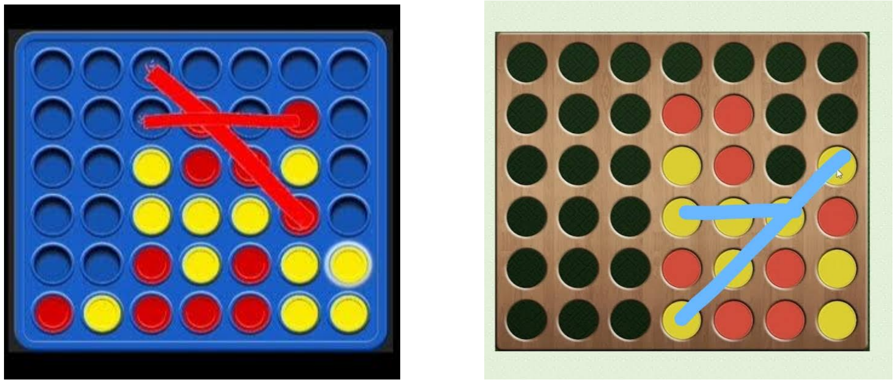

Fig 3 (a) & (b). Illustration of the “7 trap”

Minimax Algorithm

The minimax algorithm employs a depth-first search algorithm to find the best move. It is a type of backtracking algorithm that is used in decision-making and game theory to determine the best move thinking that the opponent makes the optimal moves. In this algorithm, both players are known as **maximizer** or **minimizer**. We assume the **maximizer** (the player itself) tries to score the highest possible while the **minimizer** on the opposite hand tries to score the lowest possible to make the player lose. The pseudocode snippet of the algorithm is given below.

```
function  minimax( node, depth, maximizingPlayer ) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max( value, minimax( child, depth − 1, FALSE ) )
        return value
    else (* minimizing player *)
        value := +∞
        for each child of node do
            value := min( value, minimax( child, depth − 1, TRUE ) )
        return value
```

Source: https://en.wikipedia.org/wiki/Minimax

Alpha-beta pruning

Alpha-beta pruning is not an actual algorithm, it is just an optimized form of the minimax algorithm. In an ideal condition, it reduces the computation time by a huge factor. This allows us to search for the decision tree of the game much faster than the typical minimax algorithm. It removes the game tree branches that don't need to be searched because there would already be a better move available.

In this implementation, we pass two extra parameters in the minimax algorithm named **alpha** and **beta**. **Alpha** is the current best value of the **maximizer** at that level or above, on the other hand, **Beta** is the best value of the **minimizer** at that level or above. The pseudocode implementation of alpha-beta pruning is given below.

```
function alphabeta(node, depth, α, β, maximizingPlayer) is
    if depth = 0 or node is a terminal node then
        return the heuristic value of node
    if maximizingPlayer then
        value := −∞
        for each child of node do
            value := max(value, alphabeta(child, depth − 1, α, β, FALSE))
            if value ≥ β then
                break (* β cutoff *)
            α := max(α, value)
        return value
    else
        value := +∞
        for each child of node do
            value := min(value, alphabeta(child, depth − 1, α, β, TRUE))
            if value ≤ α then
                break (* α cutoff *)
            β := min(β, value)
        return value
```

Source: https://en.wikipedia.org/wiki/Alpha–beta_pruning

Implementation of "Connect 4" game

"Connect 4" game is implemented in Java programming language, in OpenJDK environment version "11.0.17" using eclipse IDE.

Getting the depth value

The program starts by asking the user to insert the depth of the search algorithm, as some non-negative non-zero integer.

```
// getting the depth value from user
Scanner sc = new Scanner(System.in);
System.out.println("Choose the search depth: ");
int depth = sc.nextInt();
```

Determining the turn of player

Then the program determines the turn for two players randomly using `Math.random()` function. The value is saved in a variable named `turn`. If the function gives a value less than or equal to 0.5 then it's the user's turn, as the variable is assigned `turn = 0` value. Besides, the variable is assigned `turn = 1` value.

```
// choosing the turn
int turn;
if (Math.random() <= 0.5) {
      turn = 0; // represents user
} else {
      turn = 1; // represents AI
}
```

Creating the game environment

After determining the first turn, the program creates and prints the game board.

```
// creating new board
int[][] gameBoard = createBoard();
printBoard(gameBoard);
```

While creating the board, elements of array `int[][] board` are assigned value `EMPTY = 0`, representing that board has empty slots. The elements of `int[][] board` are of datatype `int` because it was easy to work with `int` rather than #, X, and O as mentioned in the given problem statement, so the variables `USER_DISC = 1` and `AI_DISC = 2` are used.

```
private static int[][] createBoard() {
	int[][] board = new int[ROWS][COLUMNS];

	int[] row = new int[COLUMNS];
	Arrays.fill(row, EMPTY);
	for (int i = 0; i < ROWS; i++) {
		board[i] = Arrays.copyOf(row, COLUMNS);
	}
	return board;
}
```

In order to print the board on the screen, there is a difference from the one in the `createBoard()` function. On the screen, empty slots are printed the same as was mentioned in the given problem statement. Empty slots are printed as #, the discs of the user are printed as X and the discs of AI (program) are printed as O.

```
private static void printBoard(int[][] board) {
	System.out.println("\n''''\n1\t2\t3\t4\t5\t6\t7\n");
	for (int i = 0; i < board.length; i++) {
		for (int j = 0; j < board[i].length; j++) {
			if (board[i][j] == EMPTY)
				System.out.print("#\t");
			else if (board[i][j] == USER_DISC)
				System.out.print("X\t");
			else if (board[i][j] == AI_DISC) {
				System.out.print("O\t");
			}
		}
		System.out.println();
```

```
        }
        System.out.println("''''\n");
}
```

Checking the current state of the game i.e. if it is over or not

After creating the whole game environment, the game starts in a while loop which terminates when the gameOver variable is set `True`. Otherwise, the `gameOver` variable is initialized as `False` at the start. During the gameplay, after each player's turn, the program checks if it reaches the end of the game stage using `boolean isWinningMove(int[][] board, int disc)` function which checks if any of the players have established a formation of four discs in a row in either direction and returns a boolean value.

```
// checking if current move resulted in the goal to win this game i.e. 4 
discs in row
if (isWinningMove(gameBoard, AI_DISC)) {
        gameOver = true;
        System.out.println("Program wins!");
}

"OR"

if (isWinningMove(gameBoard, USER_DISC)) {
        gameOver = true;
        System.out.println("Congrats! You won.");
}
```

User's turn

Now coming to the gameplay, during the **User's turn**, the program first checks the valid columns using the function `ArrayList<Integer> getValidColumns(int[][] board)` i.e. which are not filled and prints the correct range of available columns. Then asks the user to choose one of the columns where the disc is to be dropped and saves the value in the `choosenCol` variable. The program verifies the `choosenCol` variable using `Boolean isValidColumn(int[][] board, int col)`. If the user chooses the wrong column, the program warns and asks the user to choose a valid column again.

After the user chooses a valid column, the program gets the last empty row from the column using the `int getEmptyRow(int[][] board, int col)` function and saves the value in the `emptyRow` variable. Then the program updates the board by placing the `USER_DISC` using the `void placeDisc(int[][] board, int row, int col, int disc)` function, prints the current state of the board on the screen, checks if the `gameOver` variable needs to be updated, if not then update the `turn` variable so that the AI can its turn.

AI's turn

During the **AI's turn**, the program decides to choose the column using the `minimax` depth search function. The implementation of the minimax algorithm for "connect 4" is given in the following code snippet.

```java
private static Integer[] minimax(int[][] board, int depth, int alpha, int
beta, boolean maximizer) {
	// a variable to store two values i.e. column no. and highest minimax
score which
	// will be returned
	Integer[] result = new Integer[2];
	ArrayList<Integer> validColumns = getValidColumns(board);
	boolean terminalNode = isTreminalNode(board);
	// when node is terminal
	if (terminalNode) {
		if (isWinningMove(board, AI_DISC)) {
			result[0] = null; // column no.
			result[1] = positiveInf; // minimax score
			return result;
		} else if (isWinningMove(board, USER_DISC)) {
			result[0] = null; // column no.
			result[1] = negativeInf; // minimax score
			return result;
		} else {
			// no valid moves are remaining, game over
			result[0] = null; // column no.
			result[1] = 0; // minimax score
			return result;
		}
	}
	// when depth is zero
	else if (depth == 0) {
		result[0] = null; // column no.
		result[1] = evaluateConectedDiscs(board, AI_DISC); // minimax
score
		return result;
	}
	// pruning the decision tree which returns max value
	if (maximizer) {
		result[0] = validColumns.get((int) (Math.random() *
validColumns.size()));
		result[1] = negativeInf;
		for (int col : validColumns) {
			int row = getEmptyRow(board, col);
```

```
                    int[][] copiedBoard =
Arrays.stream(board).map(int[]::clone).toArray(int[][]::new);
                    placeDisc(copiedBoard, row, col, AI_DISC);
                    int score = minimax(copiedBoard, depth - 1, alpha, beta,
false)[1];
                    if (score > result[1]) {
                          result[0] = col; // updating column no.
                          result[1] = score; // updating minimax score
                    }
                    alpha = Math.max(alpha, result[1]);
                    if (alpha >= beta) {
                          break;
                    }
              }
              return result;
        }
        // pruning the decision tree which returns min value
        else {
              result[0] = validColumns.get((int) (Math.random() *
validColumns.size()));
              result[1] = positiveInf;
              for (int col : validColumns) {
                    int row = getEmptyRow(board, col);
                    int[][] copiedBoard =
Arrays.stream(board).map(int[]::clone).toArray(int[][]::new);
                    placeDisc(copiedBoard, row, col, USER_DISC);
                    int score = minimax(copiedBoard, depth - 1, alpha, beta,
true)[1];
                    if (score < result[1]) {
                          result[0] = col; // updating column no.
                          result[1] = score; // updating minimax score
                    }
                    beta = Math.min(beta, result[1]);
                    if (alpha >= beta) {
                          break;
                    }
              }
              return result;
        }
}
```

In this function, let's consider the **maximizer** from the above snippet. The function will first examine all valid positions in each column, recursively calculating the new score in the `int scoreCounter(List<Integer> connect4Array, int playerDisc)` function, and eventually updating the optimal value from the child nodes. As you can see, the new score serves as the alpha in this section. If it is higher than the current value, the recursion will stop and the new value will be updated instead, saving time and memory.

The program has to calculate scores for all possible moves, horizontally, vertically, positively diagonal, and negatively diagonal arrangements of the discs for both players using criteria that are evaluated using the `scoreCounter()` function. The program handles this utilizing the `int evaluateConectedDiscs(int[][] board, int disc)` function.

The score is determined using the `scoreCounter()` function. The function determines the criteria of rewards and penalties for this game and saves the scores in a variable named `score`. The criteria are defined below:

- If four discs of the current player are connected in a row, 100 points are awarded.
- If three discs of the current player are connected, 12 points are awarded.
- If two discs of the current player are connected, then 5 points are awarded.
- On the other hand, the player will be penalized by giving a negative score of 10 if the opponent has three discs connected in a row in the current formation, showing that it's not an optimal move.

After AI decides to choose the best move, the program performs the same way, updates the state of the board, prints it, and checks if the current state is the end of the game, if not then update the `turn` variable so that the User can take its turn and hence the game proceeds.

Results

After challenging the AI various times to play the game against multiple users, it is noticed that AI won almost all the times. In this project, AI uses the minimax algorithm to foresee the best move to outperform the human opponent by determining all possible moves rationally. An interesting fact is that if we change the depth value from 6 (high) to 1 (low), the performance of AI degrades. The implementation of the strategies and the code are checked and verified, they are working correctly.

References

- https://www.wikihow.com/Win-at-Connect-4
- https://medium.com/analytics-vidhya/artificial-intelligence-at-play-connect-four-minimax-algorithm-explained-3b5fc32e4a4f
- https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/
- https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/
- https://en.wikipedia.org/wiki/Minimax
- https://en.wikipedia.org/wiki/Alpha–beta_pruning