MIDDLE EAST TECHNICAL UNIVERSITY

NORTHERN CYPRUS CAMPUS

Department of Computer Engineering

CNG 462 - Assignment 2 Report

Zuhayr Amin 2455426
Liban Gudhane 2406064

Project Goals:

The purpose of this assignment was to create an implementation of the Othello game. We developed our game with our interpretation of the MiniMax algorithm with alpha-beta pruning.

State Space:

Before talking more about the algorithm we implemented and how it works, we will first discuss our state space. The state space included all the tiles in an 8 by 8 grid. In each cell, a player can place an ‘X’ tile while the opposing player can place an ‘O’ tile. The rules allow the players to place tiles in such a manner that at least one of the opponent’s tiles is sandwiched between the players’ tiles. In that event, all such tiles are “flipped” so that they now represent the players’ tiles. The game is finished when the board is either fully occupied, or neither player can place a tile in any of the available spots. The winner is the player with the most tiles at the end of the game.

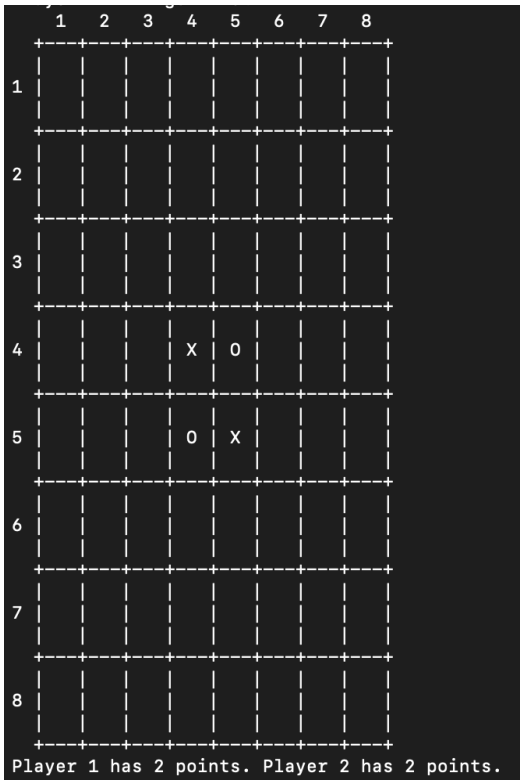

Figure 1: The initial board

The game has been developed so that a user can play against the computer using the command line, or the computer can play against itself using one of two heuristics.

Heuristics:

The first heuristic implemented is called isOnEdge(). This heuristic prefers moving towards the edges of the board and occupying the edges of the board since these pieces are difficult for an opponent player to “flip”. This is the heuristic implemented if a player plays against the computer.

The second heuristic we implemented is called bestScore(). This heuristic prefers making the moves which offer the best possible score. This heuristic is definitely inferior to the first heuristic because it allows itself to be baited into making moves which can easily be overturned by the other heuristic. This is the alternative heuristic implemented if the user chooses to run the program such that it plays against itself.

Alpha-Beta:

This is a recursive algorithm and goes down a pre-determined depth to find the best possible move at each stage. Using this information, the algorithm then takes the moves recommended by the alpha-beta pruning and applies heuristics to determine which space on the board is an ideal move to make long term. It evaluates multiple moves in front to determine what potential moves the opponent could make after the chosen move. It then chooses the move that leaves the opponent worse off. This creates a very strong algorithm.

Functions:

This code contains 15 different functions, which are all explained below.

1. **drawBoard(board):**
   This function takes in the board parameter and prints the board once it is called.

2. **resetBoard(board):**
   This function takes in the board, makes it empty and sets the starting pieces.

3. **newBoard():**
   This function creates a new 8x8 board and returns the parameter board.

4. **isOnBoard(x, y):**
   This function checks whether a given coordinate (x,y) is located on the board, returning a Boolean.

5. **getBoardCopy(board):**
   This function takes the board as a parameter creating and returning an identical parameter called boardCopy.

6. **isValidMove(board, tile, x_coord, y_coord):**
   This function checks if the (x_coord, y_coord) is allowable and returns False if it isn't. Otherwise returns a
   list of all the tiles to be filled as a result.

7. **getValidMoves(board, tile):**
   For a given board, this function checks and returns all the possible valid moves for the player with the
   specified tile.

8. **getScore(board):**
   The tiles "X" and" O" are counted in this function. The summation of each is returned as a dictionary.

9. **makeMove(board, tile, x_coord, y_coord):**
   This function makes a move for the player with a given tile on a given board at the coordinate (x_coord,y_coord).
   This function returns False if this move is invalid and returns True if the move was made.

10. **isOnEdge(x, y):**
    This function checks if the taken coordinate lays on the edge of the board and returns a Boolean accordingly.

11. **getPlayerMove(board, playerTile):**
    This function takes in the coordinate of a player's move.

12. **IsTerminal(board, tile):**
    This function checks if there exists at least one valid move returning False if it does, otherwise True.

13. **alphaBeta(board, tile, depth, alpha, beta, maximizingPlayer):**
    This is a recursive function which employs the uses of MiniMax with alpha-beta pruning to determine the best move which the computer can make.

14. **alphaBetaMove(board, tile):**
    This function returns the coordinates for the move recommended by alphaBeta(). Due to the recursive nature of alphaBeta() using the values of scores of each move, we decided to use a separate function to get the coordinates of the move recommended by alphaBeta().

15. **showPoints(playerTile, computerTile):**
Using the output from the getScore(board), this function prints out the results of each player.

Results:

We ran our program 100 times using different depths for the alpha-beta search. Player 1 uses the heuristic, which prefers the highest possible score, whereas Player 2 prefers to occupy the edges and corners of the board. The results are tabulated below:

Depth	Player 1 Wins	Player 2 Wins	Average Win Margin
1	14	86	25.16
2	12	88	26.79
3	5	95	24.23
4	19	90	26.54

Based on the Table, there does not seem to be any correlation between depth of the alpha-beta search and the distribution of wins. Clearly, Player 2's heuristic is too strong compared to Player 1. Intuitively, this makes sense because the score can be overridden by a better move later in the game, whereas moves on the edge of the board are more difficult to flip, and the corners are impossible to flip.