Othello-Reversi

DEPARTMENT OF INFORMATICS

ARTIFICIAL INTELLIGENCE
WINTER SEMESTER 2023-2024

Eleni Kechrioti
Maria Schoinaki
Christos Stamoulos

OTHELLO-RIVERSI

Introduction

Our work concerns the implementation of the game Othello - Reversi, which is the second topic of the given task. Our application allows a user to play Othello against the computer, giving him the possibility to choose the turn he wants to play and the difficulty level of the game *(1 to 5)*.
The AI algorithm we used to implement the opponent AI is Minimax with a-b pruning.

Minimax with a-b pruning

In our program, as mentioned in the introduction, we apply the minimax algorithm with a-b pruning. This algorithm easily finds the optimal move that can be made in the game, since it is a game of 2 opponents, with one being the computer. By putting minimax into practice, a tree of possible future states is generated for both players, starting with the current state of the game board. Each node of the tree represents a board state in which either the player with the white pieces or the player with the black pieces has played and is accompanied by its value, which is given by the heuristic function evaluate.
The nodes, as we said, are divided into states in which either the player with the black pieces has played last, or the player with the white pieces has played last.
According to the implementation of the MiniMax method (see below), we can say that the player with the black pieces has the max nodes and the player with the white pieces has the min nodes. Therefore, the goal of the black nodes is to maximize their value. For this reason, to find the next optimal move, for each possible move (child of the current board state), the child with the highest value is chosen, which becomes the value of the possible move. Similarly, for the white min nodes.
This whole process is time consuming for large search trees, so we want to restrict the search for possible moves to branches that are likely to be selected. This is achieved by a-b pruning.
A-b pruning has two bounds, a and b, that restrict the set of possible moves based on the tree already deployed. The a is the maximum lower value bound for the possible moves and the b is the minimum upper value bound. This avoids unnecessary generation and search for sub-trees that would not be selected.

MAIN

main

The main method of our project is **Main()**, where most of the methods of the rest of the program are connected and the result of **Othello-Reversi** is produced. Let's examine its functionality step by step.
The process of our method evolves with user input control. To achieve this, we use 2 while loop structures which display an appropriate message to the user if they assign incorrect input. We then print the starting state of the game on the screen.

Then follows the central while loop structure on which most part of **Main()** is based. If the game is not in a final state (via isTerminal()) the conflict between the player and the AI continues! Internally of the while iteration we distinguish the cases of switch /case... The switch uses the **getLastPlayer()** function of the **Board** class and through it, it determines which color of checkers was played last (***-1/B or 1/W***).

In case the last move was made by a black piece and the user has chosen to play second then it is his turn! So, we create a new user move and display the coordinates he has chosen.

Alternatively, if not, it is the system's turn to move! At this point, the **Minimax** algorithm kicks in and the system calculates the optimal movement it is allowed to make at the given moment. It displays the coordinates it chooses on the screen and finally applies its move via **makeMove()** and the corresponding data.

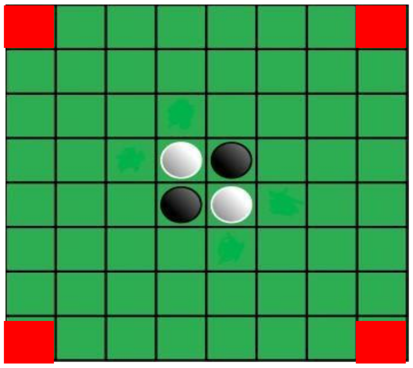
Identical functionality applies to the rest of the pieces and player turn cases.

BOARD

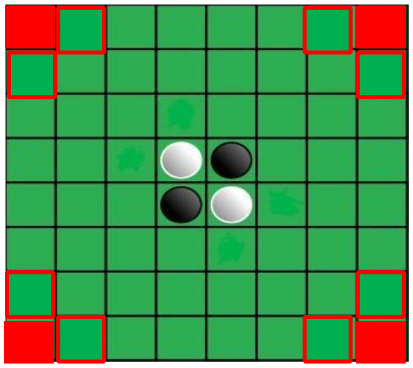
evaluate

As in any game between 2 opponents, we must apply the concept of the minimax algorithm. For our algorithm to choose the optimal according to the data next move, we need to give value to the moves. So that, out of all the possible boards where the possible moves of the computer have been played, the algorithm selects the best one. The method-function that gives value to the board moves is heuristic (***int evaluate()***). This method returns an integer, representing the value of the dashboard after some possible move.

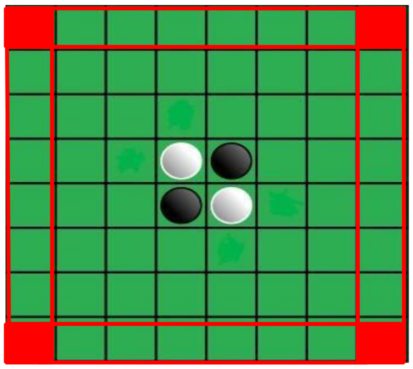
We noticed that any player who occupies one of the 4 corners of the board, after that piece cannot be flipped over by the opponent. It is fixed for as long as the game is alive. So, we considered the moves that capture the 4 corners to be the most valuable, as they not only give an advantage, but they also give the security that this piece will never be flipped over.



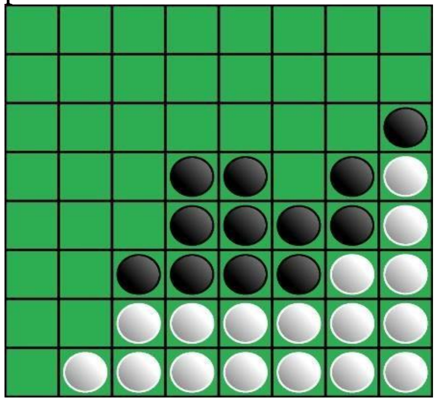
Based on this observation, another one was created. If you hold one of the 4 corners, and occupy the position to the left, right, above, or below it, you create a chain of pieces, which your opponent cannot turn over. So, surely the number of your permanent pieces will grow exponentially.



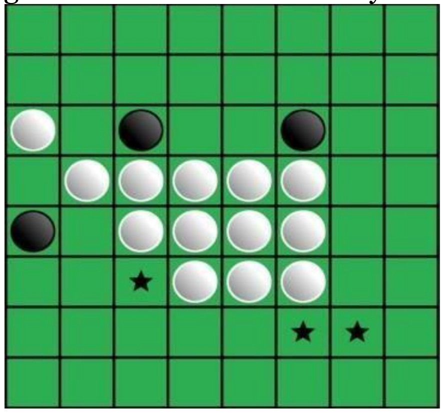
Immediately afterwards, we noticed that the pawns at the edges of the board (edge lines-columns) are also in an advantageous position. The reason is that the opponent can only flip you horizontally(not diagonally or vertically) on the edge rows, and only vertically(not horizontally or diagonally) on the edge columns. Also, by owning the edges, you can easily direct the game by flipping over many opponent's pawns.



Another observation, which is quite important, is the move that blocks the opponent. By playing a move that takes away the opponent's right to play, you not only gain one more move (consecutive), but you also control the game. The opponent now plays to your data, and even better, there are so few possible moves he might have next, that you can easily predict his move.



A similar, less important observation is that of measuring the opponent's possible moves. As mentioned above, Othello is purely a strategy game. If the move you play leaves very few possible moves for your opponent, then you become the dominant player in the game. You control every move, even the entire board, and you are definitely at an advantage. The opponent, no longer making decisions, just carrying out orders that you have implicitly given. You direct the mobility of the game and unquestionably the opponent's moves.



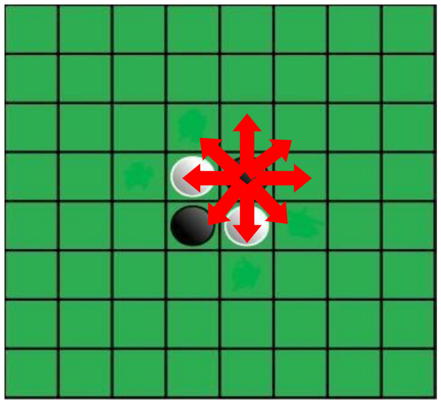
We made several observations about other parts of the game, such as the second corners, the center points, etc. However, we felt that there is no general rule that gives an advantage to these positions, only the board. Which would provide a poor, unnecessary complexity. So, any other thesis offers little value to our heuristic.

Corners	**1000**
Edges next to occupied corners	**800**
Opponent block	**700**
Edges	**500**
Possible opponent's moves	**100**
Rest positions	**50**

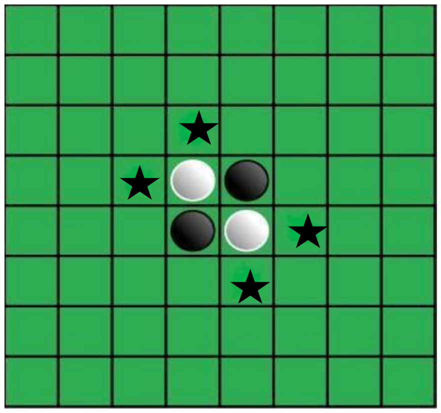
isValid

Another vitally important bool method-function is the **isValid()**. This method lets us know if a move is allowed. An allowable move is a move where pawn x is placed in a position where it can flip 1 or more of the opponent's pawns. Obviously, to achieve this, pawn x must enclose the opponent's pawns to be flipped, with the help of another pawn of the same group (color) as x. Obvious but also remarkable, is that this function, also checks if the position where the pawn is going to be placed, is available(free, empty). This method also checks if the position the pawn tends to be placed is within the boundaries of the board-table(*in 8x8 board, 0<=line<=7, 0<=column<=7*). To check if the opponent's pawn flip is performed, the method checks the 8 directions(if they are available and do not exceed the boundaries of the board-table).

For example, in the board below, the method will check all 8 directions **until** (break clause) it finds one, which satisfies the opponent's bounds, void and pawn flip. So, the black pawn, has 4 possible moves that satisfy the **isValid()** method.



The 4 possible moves are shown below.



getChildren

Returns a list of children (Board items). A child is the current state of the board and the next possible move the player can make based on the current board. Finding the children is done by traversing our board and finding all possible moves that can be made by the requested player. For each of these moves, we create the "child" as a copy of the table on which that move is then made.

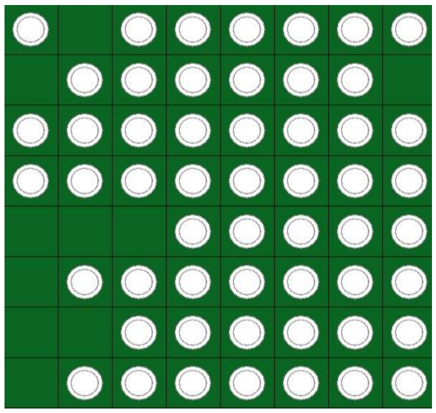
inBoarders

Checks if the move is within the table and returns True. Otherwise, returns False.

isTerminal

It traverses our two-dimensional board counting the black pieces, white pieces, and empty spaces. If there are no more empty spaces on the board, or if one of the two players has no more pieces on the board, then the game is over, and we return True. Otherwise, we return False.

For example, a finished board (game) is shown below:



Where, the player with the black pieces has no other pieces on the board, so he cannot play a move, and so does the player with the white pieces, as there is no black piece to enclose.

isBlocked

It runs through our two-dimensional table and checks if there are valid moves for the player. If there is at least one valid move, then the player is not blocked and we return True, otherwise, we return False.

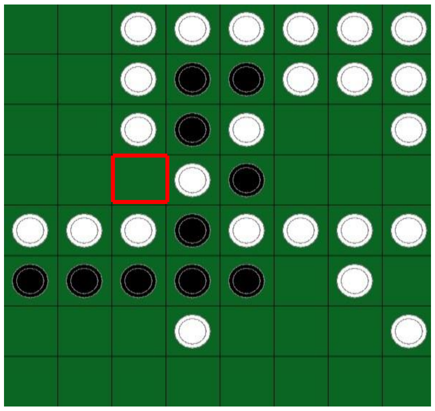
makeMove

It checks if the move the player wants to make is valid, and only then allows the move to be made by placing the player's pawn on the board and calling the flipcheckers method to flip the opponent's pawns. Otherwise, it displays an invalid move message. It also creates a new move, with the coordinates of the move just made, which it sets as the last move and the player who just played as the last player.

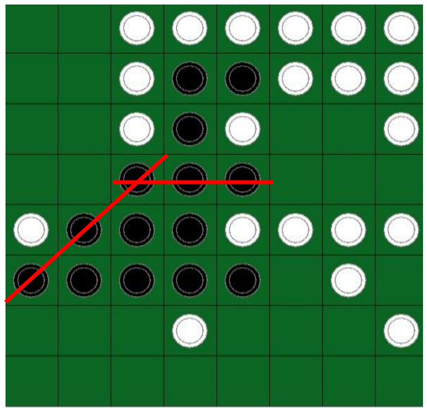
flipcheckers

Since a valid move has just been made, it flips the pieces that need to be flipped, i.e. the ones enclosed by the opponent's pieces with this move. For each direction (up, down, right, left, diagonal top right, diagonal bottom right, diagonal bottom left, diagonal top left) from the point where he makes his move he checks if there are any opponent's pieces and stops as soon as he finds a player's piece. If no pieces are found, then no opponent's pieces are trapped in that direction and therefore no opponent's pieces are flipped. However, if a player's pawn is found, then the opponent's pawns are flipped.

A simulation of the method is shown below.



If the player with the black pieces selects the selected box, then we notice that he completes a diagonal triad, and a horizontal triad, where he encloses one of his opponent's pieces in each direction. So, the board after black moves to the selected position and according to the flipcheckers method, will become:



getScores

It returns according to the given board, how many pieces each player has.

setLastMove

Determines the last move.

getLastMove

Returns the last move that was made.

setLastPlayer

Sets the last player played.

getLastPlayer

Returns the last player who played.

setGameBoard

Defines the two-dimensional table that represents our dashboard.

getGameBoard

Returns our game board.

MOVE

Constructors

The **Move** class of our program is responsible for simulating the movements that either the user or the computer will make on the board of the game (*the implementation of the movements is done through the **Makemove()** class **Board***).
The Move file is composed of the necessary coordinate fields (row - column) and a value value for setting the color of the pieces. The above data is used by 4 move constructors as well as export/return value functions (*setters/getters*).

Combining the capabilities provided by Move, any update to the dashboard can be done with minimal programming effort.

PLAYER

setMaxDepth

Sets the game's difficulty number.

setPlayerLetter

Determines the color/letter B or W for the player.

MiniMax

It calls max if the player is playing with the black pieces, because we want to maximize its value, otherwise for the white pieces it calls min.

min

Seeks to find the move with the least value (via heuristics) of the possible moves that can be made.

max

It looks to find the move with the highest value (via heuristics) of the possible moves that can be made.
Min and max are called alternately for the next level of the tree of possible moves.

WINDOW

The class in which the graphical interface of the **Othello** game is implemented.

Window

Creates a window (frame) in which the game will take place. In this class all panels and buttons that will be used to run the game are created and formatted. The panels are three and are divided into:
* home screen, where the user customizes the game as he likes and starts it
* main screen, where the user plays against the computer
* final screen, showing the final score and the winner of the round. There is an option to end the game (exit) and start a new round.

MousePressed

It is mainly used for the main screen. For each tap the user makes on the screen, we get the coordinates of the cursor on the screen and find which cell it corresponds to on the dashboard. This is done by keeping the result of integer division of the arithmetic/quantized by 60 (the number of pixels of the dimension of each cell). After this point, the same logic as main:Main is followed to perform the gestures.

Epilogue

This project was carried out in the course "**Artificial Intelligence**", which is a required core course of the 3rd year of the Department of Computer Science. The pdf has been written exclusively by the students of the group, as well as all the code implemented. The course notes, as well as the tutorials, were helpful in this effort. Useful information was also taken from the books by **S.Russell** and
P.Norvig "Artificial Intelligence, a modern approach" 4th American edition and **I.Vlahava, P.Kefalas, N.Vassiliadis, F.Kokkoras and H.Skellariou "Artificial Intelligence"** 4th edition.