**AI Project Report**

OTHELLO

Members: Đào Hồng Quân 20176850

Vũ Công Duy 20176737

Phạm Ngọc Diễn 20176715

Đoàn Minh Đức 20176717

**CONTENTS:**

I. Introduction

1. The game……………………………………………………………………………………………………………………2

2. The program……………………………………………………………………………………………………………….3

3. Programing languages…………………………………………………………………………………………….....3

II. The basics

1. Global variables………………………………………………………………………………………………………….4

2. Board representation………………………………………………………………………………………………….4

3. Legal moves………………………………………………………………………………………………………………..4

4. Searching for moves……………………………………………………………………………………………………4

III. Algorithms for the computer

1. Random………………………………………………………………………………………………………………………5

2. Moving by number……………………………………………………………………………………………………..5

3. Local maximization……………………………………………………………………………………………………..5

4. Minimax……………………………………………………………………………………………………………………..6

5. Alpha-Beta pruning…………………………………………………………………………………………………….7

6. Alpha-Beta pruning with sorted nodes……………………………………………………………………….7

7. Heuristic……………………………………………………………………………………………………………………..8

IV. Graphic User Interface……………………………………………………………………………….9

V. Overview

1. Test results………………………………………………………………………………………………………………….10

2. The algorithms…………………………………………………………………………………………………………….10

3. The coding process………………………………………………………………………………………………………11

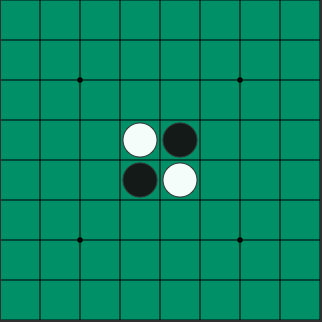
I. Introduction:

1. The game:

Othello (or Reversi) is a strategy board game for two players, played on an 8x8 uncheckered board. There are sixty-four identical game pieces called disks, which have a different color on each side corresponding to each player.

For the specific game of Othello (differing from the historical Reversi), the rules state that the game begins with four disks placed in a square (tile) in the middle of the grid. The same colored disks are on a diagonal with each other.

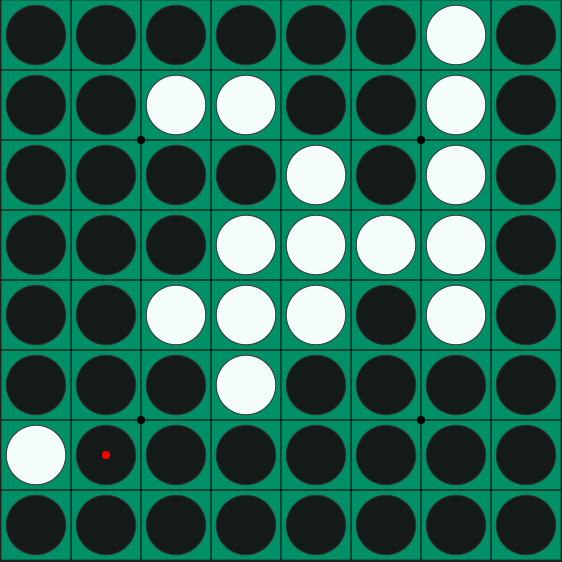
Convention has initial board position such that the disks with dark side up are to the north-east and south-west (from both players’ perspectives).



Dark must place a piece with the dark side up on the board, in such a position that there exists at least one straight (horizontal, vertical, or diagonal) occupied line between the new piece and another dark piece, with one or more contiguous light pieces between them.

After placing the piece, dark turns over (captures) all light pieces lying on a straight line between the new piece and any anchoring dark pieces. All reversed pieces now show the dark side, and dark can use them in later moves—unless light has reversed them back in the meantime. In other words, a valid move is one where at least one piece is reversed.

Players take alternate turns. If one player could not make a valid move, the turn passes back to the other player. When neither player can move, the game ends. This occurs when the grid has filled up or when neither player can legally place a piece in any of the remaining squares. This means the game may end before the grid is completely filled. This possibility may occur because one player has no pieces remaining on the board in that player's color.



The object of the game is to have the majority of disks turned to display your color when the last playable empty square is filled. In this case, Black wins over White with the score 37 to 27.

2. The program:

In the program, the color red represents the player and blue represents the computer. The user moves first.

The player gets to choose from six levels:

- Level 1: Random

- Level 2: Moving by number

- Level 3: Local maximization

- Level 4: Minimax with depth 3

- Level 5: Alpha-Beta pruning with depth 5

- Level 6: Alpha-Beta pruning with sorted nodes and depth 7

We also wrote a file to test the strength of each algorithms against one another.

3. Programing languages:

The algorithm is written in Python 3.7 with the additional library EEL to connect with JavaScript in the interface. We also use HTML and CSS to display the game on a chrome app.

II. The basics:

1. Global variables:

We represent the game board data structure as a 2-dimension array which includes each square on the board from 1 to 8 in both dimensions. We represent the value of each tile in another board for evaluation in the algorithms. Finally, we have an array of directions for a piece to move.

During the game, there are two integer variables that will be constantly modified. Turn has two values ‘1’ and ‘2’. Block is a special variable checking if there is any move left for both players, if there isn’t, the game is finished and the winner is announced.

2. Board representation:

There are 2 colors for both players, so we mark red as ‘1’ and blue as ‘2’. An empty tile is marked as ‘0’.

3. Legal moves:

A legal move in the program is represented as an array of three elements. The first two being the coordinates of the move on the board and the last element arrays the coordinate array of all pieces that will be flipped with that move.

We use arrays instead of objects to help reduce the complication of transferring data between JavaScript and Python.

4. Searching for moves:

The program checks for legal moves from existing player’s pieces.

From a player’s piece, we move in a direction as follow:

- We continue to move as long as there is an opponent piece on the tile, adding each tile to an array (which will be the third element of the move array).

- If we meet an empty tile after having passed through at least one opponent piece, we will add its coordinate to the first two elements of the move array, append the flipped array, and return the array.

- Any other instance results in returning a blank array.

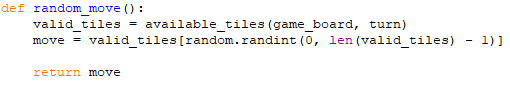
We iterate through the direction array with the function above and append all returned, non-empty array to an array that will be returned.

Finally, we perform this function on all of the player’s pieces on the board and return the complete legal array.

III. Algorithms for the computer:

1. Random:

The easiest level for the computer’s



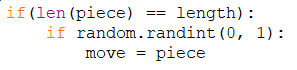
difficulty is by randomly choosing a move

from the array of available moves, with the

help of the random library.

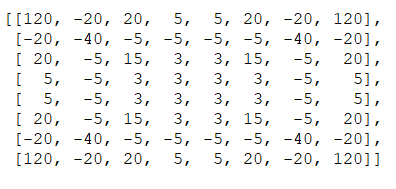
2. Moving by number:

The computer goes through the array of available moves and picks out the move with the highest number of gained pieces. If there are two pieces with the same number of flipped pieces, the computer randomly picks one.



3. Local maximization:

Using the value board, the computer checks for the legal move that gives the highest reward.



In the real local maximization method, the computer makes a move and evaluate the board afterwards to check for result. However, this method bears similarity with the base case of the Minimax method (mentioned below). Since we want to reduce the difficulty and speed of this mode, the computer only checks the reward of the coordinate on which a new piece may be placed.

4. Minimax:

This is a recursive function with three parameters: board, depth, and player. The function returns an integer value.

Instead of checking for terminal node, we decided to check for the length of the available moves regarding to the board and player argument.

The base case is when depth or the length of the moves array equals to zero. The board’s evaluation is returned in this instance.

The maximizing instance occurs when the player argument is equal to the current player. We initiate the best value with the value of -1776.

The computer iterates through the moves array as follow:

- A temporary board with the move already made is created.

- The computer gets the value of minimax with the following arguments: the temporary board, depth -1, and the opponent.

- The value will be assigned to the best value if it is larger than the current best value.

The best value is returned when the iteration is completed.

The minimizing instance occurs when the player argument is different from the current player. We initiate the best value with the value of 1776.

The computer iterates through the moves array as follow:

- A temporary board with the move already made is created.

- The computer gets the value of minimax with the following arguments: the temporary board, depth -1, and the opponent.

- The value will be assigned to the best value if it is smaller than the current best value.

The best value is returned when the iteration is completed.

To sum up the process:

- The base case returns the evaluating value.

- The maximizing instance checks for the player’s move that gains the most value.

- The minimizing instance checks for the player’s move that gains the least value.

Explanation:

- During the maximizing iteration, a player’s move is made on a simulating board and minimax is recursively called to get the best possible value out of board evaluations or lower depth’s minimizing values.

During the minimizing iteration, an opponent’s move is made instead and minimax is recursively called, which will return maximizing values. The simulated opponent will choose the move that has the least advantage for the player.

5. Alpha-Beta pruning:

This function is an improved version of minimax. The function has two new parameters: alpha and beta.

The base case remains the same.

In the maximizing instance, after comparing the best value and the returned value, the computer also compares alpha with the returned value and assigns it if alpha is smaller.

In the minimizing instance, the returned value is also assigned to beta if beta is larger.

The main difference of this algorithm is that when alpha is no longer smaller than beta, the iteration process is stopped. This break is called beta-cut-off in maximization and alpha-cut-off in minimization.

Explanation:

- Alpha represents the highest value for a move in the current node.

- Beta represents the highest value for a move under the opponent’s pressure in the parent node.

- Whenever alpha or beta is assigned a new value, this does not alter their values in the parent node. These alterations are only for the current node’s siblings.

- When alpha is larger than or equal to beta in maximization, the parent node, which maybe a minimizing node, will certainly choose the node with beta value. Therefore, no further process is needed in this node.

- When beta is smaller than or equal to alpha in minimization, the parent node, which is a maximizing node, will choose this node in favor of the previous ones, causing disadvantage to the opponent. Therefore, the current node will skip further processes.

6. Alpha-Beta pruning with sorted nodes:

This is an enhance version of alpha-beta pruning.

The main difference is that the available moves array is specifically sorted.

The sorting process:

- For each move in the array, a temporary board with that move is created and evaluated.

- The array will be sorted in descending order in accordance with its respective simulated value.

- During the maximizing instance, the array will be sorted descendingly. With the new array, the best alpha value has a higher chance of appearing early in the process, thus beta-cut-offs will appear more frequently.

- During the minimizing instance, the array will also be sorted descendingly, since the opponent’s best case is also the player’s worst case. With the new array, the smallest beta value has a higher chance of appearing early in the process, making alpha-cut-offs appear more frequently.

The increase in cut-offs frequency will reduce the number of iterations, hence greatly improve runtime.

However, the sorting function above only improves runtime and has no interference with the precision of the algorithm. Simulating the moves on each sorting instance also makes unwanted runtime increase. Therefore, like local maximization, we will only sort the array by checking the reward of the moves’ coordinates.



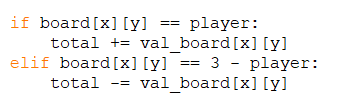
7. Heuristic:

The heuristic function of the algorithms above is the board evaluation function, which relies on the value board.

For every tile on the board:

- If there is a player disk, the value increases by the value of the same coordination on the value board.

- If there is an opponent disk, the value decreases by the value of the same coordination on the value board.



\* ***Note***: since minimax and alpha-beta only return value, the three algorithms need functions that compare the returned value of each initial move and return the best move. Thus, the former functions always get passed the initial depth minus one.

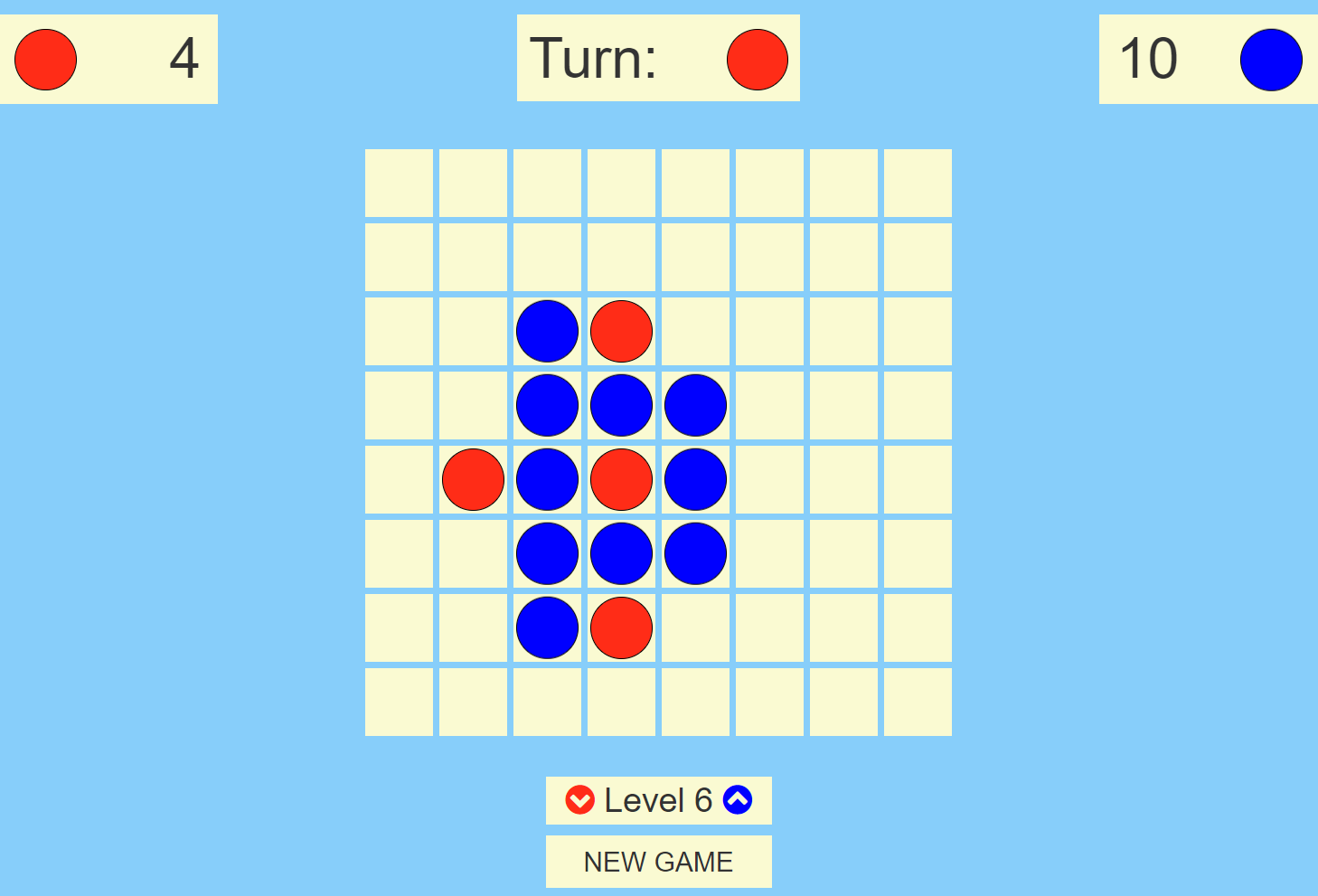
IV. Graphic User Interface:

The game is displayed as a Chrome app with HTML, CSS, and JavaScript. We also use the framework Bootstrap 3 to help with the grid system.

The algorithms, written in Python, become functions that will be exported for usage in the JavaScript file. The Python library EEL helps set up a server and allows JavaScript to call Python functions in asynchronous calls.

We only exported the function that finds legal moves and the function that returns the computer’s move (using if/else branches to find a move in accordance with the player’s choice of difficulty).

In the game, the player can choose to reset the game at will. During each of the player’s turn, you can adjust the difficulty of the game from level one to six.



*(This is a sample picture of the game when playing with the computer at level 6).*

V. Overview:

1. Test results:

Before attaching the algorithms to the interface, we made nine modes compete with a random player (50 matches each) to determine the runtime and efficiency of each mode.

The mode random is considered as an average players.

The nine modes, with the winning rates and average runtime, are:

1. Random: 49%, 0.031250s.

2. Moving by number: 46%, 0.031250s.

3. Local maximization: 75%, 0.031250s.

4. Minimax with depth 3: 84%, 9.484375s.

5. Alpha-Beta pruning with depth 3: 90%, 2.328125s.

6. Alpha-Beta pruning with sorted nodes with depth 3: 74%, 1.890625s.

7. Alpha-Beta pruning with depth 5: 84%, 66.734375s.

8. Alpha-Beta pruning with sorted nodes with depth 5: 90%, 34.703125s.

9. Alpha-Beta pruning with sorted nodes with depth 7: 80%, 407.812500s.

Random still has some advantage over modes with higher depths since the mode does not aim to undermine the opponent, thus not reaching the opponent’s best solution.

2. The algorithms:

Alpha-Beta pruning greatly reduces runtime compared to Minimax when depth becomes larger than three.

Alpha-Beta pruning with sorted nodes reduces the average runtime of the pruning algorithm by half. However, this is not sufficient to run depths greater than seven.

3. The coding process:

This is our first experience with the python programming language.

We learn the importance of variable scope. Most of the bugs we met involved not declaring global variables in functions. Python is also a tricky language as errors in functions only appear when the functions are called, while languages such as C display warnings during the compiling process. We also figured the debugging method of printing out the value of each function with its name to catch up on the process.

Using JavaScript in this local host-based app also proved to be a challenge as the language is asynchronous. It does not wait for values returned from the Python functions. We have to be careful with handling async functions.

We also find CSS hard to handle when it comes to margins and paddings.