INTERNATIONAL UNIVERSITY VIETNAM NATIONAL UNIVERSITY, HCM CITY

**School of Computer Science & Engineering**

Logo

Description automatically generated

**PROJECT**

**Lecturer: VVThanh**

**Course:** [**Algorithms & Data Structures**](https://blackboard.hcmiu.edu.vn/webapps/blackboard/execute/courseMain?course_id=_23290_1)

**Members:**

**Huỳnh Trúc Quyên ITDSIU19051**

**Table of Contents**

[I. Introduction 3](#_Toc136720755)

[1) Background 3](#_Toc136720756)

[2) Game rule 3](#_Toc136720757)

[II. Method 6](#_Toc136720758)

[1) Game play 6](#_Toc136720759)

[2) Minimax with Alpha-Beta pruning 8](#_Toc136720760)

[a. Minimax 8](#_Toc136720761)

[b. Searching with Alpha-Beta pruning 8](#_Toc136720762)

[3) Heuristic 9](#_Toc136720763)

[a. Mobility 10](#_Toc136720764)

[b. Stability 10](#_Toc136720765)

[c. Corners Captured 10](#_Toc136720766)

[III. Demo 11](#_Toc136720767)

[IV. References 14](#_Toc136720768)

[V. Contribution 14](#_Toc136720769)

**Table of Figures**

[Figure 1 4](#_Toc136720806)

[Figure 2. (Black to move, the ⭐ icon is one possible move for black) 4](#_Toc136720807)

[Figure 3. 5](#_Toc136720808)

[Figure 4 5](#_Toc136720809)

[Figure 5. Main Menu Page 11](#_Toc136720810)

[Figure 6. Rule Page 12](#_Toc136720811)

[Figure 7. Option Page 12](#_Toc136720812)

[Figure 8. Player Names Page 13](#_Toc136720813)

[Figure 9. Player vs Player 13](#_Toc136720814)

[Figure 10. Player vs BOT 14](#_Toc136720815)

# **Introduction**

This semester we constructed a game project to evaluate our understanding of Algorithms and Data Structures. Our project is a remake of the popular chess game Othello. This is a hard project, and it allows us to gain and enhance our DSA knowledge, including coding styles, design patterns, and methodologies. Our project is written in Java. We shall present our project in depth in this report, including the approaches and algorithms employed throughout development.

Here is our github: <https://github.com/huynhtrucquyen/DSA_OTHELLO.git>

## **Background**

Othello, also known as Othello, is a two-player game that has been enjoyed by people of all ages for decades. In this project, we used Java to implement the Minimax algorithm, a commonly used technique for turn-based zero-sum games. We incorporated Alpha-Beta pruning, a heuristic that often goes with the Minimax algorithm to further reduce the time complexity, allowing our bot to consider and analyze more potential moves ahead of time.

## **Game rule**

Othello is played on an 8x8 board with black discs and white discs for the black side and white side. Initially, 2 blacks and 2 whites arranged diagonally. Two players alternatively play turn by turn with black going first​.

A screenshot of a game

Description automatically generated

**Figure 1**

On each turn, the player must make a valid move​. Valid move must flip at least one opponent’s color​.

A screenshot of a game

Description automatically generated

**Figure 2. (Black to move, the ⭐ icon is one possible move for black)**

Piece is flipped when bounded by a new piece and another of the current player’s color in a straight line (vertical, horizontal, diagonal).

A screenshot of a game

Description automatically generated

**Figure 3.**

The game ends when the board has no valid moves. In the end, whoever has more pieces wins. The game is a tie when both 2 players have equal moves.

A screenshot of a game

Description automatically generated

**Figure 4**

# **Method**

## **Game play**

The "checkPosibleMove" method in the "GamePlay" class is used to determine the possible moves for a player in a game.

  It takes two parameters:

* 'check': a 2D array representing the current state of the game board, where each element indicates the player who occupies that position (-1 for empty, 1 for player 1, 2 for player 2).
* 'player': an integer representing the current player (1 for player 1, 2 for player 2).

The function initializes an empty list `arrPosibleMove` to store the possible moves. It then creates a 2D array `possible` to represent the possibility of each position on the board for the current player to make a move.

It creates another 2D array `arr` of type `cond` (a custom class) to store additional information about the cells on the board. Each element of `arr` contains four properties: 𝑎𝑟𝑟𝑖,𝑗.ℎ𝑜𝑟𝑖𝑧𝑜𝑛𝑡𝑎𝑙, 𝑎𝑟𝑟𝑖,𝑗.𝑣𝑒𝑟𝑖𝑐𝑎𝑙, 𝑎𝑟𝑟𝑖,𝑗 .𝑙𝑒𝑓𝑡𝐷𝑖𝑎𝑔𝑜𝑛𝑎𝑙, 𝑎𝑟𝑟𝑖,𝑗.𝑟𝑖𝑔𝑡ℎ𝐷𝑖𝑎𝑔𝑜𝑛𝑎𝑙.

The function iterates through the cells of the game board, starting from the top left corner and moving downwards and to the right.

If the cell is white, the `horizontal` property of the current cell is set to the same value as the `horizontal` property of the previous cell in the same row. Similarly, the `vertical` property is set to the `vertical` property of the cell above it. The `leftDiagonal` property is set to the `leftDiagonal` property of the cell diagonally above and to the left, and the `rightDiagonal` property is set to the `rightDiagonal` property of the cell diagonally above and to the right.

𝑎𝑟𝑟𝑖,𝑗 . ℎ𝑜𝑟𝑖𝑧𝑜𝑛𝑡𝑎𝑙 = 𝑎𝑟𝑟𝑖,𝑗−1. ℎ𝑜𝑟𝑖𝑧𝑜𝑛𝑡𝑎𝑙

𝑎𝑟𝑟𝑖,𝑗 . 𝑣𝑒𝑟𝑡𝑖𝑐𝑎𝑙 = 𝑎𝑟𝑟𝑖−1,𝑗 . 𝑣𝑒𝑟𝑡𝑖𝑐𝑎𝑙

𝑎𝑟𝑟𝑖,𝑗 . 𝑙𝑒𝑓𝑡𝐷𝑖𝑎𝑔𝑜𝑛𝑎𝑙 = 𝑎𝑟𝑟𝑖−1,𝑗−1.𝑙𝑒𝑓𝑡𝐷𝑖𝑎𝑔𝑜𝑛𝑎𝑙

𝑎𝑟𝑟𝑖,𝑗 . 𝑟𝑖𝑔ℎ𝑡𝐷𝑖𝑎𝑔𝑜𝑛𝑎𝑙 = 𝑎𝑟𝑟𝑖−1,𝑗+1. 𝑟𝑖𝑔ℎ𝑡𝐷𝑖𝑎𝑔𝑜𝑛𝑎l

If the cell is black, the `horizontal`, `vertical`, `leftDiagonal`, and `rightDiagonal` properties of the current cell are set to the same values as the corresponding properties of the current cell. In other words, the current cell is marked as a black cell, and all its properties indicate that there is no consecutive cell with a different value to the left in any direction.

𝑎𝑟𝑟𝑖,𝑗 . 𝑟𝑖𝑔ℎ𝑡𝐷𝑖𝑎𝑔𝑜𝑛𝑎𝑙 = 𝑎𝑟𝑟𝑖,𝑗 . 𝑙𝑒𝑓𝑡𝐷𝑖𝑎𝑔𝑜𝑛𝑎𝑙 = 𝑎𝑟𝑟𝑖,𝑗 . 𝑣𝑒𝑟𝑡𝑖𝑐𝑎𝑙 = 𝑎𝑟𝑟𝑖,𝑗 . ℎ𝑜𝑟𝑖𝑧𝑜𝑛𝑡𝑎𝑙 = 𝑤ℎ𝑖𝑡e

If the cell is empty, additional checks are performed. If the property `rightDiagonal` of the cell diagonally above and to the right is `true`, and the cell diagonally above and to the right is white (`cell[i-1][j+1]` is white), or if the property `leftDiagonal` of the cell diagonally above and to the left is `true`, and the cell diagonally above and to the left is white (`cell[i-1][j-1]` is white), or if the property `horizontal` of the cell to the left is `true`, and the cell to the left is white (`cell[i][j-1]` is white), or if the property `vertical` of the cell above is `true`, and the cell above is white (`cell[i-1][j]` is white), then the current cell is marked as a possible move.

𝑎𝑟𝑟𝑖−1,𝑗+1. 𝑟𝑖𝑔ℎ𝑡𝐷𝑖𝑎𝑔𝑜𝑛𝑎𝑙 = 𝑏𝑙𝑎𝑐𝑘 ∧ 𝑐ℎ𝑒𝑐𝑘𝑖−1,𝑗+1 = 𝑤ℎ𝑖𝑡𝑒

𝑎𝑟𝑟𝑖−1,𝑗−1.𝑙𝑒𝑓𝑡𝐷𝑖𝑎𝑔𝑜𝑛𝑎𝑙 = 𝑏𝑙𝑎𝑐𝑘 ∧ 𝑐ℎ𝑒𝑐𝑘𝑖−1,𝑗−1 = 𝑤ℎ𝑖𝑡𝑒

𝑎𝑟𝑟𝑖,𝑗−1. ℎ𝑜𝑟𝑖𝑧𝑜𝑛𝑡𝑎𝑙 = 𝑏𝑙𝑎𝑐𝑘 ∧ 𝑐ℎ𝑒𝑐𝑘𝑖,𝑗−1 = 𝑤ℎ𝑖𝑡𝑒

𝑎𝑟𝑟𝑖−1,𝑗 . 𝑣𝑒𝑟𝑡𝑖𝑐𝑎𝑙 = 𝑏𝑙𝑎𝑐𝑘 ∧ 𝑐ℎ𝑒𝑐𝑘𝑖−1,𝑗 = 𝑤ℎ𝑖𝑡𝑒

Finally, the function iterates through the `possible` array and adds the coordinates of the possible moves to the `arrPosibleMove` list.

## **Minimax with Alpha-Beta pruning**

### **Minimax**

The Minimax algorithm is a widely used technique in the field of game theory that is often applied to turn-based zero-sum games. It works by imagining all the possible future moves that both players can make and then choosing the move that is most likely to lead to a win by using the depth-first search. In the zero-sum game, the gain of one player is equivalent to the loss of the other player, meaning that the sum of the players’ gains and losses is always zero. The goal of the algorithm is to minimize the maximum possible loss or maximize the minimum possible gain, depending on the player’s perspective. This means that the player will always choose the best move based on the assumption that their opponent is making optimal moves as well. c ccc

To make things simple, assume that one player has A score, and the other has B score. To know who is winning, just consider the value of S = A - B. If S > 0 that means A wins, S < 0 means B wins, and S = 0 is a draw. So player A will try to maximize the value of S, while player B expects to minimize the value of S.

### **Searching with Alpha-Beta pruning**

This project improves upon previous work by incorporating the Alpha-Beta pruning technique to reduce the time complexity of the algorithm so that the bot can imagine and analyze further moves and various scenarios that can arise. It works by cutting off the evaluation of parts of the tree that are unlikely to lead to a better outcome, thus reducing the computation time. The algorithm maintains two values, alpha and beta, which represent the maximum score for the max player.

## **Heuristic**

In terms of Minimax search, we can imagine the algorithm will search in the graph, where each node is a state of the game, and edges are the transition move.

During the Minimax search, when visiting a leave node, the heuristic function is used to evaluate the utility values at the leave. In the previous work, we created a static board of weights associated with each coin position as shown in the below table:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 4 | -3 | 2 | 2 | 2 | 2 | -3 | 4 |
| -3 | -4 | -1 | -1 | -1 | -1 | -4 | -3 |
| 2 | -1 | 1 | 0 | 0 | 1 | -1 | 2 |
| 2 | -1 | 1 | 1 | 1 | 1 | -1 | 2 |
| 2 | -1 | 0 | 1 | 1 | 0 | -1 | 2 |
| 2 | -1 | 1 | 0 | 0 | 1 | -1 | 2 |
| -3 | -4 | -1 | -1 | -1 | -1 | -4 | -3 |
| 4 | -3 | 2 | 2 | 2 | 2 | -3 | 4 |

The main weakness of this static board heuristic is due to its lack of ability to dynamically change weights to represent the current state of the game. In this project, we proposed a heuristic that can adapt the weights depending on different situations.

Our approach uses 3 criteria to evaluate the utility values of the board state: **Mobility, Stability, Corners Captured.**

The final utility value is the linear combination of each criterion. Each of these criteria is assigned a weight based on its importance.

### **Mobility**

Mobility is the number of available moves a player has, having more moves means more opportunities to capture the opponent’s pieces, it can be considered as an indicator of the ability to control the board.

A natural tactic is to restrict the opponent’s mobility while increasing the player’s own mobility. By doing so, the player can gain more options and flexibility in their gameplay, while limiting their opponent’s options and forcing them into unfavorable positions.

### **Stability**

The stability of pieces is a crucial aspect in the game of Othello, as it determines how difficult a piece is to be flipped by the opponent. To evaluate the stability of a piece, we examine each row, column, and diagonal to check if both ends of the line are either occupied by our own piece or an empty square. If so, the piece is considered stable in that direction. By capturing stable pieces, we can limit the opponent's moves and expand our own options on the board. This heuristic is particularly significant in the early stages of the game, as controlling more stable pieces early on provides more beneficial options in the long run, resulting in increased mobility in the late game.

### **Corners Captured**

The corners in Othello are the four squares located at a1, a8, h1, and h8. These squares hold significant importance as once they are captured, they cannot be taken by the opponent, making them the most stable coins in the game. To ensure that the bot prioritizes capturing these corners, we employ the use of the static board shown above, where each weight assigned to a position indicates the likelihood of capturing a corner if we capture that particular position.

Finally, the utiliity value is calculated by the linear combination of each heuristic as follow:

***Score = Corners Captured + 5 \* Mobility + 10 \* Stability***

The Stability value has most significant weighted, according to our observation that who controls the stability can easily control all other aspects of the game.

# **Demo**

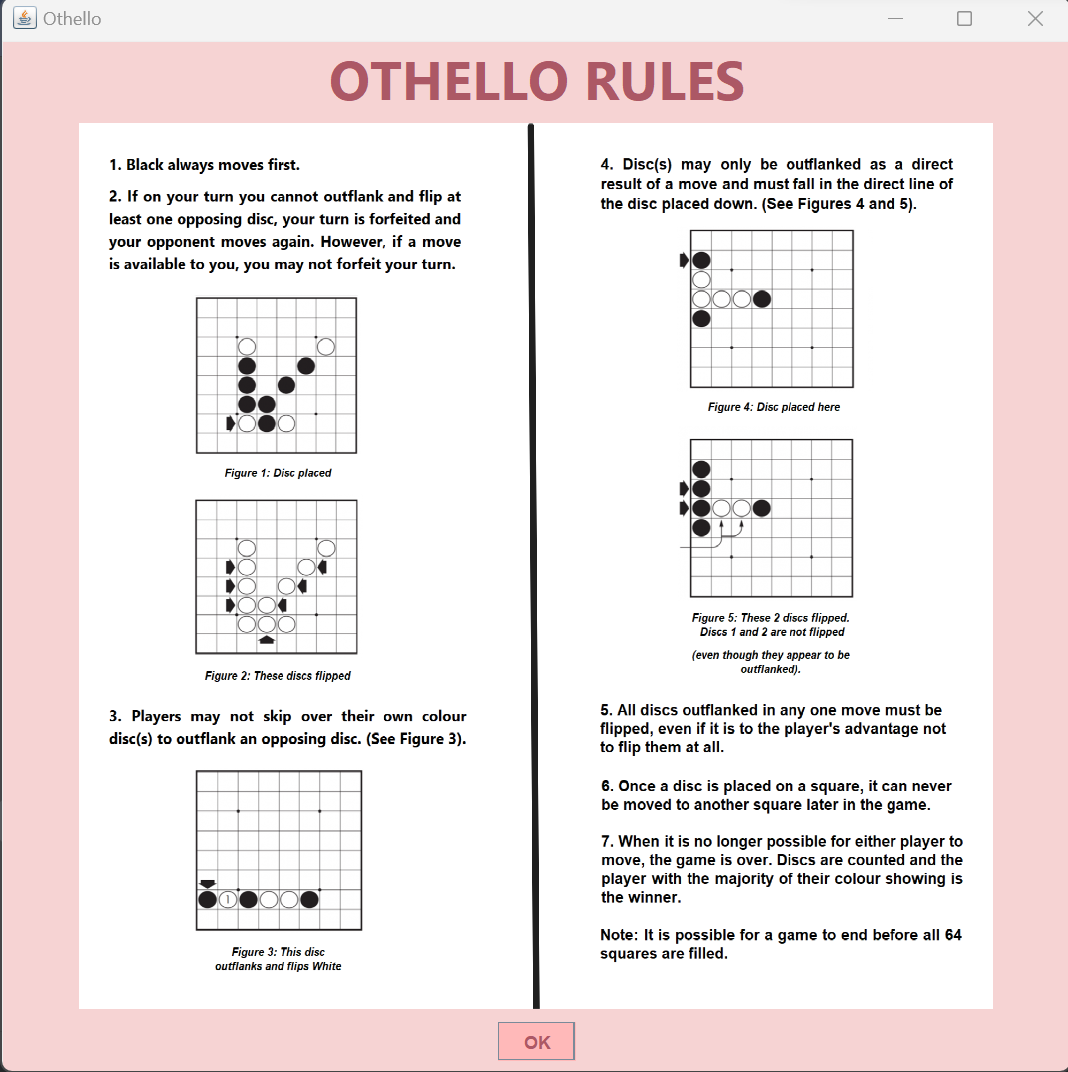
The Main Menu page contain four options: Player vs Player, Player vs BOT, Options, How to play?.

A screenshot of a computer

Description automatically generated

**Figure 5. Main Menu Page**

In the Rule page, it contains the detail Othello rules.



**Figure 6. Rule Page**

In the Options Page, we have four options for table’s background.

A screenshot of a computer screen

Description automatically generated

**Figure 7. Option Page**

In the Player Names Page, the player save their name to play game.

A screenshot of a computer

Description automatically generated

**Figure 8. Player Names Page**

We have two modes game, one is player against player.

A screenshot of a game

Description automatically generated

**Figure 9. Player vs Player**

The other is player against BOT.

A screenshot of a game

Description automatically generated

**Figure 10. Player vs BOT**

# **References**

In our project, the algorithm minimax is implemented based on this paper: [An Analysis of Heuristics in Othello](https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/RUSSIA/Final_Paper.pdf)

# **Contribution**

|  |  |  |  |
| --- | --- | --- | --- |
| **Member Name** | **ID** | **Task** | **Percentage** |
| Huỳnh Trúc Quyên | ITDSIU19051 | Algorithm for Bot making decision and Game play, GUI | 100% |