Hanoi University of Science and Technology

**School of Information and Communication Technology**

=================00o00=================

Introduction to Artificial Intelligence

Capstone Project Report

*An intelligent agent for playing Othello*

**Students**:

|  |  |
| --- | --- |
| Nguyễn Hoàng Hải | 20200193 |
| Lương Nam Khánh | 20205184 |
| Phạm Khánh Linh | 20205186 |
| Lê Minh Thịnh | 20205171 |

**Class**: 131117

**Lecturer**: Asso. Prof. Than Quang Khoat

Table of Contents

[I. Abstract 3](#_Toc108536389)

[II. Introduction 3](#_Toc108536390)

[*a.* History of *Othello* 3](#_Toc108536391)

[*b.* Rules of *Othello* 3](#_Toc108536392)

[c. Problem descriptions 5](#_Toc108536393)

[III. Algorithms 6](#_Toc108536394)

[a. Random strategy 6](#_Toc108536395)

[b. Maximum discs strategy 7](#_Toc108536396)

[c. Minimax 9](#_Toc108536397)

[d. Minimax variants 16](#_Toc108536398)

[ Alpha – Beta pruning 16](#_Toc108536399)

[ Negamax 16](#_Toc108536400)

[ Negamax with Alpha – Beta pruning 17](#_Toc108536401)

[IV. Analysis 18](#_Toc108536402)

[V. Conclusion and possible extensions 22](#_Toc108536403)

[References 24](#_Toc108536404)

[Appendix A: Project Contribution 25](#_Toc108536405)

[Appendix B: How to run our program 26](#_Toc108536406)

# Abstract

Artificial Intelligence (AI) is at the cutting edge of Computer Science. The ubiquitous application of AI in numerous devices and technologies has made our lives more productive and convenient.

The course “Introduction to Artificial Intelligence” has given us many fundamental understandings about AI. We have learnt about some types of Intelligent Agent. In this capstone project, we implement an Intelligent Agent that can play the game *Othello,* using some problem – solving techniques in adversarial search that we have learnt, namely Minimax, Minimax with Alpha – Beta pruning and some variants that we have found when we do this topic.

In this report, we will explain the rules of this game and all the algorithms that we have implemented, compare the performance between different algorithms, and give the conclusions and extensions for the project.

***Keywords***: Adversarial search, zero – sum game, Minimax, Alpha – beta pruning

# Introduction

## History of *Othello*

Othello was originally invented with the name Reversi in England around 1880. The game gained considerable popularity in England at the end of the 19th century. The modern version of the game - the most regularly used ruleset, and the one used in international tournaments - is marketed and recognized as Othello. It was patented in Japan in 1971 by Goro Hasegawa.

Hasegawa's “*How to play Othello*” (“*Osero No Uchikata*”) in Japan in 1974, was published in 1977 in an English translation entitled How to Win at Othello. This was considered as the first book that teaches people to play *Othello*.

The first Othello tournament was the Japan Othello Championship in 1973. Four years later, in 1977, the first World Othello Championship (WOC) was held in Tokyo. Five players from five countries attended. The first World Champion was Hiroshi Inoue.

## Rules of *Othello*

*Othello* is a strategy board game for two players, Black and White, played on an 8 by 8 board. The game begins with four discs placed in the middle of the board as shown in figure 1.

Players alternate taking turns, with Black moving first.

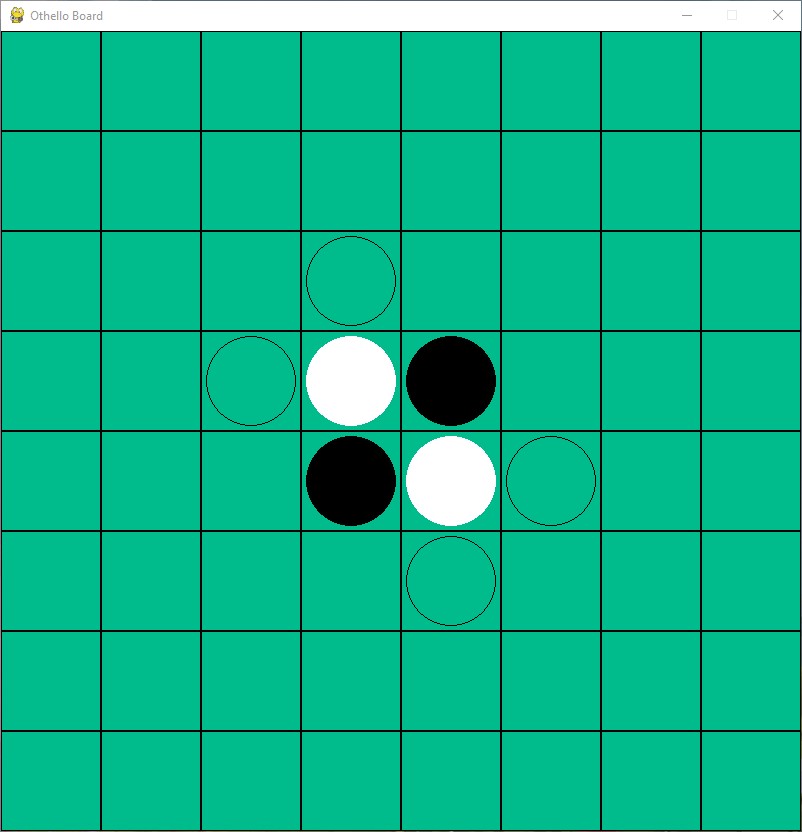


Figure 1: The initial state of the game, with possible moves for Black are shown as translucent discs

A legal move consists of placing a new disc on an empty square and flipping one or more of the opponent’s discs.

Chart, bubble chart

Description automatically generatedChart, bubble chart

Description automatically generated

Figure 2: Black's desired move indicated by red dot

Figure 3: After Black's move. White to move

Any of the opponent’s pieces which are “sandwiched” between the disc just placed on the board and a disc of the same color already on the board should be flipped. Sandwiches can be formed vertically, horizontally, or diagonally. To form a sandwich, all the squares between the new disc and the disc of the same color already on the board must be occupied by the opponent’s pieces, with no blank squares in between.

Pieces may be flipped in several directions on the same move. Any pieces which are caught in a sandwich must be flipped.

If a player has no legal moves, that player passes his turn, and his opponent continues to make consecutive moves until a legal move becomes available to that player.

If a player has at least one legal move available, he must make a move and may not pass his turn.

The game continues until neither player has a legal move. The player has more discs showing his color on the board is the winner. If both players have the same number of discs, then the game is a draw.

Background pattern

Description automatically generatedBackground pattern

Description automatically generated with medium confidenceChart, bubble chart

Description automatically generatedNote that, the game can finish before all the squares on the board are filled. In such cases, normally the number of blank squares will be added to the score of the winner. Here are some examples where the game ends before the board is completely filled:

Figure 4:Vecchi 13 – 51 Nicolas (World Othello Championship 2017, Ghent)

Figure 5:Vlasáková 1 – 63 Schotte (European Grand Prix Prague 2011)

Figure 6:Hassan 3 – 61 Verstuyft J. (European Grand Prix Ghent 2017)

## Problem descriptions

We will implement an intelligent agent that can play *Othello*. This agent has the following properties:

* Performance measure: high winning rate, good computation time, able to play moves that are equivalent to those of the experts
* Environment: board, game rules, black and white discs, players
* Actuator: make a move
* Sensor: observe the state of the board

The game is perfect information, competitive multiagent, deterministic, static, discrete, and sequential.

Based on game rules, this is a zero – sum game. With the goal to win more discs than the opponent, our agent is goal – based agent.

From those properties, the agent uses adversarial search.

The game can be represented as follows:

* Initial state:
  + An 8 – by – 8 number matrix BOARD, fulfilled by 0 to indicate blank squares, except for elements in index [3][3] and [4][4] being filled with 2 to indicate White discs and elements in index [3][4] and [4][3] being filled with 1 to indicate Black discs.
  + A 2 – element array SCORE, with SCORE[0] and SCORE[1] are the scores of Black and White players, respectively.
* Players: Player 1 – Black (AI or human) and Player 2 – White (AI or human)
* Successor function: A list of available legal moves and corresponding state of that turn.
* Goal test: if both players have no more legal moves.
* Evaluation: the scores of both players

# Algorithms

To implement the agent, we will use these algorithms:

* Random strategy
* Greedy – maximum discs strategy
* Minimax with strategy – based evaluation heuristic function

In our project, we will mainly focus on Minimax algorithm. Random and Greedy will be used as the benchmark for the performance and effectiveness of Minimax algorithm.

## Random strategy

As the name suggests, this strategy simply returns a randomly – chosen move in the list of possible moves for a player.

***Time complexity and space complexity:*** O(1), assume that the implementation of random function is negligible.

Text

Description automatically generated

Figure 7: Strategy implementation, using Random library in Python

In our program, we shuffle the list before return the random move to prevent repetitive results.

## Maximum discs strategy

This strategy can be described as “Play the move which captures the most opponent’s discs”. It is derived from the goal of the game – obtain the most discs at the end of the game.

The pseudocode for this strategy can be described as follows:

Figure 8: Pseudocode for maximum strategy

Function maximumDiscs(currentState, currentScore, player, possibleMoves)

bestScoreObtained = -∞

foreach move in possibleMoves:

possibleState = processMove(currentState, move)

possibleScore = calculateScore(possibleState)

bestScoreObtained = max(bestScoreObtained, (possibleScore[player] – currentScore[player])

return move correspond to bestScoreObtained

In the case that there are several moves give the same best score, we will randomly pick one move.

New players often use this strategy throughout every stage of the game. However, this is not an optimal strategy. The deficiency of this strategy has been documented in many papers.

In this report, for simplicity, we will show the problem related to this strategy by a game between *IAGO* and *THE MOOR* in Santa Cruz Open Machine Othello Tournament (1981). Both players were computers; *IAGO* was playing as black, and *THE MOOR* was playing as white.

Diagram

Description automatically generated with low confidence

Figure 9: (a) after 30th move – Black to play; (b) the final game state; (c) the game record

Chart, scatter chart

Description automatically generated

Figure 10: The discs count and differential throughout the game

As can be seen in figure 9, after the 30th move, *THE MOOR* was taking the score advantage (32 to 2). However, the problems are:

* White has the most discs, but it can be easy to see that nothing prevents Black from recapturing them. There are so many moves that Black can perform to recapture the discs.
* White has completely lost its *mobility*. In other words, White has a very limited number of moves that can make. Black can force White to move wherever he pleases.

From the graph, we see that *THE MOOR* maintained his score advantage until middle of the game before the described problems occurred and eventually lost the game.

The game record is shown in figure 9c. The numbered discs show when and by whom a disc was played in the associated square.

The final score for this game is 51 – 13 for *IAGO*.

More details about the failure of this strategy will be presented in the next section, but we can see that the maximum disc strategy is nonoptimal.

***Time complexity and space complexity:*** O(n), where n is number of possible moves at each state of the game, assuming that the time to calculate score and possible state is negligible.

## Minimax

Since *Othello* is two – player, deterministic, fully observable, zero – sum game, Minimax is compatible with this problem.

Minimax is a method used when evaluating a game’s decision tree to ensure that the move chosen by the player maximizes the chance of a favourable outcome for them. For instance, if there exists a move that can increase the player’s score by 3, that move would be more favoured by Minimax over a move that can only increase the score by 1.

Minimax algorithm also evaluates the opponent’s move, assuming that the opponent will always choose a move that minimizes the player’s gain from a move. Hence, the algorithm returns the best outcome for the player should they choose according to the possible moves of the opponent.

Figure 11:Pseudocode for Minimax

function minimax(node, depth, maximizingPlayer)

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

The value returned by Minimax may only be accurate up to a certain number of moves in the future, as there is a depth limit – how far the search will expand. This limit is due to the fact that exploring the whole game tree of this game is nealy impractical.

Since the computation time is limited, the quality of heuristic evaluation function determines the success of Minimax algorithm and makes the difference between decent AI and great AI.

By experimenting with different heuristic evaluation functions, as well as analyzing many features that contribute to making a competitive strategy, we have finally come up with a function that makes the AI perform quite well in practice. We will now explain the function that we have used in our program.

Our heuristic evaluation function is composed of 6 different features: Discs Different, Corner Caption, Corner Closeness, Mobility, Frontier Discs and Static Weight.

For convention, we assume that Black player is the **MAX** player, White is the **MIN** player. Black is calling the Minimax function.

* + 1. Discs Difference ()

The most straightforward feature is disc difference, which is the same as the goal of the game. It measures how many discs of each color are on the board.

Assume that there are *B* black discs and *W* white discs on the board.

If *B > W,* then we let the score *p* be defined as:

If *B < W,* then the score *p* is:

If *B = W,* we let

* + 1. Corner Caption ()

In the board there are 4 corner squares. These squares are the most valuable squares on the board, due to some reasons:

* Once Black can place a black disc on the corner, it is impossible for White to capture that piece. We can say that corner discs are the most *stable* discs on the board.
* Once Black has a corner, Black can build a number of discs that are protected by the corner and can never be captured by White.
* There is a high correlation between the number of corners captured by a player and the actual chance of winning the game.

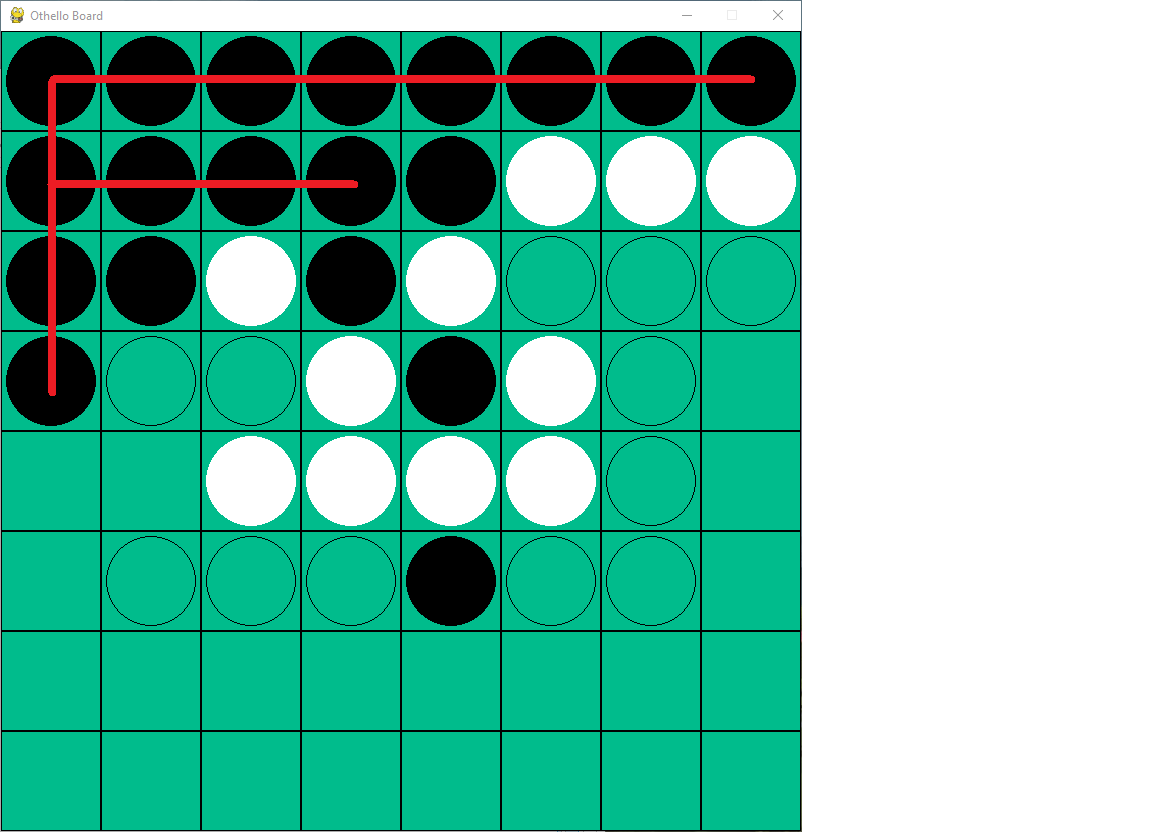


Figure 12: A example when Black has captioned 2 upper corners of the board. The black discs with red line crossed over are stable - White cannot capture them.

We count the number of black discs in corners *B* and number of white discs in corner *W.* Then, we let the score for this feature is

* + 1. Corner Closeness ()

For each corner square there are three adjacent squares (horizontal, vertical, diagonal). These squares are deadly if the corner is empty. For most of the cases, If Black has discs on these squares, White can access to the corner and control the game.

Chart, bubble chart

Description automatically generated

Figure 13: Black to move. If Black choose (1), then (1) and (2) are black discs. White then can take the corner (3).

We count the number of black discs adjacent to empty corners *B,* and number of white discs adjacent to empty corners *W.* Then, the score for this feature is

* + 1. Mobility ()

This feature measures how many moves each player has in each turn. Based on the rules, if White has no possible moves, White must pass his turn and Black can perform several consecutive moves. This gives the chance for Black to accumulate his score and make advantages on the game. Moreover, if White has limited number of moves, usually those moves are poor for him.

Given the state of the game, we calculate the number of possible moves for Black, *B,* and for White, *W.*

If *B > W,* then we let the score *m* be defined as:

If *B < W,* then the score *m* is:

If *B = W,* we let

* + 1. Frontier Discs ()

Discs that are adjacent to empty squares are frontier discs. Most of these discs are vulnerable to being flipped by an opponent’s move in the empty square. Therefore, a player should minimize the number of frontier discs he has.

We count the number of Black frontier discs, *B,* and White frontier discs, *W.*

If *B > W,* then we let the score *f* be defined as:

If *B < W,* then the score fis:

If *B = W,* we let

* + 1. Static Weight ()

We assign a value to every square on the board. Let the score . If a square has Black disc, we add the corresponding value to *s,* otherwise if that square has White disc, we subtract that value from *s.*

Since the board is symmetric both horizontal and vertical axes, we assign value to one quadrant, then mirror it. The upper left quadrant value is represented as follows:



Combining the features and using the weighted linear function, we obtain:



Figure 14: Heuristic function

Each feature of heuristic function has been multiplied by different weights. This difference emphasizes the importance of each feature.

Now we can explain the inefficiency of Maximum strategy: It ignores many features of the game and mistakenly considers the value of all positions on the board to be the same. Besides, there are more reasons for taking (or avoiding) specific squares.

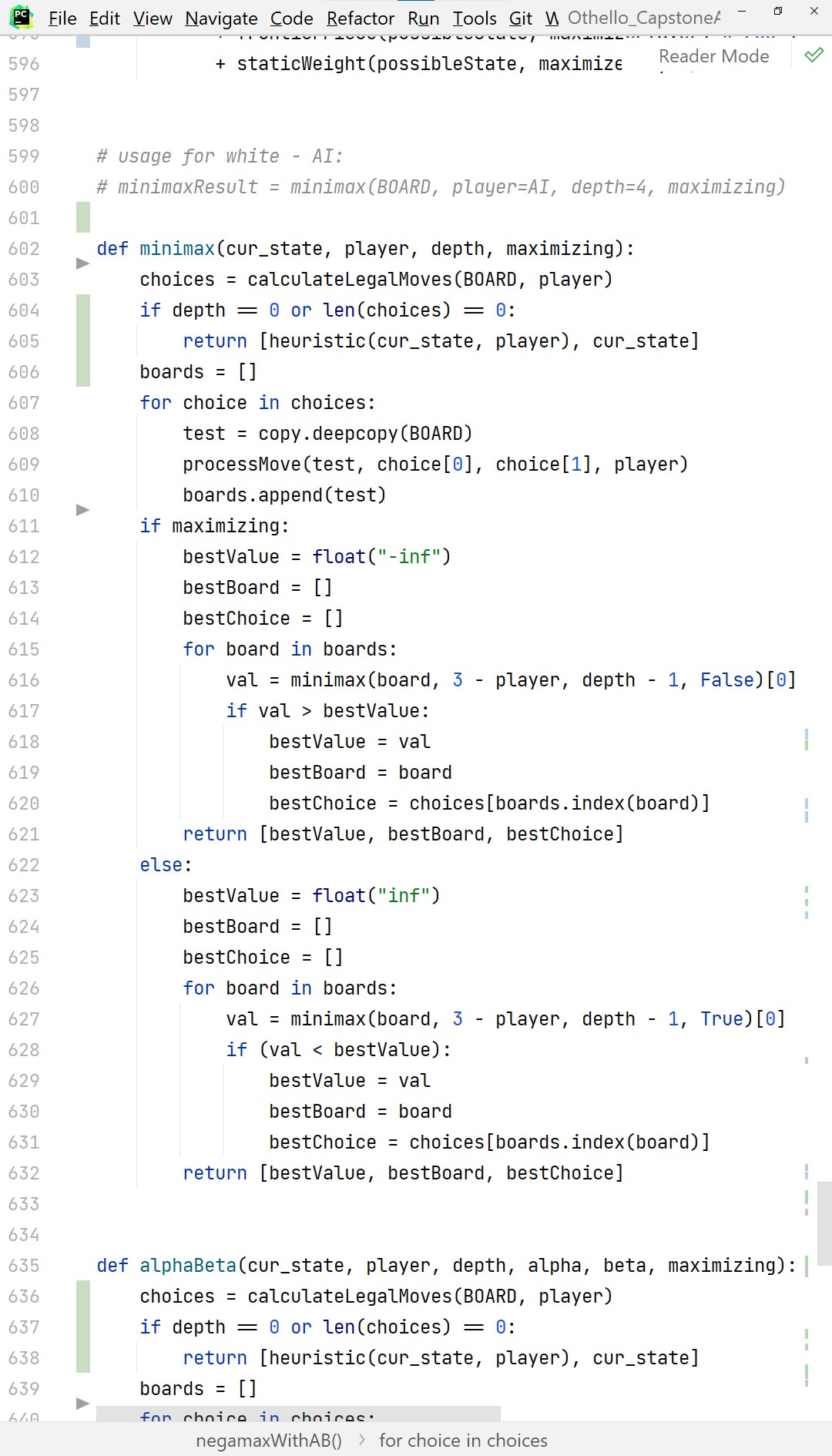
We have just examined the heuristic function. Here is the full implementation of Minimax algorithm.

Figure 15: Minimax implementation

* cur\_state: The state of the game at the time this function is called.
* player: The player to move in the next turn
* depth: Refers to the number of future moves to evaluate in game tree
* maximizing: Will be set to True for the player, otherwise False for the opponent

This function returns a list with 3 elements. The element in index 2 is used to make the move for the AI. The remaining elements are used for analyzing the algorithm.

Initial call for this function:

computerMove = minimax(BOARD, player, DEPTH, True)[2]

In our program, players will be replaced by BLACK or WHITE, depending on which side the AI is. DEPTH is set to 5 to balance the efficiency and computation time.

***Time complexity***: O(), where *b* is branching factor of game tree, *m* is the depth of the tree. In this game, *b* is equivalent to the number of possible moves in each turn.

***Space complexity***: O(), based on Depth – first search.

## Minimax variants

### Alpha – Beta pruning

The running time of Minimax is expensive since it must examine every possible state in the game tree. However, in practice, there is no need to explore all branches because there exist branches that cannot improve the evaluated value. We can “prune” them to save time without affecting the solution.

We will set 2 extra arguments: alpha and beta. Alpha denotes the best move’s value for the player calling the function (the MAX player), while beta denotes the best move’s value for the opponent (the MIN player).

By setting alpha = -∞, beta = +∞, we have initial call for this function:

computerMove = alphaBeta(BOARD, player, DEPTH, float("-inf"), float("inf"), True)[2]

### Negamax

This is a simplified variant of Minimax. The idea of this variant is from the following equation:

By using the equation, the if – else branch in Minimax can be replaced by a single line of code, as shown in the following figure:

Graphical user interface, text, application

Description automatically generated

Figure 16: Implementation of Negamax

### Negamax with Alpha – Beta pruning

Combining Alpha – Beta pruning and Negamax, we can implement a function that works efficiently like Alpha – Beta pruning but using fewer lines of code. Here is the pseudocode for this variant:

For more detailed implementation, please refer to the attached source code.

function negamaxwithAB(state, player, depth, α, β) is

if depth = 0 or no possible moves for player then

return the heuristic value of state

choices := possibleMoves(state, player)

boards := processMove(state, choices, player)

value := −∞

foreach board in boards do

value := max(value, −negamax(board, opponent, depth − 1, −β, −α))

α := max(α, value)

if α ≥ β then

break (\* cut-off \*)

return choice correspond to board

Figure 17: Pseudocode for Negamax with Alpha - Beta pruning

# Analysis

To analyze the performance of algorithms, we have set up the matches as follows:

* AI versus AI: We let the algorithms play against each other. The fixtures are shown in this table:

|  |  |  |
| --- | --- | --- |
| **Black** | **White** | 40 games for each match |
| Random | Minimax |
| Greedy | Minimax |
| Minimax | Random |
| Minimax | Greedy |

Table 1: AI fixtures

Since our Alpha – Beta pruning implementation uses the same heuristic evaluation function as we have described in section of Minimax, in every turn of Minimax we also calculate the time and number of visited game nodes that both algorithms have used.

There are some reasons to set up those fixtures: First, the player takes the first move can have a slight advantage than the opponent. Second, we want to eliminate the repetitive decisions that can occur with our algorithms. Third, the observation will be more certain and reliable.

For these matches, in all games, we record the score after each move, time taken for making a move and winning rate for every algorithm.

* AI versus human player: We only use Minimax to play with human players.

People that we have asked for help can be divided into 2 subgroups:

* Novice: These players understand the rules and have played at least 10 games. They have no or less experience, and do not have any specific strategies.
* Intermediate: These players understand the rules and have been played at least 100 games. They have more experience than Novice group, know some strategies.

Everybody is asked to play at least 5 games with our AI as both sides.

After processing the data from all matches, we obtain the following results:

* AI versus AI

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Black | White | Black wins (games) | Draw (games) | White wins (games) |
| Random | Minimax | 9 | 0 | 31 |
| Greedy | Minimax | 0 | 0 | 40 |
| Minimax | Random | 14 | 1 | 25 |
| Minimax | Greedy | 17 | 0 | 23 |

Table 2: AI versus AI - Match records

Figure 18:Average score and average running time of algorithms in every matches. The horizontal axis in all charts denotes the turns in the game.

* AI versus human

Figure 19:Comparing the average number of nodes explored throughout the game

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Black | White | Black wins (games) | Draw (games) | White wins (games) | Total games |
| Novice | Minimax | 17 | 0 | 42 | 59 |
| Intermediate | Minimax | 38 | 1 | 6 | 45 |
| Minimax | Novice | 23 | 0 | 26 | 49 |
| Minimax | Intermediate | 2 | 0 | 44 | 46 |

Table 3:AI versus human - Match records

By collecting and analyzing the activites in all games, we have following comments:

* Minimax has a better winning rate when playing as white comparing when playing as black.
* Alpha – Beta pruning works very efficacious. The running time is significantly faster than Minimax.

Also, we have received much positive feedback from our players about the AI.

However, we notice a problem relating to Minimax: It performs quite irrationally. This leads to the fact that the winning rate is not good as we have expected.

In the games that Minimax wins, it has put effort to capture the corners and then capturing the edges positions. It also tries to limit the *mobility* of the opponents in some moves.

In contrast, when it loses, we see that it makes many bad moves, for instance that move can give the corner to the opponents…

After carefully checking our implementation, we found some reasons:

* Our heuristic evaluation function is not so optimal: We have tried to find out some factors that can contribute to the success of the game, however, in practice, there are many other complex factors that are missing in our function. For example: finding stable discs, … Besides, players should adapt their strategy to any period and any state of the game, while Minimax only relies on the evaluation score.
* Errors in heuristic evaluation function: We have used “trial and error” method to find out the best weights for each feature in our function – the weights that can achieve the best decision. However, the error is inevitable, and this makes our search cannot return the most optimal move.
* Minimax works correctly assuming the opponent also plays optimally. However, we can see that Random and Greedy are not optimal. While Minimax can play well as White, it is not much better than the opponent when playing as Black. The suboptimality of the opponent is a problem of Minimax.
* Minimax cannot deal with horizon effect. It arises when the program is facing with an opponent’s move that causes serious damage and unavoidable. When playing against human player, our AI has suffered from horizon effect for almost games.
* Minimax can only make decisions at level of individual moves, while human play differently – they consider a higher-level goal, they use the goal to create plausible plans, …

# Conclusion and possible extensions

In this project, we have studied adversarial – search algorithm Minimax by implementing it in the game *Othello.* We have analyzed the game features to create the heuristic evaluation function to estimate the ultility of game state for searching. We observed the performance of Minimax algorithm in different matches against various opponents, and through observation, we have comments about its strengths and weaknesses.

We have some suggestions to extend this project:

* Improving the heuristic evaluation function, as well as considering different periods of the game (beginning, middle, end) to make the evaluation more flexible and accurate.
* Minimax is a **type A strategy** – considers all possible moves to a certain depth, then estimate the utility of that state. We can consider using some **type B strategy** methods, which ignores moves that look bad, and follow the branch that looks “reasonable” as far as possible.

Use a different, advanced search algorithms like Monte – Carlo Tree Search, … Also, some techniques in Machine Learning like Reinforcement Learning, Deep Learning, … can be used to create a more powerful AI. AlphaGo is the shining example that uses these learning techniques to defeat human players in *Othello*.

* Our program currently does not limit the time for each turn. We want to extend the functionality of our program by adding the time limit for the players. In that case, we can use some searching techniques with time constraints, like Minimax search with Iterative Deepening…

# References

|  |  |
| --- | --- |
| [1] | B. Rose, Othello and A Minute to Learn...A lifetime to Master, Anjar Co., 2005. |
| [2] | T. Q. Khoat, "Lecture slide of Introduction to Artificial Intelligence course, IT3160E," [Online]. Available: https://users.soict.hust.edu.vn/khoattq/lectures/IT3160E-AI/. |
| [3] | Z. Zhang and Y. Chen, "Searching Algorithms in Playing Othello," 2014. |
| [4] | A. Barber, W. Pretorius, U. Qureshi and D. Kumar, "An Analysis of Othello AI Strategies," 2019. |
| [5] | V. Sannidhanam and M. Annamalai, "An Analysis of Heuristics in Othello". |
| [6] | M. J. Korman, "Playing Othello with Artificial Intelligence," 2003. |
| [7] | S. J. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, 4th ed., Pearson Education, 2021. |
| [8] | "Minimax," [Online]. Available: https://en.wikipedia.org/wiki/Minimax. |
| [9] | "Alpha - beta pruning," [Online]. Available: https://en.wikipedia.org/wiki/Alpha%E2%80%93beta\_pruning. |
| [10] | "Negamax," [Online]. Available: https://en.wikipedia.org/wiki/Negamax. |
| [11] | "Reversi," [Online]. Available: https://en.wikipedia.org/wiki/Reversi. |

# Appendix A: Project Contribution

|  |  |
| --- | --- |
| Students | Task |
| Nguyễn Hoàng Hải | Analyze game strategies; implement Negamax; setup AI vs human players matches and collect records; format report; presenter. |
| Lương Nam Khánh | Implement gameplay; implement Random, Greedy algorithms; analyze game strategies; setup AI vs AI matches and collect records. |
| Phạm Khánh Linh | Analyze game strategies; implement Minimax, Alpha – beta pruning; setup AI vs human players matches and collect records; analyze game records; format report. |
| Lê Minh Thịnh | Implement gameplay, setup UI, setup AI vs AI matches and collect records; analyze game records; format report; format slides. |

# Appendix B: How to run our program

Our program is written in Python 3, using Pygame library to show the UI for the board.

To run our program, first you need to install Python compiler to your computer.

Next, install Pygame library. Usually, it can be installed by executing the command:

pip install pygame

Please note that the installation can be varied in different operating systems.

After successfully the requirements, from the terminal, navigate to the folder containing our source code. Execute our program by following command:

python othello.py

or

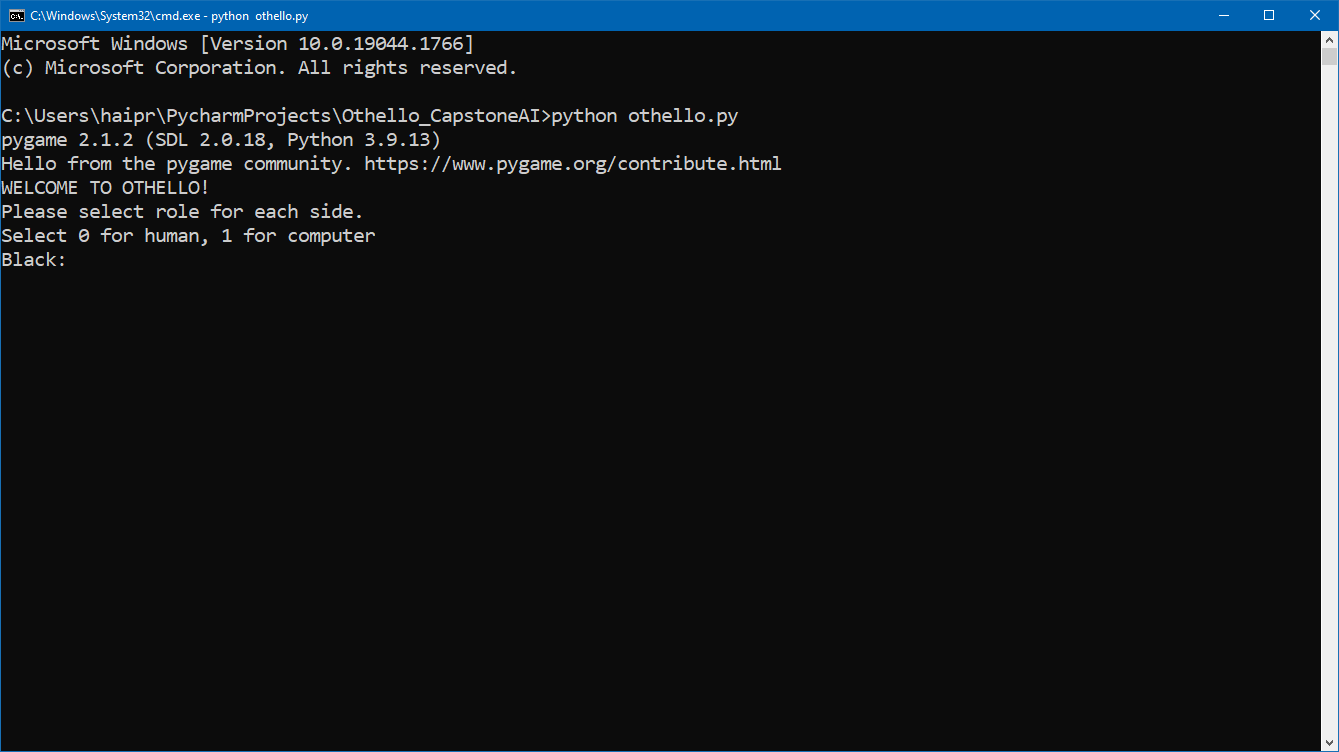
python3 othello.py

Figure i: The program starts

The program will ask for the role for Black side and White side. For example, if you want to Black will be played by human, and White will be played be AI, enter 0, then 1.

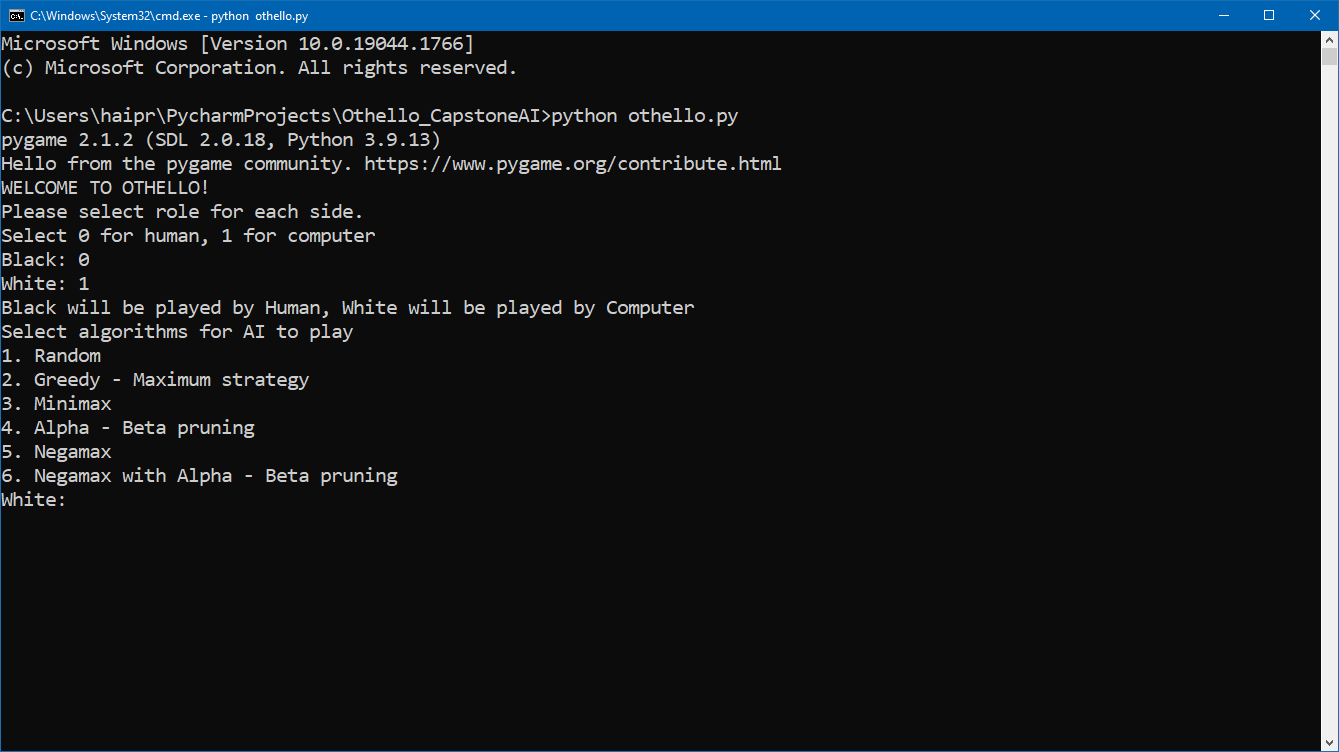


Figure ii: After choosing the role

For any AI player that you have chosen, specify the algorithm that you want the AI to use. Select any option from 1 to 6 as shown in the program.

To illustrate, we will set human play as Black and AI play as White.

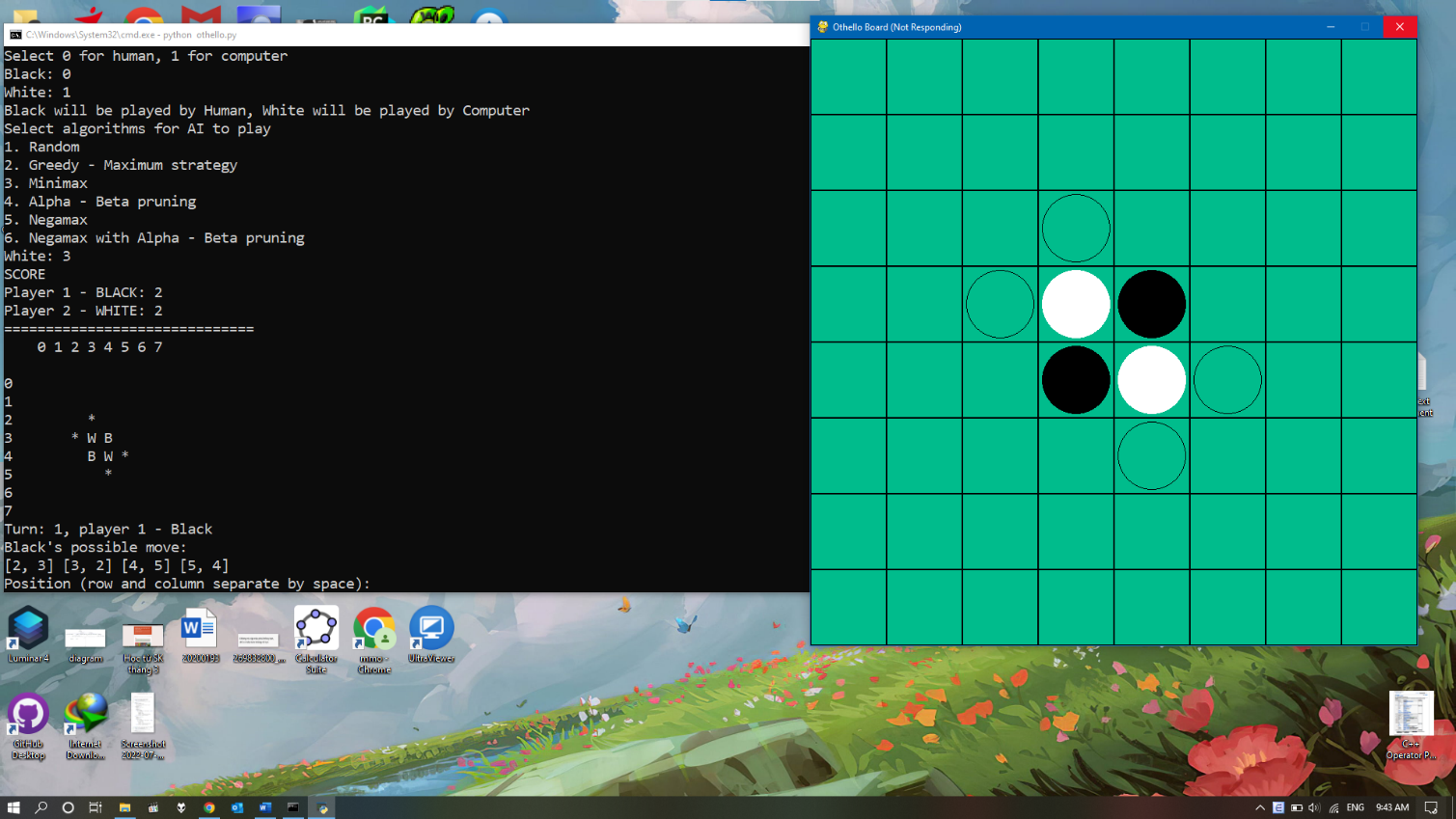
After choosing the algorithm, the game starts. For each turn, our program will display the current score of each player, the state of the board, the player currently has turn and the possible moves for that player.

Figure iii: Black to move

To make a move, enter the row and column number of the position that you choose. The program will update the game board according to the chosen move.

If you accidentally enter invalid position, it will prompt you to enter again.

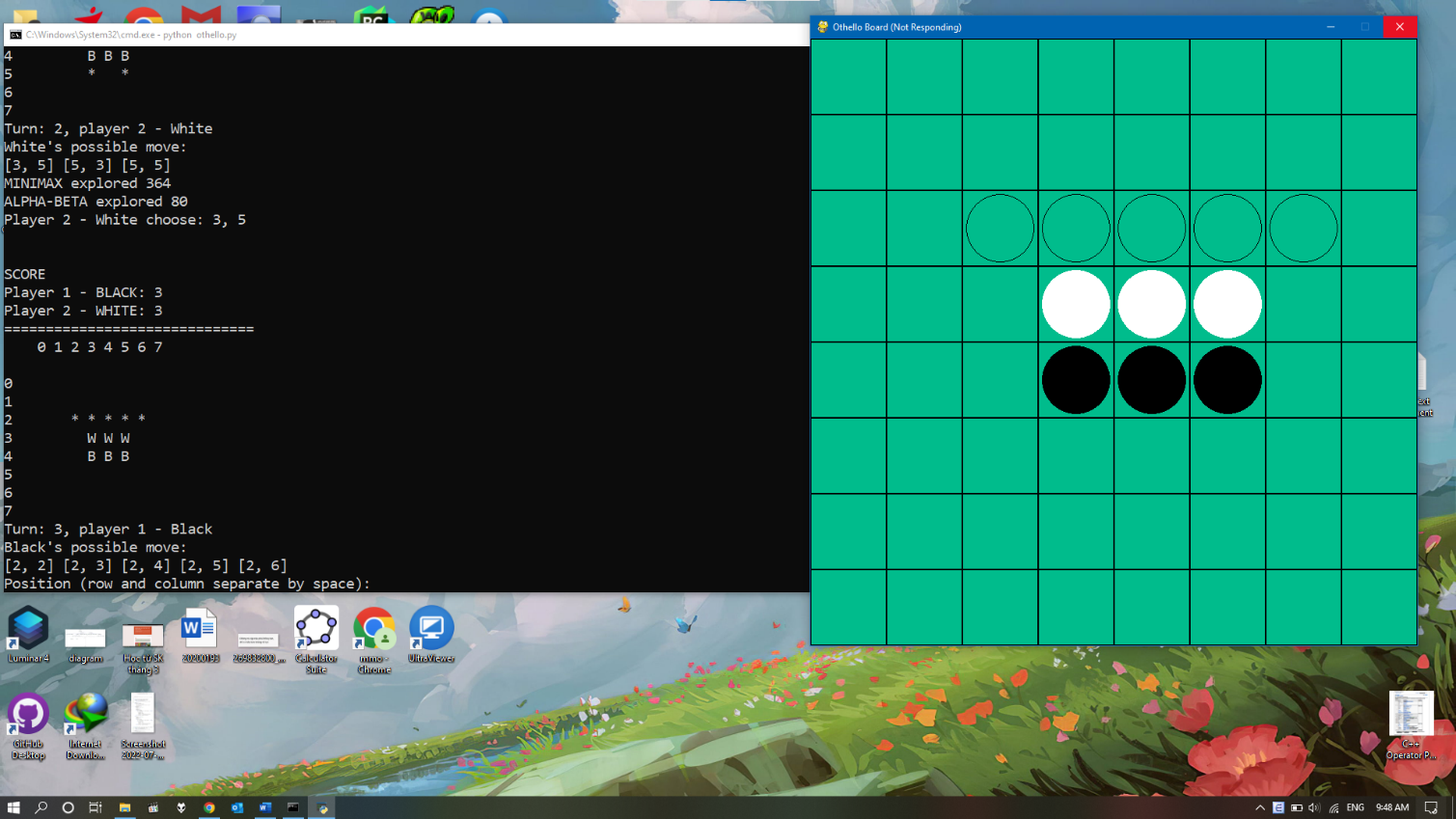
The AI will choose his own choice and the board will automatically update. Sometimes the board can be updated very fast, however you always can check the activity of the game by scrolling up the terminal window.

Figure iv:AI has performed his move, human player to move

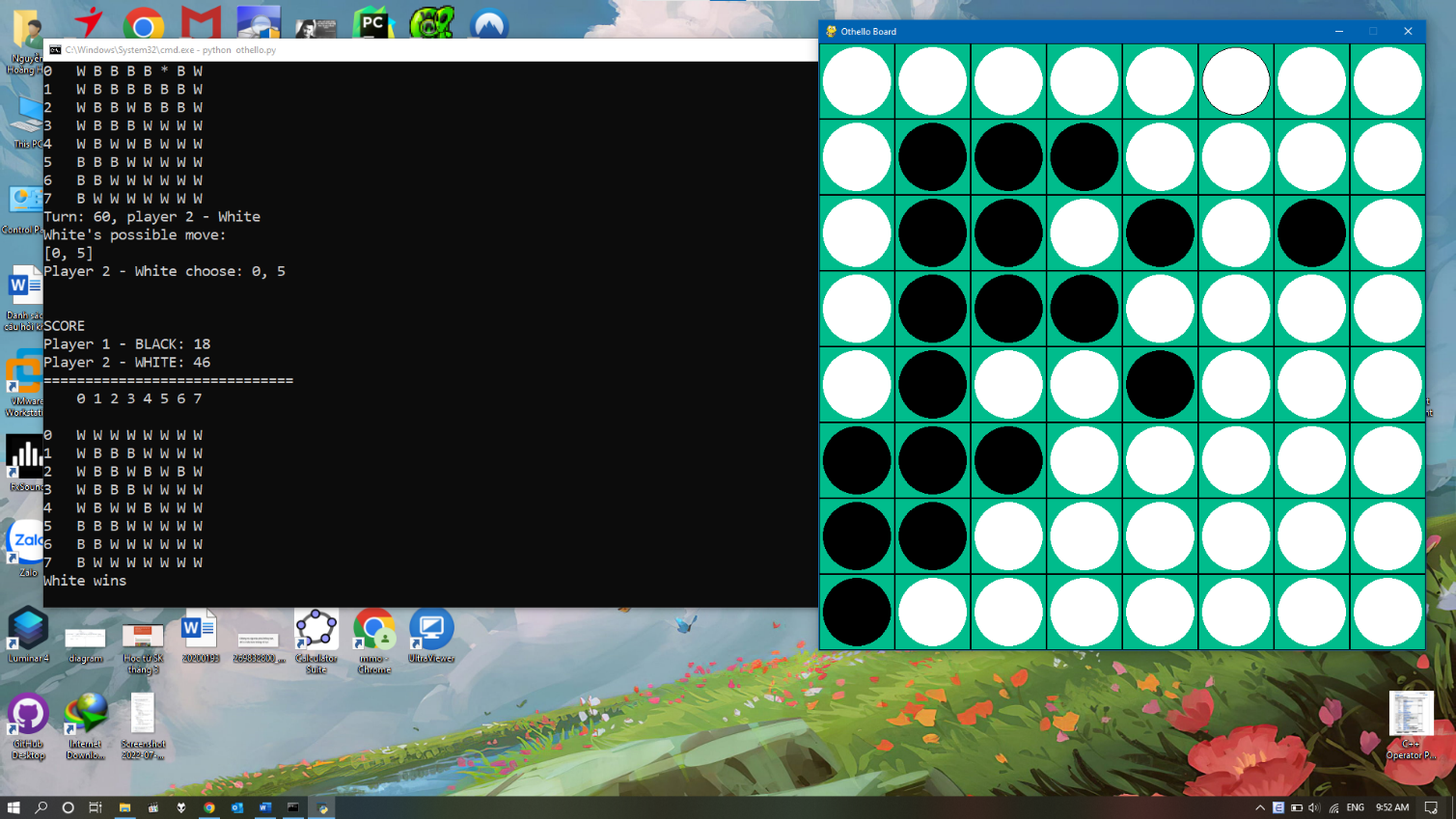
You continue to play until the game ends. When the game ends, it will print the winner of the game.

Figure v: The game finished