Implementation of an Othello Player

Sara Gholami, Sara Ghaemi

I. Introduction

The Othello game is the modern version of a board game named Reversi [1]. In this project, we have developed a player for Othello game based on Heuristic functions. Moreover, we have implemented an Othello program that has three playing modes. In this program, the user can play against another user, against a computer player or watch two computer players play against each other. We have evaluated our different Othello players based on the results of their two by two game. We also compared the best implemented player with other Othello players available online.

II. Related Work

The research about Othello in Artificial Intelligence goes back to at least 70's [2] where the strongest AI to developed to play Othello was named IAGO [3]. Rosenbloom et al. used three strategies as their heuristic factors the maximum disk strategy, the weighted square strategy, and the minimum disk strategy and used these factors as their evaluation method in their search strategy. Liskowski et al. [4] designed an Othello player using deep neural networks. They used different Convolutional neural networks (CNNs) architectures with a database of experts' moves to train their player. Benbassat et al. [5] presented the use of genetic programming for zero-sum, deterministic, full-knowledge board games and as an example Othello game. They used explicitly defined introns and a selective directional crossover method. Frankland et al. [6] investigated the incorporation of reinforcement learning and mobility with genetic programming to improve the evolved strategies in genetic programming. Hingston et al. [7] used the Monte Carlo search for Othello game and demonstrated that it is feasible to implement an Othello player with this approach. Sannidhanam et al. [8] implemented their player using multiple heuristic functions such as score, corners, mobility, stability, etc. Then they used these heuristics with three different search strategies, minimax, alpha-beta search and alpha-beta search with iterative deepening.

III. The game Othello

Othello, also known as Reversi, is a two-player combinatorial game usually played on an 8*8 board. The two players have black and white stones respectively. The winner is the player with the highest number of stones on the board. As a result, the goal of each player is to maximize their score which is the number of stones they have on the board. Usually, the black player starts the game from the initial position which is shown in Figure 1a. When each player makes a move, all opponent's stones that are enclosed from both sides horizontally, vertically or diagonally, will flip color (e.g., in Figure 1b, black has made a move from the initial position). Moreover, a move is legal if it causes at least one of the opponent's stones to flip color. In Figure 1a, red dots show valid moves for the current player which is black in this case. The players alternate turns to play on the board until none of them has a possible move. In a special case in the game, one player may not have any legal moves. In this case, the player will lose their turn, and the opponent makes a move instead.

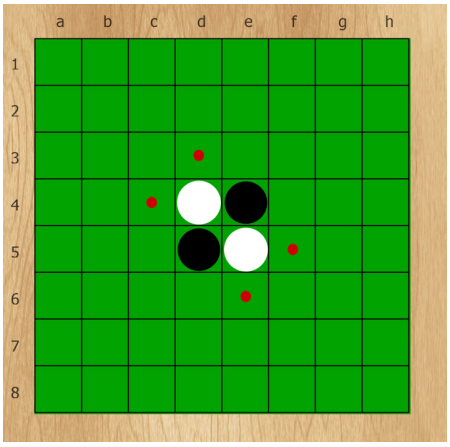
IV. Computer Player

A computer game player is composed of two important parts, its search algorithms, and its heuristic functions. These two components are explained in the following sections.

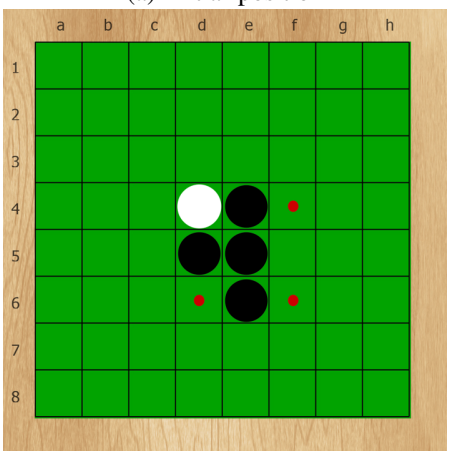
A. Search Algorithms

The search algorithm allows the computer to look at the game tree and evaluates the consequences of each move. The further the computer player looks at the game tree, the better it will be able to play as it will have a more accurate estimation of its move's outcome. There are multiple approaches to explore game tree such as minimax, alpha-beta pruning [9], negamax, etc.

We used alpha-beta pruning as the search mechanism for the computer player where it will look at the different depths of the tree based on the chosen difficulty level of the game.



(a) Initial position



(b) Black moves in e6 from the initial position

Fig. 1: Othello game boards

B. Heuristics

To evaluate the current state of the board, the computer player considers some heuristic functions. These functions provide the player with an estimation of how good the current state of the board is. In the Othello game, some specific factors influence the value of each state of the board. In this project, we have used score, mobility, potential mobility, corners captured, stability, and static weighting as our heuristic functions. We have scaled all the heuristic values to be between -100 and 100 so that they are comparable.

1) Score: For each player, score is the number of their stones. In each step computer player should try to maximize its score and minimize it's opponent's score. The score heuristic value is calculated based on the following formula:

```
Heuristic Score = 100 * (Max
 ↪ Player Score - Min Player
 ↪ Score) / (Max Player Score +
 ↪ Min Player Score)
```

Note that since there is always at least four stones in the board, Max Player Score + Min Player Score cannot be zero, and we do not need to check this condition.

2) Mobility: Mobility of a player is defined as the number of valid moves that they have. One strategy in playing Othello game is to maximize your mobility and minimize your opponent's mobility. A player with a large number of legal moves has more choices and can decide which helps improve their status in the game or gain control of the game. On the other hand, a player with a lower number of possible moves may lose their turn and have a lower chance to change the status of the game to their favor. Our computer player takes advantage of this heuristic in the following fashion.

```
if (Max Player Mobility + Min Player
 ↪ Mobility) != 0:
    Heuristic Mobility = 100 * (Max
     ↪ Player Mobility - Min Player
     ↪ Mobility) / (Max Player
     ↪ Mobility + Min Player
     ↪ Mobility)
else:
    Heuristic Mobility = 0
```

3) Potential Mobility: Potential Mobility is based on the same idea as Mobility heuristic, but it estimates the mobility in the near future. As the computer player wants to gain more mobility as a result of its next move, it wants to achieve more mobility in the future as well.

Potential Mobility can be calculated by looking at the game tree, and the more we look in depth of the tree, we can estimate it with higher accuracy. However, looking at the game tree is computationally expensive. As a result, computing potential mobility has a trade-off between computation complexity and accuracy of the result. To calculate the potential mobility, our computer player will count the number of empty squares beside at least one of the opponent's stones which are not in its current legal moves.

```
if (Max Player Potential Mobility +
 ↪ Min Player Potential Mobility) !=
 ↪ 0:
    Heuristic Potential Mobility = 100
     ↪ * (Max Player Potential
     ↪ Mobility - Min Player
     ↪ Potential Mobility) / (Max
     ↪ Player Potential Mobility +
     ↪ Min Player Potential Mobility)
else:
    Heuristic Potential Mobility = 0
```

4) Corners Captured: Corners are the most important squares on the board as if one player captures a corner, the opponent will not be able to flip its color. As a result, in many of the situations, capturing corners can determine the outcome of the game.

For the computer player to evaluate a board based on the corners, it needs to sum the number of captured corners and potential corners of each player. Each player's potential corners are the ones that can be captured in the next move only by them. There is also a factor named common corners which are the potential corners that both players can capture. From the computer player's perspective, the next move is the opponent's turn to play. Therefore, the common corners have a high probability to be captured by the opponent and they should have a negative weight in the heuristic corners.

```
if (Max Player Corners + Common
 ↪  Corners + Min Player Corners) !=
 ↪  0:
    Heuristic Corners = 100 * (Max
     ↪  Player Corners - Common
     ↪  Corners - Min Player Corners)
     ↪  / (Max Player Corners  +
     ↪  Common Corners + Min Player
     ↪  Corners)
else:
    Heuristic Corners = 0
```

5) Stability: In each state of the board, some stones are stable, and some stones are unstable. A stable stone is a stone which the opponent is not able to flip its color during the game. Based on the rules of the game the corners are the only squares which are stable from the beginning of the game. After capturing one corner, the player is able to create more stable stones upon that corner.

In order to calculate the stability of a state of the board, we assign a positive weight to the computer player's stable stones and negative weights to the opponent's stable stones.

```
if (Max Player Stability + Min Player
 ↪  Stability) != 0:
    Heuristic Stability = 100 * (Max
     ↪  Player Stability - Min Player
     ↪  Stability) / (Max Player
     ↪  Stability + Min Player
     ↪  Stability)
else:
    Heuristic Stability = 0
```

6) Static Weighting: Static weight heuristic uses a static matrix where each element of the matrix corresponds to one square on the board, and the value in that element represents the value of that square during a game.

This matrix represents an overall view of the board where corners are the most important squares, and edges are semi-important. On the other hand, squares adjacent to corners have the lowest value, and computer player needs to avoid playing in those squares since there is a high chance the opponent captures the corner in its next move.

The value of a state of the board is calculated by summing the weights of squares where a player has a stone on them. The following matrix is an 8×8 matrix corresponding to an 8×8 board, where we have similar matrices for larger boards as well.

$$\begin{bmatrix} 4 & -3 & 2 & 2 & 2 & 2 & -3 & 4 \\ -3 & -4 & -1 & -1 & -1 & -1 & -4 & -3 \\ 2 & -1 & 1 & 0 & 0 & 1 & -1 & 2 \\ 2 & -1 & 0 & 1 & 1 & 0 & -1 & 2 \\ 2 & -1 & 0 & 1 & 1 & 0 & -1 & 2 \\ 2 & -1 & 1 & 0 & 0 & 1 & -1 & 2 \\ -3 & -4 & -1 & -1 & -1 & -1 & -4 & -3 \\ 4 & -3 & 2 & 2 & 2 & 2 & -3 & 4 \end{bmatrix}$$

C. Linear Combination of Heuristic Values

To find a final evaluation of the board, we have linearly combined all the heuristic functions based on the following formula:

```
Heuristic value = Heuristic Score +
 ↪  Heuristic Mobility +  Heuristic
 ↪  Potential Mobility + 2 * Heuristic
 ↪  Static Weighting + 8 * Heuristic
 ↪  Stability + 8 * Heuristic Corners
```

We assigned the weights to each heuristic based on our interpretation of their importance and experiments. The score, mobility and potential mobility heuristics have weight one as their almost have the same level of importance. However, the static weighting heuristic has weight two as in any state of the board it has an overview of the whole board and stops the three previous heuristics from being greedy. On the other hand, the stability and corners heuristics have weight 8. The reason for this high weight is due to the importance of these heuristics. At the beginning of the game, these two heuristics are zero until one of the players is able to capture a corner square. Then the value of these heuristics needs to be higher than the rest and dominate their effect.

V. Implementation

We have implemented our game in Python and used PyQt5 for designing the graphical user interface(GUI). Our program is publicly available on

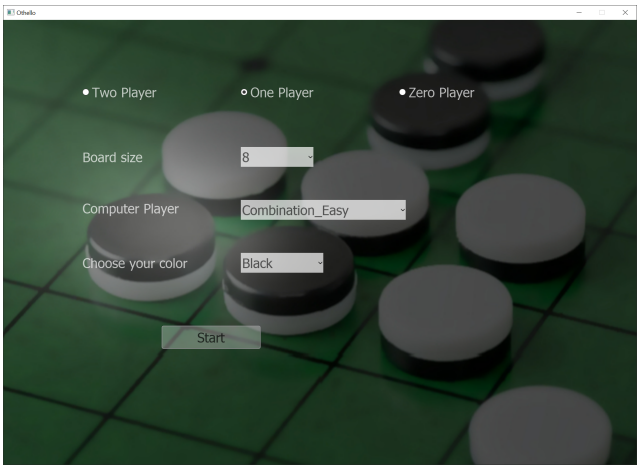
GitHub[1] and a short video showing different parts of the program is available on Youtube[2].

The implemented program has a setup page and a game page. Figure 2a shows an example of the setup page, and Figure 2b shows an example of the game page. In the setup page, the user can select the game mode which is either two player, one player, or zero player. In the two player mode, two users can play a game of Othello against each other, and they can select the board size to be 8×8, 10×10, 12×12, or 14×14. In the one player mode, a user can play against a computer player. In this mode, other than the board size, the user can also select the difficulty level of the game (Easy, Normal, Hard) and their stone color (Black or White). Finally, in the zero player mode, the user can select the difficulty level of the two computer players and the board size. In the game page, each player's scores are shown along with the player who has the current turn. The user has the option to reset the current game or go back to the setup page.

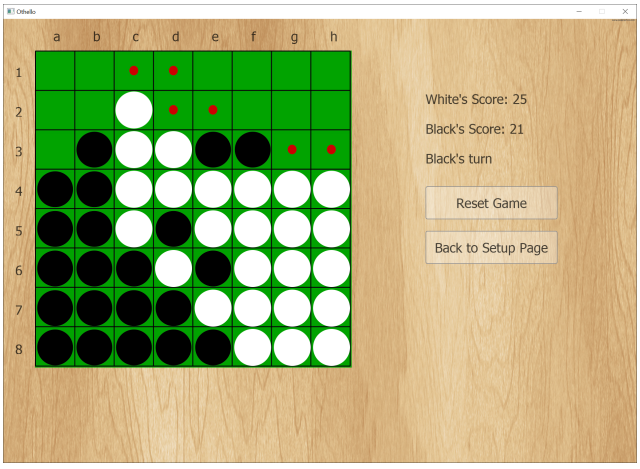
To implement the different difficulty levels of the player, we have used different depths in the search algorithm explained in Section IV-A. The easy player, normal player, and hard player search the tree with depth 1, 2, and 3 respectively. To choose what depth to use in each player, we first tried to find an appropriate depth size for the hard player. We expected the hard player to win all the other players and also some online players in a reasonable amount of time. As a result, we tested different depths for the hard player and calculated the average decision time for the player. This time depends on the number of valid moves the player has and increases dramatically when this number increases. Table I shows the average decision time for search depth of one to six. Since a depth more than 3 has a high decision time and player with search depth 3 was able to win all the online players that we tested, we chose the hard player to have a search depth of 3.

VI. Evaluation

To evaluate our player we let the different levels of our player play together. As expected, the normal player won the easy player, and the hard player won the normal player. As these players are static, this result does not change over multiple experiments. We have evaluated our program by letting its hard player play with some of the available online players. As our player evolved, it was able to defeat all of its opponents.



(a) Setup Page



(b) Game Page

Fig. 2: The implemented program

TABLE I: Average decision time for computer player for different search depths

Search depth	Average move time (S)
1	0.0347
2	0.1543
3	2.1905
4	4.1331
5	196.3013

[1]https://github.com/SaraGhlm/Othello

[2]https://youtu.be/vUpWKo6b4y0

REFERENCES

[1] B. I. Purcaru, *Games vs. Hardware. The History of PC video games: The 80's*. Purcaru Ion Bogdan, 2014.

[2] C. N. Silla, M. Paglioney, and I. G. Mardegany, "jothellot: A java-based open source othello framework for artificial intelligence undergraduate classes," in *2016 IEEE Frontiers in Education Conference (FIE)*, pp. 1–7, IEEE, 2016.

[3] P. S. Rosenbloom, "A world-championship-level othello program," *Artificial Intelligence*, vol. 19, no. 3, pp. 279–320, 1982.

[4] P. Liskowski, W. M. Jaskowski, and K. Krawiec, "Learning to play othello with deep neural networks," *IEEE Transactions on Games*, 2018.

[5] A. Benbassat and M. Sipper, "Evolving board-game players with genetic programming," in *Proceedings of the 13th annual conference companion on Genetic and evolutionary computation*, pp. 739–742, ACM, 2011.

[6] C. Frankland and N. Pillay, "Evolving game playing strategies for othello incorporating reinforcement learning and mobility," in *Proceedings of the 2015 Annual Research Conference on South African Institute of Computer Scientists and Information Technologists*, p. 16, ACM, 2015.

[7] P. Hingston and M. Masek, "Experiments with monte carlo othello," in *Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*, pp. 4059–4064, IEEE, 2007.

[8] V. Sannidhanam and M. Annamalai, "An analysis of heuristics in othello," 2004.

[9] D. E. Knuth and R. W. Moore, "An analysis of alpha-beta pruning," *Artificial intelligence*, vol. 6, no. 4, pp. 293–326, 1975.