Lebanese University

Lebanese University
Faculty of Sciences

Othello Game with AI Opponent

Final Report

Supervised By:

Dr. Kifah Tout - Dr. Rami Baida

Submitted By:

Farah Al Wardani

Mohammad Chalhoub

Yara Al Ali

INFO407

2019-2020

Table of Contents

1 Abstract

This project's objective is to create an implementation of Othello board game with an AI opponent and three levels of difficulty for the user, beginner which is random, intermediate, and expert. The implementation is done using Unity game engine to generate gameplay and the GUI, and the minimax algorithm with alpha-beta pruning was adapted to decide the moves which the AI makes during the game.

2 Introduction

Reversi is an abstract strategy board game invented during the Victorian era. It has remained popular for more than a century being given a boost in the 1970s when a version of it was re-marketed under the name 'Othello' and another in the 1990s when a computer version of it was included with the Microsoft Windows operating system.

Reversi involves play by two parties on an eight-by-eight square grid with pieces that have two distinct sides. Pieces typically appear coin-like, with a light and a dark face, each side representing one player. The goal for each player is to make pieces of their colour constitute a majority of the pieces on the board at the end of the game, by turning over as many of their opponent's pieces as possible.

3 Basic Othello Ruleset and Game Play

3.1 Game Play

- Othello is a strategy board game played between 2 players. One player plays black and the other white.
- Each player gets 32 discs and black always starts the game.
- Then the game alternates between white and black until:
    - one player can not make a valid move to outflank the opponent.
    - both players have no valid moves.
- When a player has no valid moves, he passes his turn and the opponent continues.
- A player can not voluntarily forfeit his turn.
- When both players can not make a valid move the game ends.

3.2 Valid Moves

- Black always moves first.
- A move is made by placing a disc of the player's color on the board in a position that "out-flanks" one or more of the opponent's discs.
- A disc or row of discs is outflanked when it is surrounded at the ends by discs of the opposite color.
- A disc may outflank any number of discs in one or more rows in any direction (horizontal, vertical, diagonal).

- All the discs which are outflanked will be flipped, even if it is to the player's advantage not to flip them.
- Discs may only be outflanked as a direct result of a move and must fall in the direct line of the disc being played.
- If you can't outflank and flip at least one opposing disc, you must pass your turn. However, if a move is available to you, you can't forfeit your turn.
- Once a disc has been placed on a square, it can never be moved to another square later in the game.

3.3 End of Game

- When it is no longer possible for either player to move, or the board is full, the game is over.
- The discs are then counted and the player with the majority of his or her color discs on the board is the winner.
- A tie is possible.

4 Minimax Algorithm

4.1 Introduction

Minimax is a decision-making algorithm, typically used in turn-based, two player games. The goal of the algorithm is to find the optimal next move.

In the algorithm, one player is called the maximizer, and the other player is a minimizer. If we assign an evaluation score to the game board, one player tries to choose a game state with the maximum score, while the other chooses a state with the minimum score.

In other words, the maximizer works to get the highest score, while the minimizer tries to get the lowest score by trying to counter moves.

It is based on the zero-sum game concept. In a zero-sum game, the total utility score is divided among the players. An increase in one player's score results into the decrease in another player's score. So, the total score is always zero. For one player to win, the other one has to lose. Examples of such games are chess, poker, checkers, tic-tac-toe, and othello.

Every board state has a value associated with it. In a given state if the maximizer has upper hand then, the score of the board will tend to be some positive value. If the minimizer has the upper hand in that board state then it will tend to be some negative value. The values of the board are calculated by some heuristics which are unique for every type of game.

4.2 The Algorithm

The minimax search does a depth-first exploration of the entire game tree. The heuristic functions are utilized at the leaves to provide utility values. The utility values are then

backed up all the way to the root. However, in games with huge game trees such as othello, reaching the terminal nodes is not efficient and might be even computationally infeasible. So, a certain depth should be specified. The manner in which the values are backed up to a node depends on whether the node is a min node or a max node. The max player has to make a move while at a max node, while the min player has to do so at the min node. The max player is the player who has to make the actual next move in the game, and has to maximize his/her utility value, while the min player does the reverse. A typical minimax tree is such that the min player and max player alternate. This need not necessarily hold throughout, since turns may be skipped by players in certain games, such as Othello.

Here's the algorithm's pseudo code in its basic form:

```
function minimax(board, depth, isMaximizingPlayer):
    if current board state is a terminal state :
        return value of the board
    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each move in board :
            value = minimax(board, depth-1, false)
            bestVal = max( bestVal, value)
        return bestVal
    else :
        bestVal = +INFINITY
        for each move in board :
            value = minimax(board, depth-1, true)
            bestVal = min( bestVal, value)
        return bestVal
```

4.3 Alpha-Beta Pruning

Minimax search is improved by introducing alpha-beta pruning. Alpha-beta search is similar to minimax, except that efficient pruning is done when a branch is rendered useless. Such pruning tends to be rather effective and the search can proceed to great depths, allowing the computer to implement a relatively more powerful look ahead. Pruning is done when it becomes evident that exploring a branch any further will not have an impact on its ancestors. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta: **Alpha** is the best value that the **maximizer** currently can guarantee at that level or above, it is initially negative infinity. **Beta** is the best

value that the **minimizer** currently can guarantee at that level or above, it is initially positive infinity.

Here's the algorithm's pseudo code with alpha-beta pruning:

```
function minimax(node, depth, isMaximizingPlayer, alpha, beta):
    if node is a leaf node :
        return value of the node
    if isMaximizingPlayer :
        bestVal = -INFINITY
        for each child node :
            value = minimax(node, depth-1, false, alpha, beta)
            bestVal = max( bestVal, value)
            alpha = max( alpha, bestVal)
            if beta <= alpha:
                break
        return bestVal
    else :
        bestVal = +INFINITY
        for each child node :
            value = minimax(node, depth+1, true, alpha, beta)
            bestVal = min( bestVal, value)
            beta = min( beta, bestVal)
            if beta <= alpha:
                break
        return bestVal
```

4.4 Evaluation Function

The heuristic functions control the ability of the computer to correctly determine how good a particular state is for a player. A number of factors determine whether a given state of the game is good for a player. The evaluation function is unique for every type of game. The basic idea behind the evaluation function is to give a high value for a board if **maximizer**'s turn or a low value for the board if **minimizer**'s turn. For Othello, factors such as mobility, stability, corners and coin parity determine how favorable a particular position is for a player. The most intuitive way to calculate a heuristic value is to create a linear combination of the

quantitative representation of the various important factors. We can also add a factor calculated from statically assigned weights to squares on the board.

5 Implementation and Game Representation

5.1 UI Generation in Unity

The UI was generated using Unity Canvas, and procedures were associated with events that are invoked by our game components, such as the start button, stop button, levels dropdown list, and the othello board squares, which are represented as a grid of buttons that can change images according to gameplay.

5.2 Board Representation

The board is represented in the UI by a grid, and each square of the grid is assigned a button, so we represent the board by an array of buttons where each button invokes the procedure othelloButton() with a parameter corresponding to its index in the grid when clicked. This procedure is responsible for controlling the corresponding actions.

Furthermore, the board values are represented in an array of integers. In order to know the state of a square, the entry value in this array is -1 if the square is empty, 1 if it contains a black disc, and 2 if it contains a white disc. Note that initially the four middle squares in the board are filled with two diagonal white squares, and two diagonal black squares.

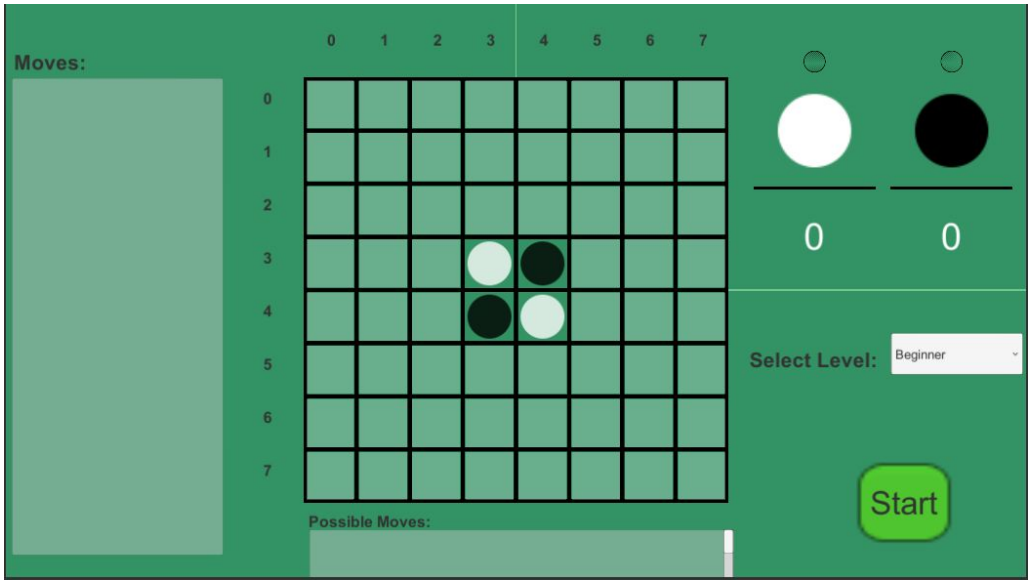

Figure 1. Main UI

5.3 Utilized Gameplay Procedures

All procedures controlling our gameplay are inside a GameController C# script. Note that two variables are initially assigned which are important to the gameflow, these variables are ‘whoTurn’ which indicates who is the currently playing player, and ‘depth’ which determines

the depth of the game tree generated in the minimax algorithm and which is defined by the chosen game level.

The utilized gameplay procedures and functions are:

- ➔ **proc Start()**: called before the first frame update, it only activates the start button and deactivate the stop button.
- ➔ **proc StartGame()**: called when the start button is clicked. It applies the needed changes in the UI, for example, it deactivates the start button. It calls **proc ResetGame()**.
- ➔ **proc StopGame():** called when the stop button is clicked. It applies the needed changes in the UI, for example, it deactivates the stop button. It calls **proc ResetGame()** and **clearPossMove()**.
- ➔ **proc HandleInputData()**: called when game level is selected from the dropdown list. For intermediate level, depth is assigned a value of 4, and for expert level, it is assigned a value of 10. Beginner level AI randomly selects a move so depth value isn't important in this case.

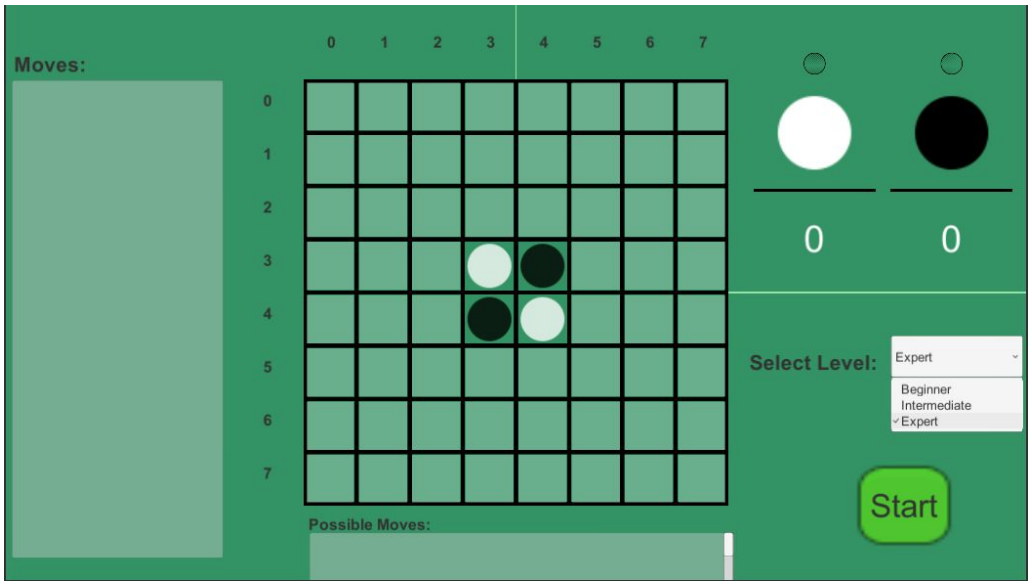

Figure 2. Choosing Level

- ➔ **proc ResetGame()**: sets the initial state of the game. It sets the 'whoTurn' variable to 0 since black always plays first. It makes sure the grid buttons other than the four middle squares are interactable and don't contain a disc image, and assigns the corresponding images to the four middle squares. It also sets the board array values to -1 for empty squares and to 1 and 2 accordingly. It resets the score in the UI, clears the possible moves from the board and shows them after calculating them again with the corresponding procedures.

- ➔ **proc printPossibleMoves()**: takes as a parameter a list of possible moves indices and prints them in the UI.
- ➔ **func checkBlackScore()**: takes as a parameter the array of board values and returns the count of black discs.
- ➔ **func checkWhiteScore()**: takes as a parameter the array of board values and returns the count of white discs.
- ➔ **func checkPossibleMove()**: takes as parameters the array of board values and the player's turn and uses some helper functions to return a list of possible moves indices corresponding to the current board state and turn.
- ➔ **proc showPossibleMove()**: takes the list of possible moves indices as parameter and shows the possible moves on the board UI.
- ➔ **proc flipPiece()**: takes as parameter the clicked square index and uses some helper functions to flip pieces in all correctly possible directions.
- ➔ **proc othelloButton()**: called when a board square button is clicked. It is also invoked by the AI coroutine. Certain steps are executed:
    - ◆ the square's button image is set to the corresponding player's disc image and the button is deactivated.
    - ◆ the corresponding entry value is set in the board values array.
    - ◆ the move is printed in the move list UI and the turn UI and turn are toggled.
    - ◆ flip pieces and regenerate possible moves for the next turn using the corresponding procedures.
    - ◆ If possible moves exist for the next player, and the new turn is the white player's turn, the AI coroutine is called.
    - ◆ if there are no possible moves for the next player, it passes and the corresponding actions are taken again for the next player to play instead. If also the other player has no possible moves, the game ends. If the black player passes and there exists possible moves for the white player, the AI coroutine explained later on has to be called.

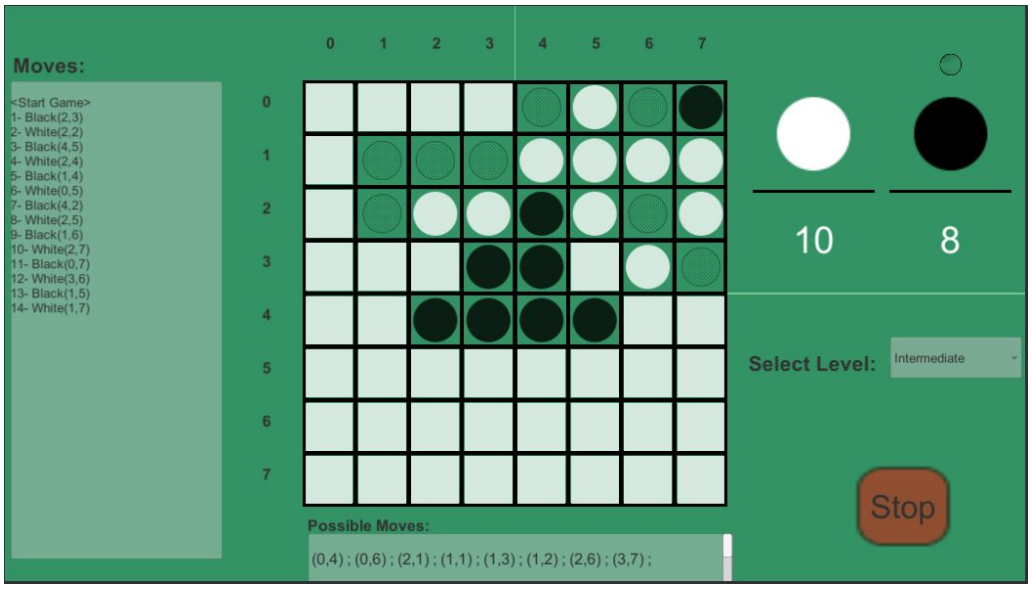

Figure 3. Before Making Move Black(3, 7)

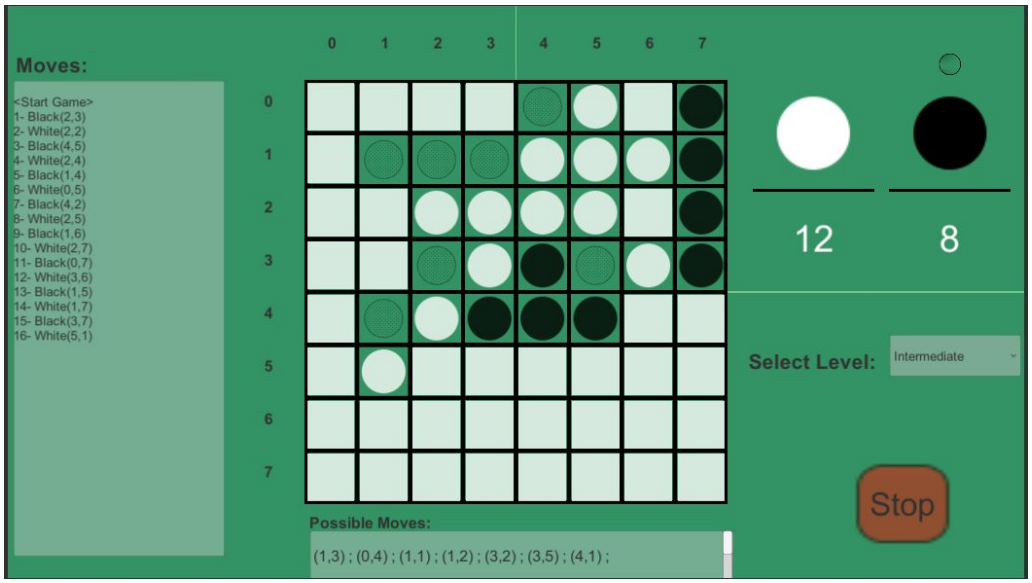

Figure 4. Coins Flipped After Making Move Black(3, 7)

5.4 Utilized AI Procedures and Evaluation Function Criterion

5.4.1 AIPlay Coroutine

This procedure invokes the on click method of the board square button corresponding to the index generated by the function **bestMove()**, which determines which is the best move that the AI can make according to the algorithm we used for deciding. So it

acts as if the player clicked on the square and does the same actions but from the AI's point of view.

5.4.2 BestMove Function

This function's goal is to return the index of the possible moves array corresponding to the best move the AI can make according to our implementation of the minimax algorithm and evaluation function.

It takes as parameters the array of board values and the depth value determined by the game difficulty level.

- ➔ If the difficulty level is beginner, a random index is returned.
- ➔ If there is only one possible move, it is directly returned.
- ➔ if there are many possible moves, the minimax approach is used where the current player which is white is minimizing. The minimax score is calculated for each possible move and the index corresponding to the minimum score is returned. However, if the tested index corresponds to a corner, it is directly returned since occupying corners is favorable in othello.

5.4.3 Minimax Function

This function is implemented in a way similar to the known minimax algorithm with alpha-beta pruning implementation [See section 4.3]. However, a detail specific to our game is added. In Othello, if a player has no possible moves in a certain board state, he passes his turn to the next player, and if both have no possible moves the game ends. This has to be represented in the minimax game tree. So, if there are no possible moves for the current player, and the other player has possible moves, minimax is called with the same parameters except the parameter indicating if the player is maximizing which is toggled. If both players have no possible moves, the evaluation value of the current board state is returned.

5.4.4 Othello Evaluation Function

For Othello, factors such as mobility, stability, corners and coin parity determine how favorable a particular position is for a player. We calculated the heuristic value by creating a linear combination of the quantitative representation of the various important factors with different weights for each factor corresponding to its importance in determining the state of the board.

The following factors were utilized in the evaluation function:

A. Coin Parity

This component of the utility function captures the difference in coins between the max player and the min player. The return value is determined as follows :

```
if(maxPlayerCoins > minPlayerCoins)
    p = (100.0 * maxPlayerCoins)/(maxPlayerCoins +
    minPlayerCoins);
else if(maxPlayerCoins < minPlayerCoins)
    p = -(100.0 * minPlayerCoins)/(maxPlayerCoins +
    minPlayerCoins);
else p = 0;
```

B. Mobility

It attempts to capture the relative difference between the number of possible moves for the max and the min players, with the intent of restricting the opponent's mobility and increasing one's own mobility. This value is calculated as follows :

```
if(maxPlayerMoves > minPlayerMoves)
    m = (100.0 * maxPlayerMoves)/(maxPlayerMoves +
    minPlayerMoves);
else if(maxPlayerMoves < minPlayerMoves)
    m = -(100.0 * minPlayerMoves)/(maxPlayerMoves +
    minPlayerMoves);
else p = 0;
```

C. Corners Captured

Corners hold special importance because once captured, they cannot be flanked by the opponent. They also allow a player to build coins around them and provide stability to the player's coins. This value is captured as follows :

```
int[] cornerIndices = {0, 7, 56, 63};
for (int i = 0; i < 4; i++){
     if (board[cornerIndices[i]] == 1)
          maxPlayerCorners++;
     else if (board[cornerIndices[i]] == 2)
          minPlayerCorners++;
}
c = 25 * (maxPlayerCorners - minPlayerCorners);
```

We also added another value which the maximizing player wants to minimize, and that is ‘corner closeness’. The maximizing player wants to have less coins neighboring the corners, because having coins near the corners would limit its chance of occupying them. This value is calculated as follows:

```
l = -12.5 * (maxPlayerCoins - minPlayerCoins);
```

where maxPlayerCoins and minPlayerCoins are calculated by testing the neighboring coins of each corner to which color belong and counting them.

D. Stability

The stability measure of a coin is a quantitative representation of how vulnerable it is to being flanked. Coins can be classified as belonging to one of three categories: (i) stable, (ii) semi-stable and (iii) unstable.

Stable coins are coins which cannot be flanked at any point of time in the game from the given state. Unstable coins are those that could be flanked in the very next move. Semi-stable coins are those that could potentially be flanked at some point in the future, but they do not face the danger of being flanked immediately in the next move. Corners are always stable in nature, and by building upon corners, more coins become stable in the region.

Calculating the stability factor with accuracy might be computationally expensive, so we made a tradeoff between accuracy and computation time by using the difference in frontier coins. A coin is a front coin if it is adjacent to an empty square and so can potentially be flipped. The maximizing player wants to minimize the number of its frontier coins. This value is calculated as follows:

```
if(maxPlayerFrontCoins > minPlayerFrontCoins)
    f = -(100.0 * maxPlayerFrontCoins)/(maxPlayerFrontCoins +
    minPlayerFrontCoins);
else if(maxPlayerFrontCoins < minPlayerFrontCoins)
    f = (100.0 * minPlayerFrontCoins)/(maxPlayerFrontCoins +
    minPlayerFrontCoins);
else f = 0;
```

In addition to that, a static factor (s) was calculated from predetermined weights assigned to each square. If the coin is black, we add the weight to s, and if it’s white, we subtract it.

The **Final Evaluation** is calculated depending on the importance of each factor as follows:

```
finalEval = (10 * p) + (800 * c) + (400 * l) + (80 * m) + (70 *
f) + (10 * s);
```

6 Conclusion and Future Work

In this project, we developed using Unity game engine an othello game with an AI opponent and with three different levels. The implemented AI used minimax algorithm for deciding on the move to make, and its efficiency was improved with introducing alpha-beta pruning. Also the specific evaluation function based on several strategies related to the Othello game helped improve the AI to win more often.

Future work would include the incorporation of learning strategies into the system. Such strategies tend to be very powerful, since they could make use of vast amounts of already existing data and can avoid pitfalls that deterministic algorithms can suffer from. We can also build more upon our framework to give rise to an extremely good Othello player. That would require careful weight modification, optimized lookup tables, and powerful learning strategies.