# COMP 1130 Othello AI Technical Report

Name: Haoting Chen

UID: u7227871

Lab Time: Thursday 8am

Tutor: Rachel Schroder, Eric Pan

# Introduction

The Haskell program “AI.hs” contains several AIs which can decide where to place their pieces in a Othello’s game state. There are two main AIs, “greedy” and “default”. The AI “greedy” contains the function "easyMode", which takes a game state and returns a move that can lead to a good next game state. The AI “default” contains the function “hardMode”, which runs differently depending on the second argument. If a 1 is input, the function will call the function “easyMode” and do the same thing as “greedy”. If any other integer is input, the function will look down to the game tree and returns a move which may lead to a good future game state.

# Assumptions

Both AIs work based on the following assumptions,

- The AI is called only if it has any legal move.
- The AI is called only if the game is not over.
- The game board is 8 x 8.
- The legal moves given by the function “legalMoves” are indeed legal.

# Documentation

### “easyMode” and “hardMode”

The function “easyMode” is used to make a quick and greedy decision for a move. When a game state is input, it will be processed through its helper function “legalMove” "filterMoves", and "bestMove" successively. The “legalMove” takes a game state and returns a list of legal moves. The “filterMoves” accepts the result of “legalMove” and filters any obviously bad moves if not all the moves are bad. It can also keep only the obviously good moves, if any. In this program, the good moves are defined as corners, and bad moves are defined as X-Squares. The refined list of moves, the current game state, and a False, is input into “bestMove” for further processing. The function "hardMode" like "easyMode". The only difference is that if its second argument is not 1, when substituting the refined list into "bestMove", a True is input instead.

### “implementMove”, “getPlayer”, and “switchPlayer”

The three functions above are the useful helper functions that are used by many other functions in the program. "implementMove” accepts a state and a move. If the move is legal, it will call the function “applyMove” to

update the game state and remove the Maybe type. If the move is illegal, it will output the same state. The function “getPlayer” takes an unfinished game state and returns the current player’s name. If a finished game state is unexpectedly input, it will return Player1 by default. The function "switchPlayer" can switch the player in an unfinished state. If a finished state is unexpectedly input, then it will output the same thing.

**“bestMove”, “scoreMove”, and “maxScoreMove”**

The job of the “bestMove” is to select the best move from a refined list of moves. If the input list is a singleton, it will only move without any analysis. If not, it will map its helper function “scoreMove” onto the moves in the list so that they are all scored. Then, it calls the other helper function “maxScoreMove” to select the move with the highest score from the list to send back to the AIs.

The function “scoreMove” takes a move and return a pair of move and its score. It brings the move and current game state into "implementMove" to get the next game state caused by this move. If there is a False input into “bestMove”, “scoreMove” will use the function “stateEvaluatePlus” to evaluate the state. Therefore, the score of the move is only based on the score of the next state. If there is a True input into “bestMove”, “scoreMove” will call the function “othelloTree” to generate a depth 6 game tree with the state as the root. Then, "miniMaxAB" accepts the tree and use it to calculate the score of the best game state that the AI may reach after the 6 turns, and then use it as the score of the move.

**“MinimaxAB”, “othelloTree”**

For the AI "defalut", the best move is selected according to the score of the state tree caused by the move. The job of “othelloTree” is to generate such a game from a state. The arguments of “othelloTree” are the depth of the tree, the player of the state of the root, and the state. The depth number is used as a counter to stop the function if the tree is deep enough. The state is used to be processed through “implementMove” so that new children states can be generated. The player's name of the root is used to check if there is a skip turn case. The function keeps swap the player's name after every loop. If the player of the root of a subtree does not match the expected player, it means the last player's turn has been skipped. Hence, “othelloTree” adds one more node to the subtree before expending the subtree. Therefore, every odd depth node is one player’s turn, and every even depth node is the other player turn. This makes sure the Minimax algorithm work effectively.

The generated tree is then accepted by the function "MinimaxAB" written based on the Minimax algorithm with alpha-beta pruning. The result of “MinimaxAB” score of the best game state that the AI may reach after the six turns. The function “MinimaxAB” has five arguments, the AI's player name, Boolean that determine whether the root of the game tree is (AI) Max's turn, game tree, initial alpha value (-9999) and initial beta value (9999). In this case, a False is input since the next state of the move is the opponent's state. Therefore,

every odd level of the tree is Min's turn, and every even level is Min's turn. The "MinimaxAB" explores the score of the leaves (states after five turn) in order. When it gets to a leaf, it brings to the state of the leaves and the AI's player the function "stateEvaluate" to get the score of the state. It then uses the score to update the node and the uppers nodes. At Max's node, the score comes from the maximum score of its children's nodes. At Min's node, the score comes from the minimum score of its children's scores. When it goes up and explores other leaves, it also updates the alpha value at even depth and beta value at odd depth. Alpha is the smallest possible score that Max will choose, and Beta is the largest possible score that Min will choose. If alpha is updated to be greater than Beta at a Max's node or Beta is updated to be less than Alpha at Min's node, the function will no longer explore the leaves of such a node since it is impossible for Max and Min to get to these states.

**"stateEvaluatePlus" "stateEvaluate" "totalPieces"**

Whether the AI is "greedy" or "default", it will always get to a stage where it needs to evaluate a specific game state. The evaluation of a game state is undertaken by the function "stateEvaluatePlus" or "stateEvaluate". Specifically, the former is designed for "greedy", and the latter is designed for "default". Both functions evaluate a game state of a player based on the number, location, and stability of the players' pieces. "stateEvaluatePlus" also takes the mobility of a player into account. "totalPieces" is the helper function for both evaluation function. It takes a game state and output the total number of pieces on the board. Therefore, the evaluation function can know how long the game lasted to adopt different evaluation methods. For example, at the start of the game, capture some key position is essential. Therefore, the function will focus on the positional score. The idea of evaluating a game state using these three components comes from the report: ""IAgo Vs Othello": An artificial intelligence agent playing Reversi" (Festa & Davino 2017) and "A New Experience: O-Thell-Us –An AI Project" (Ganter & Klink 2004), and

**"positionScore" "mobilityScore" "stabilityScore"**

The function "positionScore" is used to measure how many pieces of a player on a board and how good the pieces' locations are relative to its opponent. For the AI, every location on the board has a specific weight value. These weights are recorded in a weights matrix, such as "scoreMatrix1". The job of "positionScore" is to match every location on the board with the weights on a weights matrix. If a square has a player's piece on it, then "positionScore" will add the weight of the square. If an opponent's piece is on it, "positionScore" will subtract the weight. Therefore, a positional score of a player relative to its opponent is calculated.

The function "mobilityScore" measure the mobility of a player, that is, how many different locations that a player can choose to place their piece in their turn. The measurement is done by calculating the length of the

set of legal moves in a state. Since the next state caused by the AI's move is a opponent's state, the mobility score is always negative, which means the higher the mobility score, the fewer legal move the opponent has.

The function “stabilityScore” measure the number of stable some stable piece of a player in a state. The stable pieces are those which cannot be flipped by the opponent once. If a player captures a corner can make their pieces form a line, all the pieces on the line are stable, as shown in Figure 1. If there is a player’s piece on (0,0), the function will give the state some points. Then it continues to check if there is a player's piece at (1,0), (2,0), and so on. If so, it will give the state extra points. If not, the function will stop the loop and return the stability score of a corner. The stability score of every corner is counted separately and summed together as the output of “stabilityScore”. In this way, the AI will tend to form its pieces into a line on an edge after occupying the corresponding corner.

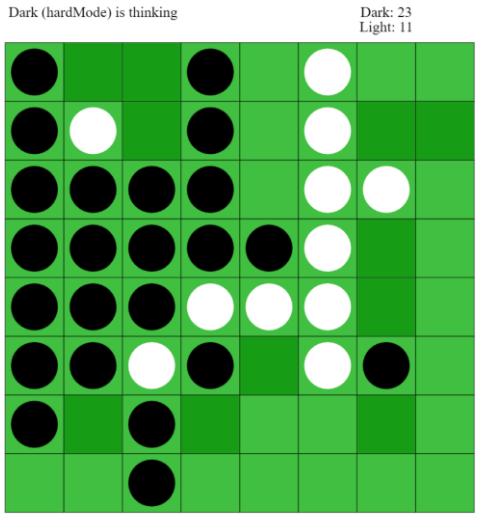


*Figure 1: pieces from (0,0) to (0,6) are all stable.*

The positional, mobility, stability score is sent back to “stateEvaluate”, “stateEvaluatePlus” For “stateEvaluatePlus”, the score of a state will be sent directly back to “bestMove” as a score of a move. For “stateEvaluate”, the score of a state will be sent to “MinimaxAB”. A score of a game subtree output from “MinimaxAB” will be used as a score of moves. Finally, “bestMove” can compare all the scored move and return the AI “greedy” or AI “default” the best one.

## Testing

To test whether function "easyMode" and "hardMode" works well, I wrote some unit tests for them. I chose some obviously good move (corner) as input and saw if the function could correctly select them. To deeply test the performance of both functions, I also used the AIs to compete with other the AIs in the phone games of Othello. When my AIs did not work as expected, I adjusted some parameter of my evaluation functions or

added more evaluation function to make them smarter. The test of “bestMove”, “scoreMove”, and “maxScoreMove” are all conducted by matching the function’s outputs with my expected outputs in unit tests.

To test if the tree generated by “othelloTree" is correct, I classify the tree type into "show" using a function that can visualise a game tree. The function is written by Antony Hosking imported from "Nim.hs" (Hosking 2016). Next, I used the function “roseMap” “getTurn” to convert every node of the tree from type “GameState” into the type “Turn” since the type “GameState” is long. In this way, I was able to check if the tree swaps the player at each level. I also set the second argument as a player different from the player of the state of the root to simulate a skip turn case. If the function is correct, it will add an extra node before generating the children game states. I also used the function “roseDepth” from Lab 9 to see if “othelloTree” to see if the function can create a rose tree to a correct depth. In order to test my “minimaxAB”, I wrote another function “miniMax” using the normal Minimax algorithm. If “minimaxAB” is correct, it should output the same result as “miniMax”.

The tests of “stateEvaluatePlus” “stateEvaluate” “totalPieces” “positionScore” “mobilityScore” “stabilityScore” are conducted in the similar ways. I created a test state and calculated each score by hand. Then I use it to compare with the actual output score of the functions through unit tests.

# Reflection

“greedy” is the first AI I made in this program. I chose not to develop “default” until my “greedy” can easily beat “firstLegalMove”. This is because the perform of “greedy” and directly reflect on the quality of my evaluation functions. And the effectiveness of “Minimax” used by “default” heavily depends on the evaluation functions. The function “filterMoves” is added after conducting several tests for the AIs. This function prevents the AI from making some silly mistake or miss a great chance, such as corner capturing. Even though the X-Squares have the smallest weight, and the corners have the largest weight on the board, there is still a chance for the Minimax algorithm to give the wrong direction because of some unexpected high results ahead. However, based on some common strategies, the advantages of placing in X-square outweigh the disadvantages. Although the decision may make the AI miss a big opportunity to achieve high a score, it also lowers the risk of loss. Another reason for making the decision is to save the AI's time to focus on evaluating other more evaluable moves.

I have decided not to include mobility score into the AI "default" because the mobility of a player in a specific stage after six turns is less significant compared to the other two measurements. Instead, it may negatively

impact the effectiveness of the total score of a game state if it has not been scaled to an appropriate factor. In addition, the simplification of the evaluation in a state decreases the time of computation. Therefore, the Minimax algorithm can explore the tree further to give a more reliable result.

Due to the limited experiments, the evaluation function is not great. The AI does not have the ability to avoid putting its piece on C-squares if there are any opponent's pieces nearby. Placing pieces on C-squares without careful analysis can make the corner be easily captured. These are the areas that can be improved in the future.

## Reference


Festa, J, Davio, Stanislao 2017, *"IAgo Vs Othello": An artificial intelligence agent playing Reversi*, viewed 1 June 2021, <https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/O-Thell-Us/Othellus.pdf>

Ganter, M, Klink, J 2004, *A New Experience: O-Thell-Us –An AI Project*, viewed 1 June 2021, <https://courses.cs.washington.edu/courses/cse573/04au/Project/mini1/O-Thell-Us/Othellus.pdf>

Hosking, A 2016, "Nim.hs", viewed 1 June 2021, <https://cs.anu.edu.au/courses/comp1100/lectures/10/>