CSC111 Project Report: AI Player for Reversi

Huiru Tan, Yupeng Chang, Xi Chen

Friday, April 16, 2021

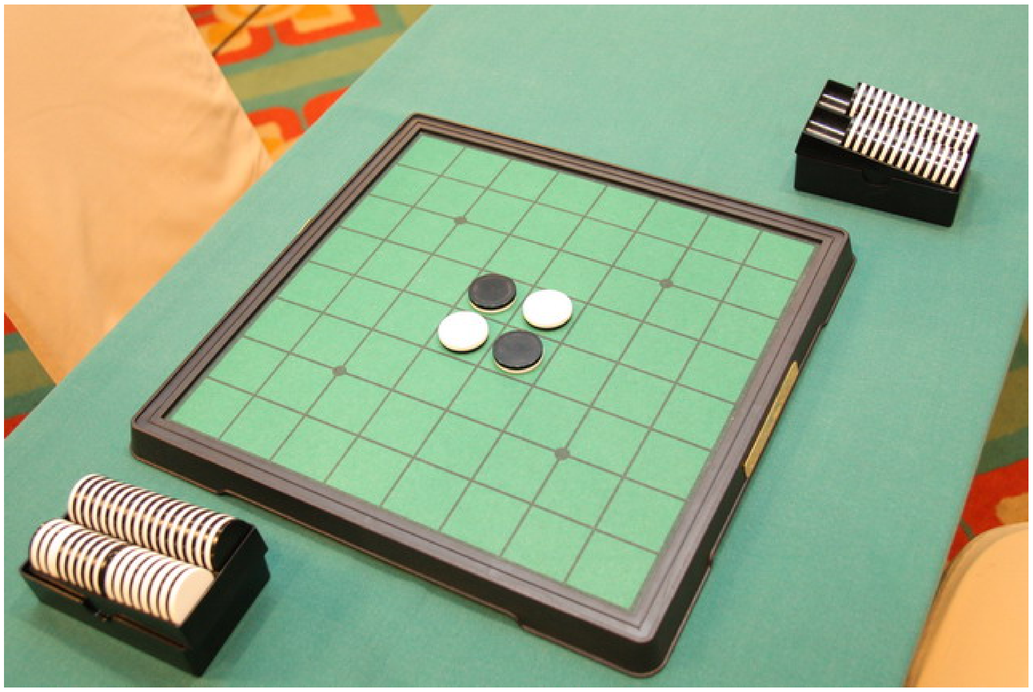
Problem Description and Research Question

Reversi is a game for two players on 8*8 board (Reversi, 2021). There are four initial pieces on the center of the grid as shown below in Figure 1. When a new black piece is placed on the board, all contiguous white pieces on the same horizontal, vertical, or diagonal line between the previous black piece and the new one will be reversed to black pieces. Similarly, white pieces follow the same rule. The player with a black piece makes the first valid move where a valid move is a move that causes at least one piece on the board reversed. Two players take alternative turns until there is no valid move for both players, then the player with more pieces in his color on the board wins.

According to the game board size and the rule of Reversi, there are numerous valid steps for each player and each step may cause a completely different situation on the board. For example, if a black player has one piece at three corners of the grid, he will have eighteen white pieces reversed to black if he put one piece at the last corner, which is a huge number for the game. Moreover, the value of each position of the grid is different. Technically, the piece at the corner has more strategic significance in most cases because they can never be reversed but they can still attack other pieces.

Therefore when Reversi AI makes one move, it needs to consider both short-term benefit (temporarily reversed pieces) and long-term benefit (the significance of the positions of each piece) by the game tree. Since the computation resources are limited, Reversi AI also needs to balance the width of the game tree and the depth of the game tree to make the optimal choice between time and space complexity. Our goal is **creating a Reversi application and an AI player which allow human to play against, then using the visual method to show how AI 'thinks' and how well it plays**.

Figure 1: The standard initial board of Reversi (Reversi, 2021)



Computational Overview

For the game board of Reversi, we use nest list to achieve. In `reversi_board.py`, we create class `Reversi` which stores the nest list which represent the board. For example, `Reversi().board[2][5]` represents the third row and sixth column of the board. The main function for the reversi game is `update_board` which has eight direction vectors. According to the rule of reversi, when a new piece is placed, it may reverse the piece in these eight directions. The function will return the total number of pieces reversed. If the number is not zero, the game board will be updated just like the real board game, otherwise, the move is not valid.

Reversi AI gets the complete game tree then uses Minimax Algorithm to find the most valuable move. First, the Minimax player call function `generate_game_tree` in reversi_player.py to generate all possible moves with a certain depth. Second, for the leafs of the game tree, it calculates the benefit score according to the evaluation function `evaluate_score_by_piece` or `evaluate_score_by_position`. The former function considers each piece on the board the same value, while the latter function each piece with different weights (constant `WEIGHT` in reversi_player.py). Third, the Minimax Player applies Minimax Algorithm to the game tree. For example, if the current player is a black player, then for the game tree of black turn, its score is the maximum of its subtrees, otherwise, the score is the minimum of its subtrees. Last, it chooses the possible move with its highest score. The game tree will be recalculated to reach the set depth after one move is made, which means the minimax player is not influenced by the previous moves since it will recalculate the game tree for the current game board.

To avoid the run time of `generate_game_tree` being too long, we add parameter `start_time` and `think_time` to the function. During the recursion, the function will check the difference between current time and start time . If the difference is larger than `think_time`, it will stop calling further recursions.

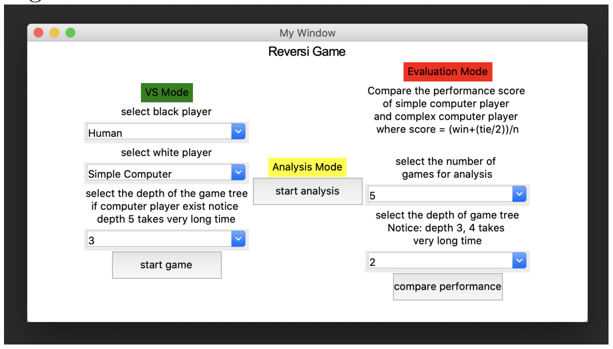
In `reversi_pygame.py`, we use **pygame** to make a Reversi application. There are two modes of application. The first mode is game mode by function `run_game(white_player, black_player, ...)`. In this mode, the players could be a human, random player, or minimax player. Therefore humans can play against computers or see how computers play against each other. The second mode is analysis mode by function `analysis_game()`. In this mode, humans can make moves on the board. At the same time, the score of each potential move calculated by the minimax algorithm will be live displayed on the board.

To evaluate how well a player plays, we simulates n games between the test player and random player. The performance score is calculated by the formula score = (win + tie / 2) / n. This is done by function `get_performance(player)` in `analysis.py`. The function `plot_data(depth, n)` use package **plotly** to plot a line graph of the performance score of random player, simple minimax player(consider piece equal weight), and complex minimax player(consider piece different weight) when the depth of game tree is range(1, depth + 1).

Finally, we use package **tkinter** in main.py to make the user interface. In this part, we use `ttk.Combobox` and `tkinter.button` to show our three main modes.

Instructions for running the program

After running main.py, you are expected to see the user interface shown below.
Figure 2: the user interface

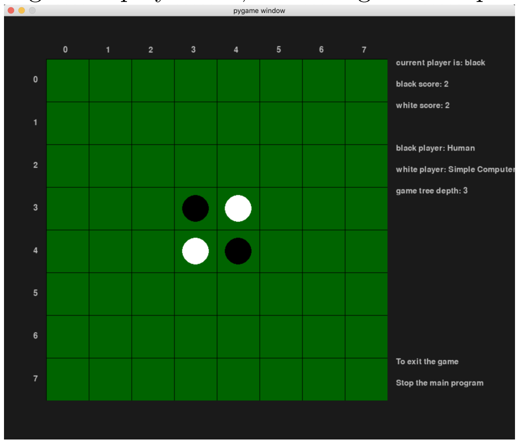


select black player: Human
select white player: Simple Computer
select the depth of the game tree if computer player exist notice depth 5 takes very long time: 3

Compare the performance score of simple computer player and complex computer player where score = (win+(tie/2))/n
select the number of games for analysis: 5
select the depth of game tree Notice: depth 3, 4 takes very long time: 2

VS MODE: You could select the black player, white player, and the game tree depth, then click the button to start the game. Notice depth 5 takes a very long time.

Figure 3: play mode, human against simple computer



current player is: black

black score: 2

white score: 2

black player: Human

white player: Simple Computer

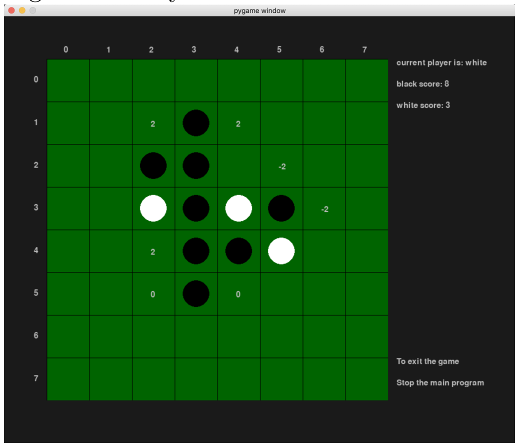
game tree depth: 3

To exit the game

Stop the main program

Analysis MODE: Click the button to start analysis, you are expected to make valid moves and see the live displayed scores of each step on the board. Notice: to exit pygame in play mode or analysis mode, please stop running main.py completely and then rerun it.

Figure 4: analysis mode



current player is: white

black score: 8

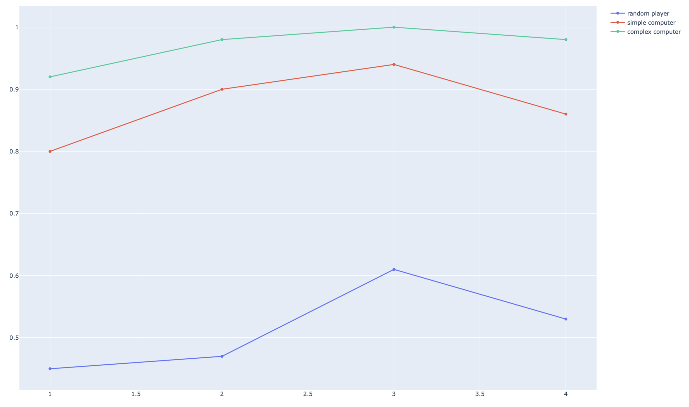
white score: 3

To exit the game

Stop the main program

Evaluate MODE: You could select the number of simulation games and the depth of the game tree to compare the performance of three different computer players. You are expected to see the resulting graph pops up in the browser after a few seconds or minutes. Otherwise, you can check the folder of the project to see the generated html result page. Notice when depth is larger than 2, each game simulation takes a much longer time. For example, evaluating 50 games with depth 2 take about 2 minutes. Evaluation of 20 games with depth 3 takes about 10 minutes.

Figure 5: The result of n=50 depth=4, time used: 4 hours



Difference between Proposal and Submission

We decided to use Minimax Algorithm based on TA feedback. We also simplified the condition for the game to end. The previous condition is that there is no valid move on the current board for both black players and white players. This means one player may make two continuous moves if there is no valid move for another player. To make the game tree more regular, we changed the condition to no valid move for the current player.

Discussion

Figure 5 illustrate a counter-intuitive fact that the win rate of minimax player slightly drops when the game tree depth increased from 3 to 4. One hypothesis is that the depth 4 game tree does not fit the Minimax Algorithm since too deep a game tree may form overfit model which causes bias. However, we were not able to get the data for depth larger than 5 limited by the computation resources.

In spite of the result of depth 4, we could conclude that the Minimax Algorithm works since the performance score of both minimax players is much higher than the random player which performance score is around 0.5. Also, as we can see from the line graph, the performance of both simple minimax player and complex minimax player increased when the depth increased(1-3).

Another finding is that the performance of a Complex minimax player(consider pieces different weight) is always higher than the simple minimax player(consider pieces same weight). This means the weight of each piece on the board is different, even though each piece counts 1 score when the game ends. Therefore now the question is how we can find a function to get the exact weight of each piece on the board. We also need to consider another fact that the weight of each piece may vary when the board changes. For example, normally a piece at the corner may have a higher potential value as we explained before. However, if the piece at the corner is surrounded by the pieces with the same color, its potential would be lower because it could not influence other pieces on the board anymore. The preliminary plan to find the objective function for the current board score is making our reversi AI learn the previous game. One approach is that we could use the matrix to get the abstract expression of the current board situation. Then we let AI guess the value of the piece in a certain situation. If the value leads to the win, AI will make the connection between the situation and the value. Otherwise, AI will guess another value and repeat the similar process.

We also wonder if there is a better algorithm to find the game tree. In our game tree generate function, we generate one complete series of moves of set depth-first then another. This means if there are four possible moves and the set depth is 5 when the think time for the function is up, we will get a tree with complete depth 5 move sequences of move1 and move2. However, there are no trees for move3 and move4. Actually, the Monte Carlo method would be more suitable here since a depth 4 game tree of all moves would be better than a depth 5 game tree of incomplete moves. Therefore the better algorithm to find the game tree would get all moves at the same depth, then it goes down to get further moves. We might need to have a different game tree structure or use a queue for this approach.
To sum up, the Minimax Algorithm does work in the Reversi game. However, there are still lots of algorithms used in the program that could be optimized to get better time and space complexity.

References

Reversi. (2021, March 9). In Wikipedia. Retrieved from https://en.wikipedia.org/wiki/Reversi

Minimax Algorithm in Game Theory — Set 1 (Introduction). (2021, March 31). In GeeksForGeeks. Retrieved from https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-1-introduction/

tkinter — Python interface to Tcl/Tk. (2021, April). In Python. Retrieved from https://docs.python.org/3/library/tkinter.html