CSC111 Project Report: Reversi AI

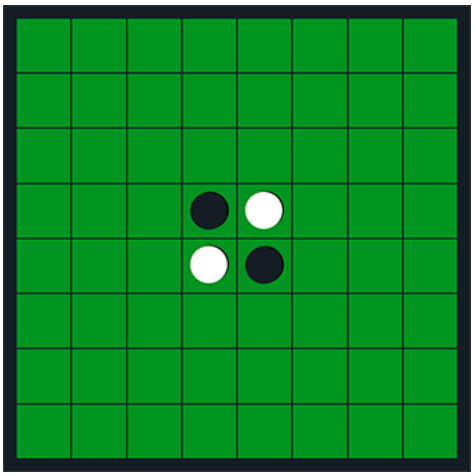
Anatoly Zavyalov, Baker Jackson, Elliot Schrider, Rachel Kim

Monday, April 19, 2021

Introduction

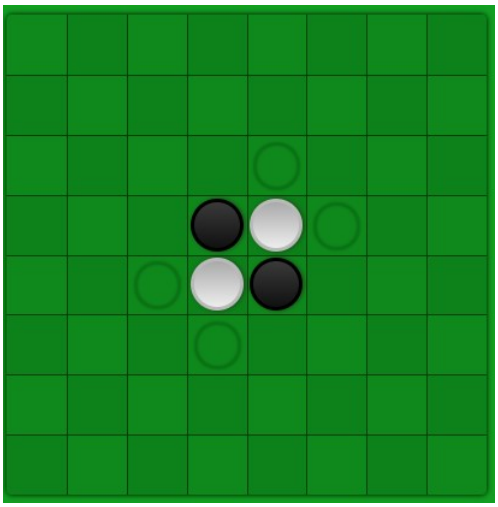
Reversi is a two-player board game. One player has white pieces, while the other has black, and the game is typically played on an 8-by-8 gridded square board. The goal of the game is to have more pieces of the player's colour than the opponent's colour by the end of the game.

The game starts with two white pieces and two black pieces on the center of the board, arranged in the following configuration:



(Gambling Sites, 2020)

The black player plays first. A move made by either player must "capture" at least one of the opponent's pieces. A player can "capture" an opponent's pieces when the player's pieces surround an opponent's pieces in a horizontal, vertical, or diagonal line. An example is below:

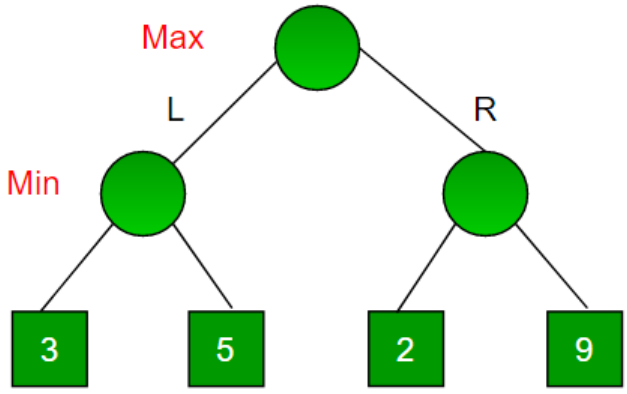


Note: The outlined circles indicate possible moves for the black player. In this case, each of the four possible moves captures exactly one white tile (CardGames.io, n.d.).

The game ends when the current player cannot make a move or when the board is full. At this point, the winner is the player with more pieces of their colour on the board.

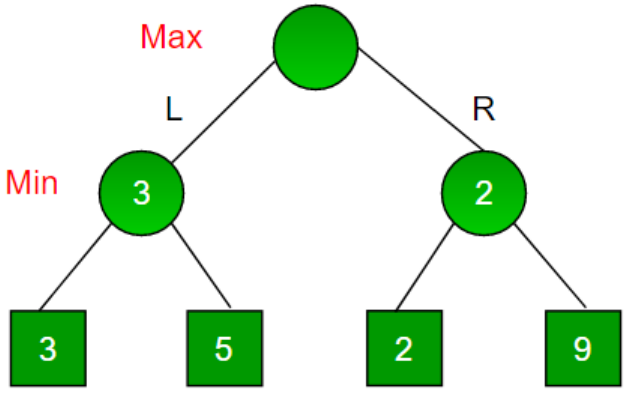
Since the 1980s, there have been Reversi programs that play at the level of human Reversi champions (Allis, 1994). However, like Assignment 2, we still believe that it is interesting to see the extent to which we, as first-year students, can create successful types of Reversi AI. CSC111 Assignment 2 serves as the main inspiration for this project. Similar to Minichess, Reversi is also a two-player, perfect information game, meaning that we can apply the notion of game trees. Furthermore, although Reversi is solved for 4-by-4 and 6-by-6 boards, it remains unsolved on the standard, 8-by-8 board (Wikimedia Foundation, 2021). We are interested in seeing how our Reversi AI performs on different board sizes and emulate different difficulty levels (beginner to advanced). Using the above as motivation, broadly, we explore and implement the following types of Reversi AI:

- *Random*: The Reversi AI chooses a random move to make out of all valid moves every turn.
- *Minimax*: The Reversi AI uses the Minimax algorithm to choose its next move. To give a broad overview of this algorithm, there are two players: the maximizer and the minimizer. These two players have diametrically opposed objectives: one tries to maximize their score (in the case of Reversi, the chance that the maximizer wins) while the other player tries to minimize their score (in the case of Reversi, the chance that the minimizer wins) (GeeksforGeeks, 2021). This score is determined by an evaluation function. Both players assume that the other plays optimally; that is, the maximizer assumes that the minimizer always plays to minimize their score, and vice versa. To illustrate using a simple example, suppose we start with the following final states, with the values in the squares determined by some evaluation function:



(GeeksforGeeks, 2021)

We work backwards from these given final states. In this example, we determine the choice that the minimizer would make; that is, the choice that minimizes the final result. Filling in these choices, we get:

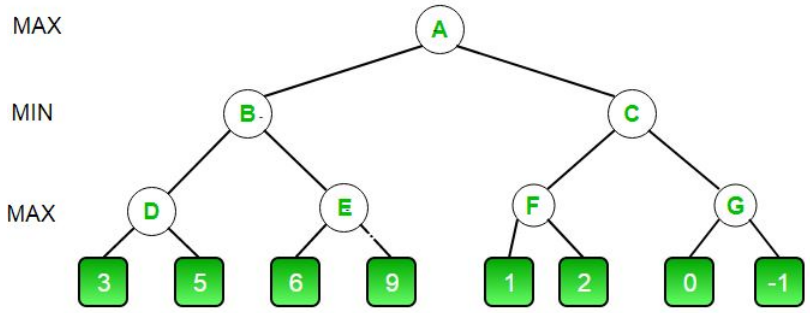


(GeeksforGeeks, 2021)

From here, the maximizer would choose the left subtree. Note that in the right subtree, there is a final value of 9, which is greater than all the final values in the left subtree; however, since the maximizer assumes that the minimizer plays optimally and chooses the left subtree instead.

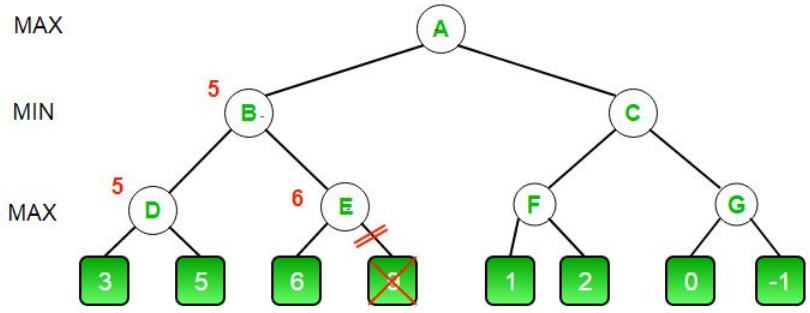
Since it is computationally infeasible to generate a full game tree for Reversi using the Minimax algorithm, we explore a variety of computationally feasible alternatives. These include restricting the depth of the game tree the Minimax Player considers when deciding the next move to make and using alpha-beta pruning.

Alpha-beta pruning is a way of reducing the number of searches into the game tree. "Alpha" represents the optimal value that the maximizer can guarantee at a certain level, and "beta" represents the optimal value that the minimizer can guarantee (GeeksforGeeks, 2019). To illustrate using a simple example, suppose we have the following game tree and we are searching left-to-right through the tree:



(GeeksforGeeks, 2019)

We start with the initial values of $-\infty$ and ∞ for alpha and beta respectively. At `D`, the left child is first considered. The value of `alpha` at `D` then becomes `max(-INF, 3) = 3`. To evaluate whether the right child should be considered, we check the condition `beta <= alpha`. This is False, since `INF > 3`, which means that the right child is indeed considered. The value of `alpha` at `D` now becomes `max(3, 5) = 3`. Then, the value of `alpha` at `D` is passed up to `B`, and the value of `beta` at `B` becomes `min(INF, 5) = 5`. Now, the value of `beta` at `B` is passed down to `E`, so the value of `alpha` at `E` is `-INF` and the value of `beta` at `E` is `5`. At `E`, the left child is first considered. The value of `alpha` at `D` then becomes `max(-INF, 6) = 6`. To evaluate whether the right child should be considered, we check the condition `beta <= alpha`. This is True, since `5 <= 6`. Thus, the right child of `E` does not have to be considered, since at `B`, the minimizer is already guaranteed a value of `5` or less through `D`, whereas through `E`, the maximizer could choose a value of `6` which would not be optimal for the minimizer. The following illustrates the "pruning" that happens through the above process:



(GeeksforGeeks, 2019)

Repeating this process can greatly reduce the total number of searches, making our Reversi AI faster.

We also include an interactive component to this project, allowing humans to play against the different AIs. Our main research question is: **To what extent can different Reversi AIs be successful at emulating different difficulty levels and beating human players?**

Computational Overview

Game Class and Logic

To represent a state of a Reversi game we implement the ReversiGame class, which contains the state of the game board, the current player's colour, the number of moves that have been made, and the human player's colour. The `reversi.py` file contains functions of the ReversiGame. The functions of this class are used to make moves in a Reversi games, update the game board, find all valid moves for the current player, and determining the winner of the game. A player can place one of their pieces on an empty square if when doing so, that player's pieces surround an opponent's pieces in a horizontal, vertical, or diagonal line. After doing so, the opponents pieces that were surrounded are changed to the current player's colour. The algorithm for finding the valid moves for the current player works by looking at each of the current player's pieces, checking in each direction to see if when placing a piece on an empty square in that direction, the opponent's pieces will be surrounded. All of these squares are stored in the list of valid moves. The path from the starting piece to the piece that is placed is also stored, since there may be multiple pieces whose path to the placed piece surrounds the opponent's pieces.

AI Players

The AI classes are in `ai_players.py` and our GameTree class is in `game_tree.py`. The different AIs are all built upon the `Player` abstract class, which has one function: `make_move`. This function has 2 parameters: the `ReversiGame` that is currently being used and the previous move played. The first player that we implemented is the `RandomPlayer`, which returns a random valid move from the position and is used as our benchmark. The more computationally expensive players are the `MinimaxPlayer` and `MinimaxABPlayer`, as these both use the `GameTree` class to make moves. The `GameTree` implementation is nearly identical to the implementation from assignment 2, with its instance attributes being its root move, a boolean representing whether it is white's move, a list of its subtrees, and a float representing its evaluation. A unique property of both of these players is that they both use the `heuristic_list` attribute of `Player`. This heuristic assigns a numerical value to each square on the board, and value is given to each position with a negative score being better for black and a positive score being better for white. We currently have two heuristics available, an absolute heuristic and a positional heuristic. The absolute heuristic simply computes the difference in the number of pieces for each player, so a score of -3 means that there are 3 more black pieces on the board than white. The positional heuristic can only be used on an 8x8 board, but the AI is much stronger when using it. It assigns more value to the corner squares and the squares on the edge. It also assigns negative values to the squares that lead to losing positions, which are the squares surrounding the corner square. These corner and side pieces are particularly important because you can capture a large number of pieces when connecting pieces on the edges.

When `make_move` is called in both of the Minimax Players, they recursively create a game tree using the `_minimax` function. If a leaf node is reached in the tree, then the `GameTree` is returned with an evaluation determined by the Player's heuristic list, which was discussed in the previous paragraph. If it is not a leaf node, then the Player checks if the game is won - if it is then it is a leaf node assigned with a large positive or negative score if a player has won, and a value of 0 if the game is a draw. After this, all possible moves are found in the given position and shuffled. They are shuffled to add variance to the players, as the `max` and `min` function in python will select the first value encountered in a list. Multiple trees may have the same evaluation, so we shuffle them so that every move with a good evaluation has a chance of being played. Finally, we iterate through each move and create a subtree for that move. `_minimax` is again called on this subtree. Finally, after all subtrees are added, the evaluation of the tree is updated as either the max/min evaluation of the subtree evaluations depending on which player is playing, with black minimizing and white maximizing. The `make_move` function will choose the max/min subtree of the returned tree depending on if it is white or blacks turn.

The `MinimaxABPlayer` optimizes the above algorithm by using the alpha-beta pruning process. Each recursive call to the function `_minimax` also passes an alpha and beta value, which allows for the pruning of trees that we know do not need to be searched. This process was explained in more detail in the introduction. These values are initialized at $-\infty$ and ∞ in the first call to `_minimax`. The algorithm follows in the same manner as the original minimax up to the creation of new subtrees. If it is white's turn, then alpha is continuously checked against the new subtree evaluation and set to this evaluation if it is greater than it. If the alpha value is greater than or equal to beta, then we `break` from the `for` loop, as we know black will never play into this position, as black must have a better move. This same process is done on black's turn, except we minimize the beta value with the subtree evaluations. This pruning process still results in the same move being generated if randomness is not used, as the pruning does not change the evaluation of the tree. This is confirmed using the `test_players` function.

Plotting Statistics

Similar to the graphs in Assignment 2, the `stats.py` module plots two graphs that show the statistics of a player's losses and wins using `plotly`. Unlike Assignment 2, for this graph, there is a parameter in the function that allows the choice of which player to focus on (that is, whether to plot the White player's win statistics or the Black player's win statistics). The first graph shows the wins, losses, and draws (which are represented by the numbers 1, 0, and 0.5) respectively for the given list of results. The second graph shows the cumulative win probability of the `focused_player`, as well as the win probability when only the most recent 50 games are considered. The win probability counts draws as "half-wins"; draws are weighted differently than both wins and losses in this calculation. This module is helpful as it provides a visual representation of how different players perform against one another over a span of numerous games. `generate_stats.py` simulates a given number of games between two players and calls the `plot_game_statistics` function in `stats.py` to plot the graphs which were used to compare the different AI in the Discussion section.

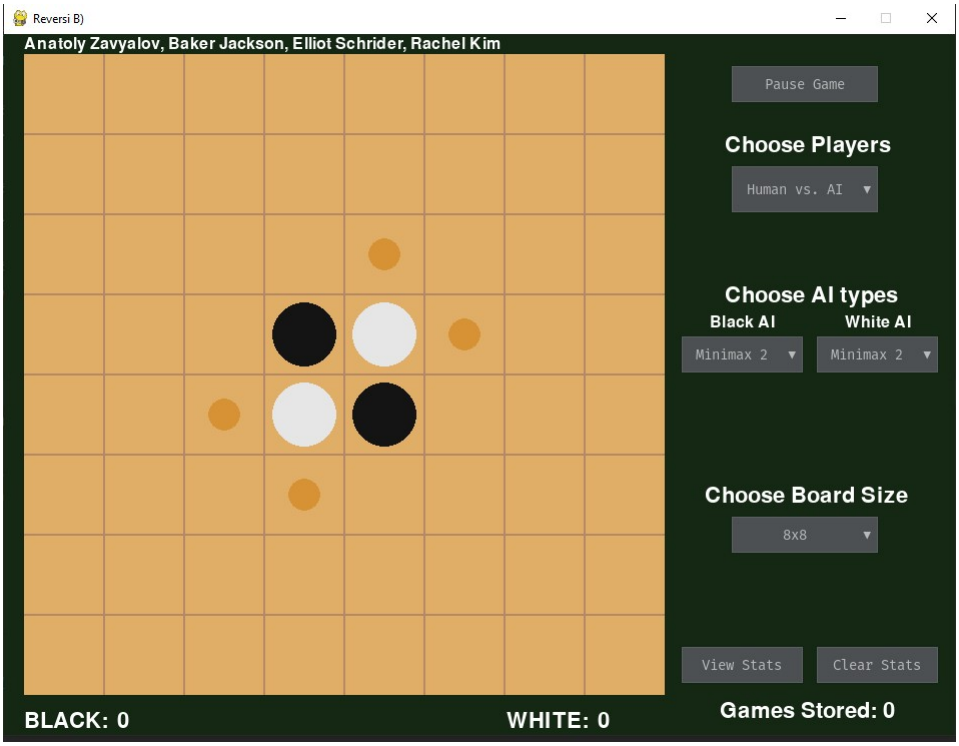
User Interface and Displaying the Game

To represent the window, we implement the `Window` class, which encapsulates a `pygame.Surface` instance (onto which the window's contents will be drawn) and contains methods for updating elements of the window, such as GUI, text, and allowing for other `pygame.Surface`s to be drawn onto the window (such as the board). The `Window` class also contains methods for drawing text by using the `pygame.font` module. To represent the board, we implement the `Board` class which stores the board pieces, board size, as well as the list of valid moves that the current player can play, as well as containing methods to update and retrieve these values. To draw the board, we implement the `BoardManager` class, which contains the method `draw_board`, which draws the board onto the `Window` attribute, and the method `check_mouse_press`, which detects clicks on squares on the board. The `BoardManager`'s `draw_board` method first draws the background of the board by drawing a plain surface with repeated vertical and horizontal lines, using `pygame`'s `pygame.draw.rect` method. The game pieces are drawn as circles (using `pygame`'s `pygame.draw.circle` method), as well as the next move indicators (which represent the squares that the current player can play). If the game is paused, an overlay is also drawn over the board along with text (using the `Window` class' `render_text` method).

For user interface elements such as buttons and dropdowns, we use the `pygame-gui` module, as it provides built-in, ready-to-use implementations of such UI elements. In `ui_elements.py`, we implemented wrapper classes for UI elements (text, button, dropdown). The abstract classes `Element` and `UIElement` contain instance attributes that provide abstract information about the UI elements. The `UIElement` class stores the actual `pygame_gui` `UIElement` instance, which is a superclass of `pygame_gui`'s `UIButton` and `UIDropDownMenu` classes. The `Button` and `Dropdown` are wrapper classes for `pygame_gui`'s `UIButton` and `UIDropDownMenu`, respectively. Both classes contain a `_function` private instance attribute, which is a lambda function that is called when the button/dropdown is interacted with. We also implemented a `Text` class, which allows us to easily draw and manipulate text on the window. These UI elements are handled and drawn in the aforementioned `Window` class. The UI elements are added into the program in `ui_handler.py`, which contains the `UIHandler` class, responsible for adding UI to a `Window` attribute (done in `add_ui`) and containing functions that are called by buttons and dropdowns when interacted with. The file also contains helper functions for several of `UIHandler`'s methods. The `UIHandler` class also contains the board size stored by the buttons (outside of the `Board` class) as well as a Boolean storing whether or not the game is paused, as to minimize parameters being passed around in the code.

Instructions to Run Program

1. After installing the Python libraries in `requirements.txt` and running `main.py`, you should see the following output:



2. The buttons and drop-downs on the right allow interaction with the user interface:

   - *Pause Game*: you can use this button to pause and resume the game.
   - *Choose Players*: the choices are all in "black v.s. white" order. By default, the program is set to "Human v.s. AI"; since black goes first, you can choose to make a move here by placing your piece on one of the smaller circles, which represent the valid moves you can make from a given position. You can also choose "AI v.s. Human," in which case the AI will make the first move. If you choose "AI v.s. AI," then you can see two AIs play on the board.
   - *Choose AI Types*: if you chose the "Human v.s. AI" option, then the selection you make for Black AI is irrelevant. Similarly, if you chose the "AI v.s. Human" option, then the selection you make for White AI is irrelevant. There are five choices for each of the AIs: Random Moves, Minimax 2, Minimax 3, Minimax 4, and Minimax 6, ordered from easiest to hardest to play against. If you choose either the "Human v.s. AI" or the "AI v.s. Human" option, you can choose which AI you want to use as your opponent. If you choose the "AI v.s. AI" option, then you can choose the two AIs that play against each other.
   - *Choose Board Size*: Here, there are four options: 8×8, 12×12, 16×16, and 24×24. Each of these board sizes works regardless of which option you choose in the "Choose Players" drop-down.
   - *View Stats and Clear Stats*: Every time a game finishes, the program stores the result of that game (the winner of the game or `'draw'`) in a list. You can click on the "View Stats" button to generate a graph for the wins, draws, and losses since `main.py` was run or since "Clear Stats" was last clicked on. You can click on the "Clear Stats" button to reset the list of results from any previous games. At the bottom, you can see the number of games that the Black Player and the White Player have won and the number of games stores since `main.py` was run or since "Clear Stats" was last clicked on.

Changes from the Project Proposal

The general direction of our project is the same as we initially proposed. There were, however, two main changes. First, we did not implement the Monte Carlo tree search algorithm, instead focusing on the Minimax algorithm and ways we could optimize it (for example, through alpha-beta pruning) and emulate different difficulty levels (for example, by adding a depth parameter). Second, rather than focusing on the board sizes of $4 \times 4, 6 \times 6$, and 8×8, we focused on $8 \times 8, 12 \times 12, 16 \times 16$, and 24×24. We incorporated some feedback from our proposal, such as letting the different AIs play against one another. There were also some details within the proposal that changed; for instance,

we realized that we did not need to create an entire class for the pieces, and instead incorporated them within the `Board` class, representing black pieces on the board with 1, white pieces with -1, and empty squares on the board with 0.

Discussion

Our research question is **To what extent can different Reversi AIs be successful at emulating different difficulty levels and beating human players?** In answering this question, we consider three subtopics: the running time of different Players, how different Players perform against one another, and human v.s. AI games.

Comparing Running Times: An important aspect of determining the "success" of Reversi AI is the extent to which the AI can challenge human players while running at a reasonable speed. One limitation earlier in our process was that `MinimaxPlayer` began to slow down significantly at depths greater than 3. Using alpha-beta pruning, however, we can increase the depth and greatly reduce the running time. Using the `check_same` function in `ai_players.py`, we can observe this difference in running time by checking the difference in the time taken by `MinimaxPlayer` and `MinimaxABPlayer` to choose a move. For depth 5, we get:

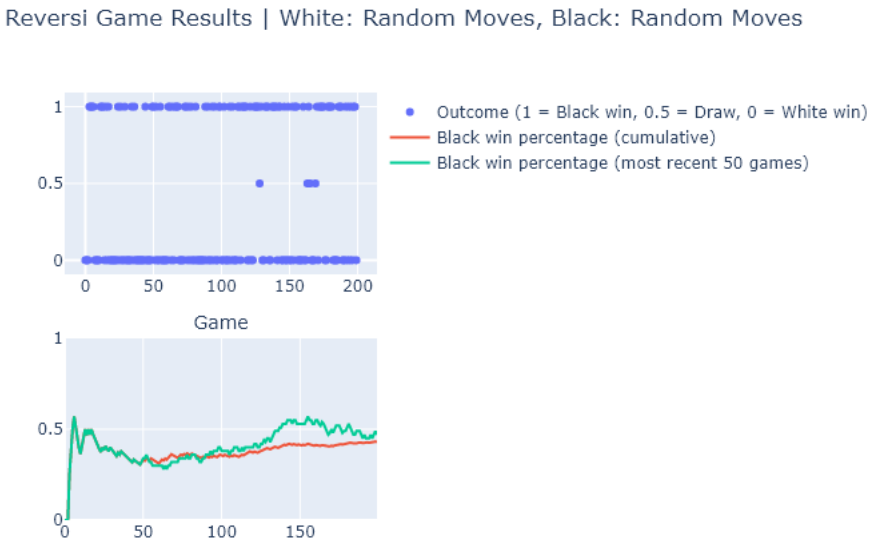
```
OLD MINIMAX CHOOSING
--- 6.556999921798706 seconds ---
AB MINIMAX CHOOSING
--- 0.6906838417053223 seconds ---
```

For depth 8, the difference is more pronounced:

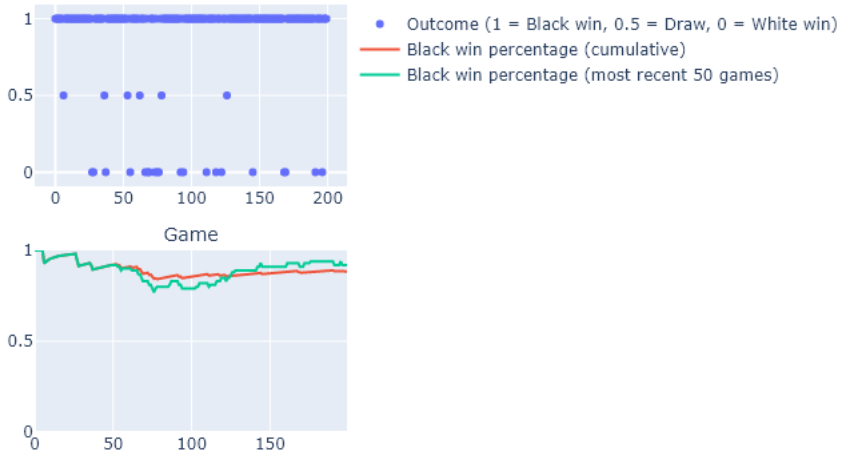
```
OLD MINIMAX CHOOSING
--- 142.9456729888916 seconds ---
AB MINIMAX CHOOSING
--- 2.9904911518096924 seconds ---
```

Overall, this indicates that we can reduce the running time enough to allow the user to play against `MinimaxABPlayer` at depths 4 and 6, thus allowing for more difficult AI opponents.

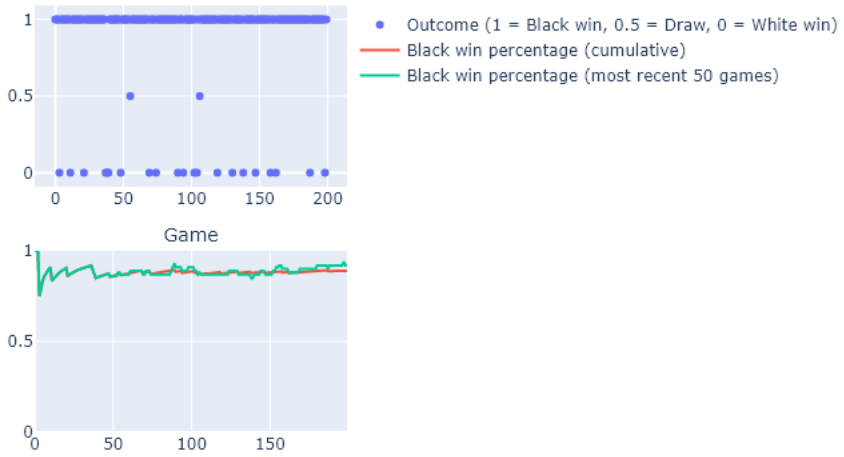
AI v.s. AI: We see whether there is a meaningful difference between the different AIs we have created with regards to how well they perform. We hypothesize that `RandomPlayer` is the "easiest" AI to play against since it chooses its moves randomly, without any algorithms to determine which move might give the AI a strategic advantage. We also hypothesize that the greater the `depth` parameter is for the `MinimaxPlayer`, the stronger it will be since the AI is able to consider more possibilities and thus has a higher likelihood of choose the optimal path. Indeed, the statistics confirm our hypotheses:

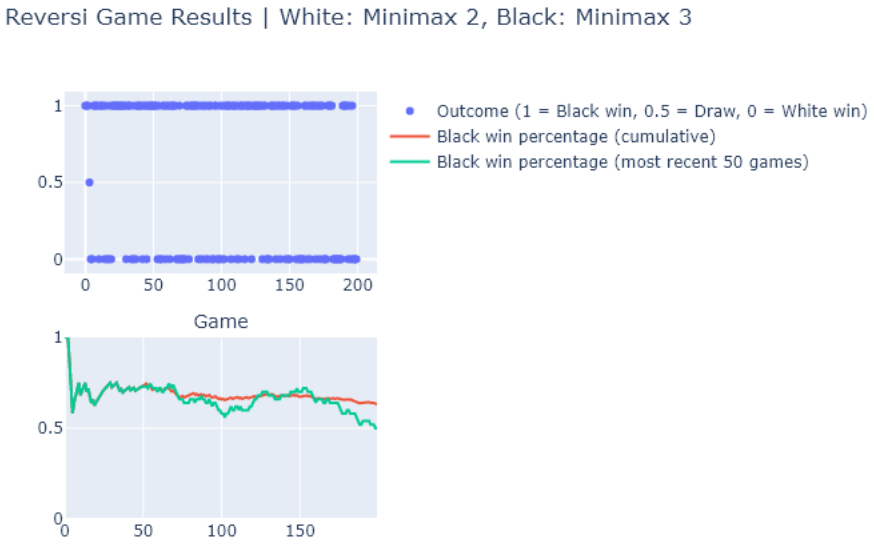


Reversi Game Results | White: Random Moves, Black: Minimax 1



Reversi Game Results | White: Random Moves, Black: Minimax 2

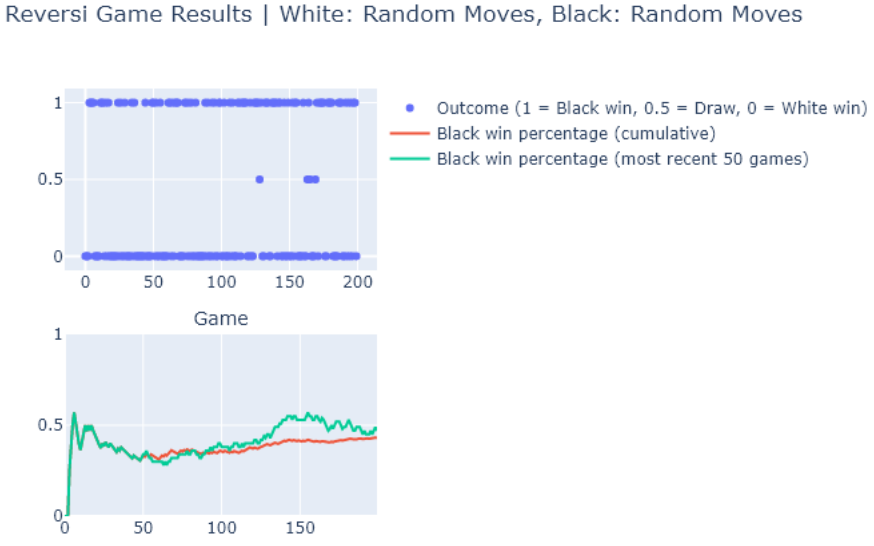


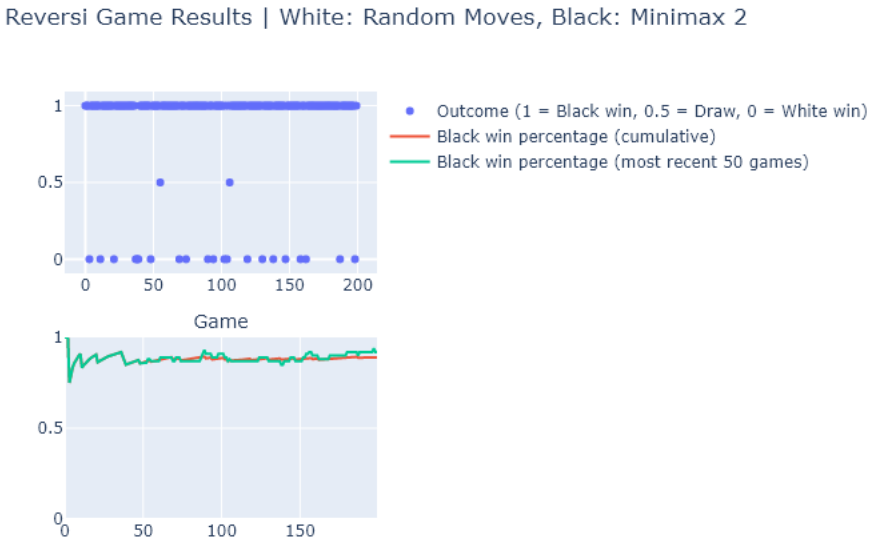


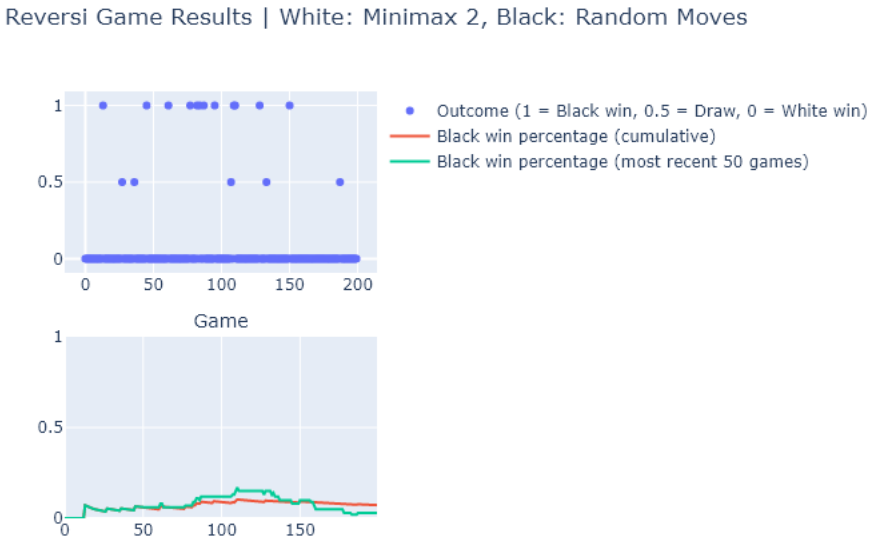
(Note: all Minimax Players we use for these graphs are `MinimaxABPlayers`)

Indeed, over a series of 200 games, we can see that `MinimaxABPlayer` at both depths 1 and 2 consistently win `RandomPlayer` and that `MinimaxABPlayer` at depth 3 performs better than `MinimaxABPlayer` at depth 2. From this, we can extrapolate that a `MinimaxABPlayer` of a higher depth outperforms one of a lower depth and that even the simplest `MinimaxABPlayer`, one with depth 1 or 2, outperform `RandomPlayer`. Thus, we can use `RandomPlayer` to represent an "easy" difficulty level, `MinimaxABPlayer` with depth 2 or 3 to represent a "medium" difficulty level, and `MinimaxABPlayer` with depth 4 or greater to represent a "hard" difficulty level (Note: we did not generate graphs for all permutations of different Players, since this would generate a large number of graphs, and instead chose a few tests that are representative of the general trends).

We also confirm that the black player and the white player perform similarly; that is, one player does not have a significant advantage over the other when all other variables are kept the same. Indeed, we see that this is the case:



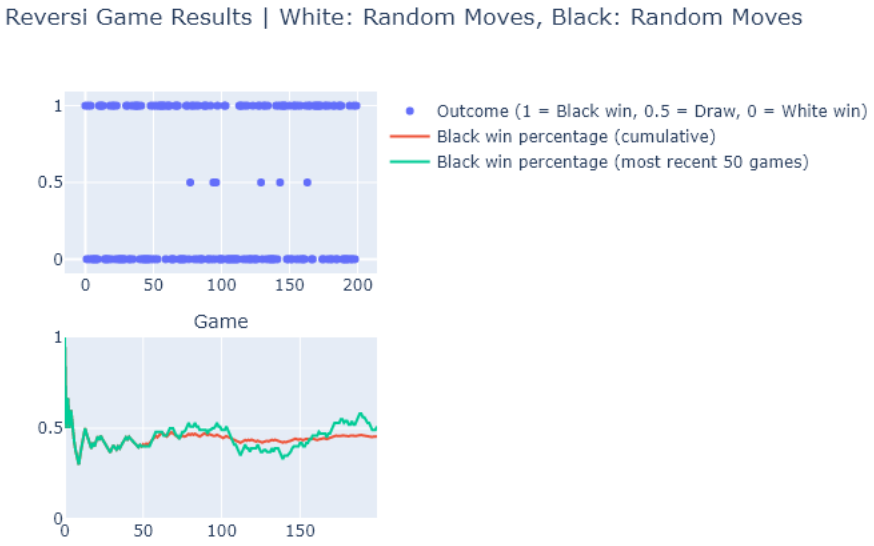


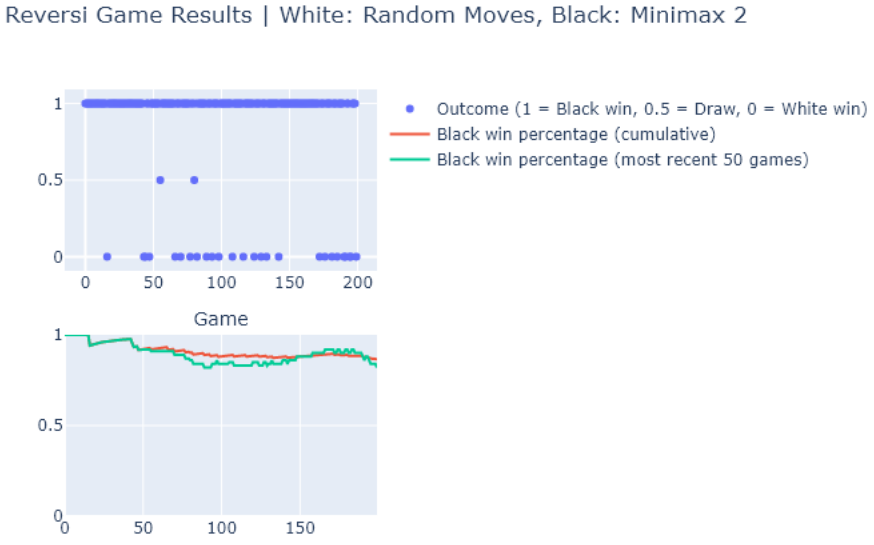


(Note: all Minimax Players we use for these graphs are `MinimaxABPlayer`s)

For `RandomPlayer` v.s. `RandomPlayer`, we see that the win probability of the black player is around 50%, as expected. By comparing two graphs where the types of players used for the black and white players is switched, we see that the win percentage of `MinimaxABPlayer` at depth 2 against `RandomPlayer` is the same, regardless of the colour of the `MinimaxABPlayer`. So, we conclude that the black player and the white player perform similarly for our AIs (Note: this does not necessarily mean that neither player has an advantage in this game; it just indicates that our AIs are not significantly affected by any differences the two players may have).

Finally, since all the graphs above were for the default 8×8 board size, we check how the different AIs perform on larger board sizes. Similar to the above graphs for 8×8, we run 200 `RandomPlayer` v.s. `RandomPlayer` games and `MinimaxABPlayer` with depth 2 v.s. `RandomPlayer` games:





(Note: all Minimax Players we use for these graphs are `MinimaxABPlayer`s)

Seeing that the graphs for both the 8×8 and 16×16 boards are similar, we can see that the different Reversi AI we created are able to effectively emulate different difficulty levels regardless of board size.

Human v.s. AI: As a moderately-experienced Reversi player, I (Rachel) was able to beat `RandomPlayer` almost every round. `MinimaxABPlayer` with depth 2 and 3 gave me mixed results; sometimes, I was able to beat the AI, while at other times I lost. I have not been able to best `MinimaxABPlayer` with depth 4 or 6 yet. Overall, this indicates that our Reversi AI can challenge and outperform the average human player.

Overall, we believe that we were quite successful answering our research question. Our Reversi AI can represent different difficulty levels of Reversi AI and some can challenge and even beat the average human player. As a next step, we could implement the Monte Carlo tree search algorithm, and analyze how this algorithm performs against the AI we already made with regards to running time and win percentage.

References

Allis, V. (1994). Searching for solutions in games and artificial intelligence (thesis). s.n., S.l. http://fragrieu.free.fr/SearchingForSolutions.pdf

CardGames.io. Reversi. Reversi — Play it online! https://cardgames.io/reversi/#rules.

Gambling Sites. (2020, January 11). A Complete Guide to Reversi - Introduction and How To Play. Gambling Sites. https://www.gamblingsites.com/skill-games/reversi/.

GeeksforGeeks. (2019, December 5). Minimax Algorithm in Game Theory: Set 4 (Alpha-Beta Pruning). GeeksforGeeks. https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning/.

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

MyreMylar/pygame_gui. (2020). GitHub. https://github.com/MyreMylar/pygame_gui

PyGame Documentation. (2021). PyGame. https://www.pygame.org/docs/

Wikimedia Foundation. (2021, February 5). Computer Othello. Wikipedia. https://en.wikipedia.org/wiki/Computer_Othello.