Final Project Report

Jacob Justice
Student at Auburn University
Auburn, AL 36830
jhj0011@auburn.edu

Alexis Myrick
Student at Auburn University
Auburn, AL 36830
amm0153@auburn.edu

Samuel Skipper
Student at Auburn University
Auburn, AL 36830
sas0006@auburn.edu

Abstract

For our final project, we created an artificial intelligence that plays the board game *Othello*. The solution will be based upon some variant of advanced search, considering there is no specific goal state, with the aim of reaching the highest possible score. We implemented three different scoring algorithms to evaluate the AI's avaliablemoves; harder difficulties are increasingly more complex. The easiest will operate on a random, non-intelligent process. The intermediate consists of a greedy search tree algorithm, which equates to the default scoring by the rules of Othello. Finally, the most advanced difficulty will be built upon the minimax algorithm, which will take into account the opponent's moves to a given depth.

1 Task definition

1.1 Othello Overview

Othello is a game consisting of two players, black or white. The board (size 8x8) starts with a 2x2 square in the center, consisting of tiles white, black, black, white (in left-right, top-down order). Each player systematically takes turns placing down a tile in an adjacent square to a cell with a tile in it already, under the condition that the tile "sandwiches" one or more of the opposing players pieces. All "sandwiched" tiles get flipped to the color of the player that initiated. This continues until both players can no longer play (which does not necessarily equate to a completed board). Black always goes first.

1.2 Input / Output

A user will be presented with various levels of difficulty when first initiating the software. These levels are used to determine the algorithm used, as well as search-depth (if using the minimax algorithm). The user will also be given an option to either play themselves, or to set another algorithm to play against the initial algorithm. At this point, the game will begin, kicking off a series of back-and-forth interactions between the user and computer, or the computers against each other.

When it is the computer's turn, it is given the difficulty level and the current state of the game. It will then use the appropriate algorithm to determine the best (as determined by the current score function) move to make, and will return the new state of the board after making that move.

If it is the user's turn, they will be given the current state of the game (to be displayed in a graphical interface). The user will be able to interact directly with the board, and the new state of the board resulting from their move is returned for the computer to evaluate their next option. If

the user opted to let the AI take over, it works the same as the computer's turn as described above.

Upon the game's conclusion, the final score is displayed, and the winner is announced.

1.3 Challenges

The main point of difficulty can be found in the minimax algorithm. The most effective way to create a perfect artificial intelligence opponent would require a search-depth without a set bound. This algorithm is a solution that does not scale without a search-depth restrain and would prove to be unreasonably costly, however, we can apply a fairly generous search-depth restriction and still receive results that are reliable and difficult for a typical user to outperform. The minimax algorithm will be able to look ahead even up to ten states and still give reasonable performance. This will ultimately require some empirical testing as to the performance of the algorithm to evaluate target search-depths that can be used for different difficulties.

Gathering accurate results has been an additional challenges with the minimax algorithm. In the early phases of development, determining if the algorithm was working correctly and effectively proved difficult, as any errors in the algorithm were not immediately obvious, and remained hidden throughout several run throughs.

2 Infrastructure

2.1 State

As previously mentioned, the state describes the Othello board at a given point in time. The state itself is an 8x8 array of cells, where each cell has a status determining if it is unowned, white, or black. The game keeps a global state, used at the start of each turn. It is worth noting that the state may not always be the current, global state of the game (consider the case of minimax which evaluates multiple consecutive turns into the future before modifying the current global state).

2.2 Utility Functions

In order to compute changes to state, we deemed a number of utility functions necessary. Although a few of these were trivial or only applied to the user interface (both of these types of functions are omitted from the paper), a select number posed interesting challenges, requiring a strong understanding of the problem in order to implement.

2.2.1 Score

Although itself being simple, the scoring function posed challenges due to its complexity. In order for our greedy or minimax AIs to evaluate the difference between two moves, they must enumerate a specific score from the resulting state. Othello has its predefined definition of score, being the current number of a given color of tiles on the board (i.e., the game starts with score(white) == score(black) == 2). This is trivial to compute, but requires checking the current status of each cell (64 in total for an 8x8 board).

2.2.2 Neighbor Cells

It is important to be able to return all "neighboring" cells for a given cell in the state (reasons for this are elaborated on in 2.2.3 and 2.2.4). Considering a given input *cell*, this means 3 cells are returned if *cell* is in a corner, 5 if *cell* is along a straight edge, and 8 otherwise. These can easily be evaluated in constant time.

2.2.3 Selectable Cells

Both the AI and player should be told which cells are available to play a piece onto. It's very simple for us as people to evaluate what moves will sandwich the cells of an opponent, but it is more complicated to evaluate this programmatically. In order to determine the set of selectable cells, the entire *state* should be iterated over, and the following three criteria must be checked for each given *cell.* Any cell that meets these criteria is appended to an array of selectable cells and returned by the utility function.

Criteria 1: Is cell of value unowned? This is simple enough to understand: if a cell has already been taken, it cannot be played over.

Criteria 2: Is there some neighbor of the opponent's color? It is important to remember the moves are only valid if they result in sandwiching an opponent's cells. As such, if *neighbor* (from the set of all *neighbors)* is either the current color or unowned, then *cell* does not sandwich and is invalid. At this point, a unit direction vector *d* should be taken between *cell* and *neighbor* (for example, (1, -1) indicates that *neighbor* is forward one row, and back one column from *cell*). Note that the first attempt at *neighbor* may not result in finding *cell* to be selectable, so this should iterate over all *neighbors* until one either results in a sandwich, or all *neighbors* terminate

Criteria 3: If continuing on in direction d, will the current team's color be reached? The distance vector *d* allows to indefinitely update *neighbor* to its adjacent neighbor in the same direction. If at any point no such neighbor exists (i.e., an edge is hit), or if *neighbor* is unowned, then *cell* is not proved to be selectable. If, however, *neighbor* eventually results in the opponent's color, *cell* can be appended to the list of selectable cells, and the search early exits to the next *cell* in *state*.

2.2.4 Flipped Cells

Evaluating flipped cells operates almost identically to evaluating selectable cells. The major difference comes in the input and output. Instead of iterating over a full state, flipped cells takes in a single *cell* within the state (a cell that has just been moved). The function seeks to return the state after making the move (i.e., update each affected *cell* in *state*). As such, during *Criteria 3* (as described in 2.2.4), there is no early exit, and any *neighbor* found is set to the same status as *cell.*

3 Approach

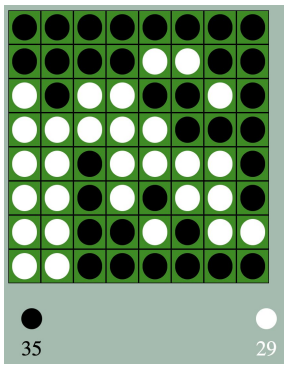
In an attempt to gather meaningful data, we had considered a number of algorithms to approach our problem with. The intention of our approach was to implement an algorithm that balanced performance and effectiveness. After we decided our approach, we began to determine which algorithms with a more narrow scope to implement as a baseline.

3.1 Random

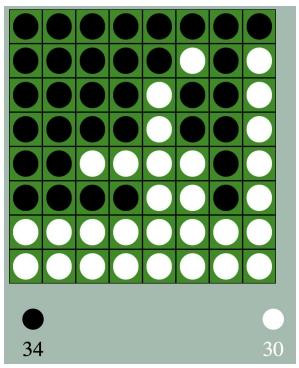
The random option is accessed by the user selecting the "Beginner" difficulty. The random baseline simply looks at all of the possible moves to make in that current states, and then at random chooses one. Random is not useful in itself for gathering meaningful information, however, it does give meaningful results for the greedy and minimax algorithm. The other implemented algorithms will give a consistent end state when they are against each other, by setting a computer against random, we can collect more varied data and capture different end states.

The following are snapshots of the random algorithm set against each of the other algorithms. In each of the snapshots the random algorithm is denoted as the black tile, who always has the

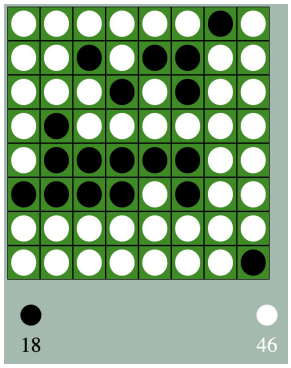
starting move.



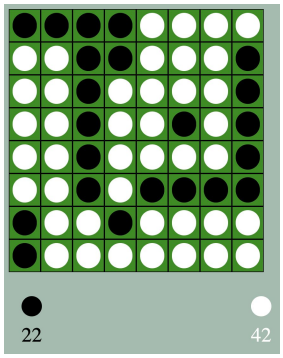
Random vs. Random



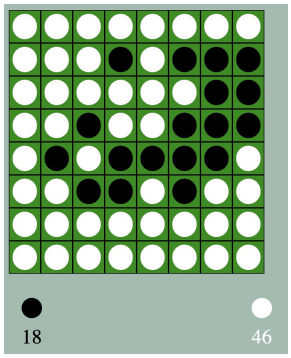
Random vs. Greedy



Random vs. Minimax d=3



Random vs. Minimax d=5

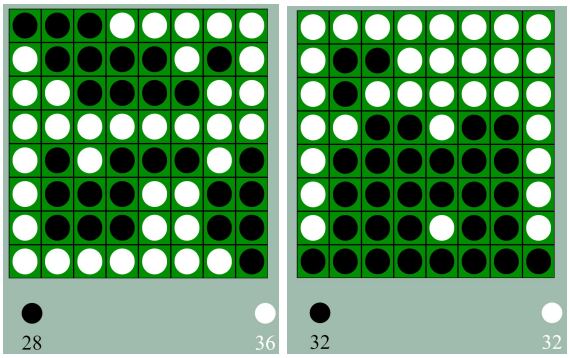


Random vs. Minimax d=7

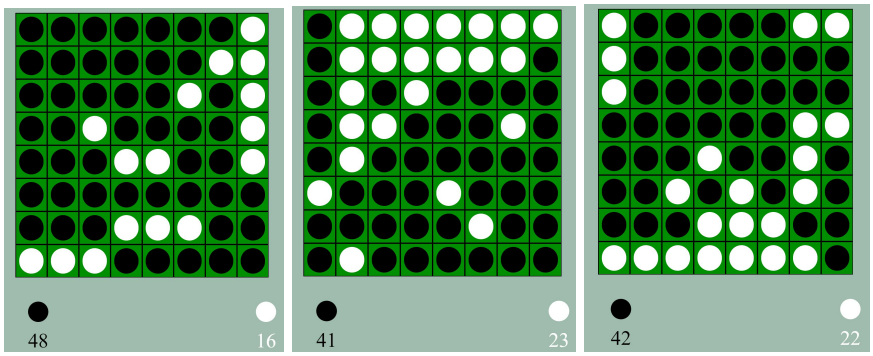
3.2 Greedy

The greedy algorithm is accessed by the user selecting the “Intermediate” difficulty. The greedy baseline chooses the best possible move to progress to the best possible state by maximizing it’s score without looking ahead at any future states. The greedy algorithm is essentially the minimax algorithm with a depth of 1. Because the greedy algorithm does not look ahead into future states, testing and confirming results as accurate can be done quickly.

The greedy algorithm was intended to commonly wins out against the random baseline, and to also occasionally outperform an inexperienced human player. In actuality, the results are not as clean cut. One interesting finding shows that playing Othello in a greedy manner can perform poor results, as the greedy algorithm is only slightly more likely to win out over a random play style. After 25 trial runs with the greedy algorithm playing the first move against the random algorithm, the greedy algorithm only came out on top 14 times, with the random algorithm winning 10 times, and the remaining 1 game as a tie. In another 25 trial runs with the greedy algorithm playing the second move, the greedy algorithm comes out on top only 13 times, versus the random algorithms 11 times, and again, 1 tie.



Greedy vs. Random Greedy vs. Greedy



Greedy vs. Minimax d=3 Greedy vs. Minimax d=5 Greedy vs. Minimax d=7

3.3 Minimax

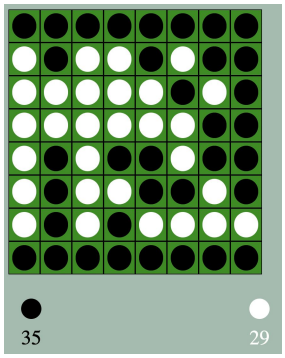
The minimax algorithm is accessed by the user selecting the "Master" difficulty. The minimax baseline aims to maximize its score while making the assumption that its opponent will consistently play in a "greedy" manner. By looking ahead into future states, the minimax algorithm will choose the best state at a specific depth (either 1, 3, 5, or 7) that the user has selected. At odd numbered states, the algorithm will choose the option that maximizes its score while taking into account the previous even numbered state. This previous even number state is a simulation of what the algorithm's opponent will move. The assumption is that this opponent will choose the option that will both maximize its score while attempting to minimize the algorithms score. The minimax algorithm continues this pattern until the end of its preselected depth. Once the maximum depth has been reached, the algorithm will choose the option that will lead to its best case state ahead in the future. Note that a state is passed within the algorithm, but the global state is not updated until the algorithm completes.

One interesting finding when putting the minimax algorithm against the greedy algorithm is that performance is not especially notable unless the selected depth is at least 5. It is reasonable to expect the minimax algorithm to outperform the greedy algorithm consistently with a depth of 3, however, from our current findings, that may not be the case.

If minimax with a depth of at least 5 is put up against the base "greedy" algorithm from the intermediate mode, one can expect minimax to outperform in every instance. The reason being that the whole idea of minimax is to outperform a greedy play style, the greedy algorithm should fail against a minimax with depth greater than or equal to 5 every time.

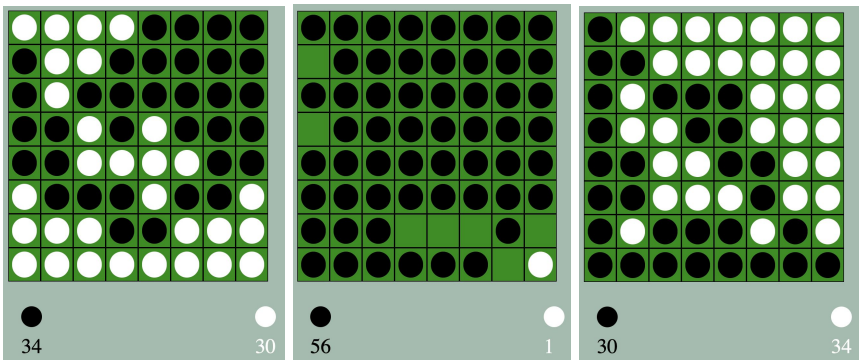
When playing the minimax algorithm against the "Beginner" option, minimax will outperform random in most, but not every, instance. The reason there is a possibility for minimax to lose against the random algorithm is due to the fact that minimax chooses its next move with

consideration that its opponent is playing logically and in a greedy manner. This finding does show a weakness of the minimax algorithm, the best bet to outplay it may be to play unlike a human would, and to choose random spots to place game pieces instead.

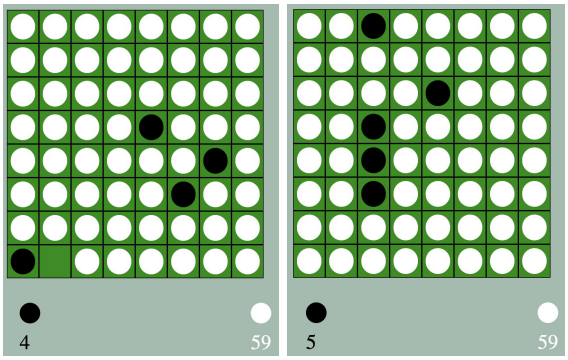


Case where Minimax was outperformed by random

The following snapshots display how the minimax algorithm performs in a typical match against the other implemented algorithms. To avoid redundancy, the minimax algorithm with a depth of 1 is not included here, as it would essentially be the greedy algorithm again.

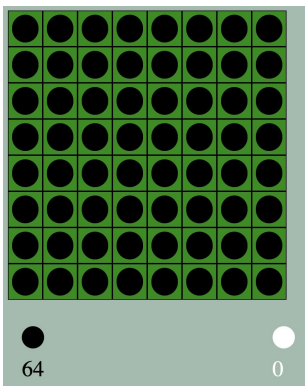


Minimax d=3 vs. Random Minimax d=3 vs. Greedy Minimax d=3 vs. Minimax d=3

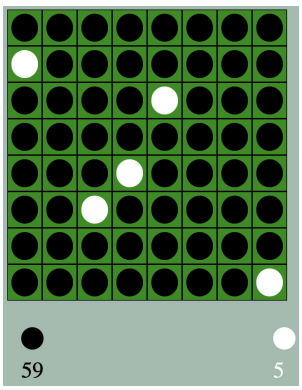


Minimax d=3 vs. Minimax d=5 Minimax d=3 vs. Minimax d=7

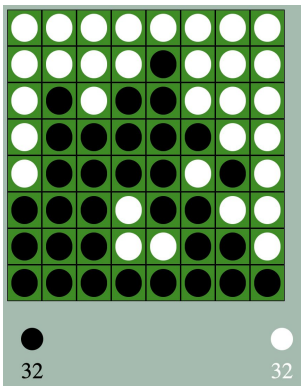
The final results match the typical results that we have found in our testing from depth 3 against every other algorithm. Depth 3 will often perform better against random and the greedy algorithm (d=1), it will have a close competitive game against its matching depth (d=3), and will lose out against the greater depths (d=5, d=7)



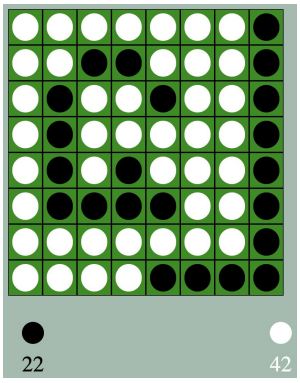
Minimax d=5 vs. Random



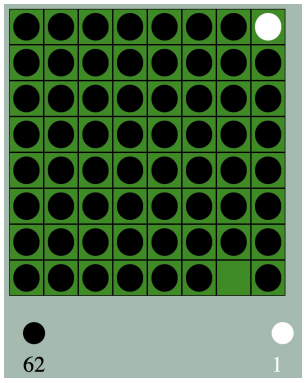
Minimax d=5 vs. Greedy



Minimax d=5 vs. Minimax d=3



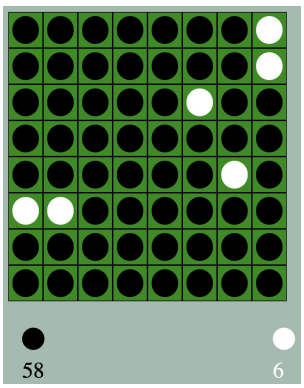
Minimax d=5 vs. Minimax d=5



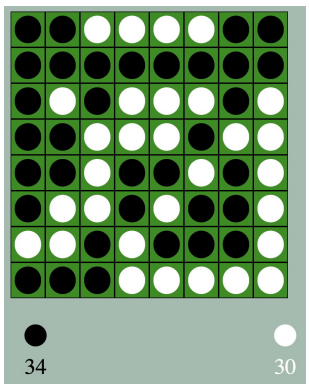
Minimax d=5 vs. Minimax d=7

The results above for depth 5 are the common final states found in our testing. These results are not quite what we would have initially expected them to be. Minimax with a depth of 5 will often win out against the random algorithm, as well as the greedy algorithm, We would have expected a depth of 5 to greatly outperform a depth of 3, yet its final state ends in a tie. Depth 5 against depth 5 gives no evidence of a player starting first receiving an advantage. The final, and perhaps most interesting results can be found from depth 5 against depth 7, in which depth 5 outperforms depth 7 to a greater extent than even the greedy algorithm.

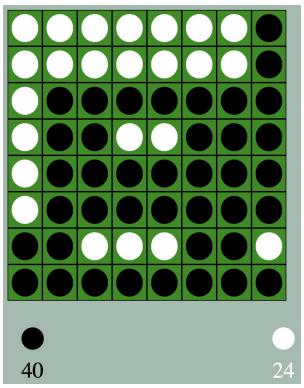
Depth 5 is also the point where the number of computations necessary to traverse all possible states increase its running time and gives noticeable slowdown. During the initial first few moves and the final few moves performance is smooth, this is due to there being less possible areas to traverse. However, by the mid-point of the game session, the runtime hits a peak and performance takes a somewhat noticeable hit.



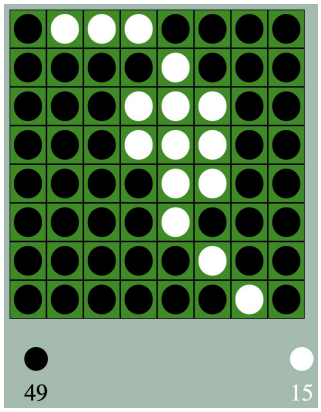
Minimax d=7 vs. Random



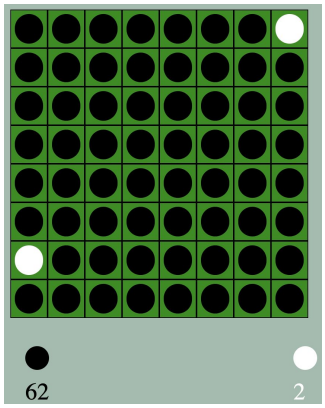
Minimax d=7 vs. Greedy



Minimax d=7 vs. Minimax d=3



Minimax d=7 vs. Minimax d=5



Minimax d=7 vs. Minimax d=7

Finally, we have the ending states for minimax with depth 7 versus the other implemented algorithms. From here, we see depth 7 performing just as expected, with wins over every other method. The only interesting outlayer is when up against the greedy algorithm, this matchup is closer than expected, but the minimax algorithm was forward thinking enough to still secure a win.

Runtime is worth noting here. The amount of time necessary to compute the actions of depth 7 is exponentially more than depth 5. Again, we see better performance during the earlier and later moves, when movement is limited, with a peak point of processing during the middle portion.

3.4 Approaches considered

Along with the options implemented within the program, there were a few other algorithms that were considered in an attempt to create an efficient artificial intelligence, and to collect data from this project. The issues with deciding on an algorithm for a game like Othello is that any possible move for a player to make will improve their situation. With this in mind, any algorithm that attempts to move towards a better state will essentially be random at worst, and greedy at best.

Both Hill Climbing and Simulated Annealing were both seriously considered for implementation. After the performance of these two became obvious, these options were discarded. In an attempt to use another baseline algorithm that would deliver meaningful data, we decided on implementing different levels for the minimax algorithm, rather than implementing either Hill Climbing or Simulated Annealing.

4 Literature review

4.1 Game Theory — The Minimax Algorithm Explained[1]

It is pretty intuitive to think of the flow of a game as a tree, where each given node is the resultant state of taking a move in the game, and the children of a node represents the next potential states from a given one. Eppes discusses the idea of creating a value that can be calculated for a given state. However, it is not this simple evaluation that distinguishes an advanced AI over a novice player, but instead, as Eppes points out, it is the ability to think ahead multiple turns.

Essentially, Eppes describes the algorithm as taking turns maximizing or minimizing the current score, back and forth recursively, to a given depth. As she says, "Your next move is only as strong as the opponent's following move is weak." In particular, a move does not matter if it nets poor score in the long run. This gets very complicated, enumerating all resulting options off of all current choices, and then all options from those, and so on.

In the example, the author speaks specifically of an implementation for chess, but in reality the concepts of minimax are applied to other board games (in this case, Othello is a prime example for minimax).

4.2 Solving Othello[2]

With the amount of possible states in a game of Othello, it seems that any sort of advantage that a player may gain by going first or second can always be overcome in perfect play. Initially the school of thought on this subject was that the second player may have an unfair advantage to come out on top, the reason for this being that the second player is more likely to place down the game piece that meets the ending condition. A database of Othello states provided from the software *Zerba Othello* strongly suggests that the ending move is more likely to flip more game pieces than the move before it.

However, the ending condition can just as well be met by the player that goes first. By creating a situation where there remains an empty spot that is not accessible by either white or black, then this advantage is true for the player who started first.

In conclusion, there does seem to be an advantage for the second player, this advantage may also be somewhat minor, as an Othello board of lesser size (4x4 or 6x6) can be strongly solved with a distinct advantage for the first player. Othello played on an 8x8 grid seems to have the advantage for the second player, this goes against our initial hypothesis that the first player had an advantage. This finding brings to light the complicated nature of game theory and the unfathomable amount of possible states. Even if our outcomes were correct and seemed biased towards the first player, these states make up such a small portion of the ending outcomes that it is difficult to draw a firm conclusion on.

5 Error analysis

5.1 Issues encountered

Possibly the biggest issue in computation stems from efficiency. This is something we were able to optimize as we continued development by decreasing complexity in the utility functions. There are perhaps still ways we could continue to optimize the code. Currently depth 7 is the highest depth that minimax could tolerably run on in our app, but even that takes longer than would be acceptable within the confines of a user application.

Throughout development, we ran into a few specific bugs upon spectating the algorithms and reviewing our data, all of which are fixed in the final code. One of the earliest bugs came from an improper implementation of minimax. In fact, the AI was originally maximizing the opposing score and minimizing their own. Although this was quick to be fixed, it paved the way for an edge case that was not as easily noticed. When a game nears its end, it is likely that one side goes multiple times. For example, if white has no moves, black keeps going until either white can move or black runs out of moves. Although we accounted for this in the actual turn logic, we did not within minimax at first. This resulted in the AI possibly seeing its action's results that did not equate to what would actually happen from that series of inputs (i.e., a move could be detrimental for it because it would allow the opponent to go 3 times in a row, but the AI would not know that). This was a pretty fascinating bug to run into, because it really showed how harmful bad data could be to the AI system.

5.2 Findings

One interesting observation is that the results of a game with a given combination of algorithms seems to vary depending on which one is white and which one is black. This may be due to the fact that black always goes first. Initially, it appears that whichever one is black has somewhat of an advantage. Even if the same algorithm wins as both white and black, it may win by a lot more if it is black. This can be seen by comparing greedy (black) vs minimax d=5 (white) and minimax d=5 (black) vs greedy (white). However, this is not always the case, as we see with the results of minimax d=5 and minimax d=3.

Another observation is that for the higher depth levels of the minimax algorithm, the calculation time increases most significantly in the middle of the game. The middle of the game is when there are the most options of squares to pick, and therefore the most calculations need to be made. Although not optimal, a practical algorithm would probably vary the depth of minimax based on the number of selectable cells.

References

[1] Eppes, Marissa. (2019) Game Theory - The Minimax Algorithm Explained.
https://towardsdatascience.com/how-a-chess-playing-computer-thinks-about-its-next-move-8f028bd0e7b1

[2] Kling, Arnold. (2012) Solving Othello: a Follow-up
https://www.econlib.org/archives/2011/02/solving_othello.html