Isabela, Amelia, James

Intro to Artificial Intelligence

Fall 2018

AI Final Project Report: Othello

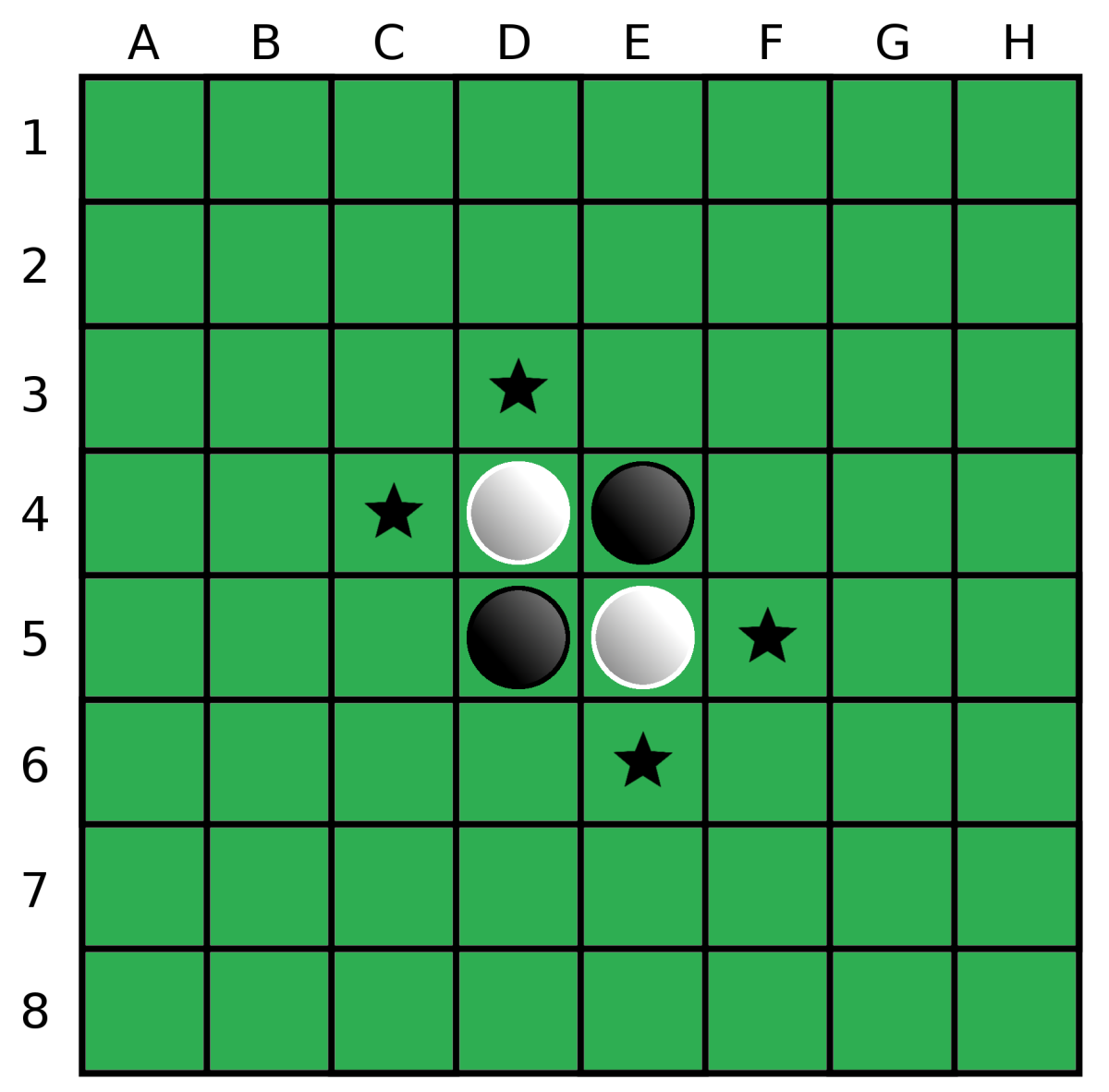
# Introduction

AI researchers are interested in games because they are hard to solve.\cite{AI\_textbook} A player has many possible moves to choose from at every turn, and depending on the complexity of the game, it may be impossible to make an optimal decision. AI considers moves that a human player would likely overlook because these moves may appear to have no immediate impact. However, unlike a human player, AI has much greater predictive power and considers moves that prove to be useful much later in the game.

For our AI project, we studied and implemented an AI that plays the game Othello. Othello is a two-player game that uses an 8-by-8 game board with columns A-H and rows 1-8 (figure).

The game starts with four discs in the center of the board: two white discs in D4 and E5 and two black discs in E4 and D5. One player uses black discs and the other player uses white discs. (In our implementation, we assigned the AI to black discs, and the AI moves first.) At each turn, a player places one disc on the board. A player can only place a disc in positions where the opponent's disc, or a row of opponent's discs, is flanked by the player's discs. Once a player places a disc, all of the opponent's discs in between the player's discs that are a straight line distance from each other will reverse in color. A player can capture vertical, horizontal, and diagonal rows of discs that connect one disc with the disc that they will place next. Also, more than more row can be captured by one disc placement. The objective of the game is to have more discs than the opponent when the game is over. The game ends when neither player has a move, which typically happens when the board is full. If a player’s opponent cannot make a move, the player continues to make moves until the opponent can move again.

* **Include figure of othello game board**



# Background

\*\*\* written in the latex document\*\*\*

\section{Background}

\subsection{Adversarial Search}

Game theory is a branch of mathematics that provides tools for analyzing situations in which parties, or players, make decisions that are interdependent.\cite{game\_theory\_brit} Each player considers the other player's potential set of moves when deciding what move to make next.

In AI, the most common games are deterministic, turn-taking, two-player games in which one player wins and the other loses -- making the situation adversarial. The most well-known and effective algorithm for choosing optimal moves in two-player adversarial games is minimax. In minimax, there are two players, MAX and MIN. MAX typically moves first, followed by MIN. A game can be defined as a search problem since MAX and MIN are each searching for the best move from their perspective at every turn. MAX considers all of MIN's potential moves and chooses the move with the maximum utility for MAX. MIN then considers all of MAX’s next potential moves and chooses the move with the minimum utility for MAX. MAX and MIN continue taking turns until the game ends.

For most games, the search space is exponential, so constructing a complete minimax tree is not feasible. A more practical approach is $\alpha\beta$ pruning with a depth-limited minimax tree. The $\alpha\beta$ algorithm is a depth-first search that computes bounds, alpha and beta, on internal nodes. MIN updates beta, where beta represents the minimum value of its children. MAX updates alpha, where alpha is the maximum value of its children. The algorithm prunes whenever alpha is greater than or equal to beta. It is important to note that $\alpha\beta$ pruning never removes the winning strategy: it only removes nodes when a better alternative exists, and the position is provably bad or the adversary can force a bad position. $\alpha\beta$ pruning can therefore eliminate large portions of the tree from consideration while still returning the same move as minimax. \cite{AI\_textbook}

\subsection{Typical Game Play Heuristics and Search}

Othello is a deterministic game with a low-branching factor. The most common approach to implementing an AI for Othello is performing $\alpha\beta$ search on a partial game tree. While most $\alpha\beta$ search implementations are similar, the differences lie in the heuristic evaluation function. $\alpha\beta$ search is highly effective when the heuristic function can accurately evaluate the utility of any board position. \cite{UW\_heuristics}

There are four well-known properties that factor into the expected value of each node: mobility, coin parity, stability and corners-captured aspect. \cite{WA\_monte\_othello} These properties vary in importance and should therefore be weighted accordingly. The heuristic evaluation function scores each of these properties and adds them together. Some heuristics use static weights, but dynamic weights are more effective.\cite{UW\_heuristics As the game of Othello progresses, the value of board positions changes as well as the importance of mobility, stability, etc.

Improved game play in Othello can result from improvements in the search strategy, better heuristic functions, using learning methods to enable the computer to learn for itself from massive amounts of data, or adding extra hardware support. In our project we focus on improving the heuristic and search strategy, and we attempt to work with learning methods. We do not add extra hardware support since we want the game engine to play as optimally and efficiently as possible on an average laptop.

\subsection{Alternatives}

Aside from the aforementioned basic methods for searching and heuristics, there are a few methods that can be added to potentially increase the optimality of our game engine.

**~~\subsubsection{Reinforcement Learning}~~**

\subsubsection{Monte Carlo Search}

While minimax considers all legal moves from a given state, MCTS samples moves instead. MCTS can therefore handle large search spaces with high branching factors. MCTS is especially useful in non-deterministic games where it is difficult to derive an accurate evaluation function.\cite{hingston\_masek\_2007} Othello is a deterministic game with a low branching factor, so minimax is a more common choice, but MCTS can still be effective.

Monte Carlo Tree Search (MCTS) is a best-first tree search algorithm based on simulated games as state evaluations. MCTS has four main steps: selection, expansion, simulation, and backpropagation. In the selection step, MCTS starts at the root node, the current game state, and selects the most promising child nodes until it reaches a leaf node that has no recorded playouts. From this leaf node, MCTS adds a child node during the expansion step. MCTS then simulates one random playout from this child node. When the game ends, the playout score (either a win or a loss) is backpropagated up the path to the root node, updating the playout scores of each node along the way.

One of the challenges of MCTS is balancing exploitation and exploration during the selection stage. While it is tempting to only exploit nodes with the best playout scores, it is also important to explore new nodes; it is possible that an unexplored node may have an even higher probability of success than any nodes visited thus far. To balance exploitation with exploration, most MCTS algorithms rely on the Upper Confidence Bound (UCB) formula.



The first term represents the exploitation score while the second term represents the exploration score. MCTS selects the most promising nodes -- those with the highest UCB value. \cite{hingston\_masek\_2007}

RESULTS:

Game engines that we played against. (on highest difficulty possible)

Also, the benchmark that he gives us. -- this will go in the appendix

Because I am testing manually I will run 5 times each. On depth 6, no monte carlo

<https://playpager.com/play-reversi/index.html>

Wzebra: <http://www.radagast.se/othello/>

|  |  |  |  |
| --- | --- | --- | --- |
| Other Othello Game Engines | Our AI wins / losses | Our AI number of black discs each game | Other AI number of white discs |
| PlayPager (hardest of 3 difficulties) | 5/0 | 61, 44, 50, 54, 59 | 3, 20, 13, 10, 5 |
| WZebra (Depth 1) | 4/1 | 37, 26, 37, 37, 37 | 27, 38, 27, 27, 27 |
| WZebra (Depth 2) | 4/1 | 25, 39, 64, 44, 41 | 39, 25, 0, 20, 23 |
| WZebra (Depth 3) | 0/5 | 25, 14, 25, 20, 16 | 39, 50, 39, 44, 48 |
| Reversi Ultimate on iPad  (hardest of 5 difficulties) | 5/0 | 60, 60, 58, 50, 61 | 4, 4, 6, 14, 3 |

**METHODS**

**I) Depth-limited Minimax and Alpha-Beta Pruning**

**A) Depth-limited Minimax**

Othello has a relatively small branching factor but a considerable depth because the game often does not end until the board game is completely full. The complete game tree would therefore be large, so implementing traditional minimax would be impractical from the perspective of both time and memory. Instead, we implemented depth-limited minimax, which is more practical since it generates the game tree only down to a certain depth. While depth-limited minimax is suboptimal, it still works well given a decent heuristic function. At every turn in the game, we created a depth-limited minimax tree (initially with a depth of 3 and later increased to a depth of 6.)

We also considered negamax, which is similar to depth-limited minimax. Instead of having separate minimum and maximum values, the minimum value is the negation of the maximum value. Although the code is simpler, negamax is essentially the same algorithm as minimax. We decided to use our original depth-limited minimax implementation.

**B) Alpha-Beta Pruning**

Although the depth-limited minimax tree is much smaller than the complete minimax game tree, the search space is still large. Alpha-beta pruning can further reduce the search space while still returning the same minimax values. \cite{AI\_textbook} So, we decided to implement alpha-beta pruning using depth-first-search. With our depth-limited minimax tree of depth 6, we did not observe noticeable delay when the algorithm was determining the next move. But, if we were to significantly increase the depth of the search tree, then iterative-deepening search would become the better option over depth-first search.

A common improvement to alpha-beta pruning is ordering the subtrees to maximize pruning. This is a useful technique when generating the complete or partial minimax tree first, evaluating the leaf nodes, and then applying alpha-beta pruning. \cite{AI\_textbook} To save memory, we decided to generate the partial minimax game tree and prune branches along the way. Since we do not evaluate nodes until we reach them, ordering the subtrees is not relevant in our implementation.

**II) Heuristic calculation**

**A) Heuristic components**

**i) Unweighted disc count**

The disc count is the number of discs that a player has placed on the board where each disc is equally weighted. This heuristic can be misleading because the player who has more discs on the board, especially early on, is not necessarily winning the game. When playing our AI with only the unweighted disc count heuristic against random moves, we found the win rate to be about 75 percent, shown in Table **\ref{table:AI\_vs\_rand}**. Because the win rate exceeds 50 percent, we concluded that the unweighted disc count had a positive effect on the evaluation function. We decided to include this component as part of our cumulative heuristic, but only in the end of the game where having the most pieces becomes important.

To calculate the unweighted disc count score of a particular board position, we counted the total number of black discs and the total number of white discs. We then took the difference between the sums.

dcount\_score = player\_dcount - AI\_dcount

**ii) Dynamically weighted disc count**

Not all positions on the board are equally valuable. Corner discs are the most desirable (especially in the beginning) because a player can use a corner to increase stability in that area and take control of the game. Similarly, edges are also valuable for this reason. However, a player should avoid placing a disc immediately adjacent to corners because doing so enables the opponent to take the corner. It is also important to note the significance of positions C3, C6, F3, and F6. These positions should be weighted more heavily than adjacent positions because they are stepping stones to acquiring edges and corners. \cite{UW\_heuristics}

Using this knowledge, we constructed an approximate board position weights matrix **[Tables \ref{table:levels0-20}, \ref{table:levels20-40}, \ref{table:levels40\_plus}]**. We played our AI with only the weighted disc count heuristic against random moves to further refine the board position weights. When the AI win rate consistently increased, we concluded that we must have chosen more accurate board position weights. The highest win rate we achieved was approximately 96 percent [Table **\ref{table:AI\_vs\_rand}]**. Since this win rate is significantly higher than the win rate of the unweighted disc count, we concluded that the weighted disc count is much more important to the heuristic function than the unweighted disc count.

We also noticed that certain board positions such as corners and edges become less valuable as the game progresses. To account for this, we added dynamic weighting. When playing our AI with only the dynamically weighted disc count heuristic against random moves, we found the win rate to be approximately 97 percent [Table **\ref{table:AI\_vs\_rand}].** This was a slight improvement over the static unweighted disc count, so we concluded that dynamically weighted disc count improved the heuristic function.

To calculate the weighted disc count score, we multiplied each black disc by the its respective board position weight and summed the scores. We did the same for the white discs and then took the difference between the two sums:

wdcount\_score = player\_wdscore - AI\_wdscore

**Board Position weights:**

Levels 0-20:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E | F | G | H |
| 1 | 150 | 0 | 12 | 3 | 3 | 12 | 0 | 150 |
| 2 | 0 | 0 | 0.5 | 0.5 | 0.5 | 0.5 | 0 | 0 |
| 3 | 12 | 0.5 | 6 | 1.8 | 1.8 | 6 | 0.5 | 12 |
| 4 | 3 | 0.5 | 1.8 | 1.8 | 1.8 | 1.8 | 0.5 | 3 |
| 5 | 3 | 0.5 | 1.8 | 1.8 | 1.8 | 1.8 | 0.5 | 3 |
| 6 | 12 | 0.5 | 6 | 1.8 | 1.8 | 6 | 0.5 | 12 |
| 7 | 0 | 0 | 0.5 | 0.5 | 0.5 | 0.5 | 0 | 0 |
| 8 | 150 | 0 | 12 | 3 | 3 | 12 | 0 | 150 |

Levels 20-40:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E | F | G | H |
| 1 | 75 | 0 | 6 | 1.5 | 1.5 | 6 | 0 | 75 |
| 2 | 0 | 0 | 0.25 | 0.25 | 0.25 | 0.25 | 0 | 0 |
| 3 | 6 | 0.25 | 3 | 0.9 | 0.9 | 3 | 0.25 | 6 |
| 4 | 1.5 | 0.25 | 0.9 | 0.9 | 0.9 | 0.9 | 0.25 | 1.5 |
| 5 | 1.5 | 0.25 | 0.9 | 0.9 | 0.9 | 0.9 | 0.25 | 1.5 |
| 6 | 6 | 0.25 | 3 | 0.9 | 0.9 | 3 | 0.25 | 6 |
| 7 | 0 | 0 | 0.25 | 0.25 | 0.25 | 0.25 | 0 | 0 |
| 8 | 75 | 0 | 6 | 1.5 | 1.5 | 6 | 0 | 75 |

Levels 40+:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E | F | G | H |
| 1 | 30 | 0 | 3 | 1 | 1 | 3 | 0 | 30 |
| 2 | 0 | 0 | 0.15 | 0.15 | 0.15 | 0.15 | 0 | 0 |
| 3 | 3 | 0.15 | 1.5 | 0.4 | 0.4 | 1.5 | 0.15 | 3 |
| 4 | 1 | 0.15 | 0.4 | 0.4 | 0.4 | 0.4 | 0.15 | 1 |
| 5 | 1 | 0.15 | 0.4 | 0.4 | 0.4 | 0.4 | 0.15 | 1 |
| 6 | 3 | 0.15 | 1.5 | 0.4 | 0.4 | 1.5 | 0.15 | 3 |
| 7 | 0 | 0 | 0.15 | 0.15 | 0.15 | 0.15 | 0 | 0 |
| 8 | 30 | 0 | 3 | 1 | 1 | 3 | 0 | 30 |

**iii) Mobility**

The mobility score counts the immediate number of possible moves available from a particular board position. The player with the greater mobility has more options than the other player and is less likely to be forced into a bad position. \cite{WA\_monte\_othello} When first considering mobility, we underestimated its importance. When playing our AI with only the mobility heuristic against random moves, we achieved a win rate of approximately 80 percent [Table **\ref{table:AI\_vs\_rand}].** We therefore concluded that mobility is important to the evaluation function, but not as important as weighted disc count, which had a much higher win rate of approximately 96 percent.

We also considered potential mobility, a measure of future possible moves. Potential mobility is much harder to quantify. To calculate it, we counted the number of empty spaces around opponent's pieces. When playing our AI with only the potential mobility heuristic against random moves, we found the win rate to be approximately 67 percent [Table **\ref{table:AI\_vs\_rand}].** We concluded that potential mobility has a slight positive effect on the evaluation function, but we decided that the effect was not significant enough to include in our final cumulative heuristic.

To calculate the mobility score of a particular board position, we counted all of the possible moves a player could make from that position. We did the same for the other player and then took the difference between the two counts.

mobility\_score = player\_mobility - AI\_mobility

**iv) Stability**

The stability score measures how likely a disc is to being reversed by the other player. A piece on the board is stable if it is impossible for the other player to reverse it. We assigned stable board positions positive weights. Conversely, an unstable disc could be reversed by the other player’s next move. We assigned these board positions negative weights. Semi-stable discs could potentially be reversed at some point in the future, but they cannot be reversed immediately. We assigned these board positions a weight of zero. \cite{UW\_heuristics}

Corners are perfectly stable pieces, so we assigned corners the highest stability weights. The positions adjacent to corners are highly unstable because they can easily be reversed. Edges are somewhat stable, but it is still possible (although difficult) to reverse them. In contrast, the positions on the interior adjacent to the edges are highly unstable. \cite{UW\_heuristics}

Using this reasoning, we constructed an approximate stability matrix **shown in Table \ref{table:stability\_weights}.** We played our AI with only the stability heuristic against random moves to further refine the stability weights. When the AI win rate increased, we concluded that we must have chosen more accurate stability weights. From Table **\ref{table:AI\_vs\_rand}, we can see that** the AI win rate was approximately 77 percent. We concluded that stability carries about the same importance as mobility.

To calculate the stability of a particular board position,we multiplied each black disc by the stability weight of its respective position and summed the scores. We did the same for the white discs and then took the difference between the two sums:

stability\_score = player\_stability - AI\_stability

**Stability Weights:**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E | F | G | H |
| 1 | 6 | -2 | 2 | 2 | 2 | 2 | -2 | 6 |
| 2 | -2 | -4 | 0 | 0 | 0 | 0 | -4 | -2 |
| 3 | 2 | 0 | 2 | 0 | 0 | 2 | 0 | 2 |
| 4 | 2 | 0 | 0 | 1 | 1 | 0 | 0 | 2 |
| 5 | 2 | 0 | 0 | 1 | 1 | 0 | 0 | 2 |
| 6 | 2 | 0 | 2 | 0 | 0 | 2 | 0 | 2 |
| 7 | -2 | -4 | 0 | 0 | 0 | 0 | -4 | -2 |
| 8 | 6 | -2 | 2 | 2 | 2 | 2 | -2 | 6 |

**v) Reinforcement Learning**

**Reinforcement learning is useful when running many games and saving the game tree between games. Reinforcement learning is used by giving a reward to the AI if it wins the game, and subtracting a reward to the AI if it loses the game. We do this by starting at the end node after the game finishes, and traverse up the tree. While at the current node, we give it a reward unit, like a value of 1. We then go to its parent, and give it a fraction of the reward unit, like a value of 0.5. We then go to its parent and give it a fraction of the fraction of the reward unit, like a value of 0.25. And we continue this until we reach the root node. If we were to run this thousands or millions of times, our gametree would have highly rewarded decision lines that lead more often to wins. This factors into the heuristic, leading the AI to choose lines that have proven to be successful, over lines that just look favorable to the heuristic.**

When playing our AI with only the reinforcement heuristic against random moves, we found the win rate to be approximately 39 percent. With this rate below 50 percent, it appears that reinforcement learning actually worsened the performance. However, reinforcement learning needs many games to learn from, so we concluded that it could be useful when storing the search tree between games and playing more often.

**(INSERT JAMES’ TABLE) - i moved it to top of heuristics section in latex document**

Table \ref{table:AI\_vs\_rand}: We tested each component of our heuristic function separately to evaluate its usefulness. We played our AI against random moves so that we could run many trials to make statistically sound conclusions. Each combination of heuristics was tested in 100 runs using our minimax method. Player 1 and Player 2 are referred to as P1 and P2.

Heuristic components:

\begin{itemize}

\item 0) Random

\item 1) Unweighted disc count

\item 2) Weighted disc count

\item 3) Mobility

\item 4) Potential Mobility

\item 5) Stability

\item 6) Reinforcement learning

\item 7) MCTS

\item \*= Dynamic

\end{itemize}

**B) Cumulative Heuristic with Dynamic Weighting**

Our cumulative heuristic accounts for all of the components discussed: disc count, weighted disc count, mobility, stability, and reinforcement learning.

heuristic = (player\_dcount - AI\_dcount) + (player\_wdscore - AI\_wdscore) + (player\_mobility - AI\_mobility) + (player\_stability - AI\_stability) + (good - bad)

After evaluating each heuristic individually by playing against random moves, we had a general understanding of which components of the heuristic were more important than others. We knew, for instance, that weighted disc count had the greatest effect on the win rate, so we needed to weight it heavily. At first, we used static weights, but we realized that the importance of certain heuristic components changes as the game progresses, so we introduced dynamic weighting. To determine the dynamic weights of each heuristic component we made estimates based upon known properties of the game:

The weighted disc count is more important in the beginning of the game than at the end of the game. The opposite is true for the unweighted disc count. In the beginning of the game, a player wants to capture the more heavily weighted discs (corners and edges) to take control of the board. In contrast, towards the end of the game, the players have already placed discs in most of the important positions. At this point, having more discs than the other player (ignoring disc weights) will determine who wins the game. \cite{UW\_heuristics}

Mobility is essential at all times in Othello. Therefore, we did not adjust its weight as the game progresses. In general, the player with the greater mobility is much more likely to win than the other less mobile player. In fact, towards the end of the game, the less mobile player may have zero possible moves. In this case, the more mobile player will be able to move for “free” and easily win the game. \cite{UW\_heuristics}

Stability is important especially in the beginning of the game. By placing discs in positions that are difficult to reverse, a player can quickly take control of a region of the board. However, as the game progresses, stability lessens in importance. The board itself naturally stabilizes since players can no longer access and reverse certain areas of the board. \cite{UW\_heuristics}

Through trial and error, we arrived at our final dynamic weights of each heuristic component, which is shown in Table **\ref{dynamic\_weights}.** Because the weights are interdependent, we could not test each dynamic weight individually. Instead, we adjusted the dynamic weights and then used the cumulative heuristic to play against random moves and assessed whether the win rate significantly improved.

At first, we were surprised that the AI using the static heuristic weights had a 100 percent win rate against random moves while the AI using the dynamic weights against random moves had a slightly lower win rate of approximately 96 percent. When we tested the AI with dynamic weights against the AI with static weights, the AI using dynamic weights had a win rate of approximately 83%. We therefore concluded that using a dynamically weighted heuristic was an improvement over static weights. **[Table**  **\ref{table:AI\_vs\_rand}]**

In our table of dynamic heuristic weights, mobility appears to be weighted the highest. However, it is important to note that the weighted disc count has its own matrix of board weights where corners, for instance, are weighted very heavily. Thus, even though mobility is weighted more heavily than weighted disc count in the cumulative heuristic, the weighted disc score by itself tends to be much greater than the mobility score. This anomaly arose because we used numerical scores rather than percentages to weight our heuristic.

**Reinforcement**

**Dynamic heuristic weights**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Levels 0-40 | Levels 40-50 | Levels 50-55 | Levels 55-60 |
| Unweighted disc count | 0 | 0 | 1 | 3 |
| Weighted disc count | 3 | 2 | 1 | 1 |
| Mobility | 10 | 10 | 10 | 10 |
| Stability | 5 | 1 | 1 | 0 |
| Reinforcement | 1 | 1 | 1 | 0 |

**C) Alternative Heuristic: Monte Carlo Tree Search**

Othello is a deterministic game with a relatively small branching factor, so devising a decent heuristic evaluation function is quite feasible. Thus, we dedicated most of our time to perfecting our heuristic, but we still considered Monte Carlo Tree Search (MCTS).

MCTS is a more common approach when playing non-deterministic games and/or games with high branching factors where deriving an accurate heuristic function is difficult. However, MCTS can be used for Othello. \cite{hingston\_masek\_2007} So, we also attempted to implement it.

When playing our AI using MCTS against random moves, we found the win rate to be 100 percent. **[Table**  **\ref{table:AI\_vs\_rand}]** Although MCTS achieved a perfect win rate, so did our cumulative dynamically weighted heuristic using a minimax tree. We concluded that our cumulative dynamically weighted heuristic was slightly superior to MCTS because its average number of pieces was slightly higher. However, the main reason we chose minimax over MCTS is because the run-time of MCTS was much too slow to be practical.

We also attempted to combine MCTS with our existing depth-limited minimax algorithm. We hypothesized that adding the playout scores from MCTS to our existing heuristic function would further improve the performance of our AI. However, we soon realized that running minimax and MCTS simultaneously was too computationally expensive. We noticed a significant delay when our program was determining which move to choose next. We were therefore unable to test whether a MCTS-Minimax hybrid would be an improvement over Minimax or MCTS alone.

Due to time constraints, we ultimately decided to omit MCTS from our code. However, we discuss the hybrid model more thoroughly in the discussion.

**III) Memory Usage**

We initially generated a partial depth-limited minimax tree up to depth 3. After making the following adjustments, we were able to increase the depth level to 6, which allowed us to expand 1000x more nodes:

1. At first, we stored a Board object in each node in the search tree. To save memory, we decided to store a board as 2D pointer array instead. This did not make a significant difference in memory usage.
2. Next, we decided to store the board as a 2D char array instead of a pointer array. This made a huge difference in memory usage. Each index of the array is one byte instead of eight bytes needed for a pointer. We therefore downsized our 8x8 Othello board from 512 bytes to 64 bytes. Each node of the decision tree has its own board, so this results in many GB of saved space when expanding millions of nodes throughout a game.
3. Finally, we deallocated memory where possible and fixed all memory leaks.

In order to achieve a depth 6 without using any significant amount of RAM, we had to clear the part of the tree that wasn’t used. During the game, once the AI makes a move, the current node is now the child node with the best heuristic. So to reduce amount of memory, we delete the other child nodes that we considered but didn't move to. That greatly reduces amount of RAM to basically 0.5GB. That way depth 6 doesn't use up ~8GB RAM like before, with no extra time added to the moves. The drawback of this, is that we can’t use reinforcement learning since it relies on a propagating tree during the game and between games.

**DISCUSSION**

In deterministic games such as Othello, it is relatively easy to calculate a decent heuristic for minimax using well-known strategies. However, perfecting an evaluation function is much more challenging. Othello is a well-studied game with many viable strategies, so we first needed to decide what strategies to account for in our heuristic. We considered unweighted disc count, weighted disc count, mobility, and stability, and reinforcement learning.

Determining the weights of each component of the heuristic was not a straightforward process. We realized, for instance, that unweighted disc count should only factor into the heuristic calculation towards the end of the game. But, we were unsure of the exact point in the game where unweighted disc count would become useful to the heuristic calculation. Furthermore, we needed to determine how important the unweighted disc count was relative to other components such as mobility. We largely used trial and error to determine weights. To evaluate our heuristic, we played our AI against itself by testing different components of the heuristic separately. **[Table**  **\ref{table:AI\_vs\_rand}]** While this gave us a general idea of the importance of each heuristic component to the overall performance, it was still hard to make precise conclusions about weights

In addition to weighting each component of the evaluation function, we also weighted individual board positions. While it is obvious that corner discs, for instance, are extremely valuable, determining the exact weight to assign to corner pieces is much less intuitive. When we under-weighted the corner positions, we observed that the other player was more likely to place a disc in the corner first. However, when we over-weighted the corner positions, this weight drowned out other important components of the heuristics.

In addition to the challenge of defining a precisely accurate heuristic, we encountered another drawback of depth-limited minimax: its suboptimality. A complete minimax game tree guarantees optimal move selection, but constructing the entire tree is impractical. With a depth-limited tree, minimax selects the best possible moves given a partial tree, but without complete knowledge of the tree, these moves are suboptimal. \cite{AI\_textbook} Yet, with a carefully crafted evaluation function, minimax is still a powerful algorithm. With increased depth levels, minimax has greater foresight and chooses better moves. We increased our depth level to 6, but we could not increase it further due to limited RAM on our personal computers.

Despite the weaknesses of minimax in this setting, we still achieved decent performance against several game engines. We also considered MCTS because of its different strengths. MCTS does not require an explicit heuristic evaluation function like minimax does. MCTS relies upon thousands of simulations and records playout scores, which represent the probability of winning from a particular node. \cite{WA\_monte\_othello} Because of this, MCTS appeared to offer greater precision than our attempt of trial-and-error when refining the heuristic for minimax. However, while MCTS is capable of converging to a near-perfect heuristic for some games, it does not converge for Othello. It is not clearly understood why this is the case. One theory is that MCTS probabilities can be misleading in Othello because they do not take into account mobility, a critical aspect of the game. \cite{hingston\_masek\_2007}

Due to the incredibly slow run-time of minimax together with MCTS, we did not investigate whether MCTS improved the move selections of minimax. We suspect that MCTS may actually misinform the minimax heuristic by providing potentially misleading probabilities.

When comparing the performances of minimax to MCTS for Othello, minimax is the better algorithm. \cite{WA\_monte\_othello}. Therefore, it actually seems counterproductive to add properties of the inferior algorithm, MCTS, to the superior minimax heuristic. On the other hand, it is reasonable to infer that adding minimax properties to MCTS could improve the performance of pure MCTS. In fact, there are hybrid minimax-MCTS algorithms that perform better than pure MCTS, but only in some games such as Catch the Lion and Breakthrough. Hybrid minimax-MCTS poses no significant advantage over pure MCTS for Othello. \cite{ieee\_hybrids} Unlike games such as chess, Othello does not have shallow traps until the end of the game. (A move is considered a shallow trap if it guarantees that the other player will win.) \cite{Ramanujan\_2010} A poor choice of move can quickly lead to the end of the game in chess, but Othello does not end until the game board is full or neither player can move.

Even if we were to dramatically increase the number of simulations for MCTS, the playout scores would still not account for critical aspects of the game such as mobility. Therefore, given more time, instead of improving MCTS, we would further refine our minimax heuristic function. With a more precisely weighted heuristic, a greater minimax tree depth, the inclusion of an end-game table, and other improvements, we could make more informed move selections.

**Also say something about reinforcement learning.**

**--------------------------------------------------------------------------------**

Realistically, constructing an entire game tree is not feasible. With only a partial game tree, the move selections are not guaranteed to be optimal.

Appendix

Playing for the benchmark:

go to this website

https://othello-reversi.com/

and have your program play against this program.

There are 3 difficulty levels (1, 2, 3 stars). Play in "1 player" mode;

the "1 player" is your program.

Play 20 games, starting at the easiest level. For each game record/store

the log of the play (your moves and opponents moves) and the runtime.

If you can win most games at some level, go to next level and repeat.

|  |  |  |  |
| --- | --- | --- | --- |
| Mode | Win Rate | Our AI Wins/Losses | Number of black disks/white disks |
| Easy | 100% | 1/0 | 56/8, 57/4, |
| Medium | 100% | 1/0 | 60/4 |
| Hard | 100% | 10/0 | 49/15,  59/5,  54/10,  55/9,  52/12,  61/3,  54/10,  50/14,  54/10,  59/5 |