Alpha-Beta Pruning with Iterative Deepening, a Transposition Table and Heuristics for Reversi Othello

Isabelle Champion and Cienna Gin-Naccarato

December 5, 2024

1 Introduction

This paper provides a detailed explanation and analysis of the development of an AI agent designed to play the game Reversi Othello. We begin by outlining the design process of our agent, with emphasis on the benefits of implementing a combination of the Alpha-Beta pruning algorithm with iterative deepening, move ordering and a transposition table. A quantitative analysis follows, along with a discussion of various advantages and disadvantages of the method chosen. We examine how factors such as the depth and breadth of the search tree, board size and heuristic functions impact our agent's performance. Finally, we discuss improvements we could have made to our design, centered around improving memory efficiency.

2 Strongest Algorithm Found For Reversi Othello

This project used alpha-beta pruning with iterative deepening and move ordering, in combination with a transposition table and a heuristic function composed of multiple factors. The strongest algorithm we found to play Reversi was alpha-beta-pruning with iterative deepening and a transposition table. We followed an iterative design process to construct this agent, methodically adding and eliminating modifications stage by stage. We read a number of articles and online sources to determine which methods would be best to try. For each method, we created a new, identical agent to our best design so far, added the method to it, and then compared performances between the new and original agent on the different board sizes. If the addition improved the performance of the original design, we added it in. We found that alpha-beta pruning performed better than Monte-Carlo tree search, and that adding iterative deepening to the alpha-beta pruning improved performance considerably, and that it worked well considering the time constraint. We also added a transposition table to keep track of previous best moves found for board configurations, in order to improve the efficiency of the iterative-deepening search. Move ordering based on static weights of pieces helped improve the efficiency of the alpha-beta pruning even more, and led to better outcomes. We constructed a heuristic function, based on the following factors: coin parity, mobility, stability and corners. The calculations of these heuristic values were based on mathematical principles, and the weights were determined from a series of experiments with different weight combinations, and our own personal experience from playing Reversi Othello. We believe we achieved a play quality of 100% against both the random agent and the gpt-greedy-corners agents.

.

3 Agent Design

3.1 Alpha-Beta Pruning

We started our design process with research online, reading a number of sources, including academic papers, to determine which search methods would work best for Reversi Othello. The vast majority of these sources pointed to either alpha-beta pruning or Monte-Carlo Tree Search. Taking this into account, we constructed two agents, one to perform alpha-beta pruning and another to perform Monte-Carlo Tree Search, both equipped with the same rudimentary heuristic (a check for number of player vs opponent pieces and a count for the number of corners obtained, with a slightly higher weighting). After a series of trials on the different board sizes, we found that the alpha-beta pruning was the most effective, so discarded the Monte-Carlo tree search method. Alpha-beta pruning relies heavily on move-ordering for success, and can perform worse when the breadth of the search tree is large, in contrast with Monte-Carlo tree search, which is usually better suited to search trees with a greater breadth. In Othello however, the search tree isn't that broad [SA], especially in the smaller-board games where the number of possible moves at each stage is relatively small. This could have led to the success of the pruning method over the Monte-Carlo tree search.

3.2 Iterative-Deepening Search

The next approach we wanted to try was iterative deepening search (IDS). We implemented this and, in keeping with the iterative design process, tested this modified agent against the original alpha-beta pruning agent with the same rudimentary heuristic. We found that the IDS agent outperformed the original agent. IDS is an any-time function, so it can be stopped at any point and will return the best solution found so far. This worked well for the project since the agent had a time constraint. The IDS used the time a lot more efficiently, as it was able to search up to the time barrier and then return a solution. The alpha-beta pruning we implemented stopped searching at a predetermined depth, regardless of time remaining or time passed, therefore not using the full decision time optimally. In order for it to not go over the time limits for more complex move calculations, this depth was relatively small, so the pruning wasn't able to obtain as much information in the same time-frame. The iterative deepening search was given a limit of 1.9 seconds, giving the algorithm plenty of time to do some final evaluations before returning a result.

3.3 Move Ordering

To speed up the search algorithm further, we implemented move ordering based on static weights matrices. For each board size, we created a matrix with a weighting associated with each possible position. In Othello, certain pieces are more valuable to obtain, since they are at less (or no) risk of being flanked by an opponent's disk. In particular, corners are the most valuable pieces on the board, as they can never be flipped. Edges (except for the squares immediately adjacent to corners) are also valuable pieces, as there is less opportunity to flip them as well; an edge piece can only be flipped from a single direction, when the opponent's pieces are below and above the edge piece, in contrast to pieces in the center of the board that can be flipped from four possible directions. The squares immediately adjacent to corners are the least valuable pieces to obtain as they give the opponent an easy opportunity to take a corner. The same goes for pieces immediately adjacent to edges (although these aren't quite as bad). The static weights matrices were composed with these premises in mind. At each successive call to the function to obtain the possible moves available to a player, these moves were then sorted in order of desirability based on the appropriate weights matrix. This significantly improved the performance of the agent. The move-ordering prioritized stronger moves, correlated with higher scores (based on the heuristic function to be discussed further on). Considering the strongest moves first is correlated with a greater pruning ratio, which improves efficiency and the extent to which the algorithm can search the tree in the allotted time.

3.4 Transposition Table

The final component added to the search algorithm was a transposition table. Numerous online sources [H.S20] [DPV06]suggested that this would drastically improve the efficiency of the iterative-deepening search. Here, we were faced with a trade-off between space-time complexity. Although the transposition table could improve the speed of iterative deepening, there is a memory cost of storing entries in a table and associated information. Additionally, we were unsure if the process of checking the table for entries, potentially linear in the number of entries, would add additional time that would take away the ability of the agent to search deeper in the tree. Taking these factors into account, we considered a number of approaches.

The first idea was to store a dictionary of board states and the associated 'best moves' with their scores. We quickly realized that storing an entire board would consume too much memory. We therefore opted for a different approach, to use hash values for the board states. From online research, we came across the concept of Zobrist hashing, which uses bit operations to speed up the process and perform calculations very quickly. After several attempts at implementing this method, we were unsure if it was totally error-free, so settled on using python's built-in hash function. We implemented our transposition table by storing the hash codes related to board states, along with other useful information, in a dictionary. We found various academic sources [Sel08] [Fis81] [Bre98] on how to construct a transposition table for a Negamax algorithm (closely related to the Minimax algorithm), along with a helpful YouTube video [H.S20] that explained the creation of this table in closer detail.

In our alpha-beta pruning method, a tuple (value, move) is returned by the appropriate player representing the best move they have found with respect to the alpha-beta pruning process. Before this return, we store the move and value in the transposition table dictionary at the appropriate hash value. We also have three flags: 'upper', 'lower', and 'exact'. The 'upper' flag signifies an upper-bound on the optimal value and move, and is associated with a value found greater than the current beta value. The 'lower' flag signifies a lower-bound on the optimal value and move, and is associated with a value that is lower than the current alpha. The 'exact' flag represents a move that has a value between the current alpha and beta value. Another flag 'depth' specifies the depth of the current search. [DPV06]

At the beginning of the alpha-beta function, the hash value of the current board state is first searched for in the transposition table. If an entry exists, the associated flag determines what the algorithm should do next. If the flag is 'upper', this means that this is an upper-bound on the optimal value. This is compared to the current beta value, and the minimum of the two is set as the new beta. Similarly, if the flag is 'lower', alpha is set to the maximum between the current alpha and 'lower'. This helps to reduce the bounds of the search and speed up the pruning process. If the flag is 'exact', the move and value are returned.

There is another factor considered, however, before this process is carried out. If the depth of the entry is greater than or equal to the current depth of the search, then the above process goes ahead and alters parameters alpha and beta, potentially returning a move and value, accordingly. This is because this entry represents a move evaluated at a more (or equally) informed point in the search. On the other hand, if the depth of the entry is less than the current depth, the above procedure is not carried out, and the algorithm continues with the regular pruning steps. In this case, the table entry represents an evaluation from a less-informed point of the search, so will provide less useful and accurate information. In keeping with our iterative design process, after implementing this table, we compared its performance to an agent without the table but that was otherwise identical, over a series of trials. For the majority of trials, the agent with the table won, so we decided to move forward with this method.

3.5 Heuristic Functions

Finally, we worked on perfecting our heuristic function. Numerous online sources pointed to the same group of techniques to try: coin parity, mobility, corners and stability. We designed the heuristic functions so that each would output a value on a scale of -100 to 100, with negative values corresponding to an opponent's advantage, and positive values to our player's advantage. A score of 0 represents an equal advantage to both. This weighting method was based on analyses from the paper An Analysis of Heuristics in Othello, from the University of Washington [SA]. This method is useful in that it generates a common scale from which the heuristic values can be easily compared.

We then conducted a series of trials to determine the relative scaling factors that should be associated with each heuristic. Each trial consisted of four sub-trials, 100 games for each valid board size between 6 to 12. The first heuristic we implemented was the simplest, coin parity, a measure of the number of player pieces vs the number of opponent pieces on the board. The next was mobility, which compares the number of possible player and opponent moves. The aim of the player is to maximize their mobility while reducing the mobility of their opponent. The corners heuristic measures the number of player vs opponent corners captured. Finally, stability measures the likelihood that a piece will be flanked [SA]. A stable piece cannot be flanked from any direction. We used a recursive method to evaluate the stability of each piece from the stability of its immediate neighbors [SA], and evaluated player vs. opponent stability. The final heuristic chosen was a weighted combination of mobility and corners based on their individual impacts, as will be discussed further on [Fen17].

4 A Quantitative Analysis of Our Agent

4.1 Depth Analysis

An analysis of our agent's quantitative performance follows. The first factor we consider is depth. Since we use iterative deepening, the depth our agent obtains is relative to however far it can search given the two second time limit.

In terms of a formula to represent this depth, let IDS(t) be a relation such that IDS(t) be the maximum depth that the iterative deepening search obtained in time t. Then, the maximum depth we obtain is IDS(1.9). On average, however, the agent obtains a depth of four or five on the smaller game boards, and three or four on the larger boards before returning. The depth is the same for all branches, except for the branches that are pruned. The construction of the search algorithm contributes to the depth of the search; the extent to which the iterative deepening can search determines the deepest depth discovered, and the pruning cuts off certain branches. This pruning is affected by move ordering (detailed above), so the move ordering also indirectly affects the search depth.

4.2 Breadth Analysis

The breadth achieved during the search depends on whether or not the neighboring branches have been pruned; if none have been pruned, then the full breadth of the tree is explored. A formula to represent this can be constructed as follows: let X_i be the number of nodes at level i, and Prune(X_i) be a function that outputs the number of these nodes that the alpha-beta algorithm prunes. Then, the breadth of level i achieved is equal to $|X_i|$ - Prune(X_i), where $|X_i|$ is the length of set X_i. Every move is considered at each round of the play, and this is the same for both maximum and minimum players- we wanted to ensure all options were considered and that the opponent is of equal strength. We used static weights based on move-ordering (detailed above) to preemptively select better moves, and lead to a greater degree of pruning.

4.3 Board Size Analysis

The board size influences the depth and breadth of the search. In games with a larger board size (10,12), the search tree has a greater breadth. This is because there are more moves to consider at each stage, especially in the middle of the game. This means the pruning cannot search to as deep a level. In games played on the larger board sizes, we found that our pruning only achieved an average search depth of 3 mid-game, whereas in smaller games, it achieved a depth of 4 or 5 mid-game. We altered the weights of the heuristic based on the size of the board, and had corners more heavily weighted in the larger board games due to the results of many experimental trials.

4.4 Heuristics Analysis

The heuristics we tried implementing were corners, mobility, stability and coin parity. We also tried a static board value-based move ordering technique and implemented pruning. We measured the impact of each in a series of trials, where we constructed an agent for each heuristic function (which differed only in the heuristic weighting in the evaluation function) and played a series of games between each agent pair at each fixed board size. The heuristic with the most impact was corners, and the heuristics with the least impact were coin parity and stability. In Othello, winning the game is strongly correlated with obtaining the corner pieces, since these cannot be flipped. Coin parity also changes drastically from turn to turn, therefore, high coin parity at one point in the game does not necessarily correspond to an advantage for any player at a given time. What matters more is where on the board your pieces are and how safe they are from being flipped. We found mobility had more impact than stability, perhaps due to its calculation being less time and memory-complex. This may have enabled IDS to reach greater depths. An agent constructed with only corners and mobility consistently performed better than an agent with corners mobility and stability, where the corner weighting was

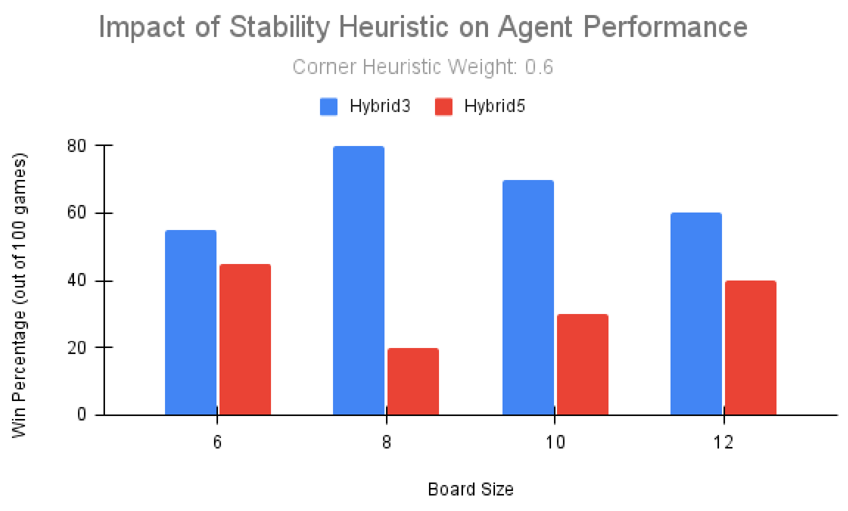

Figure 1: The weightings for the heuristics of the two agents compared are as follows: Hybrid3 = 0.6*corners + 0.4*mobility; Hybrid4= 0.6*corners + 0.2*mobility + 0.2*stability.

the same for both agents as shown in Figure 1 below. Our final agent uses the corners and mobility heuristic. The pruning method was helpful in speeding up the Minimax search by eliminating useless branches corresponding to moves that would never be selected. The move ordering was also effective as considering better moves first is a method that optimizes pruning.

We predict a 100% win rate against the random agent, a 70% win rate against Dave, and a 50% win rate against our classmates.

5 Advantages and Disadvantages of Our Agent

A main advantage of our approach is the use of iterative deepening, which maximizes the depth of the search given the allotted time. Another advantage is our use of pruning, and how the static-weights-based move-ordering enables a greater degree of pruning. The move-ordering ensures that, if a corner capture is a possibility, this will be considered first, and (non-corner-adjacent) edge moves second; these moves are the most valuable logistically. Another strength is our transposition table, which caches previously seen board states and can immediately return either a move, eradicating the need for a further search down that path, or a narrower window for the alpha and beta values, leading to a greater chance of pruning occurring. Our method for constructing our agent is also strong; we opted for an iterative design process that slowly built on previous versions, ensuring that each new modification was contributing positively to the performance of our agent. We also made sure to run multiple tests before integrating new features and considered a broad range of features. The research we did prior to starting to build our agent also played to our advantage, as the results from experiments conducted by others [SA] [Gal24] helped us choose which methods to prioritize and test, helping us to narrow our focus. A disadvantage of our approach is the memory usage of the transposition table. Our overall memory usage comes relatively close to the 500MB maximum with games on the 12x12 board. Another disadvantage is the time complexity of the heuristic functions. Mobility, coin parity and stability all iterate over each coin on the entire board, which is quadratic in the board length. This is inefficient and takes time from reaching deeper levels in the search tree. These heuristic functions could certainly have been optimized to improve efficiency.

6 Improvements

Our heuristic functions could have been improved with a bit-board approach. The complexity of the calculations mentioned above would have been significantly reduced with this method. Instead

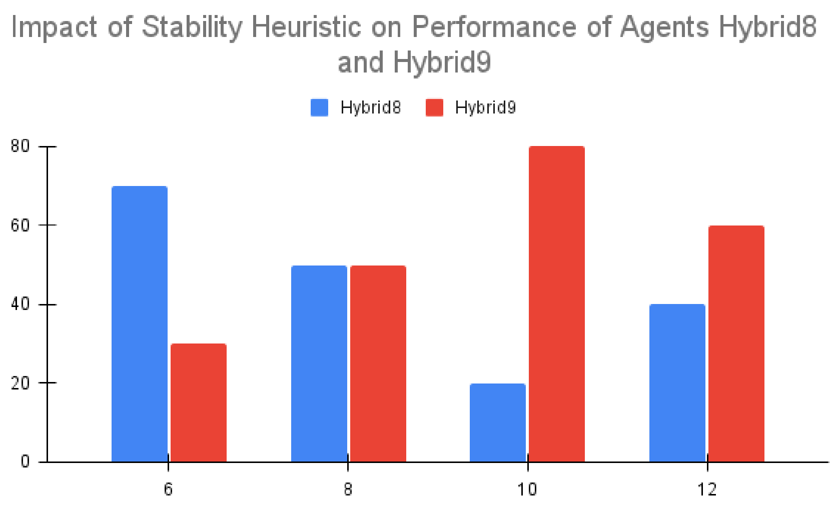

Figure 2: The heuristic for Hybrid8 is 0.7*corners + 0.3*mobility, while the heuristic for Hybrid9 is 0.8*corners+0.2*mobility. Here, Hybrid8 prevails on board size 6 before tying on board size 8 and losing on the larger boards.

of iterating through the chess_board object, a two-dimensional numpy array, instantaneous bitwise operations could have been performed on a bit string representation of the board.

Another way to improve the agent would have been to implement a more effective stability heuristic. The stability heuristic that was implemented calculated the number of stable pieces a player had on the board compared to the opponent. Improving this heuristic might entail applying similar logic to determine if a piece is unstable (able to be immediately flanked) or semi-stable (not able to be immediately flanked, but could possibly be in the near future). With this version of the stability heuristic, pieces that do not meet the criteria of being stable in all direction-pairs can still have some impact on the game, instead of being discarded. While the current stability heuristic sums the total amount of stable pieces, the proposed improved heuristic could assign a positive weight to fully stable pieces, a negative weight to unstable pieces, and a neutral weight to semi-stable pieces [SA] [AB20]. This modification might help better predict optimal moves based on the locations of known unstable and semi-stable pieces.

While we adjusted the heuristic weightings based on the board size, further refining the functions, (for example, including a different set of heuristic functions per board size), could help improve our agent's overall performance. During our trials, we constructed various agent variations with different combinations and weightings of the heuristic functions. We had some conflicting results where, for example, certain agents (over a series of 100 games), would win on, for example boards 8 and 12 and then lose on 6 and 10, not demonstrating a clear advantage of certain methods on smaller or larger board sizes. This trend appeared in a number of trials, making it difficult to draw conclusions about which tactics were better suited to smaller vs larger board sizes. In terms of a future improvement, we would like to have been able to understand the cause of this variation, whether it was due to the agent's design or if it was more random. An example of these unclear results is demonstrated in Figure 2. below. Another improvement could be to run more trials, which would smooth out results, and give us a better view of overall trends. Also, to run trials on an even broader range of heuristic combinations and weightings.

Once the optimal heuristic is determined for each board size, it may be additionally useful to add parity as a strategy to come into effect near the end of the game. This could be implemented by dynamically changing the weight to the parity heuristic with an exponential function as the game progresses. When we tried this strategy on some of our stronger agent cases in the trial period, we found the addition of parity to be generally ineffective when the maximum weight parity could achieve was 0.3. Testing this maximum weight achievable by the parity heuristic is similar to refining the heuristic weightings for each board size and could lead to an even more optimal agent.

In terms of input from outside sources, we used the method of obtaining heuristic values on a scale of -100 to 100, and subtracting the opponent's score for a heuristic from the player's score, as detailed in the paper An Analysis of Heuristics in Othello, [SA]. Additionally, the transposition table's logic is based on the Negamax wikipedia entry [Bre98] [Fis81] [Sel08]. We used ChatGPT to help debug our code, with the prompts 'why would X error be happening in this code', and we found it helpful for finding problems.

References

[AB20] Uzair Qureshi Dhruv Kumar Alec Barber, Warren Pretorius. An analysis of othello ai strategies. Undergraduate Term Paper, Trinity College Dublin, 2019/2020.

[Bre98] Dennis M. Breuker. *Memory versus Search in Games.* PhD thesis, Maastrich University, 1998.

[DPV06] Sanjoy Dasgupta, Christos H. Papadimitriou, and Umesh Vazirani. *Algorithms.* McGraw-Hill, 2006. Dynamic Programming.

[Fen17] Matthew Fennel. Determining stable disks in othello, 2017.

[Fis81] John P. Fishburn. *Analysis of Speedup in Distributed Algorithms.* PhD thesis, Princeton University, 1981.

[Gal24] GalaX1us. Othello-agents, 2024.

[H.S20] H.Sperr. Stackit 9 - transposition table v2 short, 2020.

[SA] V. Sanniddhanam and M. Annamala. An analysis of heuristics in othello. Undergraduate Term Paper, University of Washington.

[Sel08] George T. Heineman; Gary Pollice Stanley Selkow. *Chapter 7: Path Finding in AI.* Orielly Media, 2008.