Minimax With Alpha-Beta Pruning for Othello

Ryan Bergman
Department of Computer Science
University of New Hampshire
Durham, NH 03824
rpb1017@wildcats.unh.edu

Abstract

Turn-based, two-player games such as Othello have seen great popularity in AI research. Simulating every move throughout an entire game to find the best possible move is unreasonable, given Othello has an average branching factor of about 10. The minimax algorithm is one algorithm which can be used to make decisions in these types of games, while also being able to be completed within a reasonable amount of time. Though minimax is effective, there are many improvements that can be made to this algorithm that will improve its time efficiency.

Introduction

Othello is a game which has seen significant competitive success within the past few decades. It is played on an 8x8 board with two-sided black and white discs. Players take turns placing discs down on the board. A piece can be placed in a position if in any direction (diagonal included) there exist one or more of the opponent's pieces, followed by another one of the player's pieces. Once a piece is placed, all pieces between the placed piece and the other piece of the player are flipped, making them now owned by the player. Once no moves are available for either player, the game is over. At the end of the game, the player who has the most discs is the winner.

In competitive Othello, there is typically a time limit imposed on the players to ensure no player significantly prolongs the game. Other than that, the rules are generally the same as casual Othello.

Given that Othello has seen such great competitive success, it is often used as a game to demonstrate the power of artificial intelligence. Two of these programs are Bill (Lee and Mahajan 1986) and Iago (Rosenbloom 1981). Both of these programs use many different strategies to make the best search given the time limit imposed. A lot of balancing has to be done between time allocation and search accuracy. If too much time is allocated for a search, then later searches will not have as much time, or worse, the time limit will be reached. If not enough time is allocated for a search, the search will not be very good, and the chances of winning the game are decreased.

To make a successful Othello program that can play competitively, both time and success must be taken into account. In other words, how can an Othello program be made to be able to play competitively, both optimizing for time as well as success?

Approach

Minimax

The minimax algorithm is an algorithm that can be used to find the best move for a player in a two-player turn-based game. It looks ahead multiple moves in the game, for both the player and the opponent, to better predict how effective moves will be long-term.

The algorithm makes two assumptions: 1) the player will always make the move that benefits them the most, and 2) the opponent will always make the move that will negatively affect the player the most. These assumptions prove to be rather effective, since the result gives the player the best move they can force against their opponent. Thus, the player is always making the best move available to them, given the information they currently know.

All nodes in the search tree represent a state of the game board. The children of any given node are the resulting game boards from a move of the player whose turn it is.

The algorithm is typically implemented with the concept of *min nodes* and *max nodes* (hence the name *minimax*), which alternate on every level of the tree. Min nodes represent some minimizing player (the opponent of the player) trying to minimize the player's score across all possible moves, whereas max nodes represent some maximizing player (the player themself) trying to maximize their own score across all possible moves. The search is limited to some depth, at which point the heuristic value of the board is returned.

Heuristic Functions The minimax algorithm is effective, but only so when an effective heuristic evaluation function is used with it. As an extreme example, a random heuristic would give no meaningful information about the game state, and thus using the minimax algorithm with it wouldn't result in a good move, but rather a random move.

There are a couple of criteria on which a heuristic function can be evaluated: 1) accuracy; the function should be able to accurately represent the advantage one player has at any game state, and 2) time; the function should be able to run within a reasonable time limit; taking 100 seconds to evaluate the board at one node is much too long. These criteria can easily be analyzed by empirical evidence. Accuracy can be measured by winrate, i.e. how often one function wins against another function. Time can be measured by average search time, i.e. how long one function takes to evaluate a random game board compared to another function.

There are many different ways to evaluate boards in the game of Othello. The most simple of these methods is rather simple: assign unit weights to each piece, so the heuristic value is the number of pieces the player has minus the number of pieces the opponent has. This is surely better than a random heuristic, but doesn't show too much insight into the intricacies of Othello.

A much more useful heuristic would be to weight each player's discs depending on their position on the board. For example, corners are weighted the most, because discs in the corner can never be flipped. Similarly, the tile directly next to the corner are weighted very low, because it has both a high probability to flip, as well as a high probability to give the corner to the opponent. This is more effective at guessing which moves are good, because it acknowledges the fact that having discs in some positions is more advantageous than other positions. The only problem with this heuristic is that the premise that certain moves are not good (e.g. moves next to the corner) is only true in some scenarios. If a player owned the corner piece, it would actually be beneficial to own the piece next to it as well.

Another heuristic function takes into account the notion of *mobility*. Mobility is simply a measure of how many moves a player is able to make at a certain point in the game. The idea behind this is that having more moves is more beneficial because with more moves there is a higher probability that one of those moves is good. However, this alone is not a very good predictor of the game state; it doesn't say how much either player is winning by. A better function would combine mobility with one of the previously mentioned heuristic functions, such as the weighted discs. This takes into account both the current state of the board, as well as the possibilities of the player, resulting in a much better overall representation.

Pseudocode Figure 1 shows a simple pseudocode implementation of the minimax algorithm. Lines 2-3 handle leaf nodes; once the algorithm hits the depth limit, it returns the heuristic value of the current game state.

Lines 5-11 handle if the algorithm is at a max node. It enumerates all successors of the current node and recursively calls the minimax function on each of the new nodes. It keeps a track of the maximum minimax value achieved so far then finally returns it once all children have been visited.

Lines 12-18 are the opposite of lines 5-11. All successors are enumerated, and the function is called recursively, but the minimum minimax value is kept track of instead of the maximum value.

Figure 1: Minimax Pseudocode

```
 1: function MINIMAX(s, node, depth)
 2:     if depth = 0 or no moves available then
 3:         return heuristic(s)
 4:     end if
 5:     if node is a max node then
 6:         maxValue ← −∞
 7:         for s′ in successors(s) do
 8:             value ← minimax(s′, min, depth − 1)
 9:             maxValue ← max(value, maxValue)
10:         end for
11:         return maxValue
12:     else ▷ node is a min node
13:         minValue ← ∞
14:         for s′ in successors(s) do
15:             value ← minimax(s′, max, depth − 1)
16:             minValue ← min(value, minValue)
17:         end for
18:         return minValue
19:     end if
20: end function
```

Alpha-Beta Pruning

The minimax algorithm is effective at finding the best move, but is rather inefficient because it needs to visit every node in the search tree. Certain subtrees of the search are unnecessary to visit because of how max nodes and min nodes work. For example, remember that max nodes will always take the maximum value of their children and min nodes will always take the minimum value of their children. If a min node encounters a value that is less than the value of one of its siblings that has already been visited, the search on this mode does not need to continue; since the max node (which is the parent of the min node) always takes the maximum value, it will take its sibling which has a higher value. The opposite applies for the opposite configuration as well.

This concept is the premise of the Alpha-Beta Pruning algorithm. It is an extension of minimax which keeps track of two more values, namely *alpha* (α) and *beta* (β). α represents the best possible value that the maximizing player can force so far, and β represents the worst possible value that the minimizing player can force so far. These values are propagated down the search tree, but not upwards. Once $\alpha >= \beta$, the node stops being explored. This is equivalent to the concept talked about previously, where a min node will stop being explored once it encounters a value less than one of those of its siblings.

Pseudocode Figure 2 shows a simple pseudocode implementation of the alpha-beta pruning algorithm. Similar to the minimax pseudocode, lines 2-3 handle leaf nodes.

Lines 5-15 handle if the algorithm is at a max node. Similar to minimax, it enumerates all child nodes and finds the maximum value of recursive alpha-beta function calls. However, it also keeps track of the maximum α so far. Note that α is a parameter which gets passed down the tree, but not up it.

Lines 16-26 are the opposite of lines 5-15. All successors are enumerated, the function is called recursively, and the minimum alpha-beta value is determined by recursive function calls. However, it also keeps track of the minimum β so far. Similar to α, β is a parameter to the function which gets passed down the tree.

As stated previously, if at any time α is greater than or equal to β, the algorithm breaks out of the loop; it is already known that the algorithm will not pick this subtree as the best move of the parent node, so exploring further is unnecessary.

Figure 2: Alpha-Beta Pseudocode

```
1:  function ALPHABETA(s, node, depth, α, β)
2:      if depth = 0 or no moves available then
3:          return heuristic(s)
4:      end if
5:      if node is a max node then
6:          maxValue ← −∞
7:          for s′ in successors(s) do
8:              value ← minimax(s′, min, depth − 1)
9:              maxValue ← max(maxValue, value)
10:             α ← max(α, maxValue)
11:             if α >= β then
12:                 break
13:             end if
14:         end for
15:         return maxValue
16:     else ▷ node is a min node
17:         minValue ← ∞
18:         for s′ in successors(s) do
19:             value ← minimax(s′, max, depth − 1)
20:             minValue ← min(minValue, value)
21:             β ← min(β, minValue)
22:             if α >= β then
23:                 break
24:             end if
25:         end for
26:         return minValue
27:     end if
28: end function
```

Move Reordering Minimax with Alpha-Beta pruning can be further improved by changing the order in which nodes are visited. If the nodes are sorted in such a way that makes pruning occur faster, then less nodes would need to be expanded overall. This is done by making one assumption: moves that are good in the short-term are more likely to be good in the long-term than moves that are not. Of course this is not always the case, but it is a rather effective predictor of good moves.

This is done by performing the evaluation function on the intermediate nodes in the tree, not just the leaf nodes. So, when the successors of a node are visited, they are visited in sorted order according to the evaluation function. At min nodes, nodes are sorted from least evaluation to greatest evaluation. At max nodes, nodes are sorted from greatest evaluation to least evaluation.

Evaluation

Methods

Empirical evaluations were performed based on four main metrics. Three of these were performance metrics, namely nodes generated, nodes expanded, and search time. The fourth metric that was evaluated was winrate.

The performance metrics were gathered over 50 trials. For each trial, a game board would be initialized with a random number of random moves. Then, for each algorithm/heuristic, a search was performed and metrics were gathered. At the end of all trials, all metrics were averaged.

The number of nodes expanded was measured by the number of times the searching function was called. The number of nodes generated was measured by the sum of the number of children at every search node. The search time was measured by taking the start time directly before the search function call and the end time directly after the search function call. The total search time was the difference in time between the end time and the start time.

The winrate metric was gathered over 20 trials each. For each combination of all algorithms/heuristics (i.e. all heuristics play against each other), a full game was simulated. The initial board was initialized with a random number of random moves, and then each algorithm/heuristic took turns making moves. This ensured that the algorithms/heuristics that won were able to consistently do so given any game board. After every game, the player who took the first turn was switched. This ensured that any advantage gained from taking the first move was eliminated.

Empirical Results

Winrate of Various Depths To ensure that looking further ahead in the game tree is more beneficial, winrate was compared between searches of various depths. Three depths, 1, 3, and 6, were tested against each other. The alpha-beta pruning search method with the weight heuristic was used for all searches.

As can be seen in Table 1, searches with a higher depth win 100% of games against searches with a lower depth. This result verifies that looking ahead in the game tree is beneficial, rather than just making decisions based on the next level of the game.

Table 1: Winrate of Player 1 for Various Depth Bounds (N=20)

Player 1	**Player 2**		
	Depth = 1	Depth = 3	Depth = 6
Depth = 1		0	0
Depth = 3	1		0
Depth = 6	1	1	

Winrate of Search Algorithms To ensure the alpha-beta pruning algorithm as well as the move reordering addition

were both implemented correctly, winrate was compared between the both of them as well as the basic minimax algorithm. The weight heuristic with depth 5 was used for all searches.

As can be seen in Table 2, the winrate of all algorithms playing against each other is exactly 0.5, for any combination. This verifies that all algorithms are implemented correctly. If the winrates had not been 0.5, that would be a sign that some algorithms were not implemented correctly. Recall that in alpha-beta, only subtrees which will not be picked are pruned. If more subtrees were being pruned than necessary, then the winrate would likely not be 0.5.

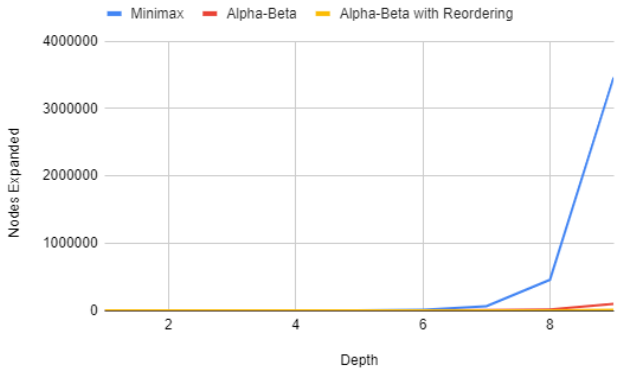
Table 2: Average Winrate of Player 1 for Various Search Algorithms (N=20)

Player 1	**Player 2**		
	Minimax	Alpha-Beta	Reordering
Minimax		0.5	0.5
Alpha-Beta	0.5		0.5
Reordering	0.5	0.5	

Performance of Search Algorithms Since it has been shown that all search algorithms produce equivalent results (i.e. all winrates are 0.5), the performance of each algorithm must be further analyzed. The performance of all algorithms was compared using the weight heuristic and monotonically increasing depth values.

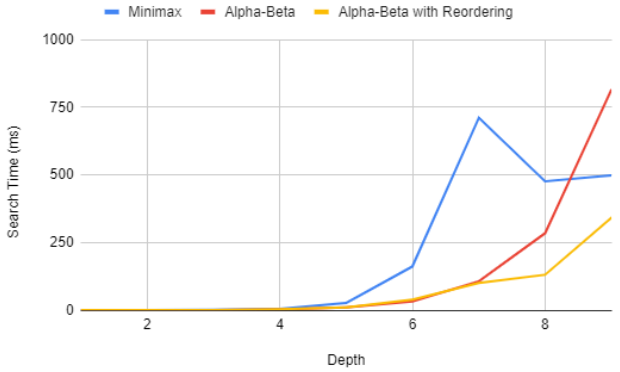
As can be seen in Figure 3, minimax expands significantly more nodes than the other algorithms. Alpha-beta expands more nodes than alpha-beta with move reordering. Thus, we can see that the alpha-beta algorithms are doing what they are supposed to by limiting the search space to only relevant nodes.

Figure 3: Average Nodes Expanded of Various Search Algorithms Against Depth (N=50)



As can be seen in Figure 4, at low to medium depth, alpha-beta and alpha-beta with move reordering both take about the same time, whereas minimax takes significantly longer. At higher depth, alpha-beta with move reordering beats out both of the other algorithms by a significant margin. What's interesting is that minimax at high depth seems to lower its search time, so much so that it ends up being faster than alpha-beta. This result was confirmed over multiple different testing runs. This is most likely due to some compiler or allocator optimizations at higher memory usage.

Figure 4: Average Search Time of Various Search Algorithms Against Depth (N=50)



Winrate of Heuristic Functions To figure out which heuristic is the best to use, winrate was compared between all heuristics. The heuristics used were a random heuristic, the unit weight heuristic, the weighted discs heuristic, and the weighted discs + mobility heuristic. All heuristics were tested using the basic alpha-beta algorithm at depth 5.

As can be seen in Table 3, all "smart" heuristic functions are able to beat a random heuristic function consistently. The weight and weight + mobility heuristics are significantly better than the unit heuristic, though the weight and weight + mobility heuristics do not have much of a difference between each other.

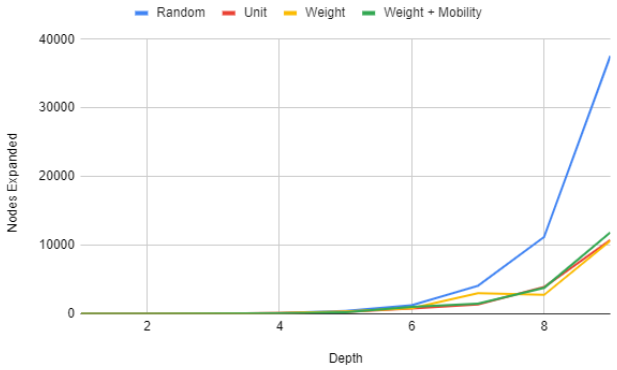
Table 3: Average Winrate of Player 1 for Various Heuristic Functions (N=20)

Player 1	**Player 2**			
	Random	Unit	Weight	Mobility
Random		0.3	0	0.05
Unit	0.7		0	0
Weight	1	1		0.5
Mobility	0.95	1	0.5	

Performance of Heuristic Functions Recall that it was previously discussed that heuristic functions must not only be accurate, but also be able to run in a reasonable amount of time. Thus, to figure out the fastest heuristics, the performance of all heuristics was evaluated. The searches used the alpha-beta with move reordering search algorithm and monotonically increasing depth values.

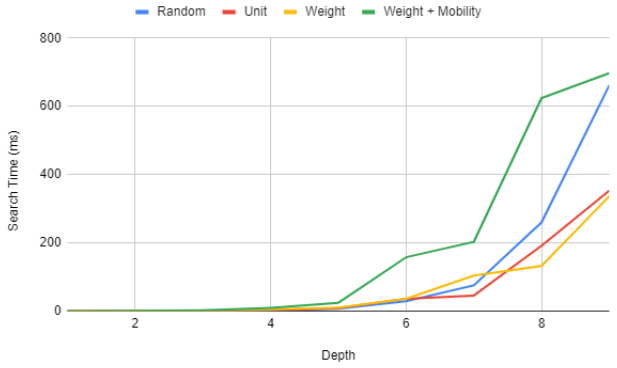
As can be seen in Figure 5, all of the "smart" heuristics have about the same number of nodes expanded, while the random heuristic has significantly more nodes expanded.

Figure 5: Average Nodes Expanded of Various Search Algorithms Against Depth (N=50)



As can be seen in Figure 6, the weight + mobility heuristic has by far the highest average search time, followed by the random heuristic, then the unit heuristic, then the weight heuristic. In fact, the weight + mobility heuristic takes nearly twice as long as just the weight heuristic on its own. This shows that it may potentially be worth using either the unit or weight heuristic if time is a significant constraint.

Figure 6: Average Search Time of Various Search Algorithms Against Depth (N=50)



Discussion

It is now clear that alpha-beta provides a large benefit over basic minimax. Its search time and number of nodes expanded is significantly lower than that of minimax. Additionally, its winrate between itself and minmax is exactly 0.5, so there is no benefit to using minimax at all.

The same could be said for the difference between alpha-beta and alpha-beta with move reordering. Although the difference between the two is not as significant as that of the difference between alpha-beta and minimax, there is still nonetheless a significant difference. Moreover, the winrate between alpha-beta and alpha-beta with move reordering is exactly 0.5, so again there is no benefit to using basic alpha-beta instead of alpha-beta with move reordering.

As for heuristic functions, the weight heuristic is the best heuristic to use, given that it has the highest winrate among all heuristics. It also has a comparable search time to the other heuristics, even being the fastest heuristic at depths 8 and 9.

It has also been shown that increasing the depth bound of the search increases the winrate of all search algorithms. In a competitive situation, it would be worth increasing the search depth to a level that maximizes the time available to the player. This would give better searches while making the best use of the available time.

Future Work

As seen before, one thing which could be improved is the weight + mobility heuristic. It may be worth weighting the mobility term by some constant to make it count for more in the heuristic evaluation. The weights of the discs may currently be too high, resulting in mobility not making enough of a difference in the evaluation.

(Norvig 1992) discusses many other methods that Othello programs use to increase both their search time usage and the accuracy of their heuristic functions.

Edge Stability

One improvement that can be made to the heuristic function uses the notion of *edge stability*. Playing on the edge pieces is important in Othello because no internal moves can flip edge discs, only other edge moves.

The stability of an edge disc is determined by the pieces around it as well as its position. An edge can be one of three stability states: stable, semi-stable, or unstable. If an edge piece can never be flipped (e.g. a corner or a piece next to a corner when the same player owns that corner), then it is stable. If an edge may be able to be flipped in the future (e.g. the disc is surrounded by two of its own pieces but could be captured in the future), it is semi-stable. If an edge can be instantly captured (e.g. the disc is surrounded by one empty space and one opponent disc) then it is unstable.

While this is an accurate heuristic, it is rather expensive to compute. For this reason, many Othello programs choose to use a pre-computed lookup table of edge stabilities instead of computing it on-the-spot.

Iterative Deepening

As discussed previously, in competitive Othello, it is important to not go over the time limit allowed for each player. Since these search algorithms are inherently depth-first, iterative deepening is a technique that can be used to maximize time usage.

For each move, a certain amount of time could be allocated, for which iterative deepening could be used to find progressively better solutions. Once the time limit has been reached, the search would stop, and the best solution so far would be used as the move.

The only downside to this technique is that some of the search time is wasted on earlier depth values. The time used for earlier depths could be used for a deeper search instead. Nonetheless, it may be worth implementing in the future.

Forward Pruning

As with any game, some moves are objectively poor. The minimax algorithm does not take this into account, however, if some moves are ignored, significant parts of the search tree could be pruned.

For example, imagine a player is considering a move next to a corner. If the opponent has a disc next to the square the player is attempting to take, this is an objectively bad move. In this situation, the opponent would be able to put a disc in the corner, flipping the disc the player put down.

In situations like this, it is most likely not worth exploring down that move's subtree. Since time is so important in Othello, it is more useful to put time towards performing a deeper search than it is to consider objectively bad moves.

Conclusion

In conclusion, alpha-beta pruning provides great improvement to the basic minimax algorithm. Moreover, changing the order in which child nodes are visited in alpha-beta pruning is also a great improvement. Both of these changes have been shown to significantly decrease search time, while still keeping the integrity of the algorithm. The weight heuristic has been shown to have a comparable search time to other heuristics as well as the best winrate, therefore it is the best heuristic to use for all scenarios. It is by these points that we can see using alpha-beta with move reordering as well as the weight heuristic would be the best fit for a competitive Othello program. It is able to see great success compared to other heuristic functions, as well as using the fastest search algorithm and a comparably fast heuristic function.

References

Lee, K.-F., and Mahajan, S. 1986. Bill : a table-based, knowledge-intensive othello program.

Norvig, P. 1992. Chapter 18 - search and the game of othello. In Norvig, P., ed., *Paradigms of Artificial Intelligence Programming*. San Francisco (CA): Morgan Kaufmann. 596 – 654.

Rosenbloom, P. S. 1981. A world-championship-level othello program.