Applications of Artificial Intelligence and Machine Learning in Othello

Jack Chen

TJHSST Computer Systems Lab 2009–2010

Abstract

This project explores Artificial Intelligence techniques in the board game Othello. Several Othello-playing programs were implemented and compared. The performance of minimax search algorithms, including alpha-beta, NegaScout and MTD(f), and of other search improvements such as transposition tables, was analyzed. In addition, the use of machine learning to enable AI players to improve play automatically through training was investigated.

1 Introduction

Othello (also known as Reversi) is a two-player board game and abstract strategy game, like chess and checkers. I chose to work with Othello because it is sufficiently complex to allow significant exploration of advanced AI techniques, but has a simple set of rules compared to more complex games like chess. It has a moderate branching factor, larger than checkers and smaller than chess, for example, which makes advanced search techniques important without requiring a great deal of computational power for strong play. Although my AI programs are implemented to play Othello, most of the algorithms, data structures, and techniques I have investigated are designed for abstract strategy games in general instead of Othello specifically, and many machine learning algorithms are widely applicable to problems other than games.

2 Background

The basic goal of an AI player is to consider the possible moves from the current game state, evaluate the position resulting from each move, and choose the one that appears best. One major component of an AI player is the static evaluation function, which heuristically estimates the value of a position without exploring moves. This value indicates which player has the advantage and how large that advantage is. A second major component is the search algorithm, which more accurately evaluates a state by looking ahead at potential moves.

3 Static Evaluation

3.1 Features

For positions at the end of a game, the static evaluation is based solely on the number of pieces each player has, but for earlier positions, other positional features must be considered. The primary goals before the end of the game are mobility, stability, and parity. The major features used in my static evaluation function reflect these three goals. The overall evaluation is a linear combination of the features, that is, it is a weighted sum of the feature values. Features that are good have positive weights, while features that are bad have negative weights, and the magnitude of a feature's weight reflects its importance.

- Mobility

  Mobility is a measure of the number of moves available to each player, both at the current position and in the future (potential mobility). Mobility is important because a player with low mobility is more likely to be forced to make a bad move, such as giving up a corner. The goal is to maximize one's own mobility and minimize the opponent's mobility.

  - Moves

    The number of moves each player can make is a measure of current mobility. The moves differential, the number of moves available to the player minus the number of moves available to the opponent, is one of the features used in my static evaluation function. Positions with higher moves differential are better for that player, so this feature has a positive weight.

  - Frontier squares

    Frontier squares are empty squares adjacent to a player's pieces. The number of frontier squares is a measure of potential mobility, because the more frontier squares a player has, the more moves the opponent can potentially make. Having fewer frontier squares than the opponent is good, so the frontier squares differential is weighted negatively.

- Stability

  Pieces that are impossible to flip are called stable. These pieces are useful because they contribute directly to the final score.

  - Corners

    Corners are extremely valuable because corner pieces are immediately stable and can make adjacent pieces stable. They have the largest positive weights of all the features I use.

  - X-squares

    X-squares are the squares diagonally adjacent to corners. X-squares are highly undesirable when the adjacent corner is unoccupied because they make the corner vulnerable to being taken by the opponent, so they have very negative weight.

  - C-squares

    C-squares are the squares adjacent to corners and on an edge. C-squares adjacent to an unoccupied corner are somewhat undesirable, like X-squares, but they are much less dangerous. In addition, C-squares can contribute to edge stability, which makes them desirable in some cases. Generally, C-squares are weighted fairly negatively, but to a much smaller extent than X-squares.

- Parity

  Global parity is the strategic concept that the last player to move in the game has a slight advantage because all of the pieces gained become stable. White therefore has an advantage over black, as long as the parity is not reversed by passes. In addition, in the endgame, empty squares tend to separate into disjoint regions. Local parity is based on the idea that the last player to move in each region has an advantage because the pieces that player gains are often stable. I use global parity as a feature, but do not consider local parity.

3.2 Game Stages

The importance of the features used in the static evaluation function depends on the stage of the game. For example, one common strategy is to minimize the number of pieces one has early in the game, as this tends to improve mobility, even though this is contrary to the ultimate goal of the game. It is useful, then, to have different feature weights for different game stages. In Othello, the total number of pieces on the board is a good measure of the game stage.

4 Search Algorithms and Data Structures

Static evaluation is often inaccurate. For example, it is difficult to detect traps statically. When evaluating a position, it is therefore important to consider possible moves, the possible moves in response to each of those moves, and so on. This forms a game tree of the possible sequences of moves from the initial game state.

4.1 Minimax

The minimax search algorithm is the basic algorithm to do this exploration of the game tree. Minimax recursively evaluates a position by taking the best of the values for each child position. The best value is the maximum for one player and the minimum for the other player, because positions that are good for one player are bad for the other.

The number of nodes searched grows exponentially with search depth, which is measured in ply (one ply is a move by one player). The rate of growth is the branching factor, which is the average number of children of each node, or the average number of moves from each position. In Othello, the branching factor averages about 10, although it tends to be higher in the midgame and lower in the endgame. Because of this extremely large growth rate, searching the entire game tree down to the end of the game is not practical. Therefore, minimax search can only look a limited number of moves ahead. The terminal positions at the end of this lookahead are evaluated with the static evaluation function.

Figure 1 shows a minimax search on an example game tree. Player 1 is the minimizing player and player 2 is the maximizing player. The letters represent game states, and the numbers next to each node represent its evaluation. The search has depth 3, and the values of the leaf nodes represent values from a static evaluation function.

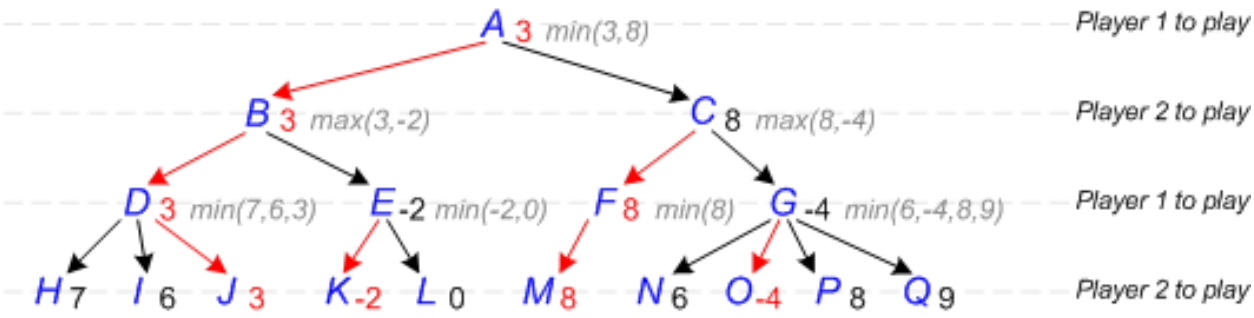
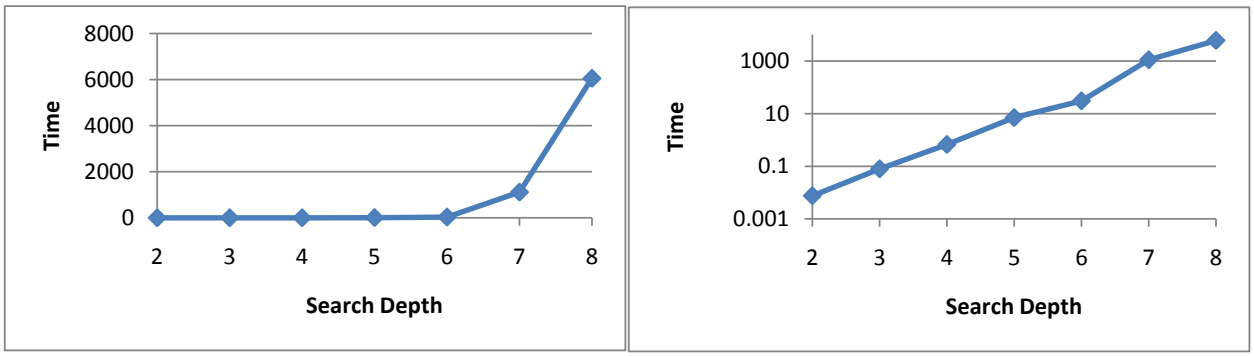


Figure 1: An example minimax search

Figure 2 shows the performance of minimax search at various seach depths. The plots show the average time required per game (in seconds) when using a minimax search of fixed depth to play several randomized games, on linear and logarithmic scales. The time grows exponentially with a base of about 10, which matches the average branching factor.



(a) Linear scale

(b) Semilogarithmic scale

Figure 2: Time required for minimax search vs. search depth

4.2 Alpha-Beta Pruning

There are many minimax algorithms that are much faster than naive minimax. Alpha-beta pruning is an extremely important improvement on which several others depend. Alpha-beta search greatly reduces the number of nodes in the game tree that must be searched. This search algorithm maintains two values, alpha and beta, that represent the window between the best values the players can be assured of from the search so far. If the algorithm finds a bound for a node's value that is outside the alpha-beta window, then the node and its subtree can be safely pruned because the node's value cannot affect the value of the root.

Figure 3 shows an alpha-beta search on the same game tree shown in Figure 1. Nodes L and G, along with their subtrees, are pruned, significantly reducing the computation time spent searching this game tree.

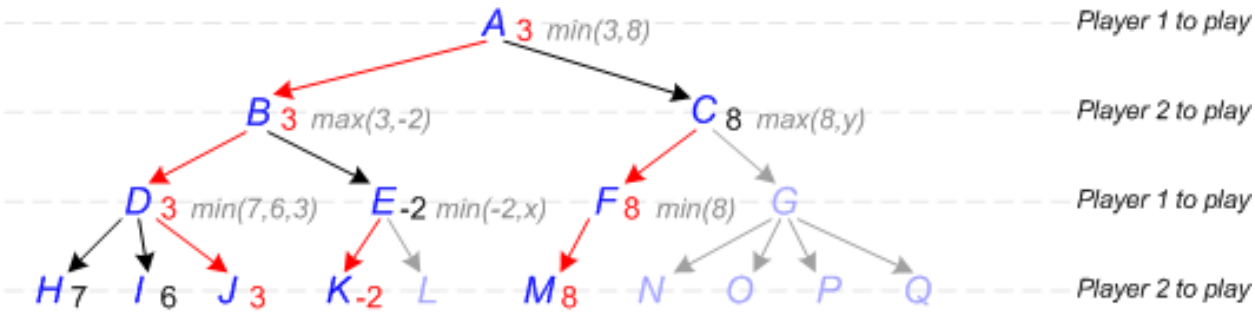


Figure 3: An example alpha-beta search

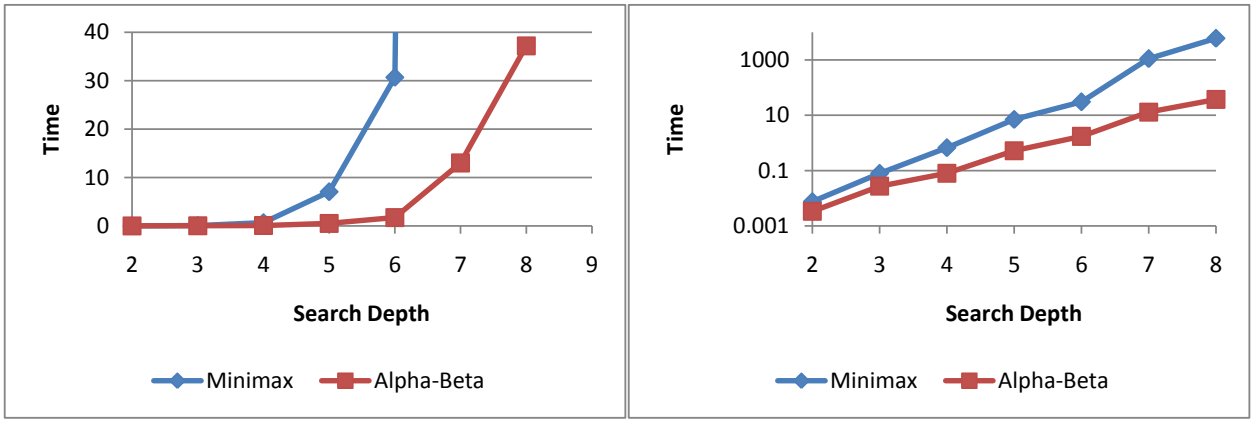
In the best case, if moves are searched in order from best to worst, then the effective branching factor of alpha-beta search is reduced to the square root of naive minimax's branching factor, meaning a search twice as deep is possible with about the same computation time. Of course, the correct ordering of the moves is not known, or a search would be unnecessary. However, even with random move ordering, alpha-beta pruning dramatically reduces the number of nodes searched. There are many methods that can be used to improve move ordering, such as previous estimates from shallower searches in iterative deepening, killer moves, and history heuristics.

Figure 4 compares the performance of alpha-beta and minimax at various seach depths. For alpha-beta, the time grows exponentially with a base of about 5, which is a huge improvement over minimax's branching factor of approximately 10.

4.3 Transposition Table

Move sequences that result in the same position are called transpositions. For example, the two opening move sequences shown in Figure 5 result in the same position.

An important way to improve search speed is to cache information about positions that have already been searched in a data structure called a transposition table. Transpositions could cause a program to repeatedly analyze the same position. Storing previous results in



(a) Linear scale (b) Semilogarithmic scale

Figure 4: Comparison of time required for alpha-beta and minimax search vs. search depth

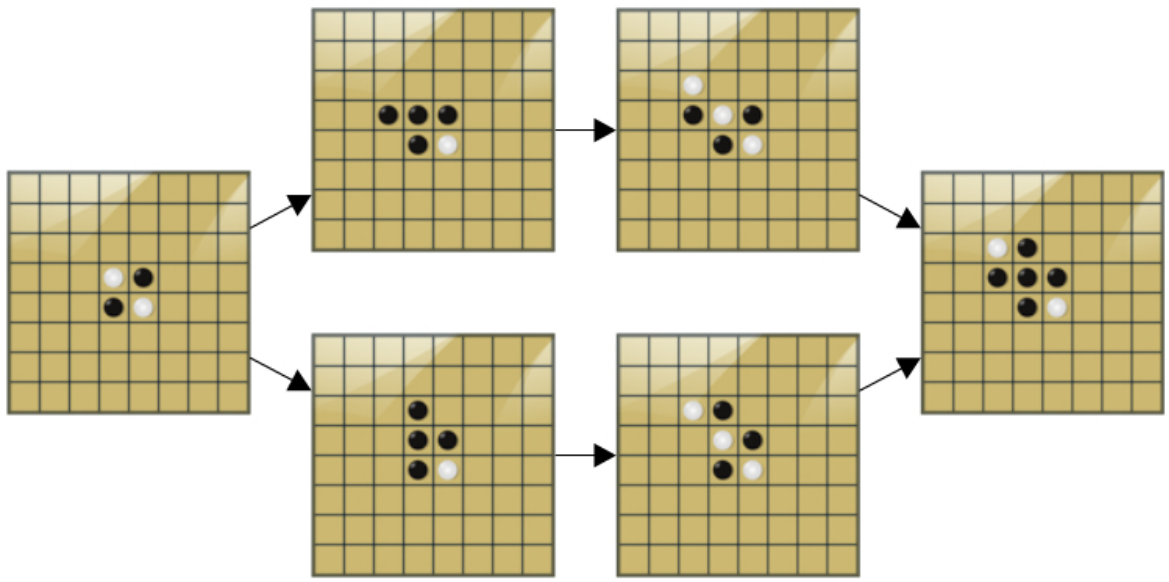


Figure 5: An example transposition

a transposition table allows the program to avoid this problem. In addition, a transposition table can be used to improve move ordering by storing the best move found for each position and searching this move first. This is especially useful with iterative deepening, as the best moves found in shallow searches often remain good moves for deeper searches. I found that this improved move ordering is a much more important use of the transposition table in terms of increasing search speed.

The transposition table is implemented as a hash table to allow efficient access. One useful method for hashing positions in games like Othello and Chess is Zobrist hashing [12]. A Zobrist hash consists of an XOR sum of several bitstrings. For each square on the board, there is one randomly generated bitstring representing a black piece and another representing a white piece. A position's Zobrist hash is formed by XORing together the appropriate bitstrings. The primary benefit of Zobrist hashing is that it can be incrementally updated very quickly by XORing it with the bitstrings for the pieces that have changed. Zobrist hashes also have the advantage of uniform distribution.

Alpha-beta search may not yield an exact value if the true value lies outside the alpha-beta window. Instead, the search may yield only an upper or lower bound. However, this limited information is still useful in later searches, as it can reduce the alpha-beta window size or result in cutoffs.

For each position in the transposition table, the following information is stored:

- The hash key or another hash of the position, which is used to detect collisions. This takes less memory than storing the entire board, although there is a possibility that two different positions will have the same hashes. Such a collision would cause inaccurate information to be used. However, with a sufficiently long hash key, such as the 64-bit hash keys I use, the probability of a collision is extremely small and is outweighed by the time and memory savings.

- Information about the position's value from previous searches, including an exact value, an upper bound, or a lower bound.

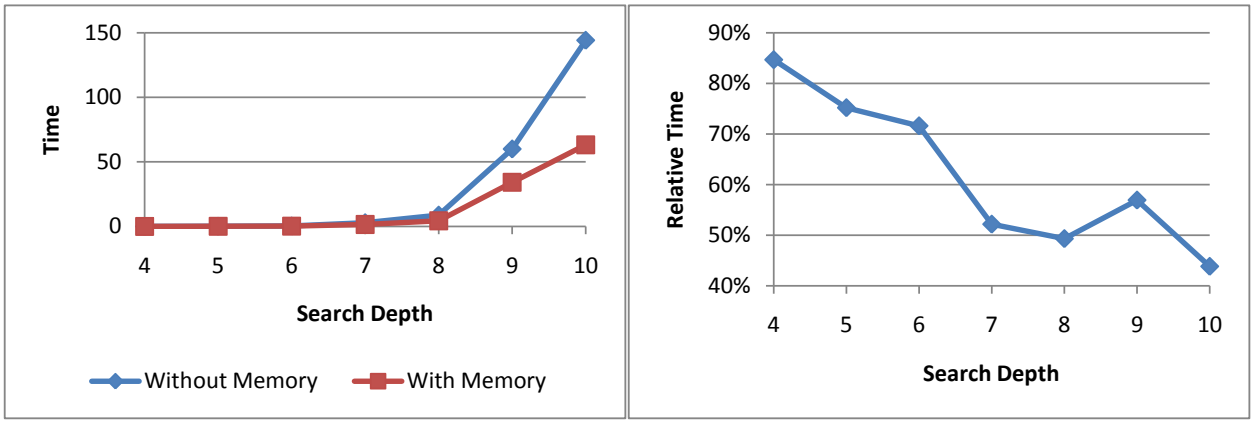
- The best move found so far.

- The depth of the search this information is from. If the depth is less than the depth of the current search, the evaluation information should not be used, but the move information can still be used to improve move ordering.

Figure 6 shows the performance of alpha-beta search with and without a transposition table for memoization and for move ordering. Based on these results, search with memory is an average of 1.5 to 2 times as fast.

4.4 NegaScout

NegaScout [11] (which is similar to Principal Variation Search [9]) is an enhancement of alpha-beta search that can reduce the number of nodes that must be searched. NegaScout



(a)

(b) Time for alpha-beta with memory relative to alpha-beta without memory

Figure 6: Comparison of time required for alpha-beta search with and without a transposition table vs. search depth

searches the first move for a node with a normal alpha-beta window. It then assumes that the next moves are worse, which is often true with good move ordering. For the remaining moves, it uses a null-window search, in which the alpha-beta window has zero width, to test whether this is true. The value must lie outside the null-window, so the null-window search must fail. However, this yields a lower or upper bound on the value if the null-window search fails high or low, respectively. If it fails low, then the test successfully shows that the move is worse than the current alpha and therefore does not need to be further considered. Otherwise, the test shows that the move is better than the current alpha, so it must be re-searched with a full window.

Null-window searches are faster because they produce many more cutoffs. However, even though NegaScout never explores nodes that alpha-beta prunes, NegaScout may be slower because it may need to re-search nodes several times when null-window searches fail high. If move ordering is good, NegaScout searches faster than alpha-beta pruning, but if move ordering is poor, NegaScout can be slower. See Section 4.6 for analysis of the performance of NegaScout.

A transposition table is even more beneficial to NegaScout than to alpha-beta because stored information can be used during re-searches, for example, to prevent re-evaluation of leaf nodes.

4.5 MTD(f)

MTD(f) [10] is another search algorithm that is more efficient than alpha-beta and outperforms NegaScout. It is efficient because it uses only null-window searches, which result in many more cutoffs than wide window alpha-beta searches. Each null-window search yields a bound on the minimax value, so MTD(f) uses repeated null-window searches to converge

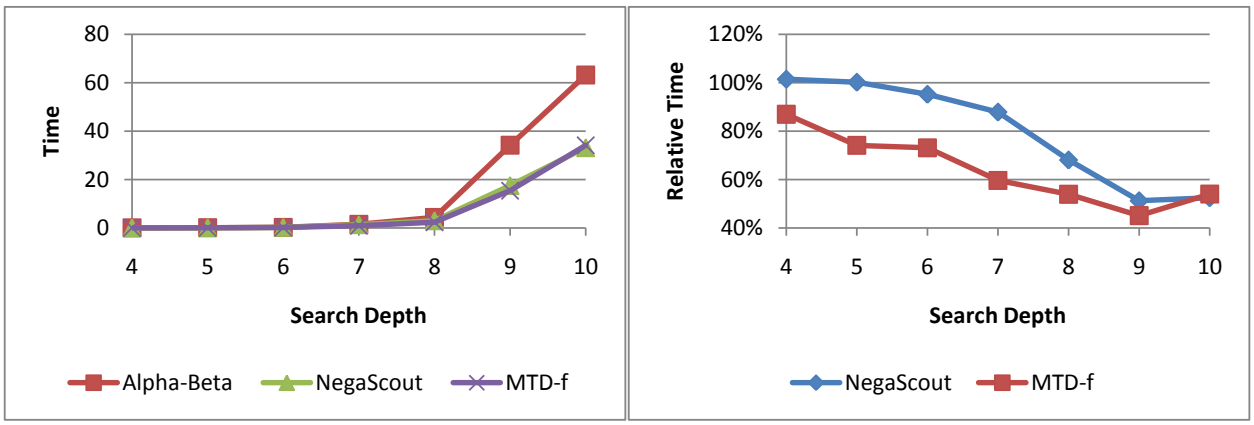
on the exact minimax value. Because many nodes need to be re-evaluated several times, a transposition table is crucial for MTD(f) to prevent excessive re-searches.

MTD(f) starts its search at a given value, f. The speed of MTD(f) depends heavily on how close this first guess is to the actual value, as the closer it is the fewer null-window searches are necessary. It is therefore useful to use iterative deepening on MTD(f), using the value of the previous iteration as the first guess for the next iteration.

While MTD(f) is theoretically more efficient than alpha-beta and NegaScout, it has some practical issues, such as heavy reliance on the transposition table and search instability. The performance of MTD(f) is discussed in Section 4.6.

4.6 Performance Analysis

Figure 7 compares the performance of NegaScout, MTD(f), and alpha-beta search. All of these searches are done with a transposition table. Different searches were done with transposition table sizes of 2^{20}, 2^{22}, or 2^{24} positions, and the plotted time is the minimum of these times. These results indicate that NegaScout is ineffective at small depths, but is significantly faster than alpha-beta on deeper searches. MTD(f) is faster than both alpha-beta and NegaScout overall.



(a)

(b) Time for NegaScout and MTD(f) relative to alpha-beta

Figure 7: Comparison of time required for NegaScout, MTD(f), and alpha-beta vs. search depth

However, as search depth increases, the relative speed of MTD(f) worsens. This is due to its heavy transposition table dependence, as can be seen in Figure 8, which shows the performance of NegaScout and MTD(f) with different transposition table sizes. When the transposition table is too small, MTD(f) takes a heavy performance hit from repeated re-searching. For 9-ply MTD(f) searches, a size of 2^{22} appears sufficient, while for 10-ply searches, this is also too small. On the other hand, NegaScout is barely affected by these changes in transposition table size.

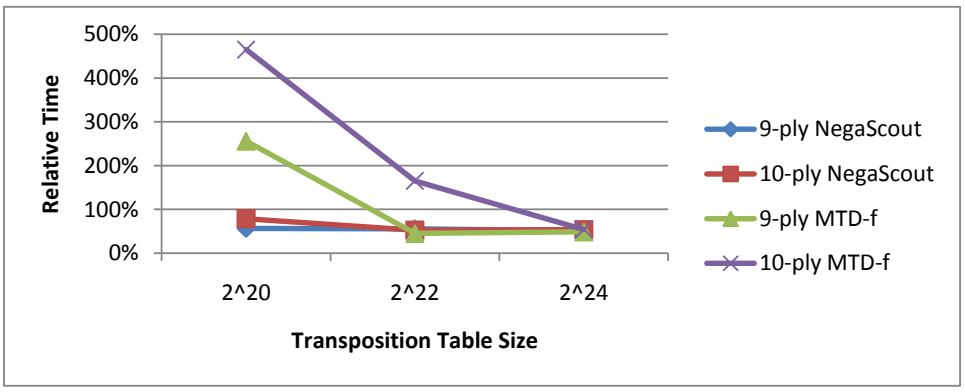


Figure 8: Time for NegaScout and MTD(f) with 9-ply and 10-ply search depth relative to alpha-beta vs. transposition table size

5 Bitboards

A bitboard is a data structure that can be used to represent a game board in which each bit corresponds to one of the squares on the board and indicates whether or not a certain piece is on that square. For Othello, I use bitboards consisting of two 64-bit bitstrings. One of the bitstrings represents black's pieces and another represents white's pieces, and each bit of these bitstrings represents one of the 64 squares on the board. The use of bit manipulation techniques on bitboards allows great speed improvements in certain operations, such as finding the possible moves and counting frontier squares. Bitboards are also advantageous in terms of memory use, since they are very compact. The use of bitboard optimizations made the AI player about 5 times as fast, as shown in Figure 9. This is enough to search about one ply deeper, a significant advantage.

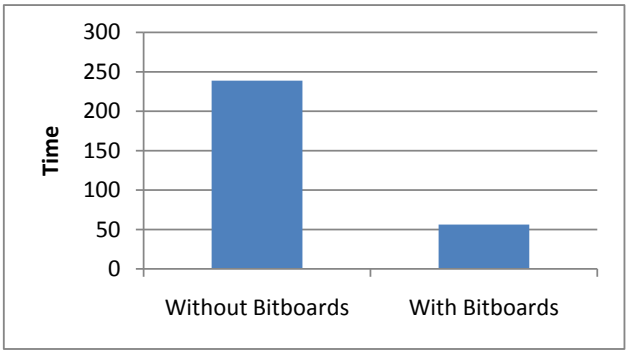


Figure 9: Comparison of time for alpha-beta search with and without bitboard optimizations

6 Time Management

My time management strategy dynamically allocates a certain amount of time to each move based on the amount of time available and the number of moves remaining in the game. If a move takes more or less time than allocated, or if the number of moves that my AI player needs to make changes due to passes, the time allocated to later moves changes accordingly.

Once time is allocated for a move, my AI player searches with iterative deepening as deeply as possible given the time limit. One simple way to do this is to continue searching until all of the allocated time has been spent, then aborting the search immediately. However, this wastes some time spent on the final search iteration. To help avoid aborting a search, my time management strategy estimates the branching factor, based on the number of moves available in the leaf nodes of the previous search iteration, and uses this to predict the time for the next search iteration. If this search is predicted to exceed the allocated time, then the iterative deepening is ended, thus saving the remaining allocated time for future moves.

It is possible to estimate the time required to search to a given depth without iterative deepening based on the branching factor and time of earlier searches. However, the branching factor depends on several factors including the search depth and the game stage, so iterative deepening allows more accurate time estimation. In addition, iterative deepening helps increase search efficiency by improving move ordering. Although nodes at the first few levels may be re-searched several times, the time spent searching at lower depths is much less than the time spent on the deepest search iteration whenever the branching factor is reasonably large, so the extra time spent on shallower searches is usually outweighed by the time saved on deeper searches. During the endgame, the branching factor is low enough that this is not the case, so my AI player does a full-depth search to the end of the game instead of the normal iterative deepening search. This endgame search also is much faster per node because all static evaluation at the terminal nodes is based soley on the number of pieces each player has.

7 Training

Initially, my static evaluation function's feature weights were set manually based on human strategy, and hand-tuned somewhat with a manual hill-climbing process. However, this process is slow and ineffective. A much better way to set feature weights is to use machine learning to automatically train the static evaluation function by optimizing the feature weights.

First, I generated a large set of example positions by playing several thousand games with a stochastic version of my untrained AI player. In order to generate a diverse training set that also reflects realistic play, this AI chooses moves with probability based on the move's evaluation, with high probability for the best move(s) and decreasing probability for increasingly suboptimal moves. Later, after training using these examples, I used the trained AI player to generate additional example positions in a bootstrapping process.

I divided the game into 59 stages, each stage representing positions with a certain number

of total pieces from 5 to 63, and trained a separate set of weights for each stage. The training process starts with the last stage and proceeds through the game stages in reverse order. For each stage, the set of example positions matching the stage are evaluated with a fairly deep search. For the last few stages of the game, these evaluations are exact because the search can reach the end of the game. As earlier stages are trained, the leaf nodes of the searches are statically evaluated with the weights for a later game stage, which have already been trained, making the evaluations quite accurate. These evaluations are the target values for the static evaluation function at the current game stage. To optimize the current stage's weights, I used a batch gradient descent method.

7.1 Gradient Descent

Gradient descent is an optimization algorithm that finds a local minimum of a function by taking steps in the direction of steepest descent, proportional to the negative gradient of the function. For example, Figure 10 shows a contour plot of the function being optimized with the sequence of steps taken by the gradient descent algorithm. The steps are orthogonal to the contours because they are in the direction of the gradient, and they become progressively smaller as the algorithm converges on the local optimum.

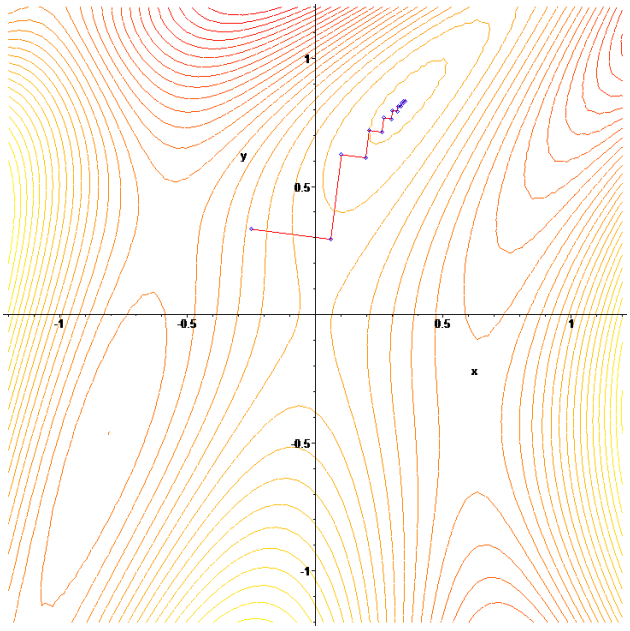


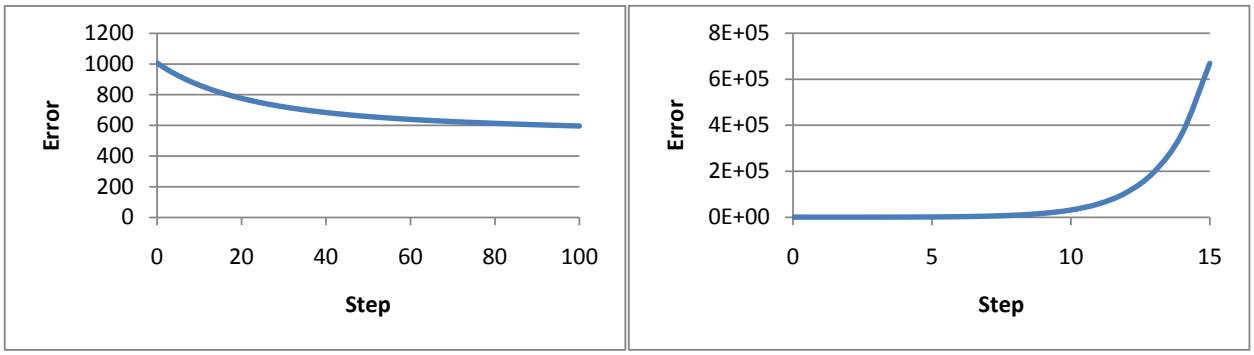
Figure 10: Example of gradient descent

As applied to static evaluation training, the function to minimize is a measure of the error for a given set of weights, based on the difference between the target value for each position and the value produced by the static evaluation function with the given weights. The gradient descent method starts with an arbitrary set of weights and then then repeatedly

takes steps in the direction that reduces error most until it reaches convergence at a local minimum.

A basic form of gradient descent takes steps of size directly proportional to the magnitude of the gradient, with a fixed constant of proportionality called the learning rate. However, a learning rate that is too small may result in extremely slow convergence, while a learning rate that is too large may converge to a poor solution or fail to converge at all. The use of a dynamic learning rate helps to avoid these problems. In each step, a line search is used to determine a location along the direction of steepest descent that loosely minimizes the error.

To illustrate the benefits of a dynamic learning rate, I compared the performance of training with the line search method and with two fixed learning rates. Training with a small fixed learning rate of 0.003 converged to almost the same solutions as the method with a line search, but in about four times as many steps. Although using a line search requires more time per step, the line search method still completed training in about half as much time. On the other hand, a method with a slightly larger fixed learning rate of 0.005 failed to converge when training weights for the last stages of the game. As the weights for all earlier stages of the game depend on these, the training results are all highly suboptimal. Figure 11 shows the error at each of the first few steps in the training of the last game stage (63 total pieces) with these fixed learning rates. As we can see, in the training with learning rate 0.005, the error grows exponentially. To see why, we plot the value for one weight at each of the first few steps in Figure 12. After a few steps, the value for this weight began oscillating between positive and negative values with exponentially growing magnitude. The other weights behaved similarly.



(a) Learning rate 0.003

(b) Learning rate 0.005

Figure 11: Comparison of error vs. step with two fixed learning rates

7.2 Overfitting and Cross-Validation

The goal of training is to not only have the static evaluation function fit the example positions well, but to generalize this learning to fit other positions well. However, if a game results in positions unlike the training examples, the fit may be poor. This is especially problematic

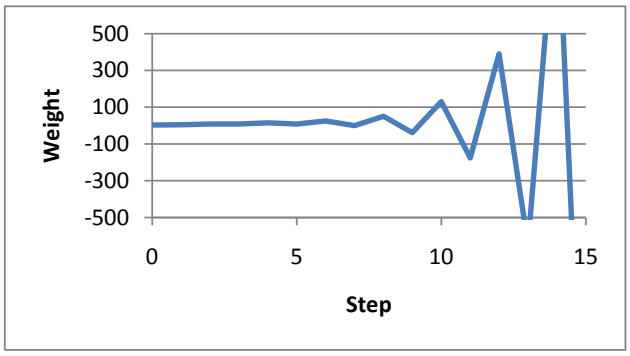


Figure 12: Value of weight for corner feature vs. step with learning rate 0.005

when the number of examples is too small, which can result in overfitting to the examples that actually worsens performance on general positions.

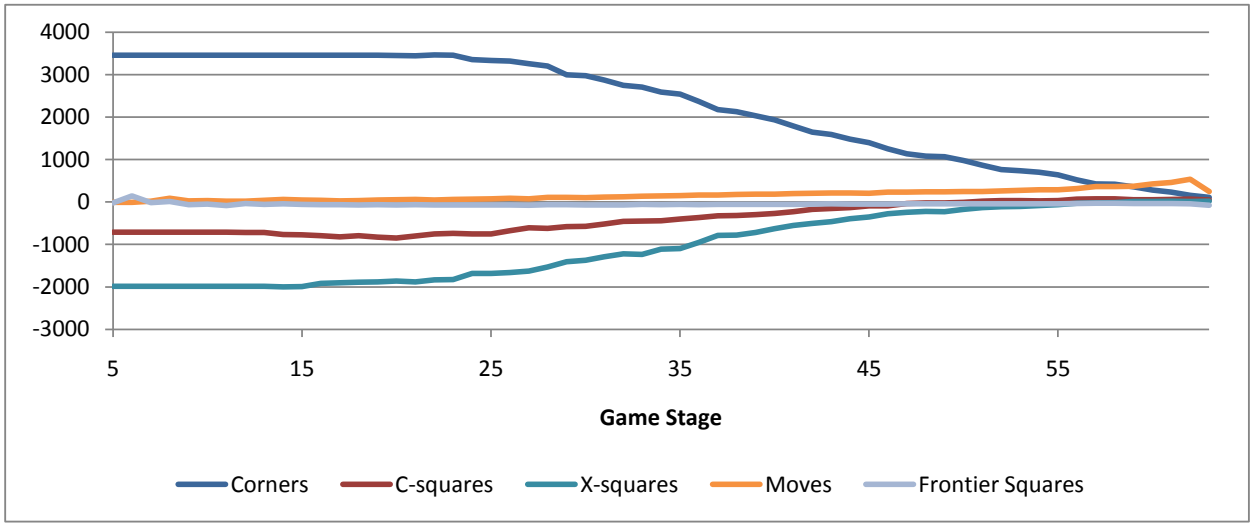
To avoid overfitting, the example positions are partitioned into a training set and a validation set. The training is done with the examples in the training set only, and after each step in the training, performance is evaluated on the validation set. If this performance stops improving, even if performance on the training set is still decreasing, training is stopped. This early stopping method is a simple and effective way to prevent overfitting.

7.3 Training Results

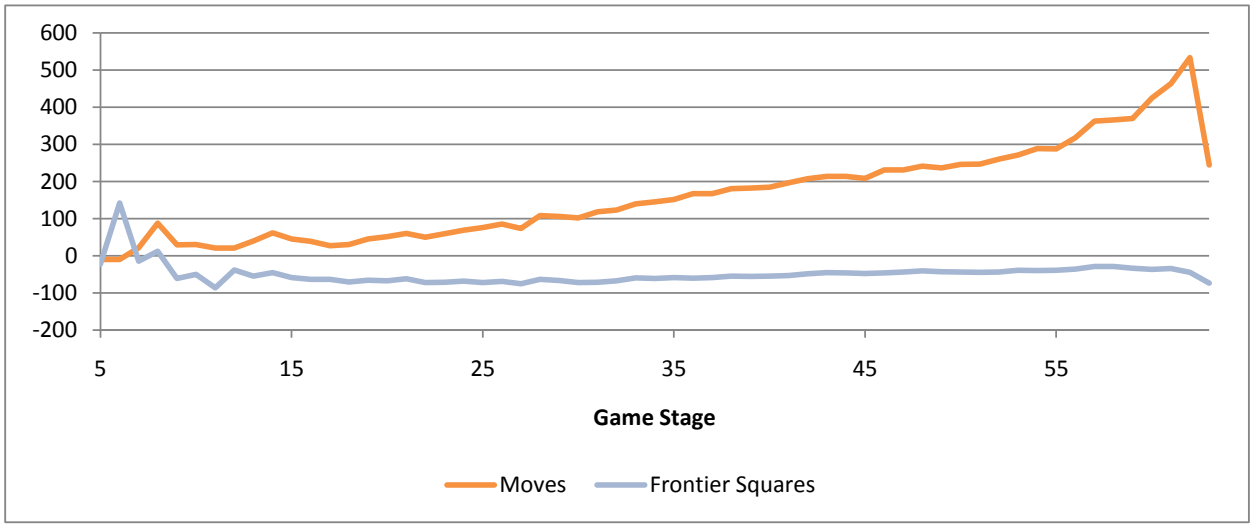
Figure 13 shows the trained weights for some of the more important features over each game stage. We can see that corners and adjacent squares are extremely important, especially early in the game, while moves become increasingly important near the end of the game. Frontier squares are consistently weighted slightly negatively.

8 Referee

Another major part of this project was the development of an Othello referee program to run the game. The referee acts as the interface between two players playing a game. The referee supports AI player programs written in multiple programming languages as well as human players. It keeps track of the game state, enforces rules and time limits, and handles scoring. The referee has a graphical user interface that displays the board, animates the players' moves, and allows a human to play easily by clicking on the board. The referee is highly customizable and supports a number of additional features, such as tournaments between several players and automatic statistical testing to determine whether one player plays significantly better than another.



(a) Zoomed-out view of largest weights



(b) Zoomed-in view of smaller weights

Figure 13: Trained weights for selected features over each game stage

9 Conclusions

I implemented and compared several Othello programs using various static evaluation functions, search algorithms, and data structures. Search improvements such as transposition tables and bitboards greatly improve performance, and efficient search algorithms such as NegaScout and MTD(f) are much faster than the basic alpha-beta search algorithm. I found that MTD(f) outperformed the other search algorithms I tested. The use of machine learning to optimize the static evaluation function was successful in improving the static evaluation's accuracy, resulting in better play.

My best AI players were fairly strong, able to easily defeat me and other amateur human players even with extremely small time limits. However, they were not nearly as strong as Othello programs such as Michael Buro's Logistello [6]. There are many other AI techniques that could be explored in future research.

10 Future Research

Selective search algorithms, such as ProbCut [2] and Multi-ProbCut [4] can further enhance game-tree search by pruning parts of the game tree that probably will not affect the overall minimax value. This allows the player to search much deeper in the relevant parts of the game tree.

Opening books [5] allow much better and faster play in the early game by storing previously computed information about early game board states.

Another potential area of investigation is parallelization. Splitting up searches between several processors can greatly increase the search speed.

Traditionally, the static evaluation function is based on human knowledge about the game. In Othello, static evaluation is usually based on features related to human goals, such as mobility, stability, and parity. However, using pattern-based features as discussed in [3] can improve static evaluation.

There are several machine learning techniques that can be applied to the training of the static evaluation function. Among the algorithms I investigated but did not implement are genetic algorithms, particle swarm optimization, and artificial neural networks.

There are many other machine learning methods that can be used to improve the quality and speed of an AI player based on experience. For example, [8] describes the use of an experience base to augment a non-learning Othello program, and [1] describes a chess program that learns as it plays games using the TDLeaf(λ) algorithm. Another interesting idea is to learn a model of an opponent's strategy and incorporate that into the minimax search [7].

References

[1] J. Baxter, A. Tridgell, and L. Weaver. Experiments in parameter learning using temporal differences. *International Computer Chess Association Journal*, 21(2):84–99, 1998.

[2] M. Buro. ProbCut: An effective selective extension of the alpha-beta algorithm. *International Computer Chess Association Journal*, 18(2):71–76, 1995.

[3] M. Buro. An evaluation function for othello based on statistics. Technical Report 31, NEC Research Institute, 1997.

[4] M. Buro. Experiments with Multi-ProbCut and a new high-quality evaluation function for Othello. In H. J. van den Herik and H. Iida, editors, *Games in AI Research*, pages 77–96. Universiteit Maastricht, 2000.

[5] M. Buro. Toward opening book learning. In J. Fürnkranz and M. Kubat, editors, *Machines that Learn to Play Games*, chapter 4, pages 81–89. Nova Science Publishers, Huntington, NY, 2001.

[6] M. Buro. The evolution of strong Othello programs. In *Proceedings of the International Workshop on Entertainment computing (IWEC-02)*, Makuhari, Japan, 2002.

[7] D. Carmel and S. Markovitch. Learning models of opponent's strategy in game playing. Technical Report CIS #9305, Centre for Intelligent Systems, Technion - Israel Inst. Technology, Haifa 32000 Israel, Jan. 1993.

[8] K. A. DeJong and A. C. Shultz. Using experience-based learning in game-playing. In *Proceedings of the 5th International Conference on Machine Learning*, pages 284–290, 1988.

[9] T. A. Marsland and M. S. Campbell. Parallel search of strongly ordered game trees. *ACM Computing Survey*, 14(4):533–551, Dec. 1982.

[10] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. A new paradigm for minimax search. Research Note EUR-CS-95-03, Erasmus University Rotterdam, Rotterdam, Netherlands, 1995.

[11] A. Reinefeld. An improvement of the scout tree search algorithm. *ICCA Journal*, 6(4):4–14, 1983.

[12] A. L. Zobrist. A new hashing method with application for game playing. Technical Report 88, Univ. of Wisconsin, 1970.