CS 221 Othello Report
Demosthenes

Chuong Do, Sanders Chong, Mark Tong, Anthony Hui*
Stanford University
(Dated: November 27, 2002)

This report is intended to inform the reader about our experiences and strategies in programming an artificially intelligent Othello client. More specifically, we will describe what went right, what went wrong, and what we would do if we were to do it again. There is a brief discussion on various searching, training, and evaluation techniques, accompanied by analysis on what we found to work the best. The last section provides some thoughts on future work and extension.

1. INTRODUCTION AND OVERVIEW

Games such as chess and Othello have showcased the power of applying artificial intelligence to develop winning strategies and out-compete humans at their own game. In this report we will consider the game of Othello to introduce ideas in programming an intelligent agent and present an interesting training algorithm that we found to work well.

With the enormity of the game the search space, games such as Othello are virtually unsolvable by humans or computers. Yet, it is not a game of chance. Good Othello players consistently do well at tournaments and novices consistently get beaten. What, then, separates the good players from the bad? Since no human or computer could look at *every* possibility, there must be some key features that distinguish a good position from a bad one. That is, there are some evaluation features that are important in playing Othello competitively. We will consider these evaluation features first.

2. EVALUATION FEATURES

Our evaluation function, given a board, returns a numerical score indicating the desirability of a particular board configuration. The evaluation function that we use in our Othello client incorporates a total of nine features, each given a particular weight in different stage of the game. They are explained as follows:

Piece Differential

Piece Differential = number of our pieces − number of the opponent's pieces

This simply measures the difference between the number of our pieces and the number of opponent's pieces. It should almost definitely be a feature in any Othello game since in the end, it is the piece differential that matters and not anything else.

Mobility Differential

Mobility Differential = number of legal moves on our side − number of legal moves on the opponent's side

This simply measures the difference between the number of legal moves on our side and the number of legal moves on the opponent's side. This feature is supported by the function MLGetMobility(), which returns the number of legal moves, given a board configuration and the current player (see section 4 for more details on optimizations).

4-Corners Differential

4-Corners Differential = number of 4-corners on our side − number of 4-corners on the opponent's side

Corners are considered to be good moves in the Othello. Not only are they stable (described later), but they also provide strategic significance for future flips. The 4 corners are defined as the top-right, top-left, bottom-right and bottom-left cells.

The 4-Corners Differential simply measures the difference between the number of 4-corners that we have and the number of 4-corners our opponent has.

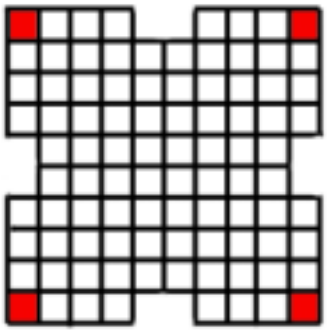

FIG. 1: 4-Corner Squares

8-Corners Differential

8-Corners Differential = number of 8-corners on our side − number of 8-corners on the opponent's side

Similar to the 4-corners, 8-corners are also considered to be good moves in the Othello game. 8-corners are defined as the following:

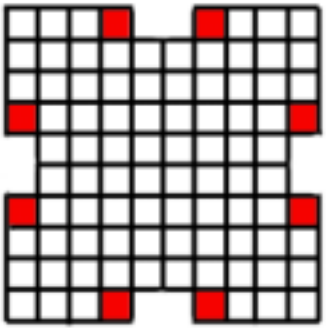

FIG. 2: 8-Corner Squares

We decided to distinguish the 8-squares from the 4-squares due to differences in diagonal control and mid-board play.

The 8-Corners Differential simply measures the difference between the number of 8-corners that we have and the number of 8-corners of our opponent.

C-Squares Differential

C-Squares Differential = number of C-Squares on our side − number of C-Squares on the opponent's side

C-squares are squares directly next to the 4-corners on the board (but not diagonally adjacent to). C-squares can potentially be good or bad. Since they are directly adjacent to corner squares they may allow opponents to grab the corner squares. However, they are also on the edge of the board, which means that they have less opportunity to be flipped. They can then setup to flip an entire row or column of enemy discs. C-squares can potentially play an important role in Othello games.

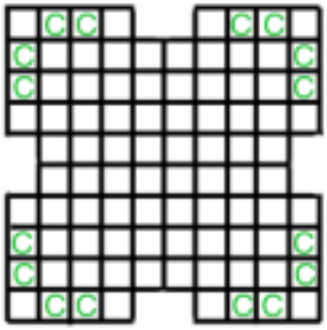

FIG. 3: The C-Squares

The C-Squares Differential simply measures the difference between the number of C-squares on our side and the number of C-squares of our opponent.

X-Squares Differential

X-square Differential = number of X-squares on our side − number of X-squares on the opponent's side

X-squares are squares directly next to the corners on the board. They are considered bad in our evaluation, since occupying X-squares tends to make it easy for the opponent to capture the corners. X-squares are considered to be undesirable moves by many Othello experts and we too, are assigning a negative weight to this feature. The differential simply measures the difference between the number of X-squares occupied for both sides.

The X-squares:

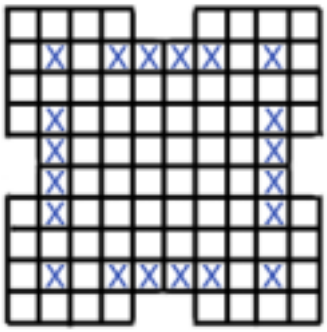

FIG. 4: The X-Squares

Frontier Differential

Frontier Differential = our Frontier measure − the opponent's Frontier measure

Frontier is a measure of the number of our pieces adjacent to an empty square in any direction (horizontally, vertically or diagonally). The more pieces we have on the frontier, the less likely we will increase our mobility. If, for instance, we surround our opponent entirely, we have no mobility, but the opponent has significant mobility. While frontier squares are certainly linked to mobility issues, they can actually distinguish between positions where the mobility measure fails to see a difference. Whereas mobility looks only at the moves that are possible starting at a given board configuration, frontier squares also mark off squares that are not directly liable to be captured at the moment, but run a high risk of being captured later in the game. In a sense, frontier squares counting helps refine the accuracy of the mobility estimation.

Stable Pieces Differential

Stable Pieces Differential = number of our stable pieces - number of the oppponent's stable pieces

Stable Pieces are simply the pieces which cannot be flipped by the opponent and cannot be flipped in the future. Unlike measuring a simple piece difference, the stable piece differential gives a more accurate measure of the material advantage of a given side during the game. In Othello, common strategies include minimizing the number of pieces relative to the opponent early in the game; however, such strategies overlook the importance of obtaining squares that will stay for the rest of the game. Ideally, the program should be able to recognize when it is grab pieces early in the game and when it is not.

Sandwich Squares Differential

Sandwich Squares Differential = number of our sandwich squares − number of the oppponent's sandwich squares

Sandwich squares are squares which are trapped between two of the opponent's pieces, horizontally, vertically or diagonally. The higher the number of sandwich squares in the board, the more favorable the board is, since there are more moves with which we can flip the opponent's pieces. Furthermore, it is less likely for our piece to be flipped.We expected that the evaluation function would give positive weight towards boards with a higher number of sandwich squares early in the game. Instead it turned out that this feature was given a strong negative weight. While researching Othello strategies, we later found out that these sandwich squares are commonly referred to as "wedges" in certain Othello literature.

Wipe-out Avoidance

We have also included a special case in the evaluation function to penalize a complete wipe out. If the number of our pieces is zero in the board, the evaluation function will simply return negative infinity. The situation of having all the pieces being completely wiped out might seem really good from the frontier minimization standpoint, thus we need to include this special case to avoid this from happening. We came across this idea when reading a paper by Jimmy et. al. [5]

3. SEARCH STRATEGIES

In the game of Othello, even an extra 2-ply look ahead over the opponent's search could make a huge difference in the outcome of the game. In this section we discuss various startegies to boost our look ahead depth and correctly distribute our time to effectively maximize our score. We also consider pre-computation strategies and a history heuristic to improve our in-game performance.

Alpha-Beta Pruning

We improve upon the normal alpha-beta pruning by employing a more efficient algorithm that we came across during research. After running many tests and evaluation games, we ended up selecting MTD(f).

NegaScout

NegaScout is an enhanced version of Principal Variation Search which makes use of restricted α-β window sizes to achieve a speed-up in search performance. In this algorithm, move ordering attempts to find the best line of play when evaluating a game node in the tree. After searching the "principal variation" with a normal α-β window, searches for later variations are performing using null windows (windows of "zero" size for which $0 < \beta - \alpha < \epsilon$ for some small value of ϵ). Null windows restrict the range of the search considerably and allow the program to quickly identify moves that lead to worse positions quickly. When a β-cutoff occurs for a variation

other than the primary line of play, then the program must perform a re-search with increased window size.

NegaScout is a frequently used algorithm for Othello programs, including Logistello. Sample code is provided at http://www.zib.de/reinefeld/nsc.html. We eventually discarded NegaScout in favor of a superior algorithm, MTD(f).

MTD(f)

MTD(f) is a minimax search algorithm simpler and more efficient than its predecessors, including NegaScout. The algorithm calls a version of alpha-beta search that stores its nodes in memory as it has determined their value, retrieving these values in subsequent searches. This is achieved in our program by calls to SmarterSearch and the transposition table, which covers the overhead of search tree reexploration.

Instead of using a wide search window as with conventional alpha-beta searches, MTD(f) performs repeated searches with windows of zero size, using each return value as an upper or lower bound on the minimax value. When the bounds converge, the minimax value and corresponding position are returned. Iterative deepening improves MTD(f)'s efficiency by providing initial guesses of the minimax value from the results of previous searches.

Less than 15 lines of code were necessary to incorporate MTD(f). More information about MTD(f) is available at http://cs.vu.nl/∼aske/mtdf.html.

Multi-ProbCut

In one version of our project we implemented Multi-ProbCut, a stochastic estimator for forward cutting search trees at multiple levels [3,4]. To do this we programmed a linear regression utility to determine the estimate for the least-squares correlation between searches of different depths at various stages in the game. Using these statistics, we could selectively prune branches of a search tree using reduced depth searches and relying on the least-squares fit line to predict the result of the deeper search in order to determine if a cutoff would occur with a high enough probability. Given a small enough variance in scores, then future scores could be predicted with relatively high probability.

After running Multi-ProbCut (MPC) with multiple parameters, we found that our implementation was not yielding the correct cuts. There was no significant boost in ply searches either. We are unsure whether the failure was due to a lack of sufficient statistical data (each MPC parameter was based on 750 sample boards due to development time constraints), instability of our evaluation function, or incorrect implementation. However, due to the success of MPC to elevate game play in programs such as Logistello, we still consider MPC a worthwhile method to look into even though it did not work for us. After testing our version against our previous versions we decided to "probCut" ProbCut.

Quiescence Search

We attempted to incorporate quiescence search into our search mechanism, but the results were not satisfactory. The essence of quiescence search is to dynamically increase the search depth when the search reaches certain points which we regard as unstable. This is to avoid the horizon effects which might lead to unreliable search results. One strategy we tried is to search a further 2-ply down when the score of the board at the end of search differs by more than a certain threshold than the score of the board 2-ply up. The reasoning is that when the difference between the scores is too big, the position we reached in the search is not a *quiet* position and the search result might not be reliable. We search further, trying to reach a position which is relatively more stable.

Another approach we tried is to dynamically increase the search depth when we hit certain moves. For example, when we hit a corner move i.e. when we place a disc on a corner, we choose to search further down the tree. The reasoning behind is that corner moves are critical in the game of Othello, and we want to see if the opponent is intentionally sacrificing a corner position to capture another position which is more valuable. We want to dynamically increase the search depth to see if this kind of behavior is occuring.

Neither approach resulted in any significant improvement in our search mechanism. In some cases, it even weakens our client, the reason we believe being that quiescence search wastes time and decreases the amount time we allocate for later searches. We tried different variations like limiting the quiescence search depth and changing some other search parameters, but the result was not satifactory.

Another strategy for quiescence search in chess is to increase the search depth when the number of moves for a particular side is 1. Note that this does not work as well in Othello since there tend to be many moves for a particular side during the game at most times (provided both sides do a decent job of maximizing mobility).

Time management

The advanced features used in time management include a branching factor time estimator for each depth level of a search, dynamic scheduling, and a deep endgame search with greedy evaluation, in addition to traditional iterative deepening. The client usually finishes with over 40 seconds remaining.

Scheduling

Each move has a maximum time limit based on the current number of pieces on the board and total time remaining. This limit is reevaluated after every turn. Due to the success of the branching factor time estimate, each move almost always has more time allocated to it than the last. The scheduler equally partitions the time between all remaining moves except for the end-game, in which it gives the greedy search slightly more time to solve the end-game configuration.

Iterative Deepening

For each move, we use iterative deepening to calculate the optimal move for successive depth levels, taking the value returned by the search of highest depth that is fully completed. Before searching a new depth, the time is checked to see if the time has been exceeded. The first 5-6 plies of each search find numerous hits from the transposition table, and thus finish almost instantaneously. Recursive calls to the search process do not execute any time-checks, which improves our efficiency while searching.

Branching Factor Time Estimation

The average mobility in the leaf nodes is likely to be highly representative of the branching factor for the next evaluation level of the tree. We multiply the time estimate by the square root of the branching factor since α-β on average (given perfect move ordering) tends to result in a square-root reduction in effective branching factor.

Thus, using the sum of mobility scores from our evaluation function, running total of nodes evaluated, and the time taken by the search at current its depth, we calculate a branching factor and use it to predict the running time of the next depth level, as follows:

$$branchingFactor = \frac{\sum_{nodesEvaluated} Mobility}{nodesEvaluated}$$

$$estimatedTime = \sqrt{branchingFactor} * levelTime$$

For the milestone, we used a basic assumption that a search of depth N+1 will take longer than a search of depth N for all nontrivial search times. Iterative deepening was thus halted whenever the time remaining for the move was less than the time taken by the search of the most recent depth, saving some time for each search. However, searches of high depth still tended to run and abort frequently, consuming time with no benefit. Using estimated time from the above calculations, the program cancels all searches that are predicted to abort.

Since our final Othello agent averaged 40 seconds left on the clock after a game, we adjusted the parameters to test different time allocations given to the search process. We found that allocating more time did not present the search with enough time to finish another ply depth and used up time unnecessarily. However, we did not test this with our time-abort mechanism turned off. The extra time, then, would not have been wasted, but it is highly likely that the time usage would have become less efficient overall.

Endgame Greedy Search

The endgame greedy evaluator starts when 14 or fewer empty spaces exist on the board, i.e. when about 14 plies remain in the search. Since the goal of the game is to win by as many pieces as possible, the greedy evaluator scores positions solely by piece differential. During the endgame, each step is allocated 50% extra time to search more deeply, as the impact of each move is signficant in determining the final outcome. A comparative advantage of just 2-3 depth levels over an opponent's endgame search is often enough to turn a close loss into a comfortable win.

By allocating almost all of the remaining time to the move at the beginning of the endgame search, the client can extend the endgame search by a few plies. However, this cramps the time of subsequent searches and sometimes comes dangerously close to running out of time, so we decided not to incorporate this under tournament conditions.

Transposition Table

The purpose of the transposition table is to exploit the information from previous searches. Much like cache in computer systems, the effectiveness of transposition tables makes use of the fact that board configurations searched now will probably also be searched in the near future. Before actually searching and spending time to evaluate a given board position, we first check to see if that board configuration is stored in the transposition table; if it is found, we will go ahead and use the best move stored in the table if the entry in the table has a depth

equal or greater than that desired by the search. This saves the program the trouble of having to recursively search and evaluate different board configurations.

The transposition table is implemented as a hash table of a fixed size. We originally stored all entries of the hash table on a priority queue so that we could sort all nodes in terms of least recently used access. However, each update required an update of the priority queue, which required many calls to *memcpy*. Instead, we ended up selecting a simpler, more efficient hash table where no update costs were necessary. The hash function we chose utilized bit shifts and masks to mix bits quickly [6]. As you will see in the table presented later in this section, it was also effective in distributing the entries (in the table, less than one percent of the buckets are left empty). Coincidentally, after implementing this hash function, we discovered that last year's Othello coompetition winners had also used this integer hashing method [5].

To find the appropriate size of our transposition table, we generated somewhat intelligent (ply search depth of 4 with random elements) game boards for various stages of a game. With 1000 boards, we did a 4-ply search on each board with the transposition table turned on. We varied our hash functions (not shown in table) and transposition table sizes. The following data was run on a Pentium 4, 2 GHz computer running Windows XP through Cygwin. Some important data is produced below:

Transposition Test Data			
Table Size	Bucket Size	Time to Complete	Bucket Distribution
65532 buckets	4	12 min 8 sec	All buckets full
2097152 buckets	4	2 min 3 sec	empty:20141, 1:83141, 2:260851, 3:528187, 4:1548334
1048576 buckets	4	2 min 4 sec	empty:106, 1:786, 2:4086, 3:13542, 4:1034928
1048576 buckets	2	8 min 20 sec	empty:411811, 1:843926, 2:2938567
1048576	8	3 min 15 sec	roughly normal distribution with mean at 4

The first row shows that table sizes too small take a large hit because either a lot of useful boards are thrown out before we can use them or there are too many replacement searches. As expected, by increasing the number of hash buckets and keeping bucket sizes small, we can reduce search time drastically to improve our performance. Experimentally, table sizes of 2097152 were too large because it would require a total of 368 megabytes of memory on the stack. We reduced our table size to 1048576 which gave us stable and fast performance on the Elaines. Keeping a small number of buckets also helped reduce search time because there is less linear searching within the bucket.

History Heuristic

The history heuristic holds a 10x10x2 table, H which contains one entry for each square of the board per side. Every time a search of depth d (at any level in the recursive search process) reports that a particular move (x, y) is good for a particular player p, then we update the history table by the update rule: $H[x][y][p] \leftarrow H[x][y][p] + 2^d$. This gives higher weights to moves that have historically been good for a particular side. These history scores are then used in move ordering.

Opening Book

By precomputation we can list out probable board configurations for games of depth 6 (determined by our previous Othello clients). From this we can apply a 10-16 ply search and store the results in an opening book. When we load our Othello client, we load in the opening book into a special opening book that is only searched in the first few moves. This gives the Othello client a large time advantage in the beginning since it can make 8-ply decisions in just the time it takes to do a lookup.

4. BIT BOARDS AND OPTIMIZATIONS

We decided to re-write the board structure in a manner that was more efficient for hashing and board evaluation. Since there are 100 total squares (including invalid squares) on the board, we chose to bit-pack the boards using four longs ($4 \times 32 = 128$ bits). This could have been done with one fewer long, as only 92 squares are relevant, but we decided to use 4 for simplicity's sake. Each "board type" would store this bit-packed board representation (one for pieces played and one for side information), the number of pieces for each side, and the player's turn for that particular board. The board that stores player side information stores a 0 for all white's pieces and 1 for all of black's. The board that

keeps track of filled cells sets a 1 in every filled cell and a 0 everywhere else. We later added depth and score for the transposition table.

There are a number of big advantages to using this bitboard representation. First, in move generation, we can efficiently calculate all possible moves by a bit shifting technique described below:

- To initialize the algorithm we may extract bitboards containing only our pieces and only our opponent's pieces.
- We can also find the empty squares by complementing the bitboard that stores the positions filled.
- Create a results bitboard to hold the results of the search
- For all eight directions (up, down, left, right, up-left, up-right, down-left, down-right) do the following:
  1. Make a copy of `ourPieces` into `temp`
  2. Move all pieces of `ourPieces` in the direction given. This may be accomplished with bitshifting.
  3. Check that an opponent's piece is found
  4. Now, continue moving pieces in that direction until an empty square is found (in which case a move exists), our piece is found (no move exists), or the edge of the board is hit (no moves exist).
  5. If a move is found, add it to our results bitboard

The following is a simplified version of the bitboard move generation technique we used:

```
if (ourSide == 0)
   ourPieces = posFilled & ~posSide & BOARD-MASK
   oppPieces = posFilled & posSide & BOARD-MASK
else
   ourPieces = posFilled & posSide & BOARD-MASK
   oppPieces = posFilled & ~posSide & BOARD-MASK
empty Squares = ~posFilled & BOARD-MASK
for(direction = 0; direction < 8; direction++)
   temp = ourPieces
   temp = temp << DIRECTION-SHIFT[direction]
      & BOARD-MASK
   temp = temp & oppPieces
   while (temp ≠ 0)
      temp = temp << DIRECTION-SHIFT[direction]
         & BOARD-MASK
      result = result | (temp & emptyPieces)
      temp = temp & oppPieces
```

Using bit masks and bit manipulation, we can accomplish the board shifting and checking for all cells at the same time! This gives us an efficient method to generate valid moves.

Another feature of the bit board implementation is an efficient mobility evaluation. From the algorithm above we already know how to get moves quickly and accurately. For a given board we can calculate the possible moves for a particular player and use an efficient bit-counting technique to sum up the mobility factor for a given player. The algorithm for counting the number of bits that are set to "1" is given below (where *Number* is the bit string):

```
while (Number)
   Number = Number & (Number - 1)
   count++;
```

Each iteration takes out the least significant bit that is a 1. The total run-time is O(T), where T is the total number of bits that are set to 1. We also did a mask check and lookup to efficiently find the location of a 1 in a bit stream.

The bit board implementation also give us a good estimation for stability. First, we set the boundary to be stable. Then, using the shifting method, we can check each cell in parallel for all cells that are stable in four directions (column, row, diagonal-negative-slope, diagonal-positive-slope). It is considered stable in a given direction if there it is adjacent to a stable disc in that direction. We loop until no more stable pieces are found. Some pseudocode is given below:

```
if (ourSide == 0)
   ourPieces = (posFilled & ~posSide) |
      ~BOARD-MASK
else
   ourPieces = (posFilled & posSide)
      | ~BOARD-MASK
stablePieces = ~BOARD-MASK
newStablePieces = 0
do
   stablePieces = stablePieces | newStablePieces
   newStablePieces = newStablePieces &
         ((ourPieces << UP) | ~BOARD-MASK) |
         ((ourPieces << DOWN) | ~BOARD-MASK)
   newStablePieces = newStablePieces &
         ((ourPieces << LEFT) | ~BOARD-MASK) |
            ((ourPieces << RIGHT) | ~BOARD-MASK)
   newStablePieces = newStablePieces &
         ((ourPieces << UP-LEFT) | ~BOARD-MASK) |
            ((ourPieces << DOWN-RIGHT) | ~BOARD-MASK)
   newStablePieces = newStablePieces &
         ((ourPieces << UP-RIGHT) | ~BOARD-MASK) |
            ((ourPieces << DOWN-LEFT) | ~BOARD-MASK)
   newStablePieces = newStablePieces & BOARD-MASK
```

until newStablePieces = 0

This method is only an estimation, however, since it will miss cases when a stable piece is sandwiched between opponent pieces that make it stable. Consider the following two examples:

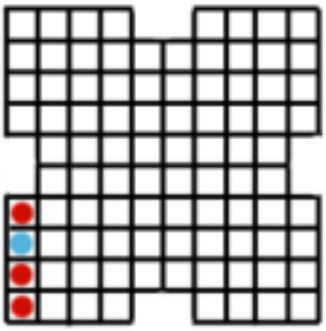

FIG. 5: Stability Example 1

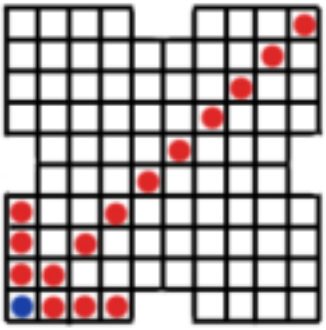

FIG. 6: Stability Example 2

In the first example, the white (blue-ish) piece is stable because no red discs can flip it. In figure 2, the piece diagonally up-right from the blue corner disc is stable as well. However, since we check for stable pieces of the same color, we do not catch this case. One way to calculate actual stability, then, would be to consider all rows, columns, and diagonals to check for filled rows and columns. Then if a particular row, column, or diagonal is filled, then stable discs are stable for any color in that given direction. If a cell is stable in all directions, it is considered stable (remember, there are 4 directions, up/down, left/right, positive sloped diagonal, negative sloped diagonal). At the time of programming, this computation seemed too expensive for quick evaluation purposes. However, if we were to do this project again, we might consider calculating true stability by using bit masks to find filled rows, columns, and diagonals. From there we can set universally stable discs and find true stability.

There is also a stability optimization that we forgot to include when finalizing our project. Since stable pieces, by definition, are stable, we can store previously calculated stable pieces and calculate new stable pieces incrementally. This would give us a greater speedup in evaluation, especially in the end where nearly all pieces are stable.

Frontier squares and sandwich squares are calculated similarly to the methods described above.

Bit board representation also gave us an easy and efficient way to translate tables and mask out certain regions using bit masks and look up tables. Overall, the bit board implementation alone gave us at least a 4 times speed up.

5. TRAINING

We implemented method #2 given in the assignment handout but found that the program generally did not converge to values that were reflective of the true "correct" values for the weights. It also seemed problematic because it would select the "true" values based on guesses from its untrained weights. The final program uses weights corresponding to 87 stages of the game as described below. The method used is based on one described by the winners of the CS 221 Othello competition two years ago[1]. Essentially, the game is divided into stages, and the weights for the stages are trained in reverse order from the end of the game towards the beginning. This ensures that the target evaluations are trained or actual (pure greedy at the end).

Stage Training

We divided the game into 88 stages ranging from when there are 5 pieces on the board to when there are 91 pieces on the board. Our evaluation function never needs to evaluate a board with only 4 pieces, and the evaluation for when all 92 pieces are on the board was chosen to be the piece difference between the evaluating player and the opponent.

Training Data

For each stage, we constructed 5000 training boards by playing the program against itself using a 4-ply greedy search. The boards were not checked for uniqueness, but we did introduce randomness in the move selection to ensure variety in the games. The 4-ply search would return a list of n possible moves ordered by decreasing expected value. The probability $P(i)$ of choosing the ith best move was set by $P(i) = 10P(i+1)$ and the sum of the probabilities set to 1.

More realistic boards could have been generated by a "bootstrapping" method in which the trained weights were used instead of the greedy player to create the boards; furthermore, the probabilities of selecting each move could be chosen as a function of the expected value for the move. Using the greedy player to generate moves was done for the sake of speed; this was not necessarily a bad choice, however, since the Othello brain should be able to handle positions against any type of player, not just boards selected to be "good", we decided that this approach would suffice.

Training Procedures and Failed Attempts

This section will describe the various methods tested to train our Othello program. The first method is a trial using the learning technique described in the Othello handout. The second is a neural network that groups our evaluation features into different categories and introduces two hidden layers. The final method is the method we eventually selected to be submitted in our final version.

Method 2 in handout

This training method looked at the current state, an opponent's move, and based on the predicted evaluation values trains the current weights to fit the predicted value. As noted by the handout, this would only work if each player selected its optimal move. There are a few problems with this assumption. First, the opponent we are competing against may not have trained weights either (which was the case when we started) and it might not take the optimal move. Second, the best we can train to is highly dependent on how effective our opponent is. If our opponent is not strong, it is unlikely that we will train to be stronger. Third, to train for the general case game (not simply for a single agent), we need to have many well-trained opponents to train against. If we use the same opponent over and over, we will most likely overtrain our agent to only defeat our training opponent. After 1000 iterations through this training method the weights still did not converge. In some cases, our original untrained player would defeat our "trained" player. This led us to seek for a training method that would train against "true" target values.

Neural Network

In designing an appropriate neural network topology for the Othello client, we created a scheme with four general categories of inputs which we chose to group as follows:

1. Positional: corner squares, X-squares, C-squares
2. Configurational: stability, frontier, sandwich squares
3. Temporal: mobility, board parity
4. Tactical: piece differential

We then used back-propagation to train the weights at each stage of the game (where stages are described above). The output layer in the last stage is simply the greedy piece differential output at the end of the game. In all other stages the target output is taken to be the evaluation 4 plies down using the trained weights from the next stage (trained before since we go from end-game to begin-game).

Linear Combination

The scheme we initially used was very similar to the method described in an earlier paper by McAlister and Wright [1]. The last stages of the game were trained first by performing stochastic gradient descent using the piece difference evaluation as the target output. Successively earlier stages were then trained by using the results of a 4-ply search based on the already trained weights. This method gives the advantage of using trained weights for establishing the target for convergence.

In our interpretation of the method, the difference between the target score and the one predicted using either piece difference or the trained stages of the evaluation function was taken as an indication of the correct direction for moving. We trained using different parameters and found learning rate values optimized for our implementation.

We also tried splitting up the weights into weights for the red player and wieghts for the white player. We thought that the strategies of both sides may be different enough to warrant different weights. However, both weights must be trained at the same time since the weights at one stage of the game for the red player depends on the weights at the next stage for the white player. Hence, simultaneous training was required and there was no need to train separately. In fact, when trained separately, our client consistently lost to the client where both sides' weights were trained together. In the end we concluded that the weights were not sufficiently different as changes in parity during the game make it nearly impossible to distinguish between the appropriate weights for the two players. Thus, our final trained version uses only a single set of weigthsf for each stage.

Our Training Strategy

By far the greatest obstacle in training our data was convergence. The normal gradient descent method posed

some major drawbacks in terms of training speed and quick convergence. Of our numerous attempts to arrive at ideal convergence, we found that a small learning rate would take many iterations to account for large disparities between target and predicted weights. On the other hand, large learning rates would miss small disparities between the target and predicted values. For instance, if we are given weight w_1 that starts out at 8.7 and weight w_2 that starts at 5.4, and the ideal weights for w_1 and w_2 are 556 and 5.8 respectively, then a learning rate of 0.05 (which we found good for the fine-tuned detail in our implementation) could take hours for w_1 to converge at 556 over a 5000-sized data set. A learning rate greater than 1 may miss the ideal weight for w_2 altogether.

What we ended up implementing to solve this problem is an algorithm that is similar to performing a line search (using the Golden Section method [2]) after batch gradient descent. In our approach, we set the learning rate to be very low (in our case 0.02) and then perform a gradient descent over all 5000 boards in the training set for a particular stage. We compute the total change in each of the weights following this training pass. Then, instead of simply just iterating until convergence as done in the method by McAlister and Wright, we attempted to see the effects of doubling the changes in all weights, quadrupling the changes, multiplying the changes by a factor of eight, etc. For each attempt, we compute the total sum squared error of the predictions on the training set. We keep doing this until we go too far in a given direction and the overall error rate increases rather than decreasing. At this point, we step back a factor of 2 and begin our individualized descent. This process is similar to the above process except here, we consider only the effects of adjusting individual weights rather than all at once. This allows us to converge extremely quickly in only a few iterations through the training data. As a result, we were able to train all 88 stages to convergence in less than one and a half hours. To note the effectiveness of this method, we must note that all weights converged and that for no stage was the training process aborted due to using too many iterations of training.

Some pseudocode is presented below:

```
for (stage = 87; stage ≥ 0; stage–)
  weights[stage] = weights[stage + 1]

  for (board = 0; board < NUM-BOARDS; board++)
    target[board] = PerformSearch (boards[board],
      weights,LOOK-AHEAD)

  for (i = 0; i < MAX-CONVERGENCE-STEPS; i++)
    oldweights[stage] = weights[stage]

    // gradient descent
    for (board = 0; board < NUM-BOARDS; board++)
      for (j = 0; j < MAX-INDIV-CONVERGE-STEPS; j++)
        result = Evaluate (boards[board], weights)
        for (k = 0; k < NUM-WEIGHTS; k++)
          weights[stage][k] += LEARN-RATE ×
            (target[board] - result) × values[k]

    // group speculative jumping
    deltas = weights[stage] - oldweights[stage]
    currError = GetError (boards, weights)
    while (true)
      weights[stage] += deltas
      newError = GetError (board, weights)
      if (newError > currError)
        weights[stage] −= deltas
        break          else
        currError = newError
        deltas ×= 2

    // individual speculative jumping
    max = 0
    for (j = 1; j < NUM-WEIGHTS; j++)
      if (abs(deltas[j]) > abs(deltas[max]))
        max = j
    deltas = weights[stage][max] - oldweights[stage][max]
    currError = GetError (boards, weights)
    while (true)
      weights[stage][max] += deltas[max]
      newError = GetError (board, weights)
      if (newError > currError)
        weights[stage] −= deltas[max]
        break
      else
        currError = newError
        deltas[max] ×= 2
    if (Converged(weights, oldweights)) break
```

Here is a chart that shows the importance of each feature over each stage of the game:

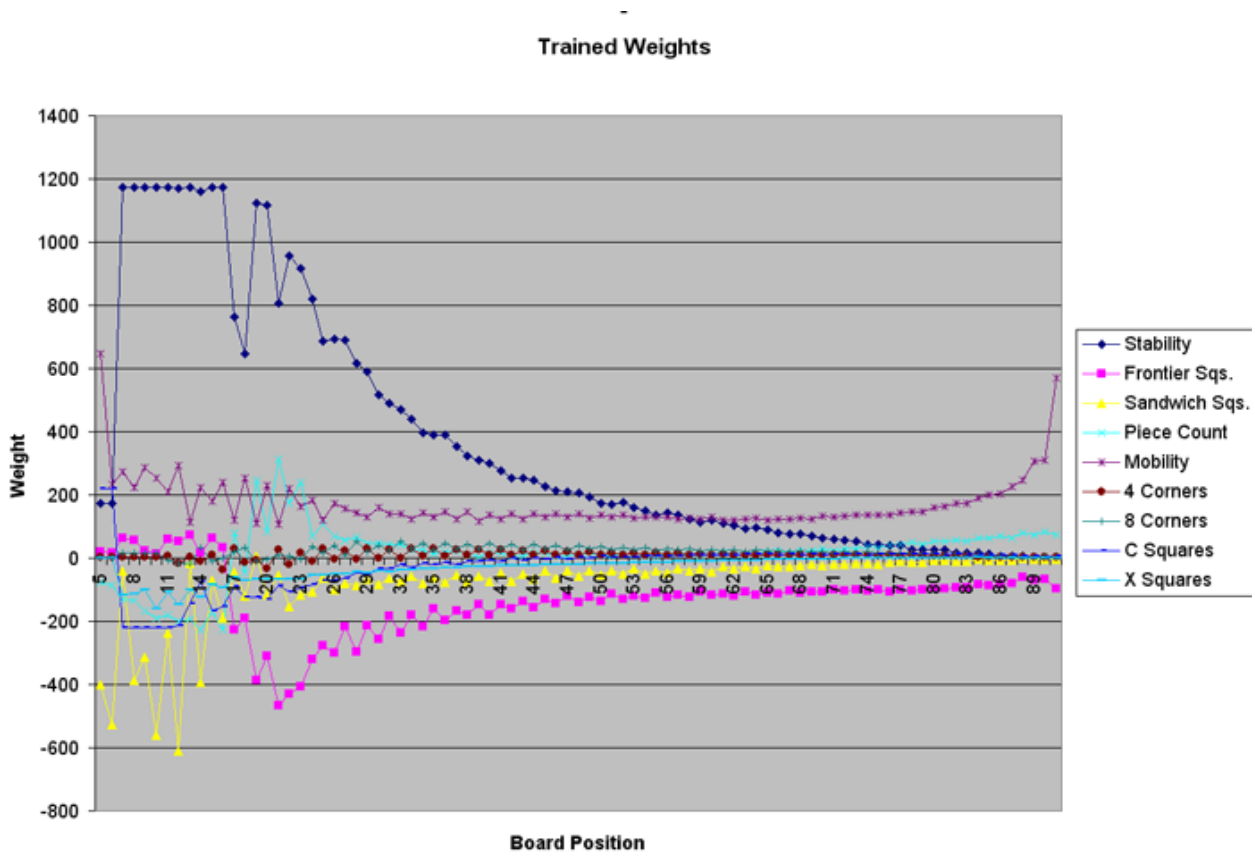

FIG. 7: Chart of Trained Weights

We noticed that around stage 16 some of our trainined weights were somewhat erratic. We think this is due to the fast convergence of the method described above, and it could actually be the case that we overtrained those weights, essentially memorizing our sample data.

We varied *MAX-CONVERGENCE-STEPS* and *MAX-INDIV-CONVERGENCE-STEPS* and found that values for *MAX-CONVERGENCE-STEPS* did not really matter because the algorithm converged very fast. If we are given a data set of many boards, *MAX-INDIV-CONVERGENCE-STEPS* also does not need to be large because there is a lot of data that allows the training to converge quickly. For a smaller data set,*MAX-INDIV-CONVERGENCE-STEPS* is important because it greatly affects the learning rate. We believe that our success in the Othello competition is due to the training strategy used and the features chosen, especially since many opponents actually sought deeper per move.

Since we set our lookahead for training to be of depth 4, we noticed that our weights were four-periodic (i.e. the weights for stage i are based on the weights for stage $i+4$). In the future it might be better to take a combination of different look aheads. Also, at a cost of greater training time, we could use a larger look ahead to increase performance.

FUTURE WORK

We ended up discovering our training algorithm too late in the project to have time to train our neural network. Using the doubling strategy we might have been able to get our neural network to learn the game of Othello.

A relatively difficult feature to do efficiently would be to include region parity in our evaluation function. This would require the agent to recognize empty regions on the board and quickly count the number of empty cells in that region.

Rather than using linear combination of the features we could instead use a third order polynomial function for each feature (whose coefficients would be determined by training). Given the fast convergence of the training method, this should be feasible. Early experiments with quadratic functions looked promising. These experiments show strong quadratic correlations.

REFERENCES

[1] Jon McAlister and Daniel Wright. Othello paper. *http://bigmac.stanford.edu*

[2] Talked to Professor Ng after the project to find out if there was a name for our training method.

[3] Buro, M. "ProbCut: An Effective Selective Extension of the Alpha-Beta Algorithm". ICCA Journal 18(2) 1995, 71-76. *http://www.cs.ualberta.ca/∼mburo/publications.html*

[4] Buro, M. "Experiments with Multi-ProbCut and a New High-Quality Evaluation Function for Othello" Games in AI Research, H.J. van Herik, H. Iida (ed), ISBN. 90-621-6416-1,2000. *http://www.cs.ualberta.ca/∼mburo/publications.html*

[5] Alvin Cheung, Alwin Chi, Jimmy Pang. "CS221 Othello Project Report: Lap Fung the Tortoise". *http://www.stanford.edu/∼hcpang/othello.html*

[6] Bit Shifting methods and Integer Hash Functions. *http://www.concentric.net/∼Ttwang/tech/inthash.htm*

---

* Electronic address: chuongdo@,chongs@,mktong@,huics@