CS 221 Programming Assignment:

Othello

ꗏ Team Desdemona ꗐ

Karen “Learningishka” Corby
Lilly “Hash-Meisterin” Irani
Patrick “O’Regan” Perry
Luke “Bit Boarder” Swartz

27 November 2002

Abstract

Our Othello program implements an optimized iterative deepening alpha-beta pruning search algorithm with cached states, using a combination of “complex” features (such as mobility approximation and corner heuristics) and “simple” features (patterns of discs), weighted using an online stochastic gradient descent learning algorithm with exponential summing. While the program had mediocre performance in tournament play, we experimented with a number of both previously documented and innovative techniques which, upon greater fine-tuning, could produce excellent results.

0 Introduction and Goals

Artificial Intelligence researchers and programmers have been trying to program computers to play the board game Othello since nearly the game’s “invention” (or, rather, refinement of the older British game Reversi) in 1971 (Buro 2002, Anderson 2002). Many advanced AI techniques can be employed on Othello, and its relatively small search space and complexity compared to chess make it possible to beat human opponents without massive computing resources. This paper describes DESDEMONA, a program that plays a modified version of Othello (with a 10 x 10 board with “chunks” cut out of the sides, as described in the CS 221 assignment handout). The program encompasses four main modules, each with a separate goal: the search algorithm (section 1), whose goal is to search through as many good board states as possible, the evaluation features (section 2), whose goal is to represent relevant aspects of board states, the learning algorithm (section 3), whose goal is to weight the features according to their relative importance throughout the game, and the simple features (section 4), an innovative approach to features and feature weighting, whose goal is to augment and/or duplicate the standard evaluation features and learning algorithm. We also detail, in section 5, what improvements we might make on the program.

1 Search

1.1 Basic Search Algorithm

The search algorithm is a standard minimax search, augmented by alpha-beta pruning (as described in Russell and Norvig 2002). As everyone was required to implement this for the milestone, we shall not go into detail about this baseline, except to note that alpha-beta pruning greatly improves performance by drastically cutting the number of nodes to be searched. Note that there are apparently some minor "improvements" on alpha-beta pruning, such as Nega-Max (mentioned in Anderson 2002), but we did not pursue these.

1.2.0 Time Strategy

DESDEMONA allocates the total time allotted uniformly across all moves. For example, if there are 100 seconds of play left, and 20 more moves to make, each move will be given 5 seconds of computation. When either player passes, the move count is updated accordingly. That is, every time the opponent passes twice, we have to make one more move, and every time we pass twice, we have to make one less move.

This uniform allocation is admittedly naïve, as moves towards the end of the game take less computation time, and as moves in the middle of the game are more important than moves at the beginning. A simple extension would be to give end and beginning moves less weight in determining the time allocation, thereby making sure that the moves deserving more computation time actually *get* more computation time. Another extension, described in Cheung, Chi and Pang (2001), is to "save" a certain amount of time for a final end-game search, allowing "perfect play" for the last few moves. A more ambitious extension would be to use machine learning to determine the timing weights rather than hand-tuning them.

1.2.1 Iterative Deepening

The goal of the move selection stage is to expand the game tree to the farthest level possible in the time allotted. Depth-first search (DFS) with a fixed look-ahead is surely not the optimal solution to the maximum depth problem, as it is likely to overstep or underutilize the computation time available. It is thus desirable to have search at variable depths, depending on the move itself. Breadth-first search (BFS) may seem to be a plausible solution at first, as the *entire* tree is expanded one level at a time. However, memory restrictions ($O(b^{d+1})$, where b is the branching factor, and d is the depth) prohibit using this strategy.

The strategy we chose, iterative deepening, combines the uniform expansion of BFS with the small memory requirement of DFS. The algorithm is as follows: First, use DFS to expand the game tree to depth 1. As long as we have time left, increment the depth by one, start from the root, and use DFS to expand the tree to the new, incremented depth.

The reader will note that nodes near the root of the tree are expanded multiple times. This may seem inefficient, but the difference in computation between search to depth d and searching to depth $d + 1$ is an order of magnitude—the repeated computation is insignificant relative to the total computation time.

1.3 Caching with Transposition Tables

In many board games, multiple paths can lead to the same board state. Because in Othello, the final piece difference determines a win without regard to the path taken to that state, game board states that are identical must have the same expected final value, given an evaluation function that remains constant for the duration of the game. Thus, it makes sense to cache board states along with the value they were found to evaluate to after some lookahead search so that if the same state is encountered again, the cached value can be used.

However, only caching board values for which we have found exact values at a given lookahead limits us. Using alpha-beta search, we would hope to be able to prune a great number of subtrees to speed up our search. Pruning a sub-tree, however, means that we cannot give an exact value to the parent node whose children were pruned. However, we can take advantage what little we do know – that the parent has *some* child with a value we have determined through search and application of our evaluation function. In a given search tree context, finding a board state cached with a non-exact value may still be enough to allow us to discontinue search down that path if:

1. We are the maximizing player and a possible successor is found to have a subtree valued lower than our current best option.
2. We are the minimizing player and a possible successor is found to have a subtree valued higher than our current "best" option (the option that would give the lowest value for the maximizing player.

To capitalize on these observations, our transposition table elements track boards for which we have an exact value, as well as the best subtree found for a node whose child(ren) is pruned and the distinction between the two types is maintained.

1.3.1 Hash Functions

We considered several options for hashing our board. We considered a good hash function to be **quick to execute**, and **uniformly distributed**. To satisfy the first constraint, we tried to come up with functions that avoid expensive operations such as mod, multiply, or divide. In order to achieve uniform distrubution, we began by trying to think about higher-level features than simply the bits in the board as a string, since we intuited that high-level features would distinguish between boards and thus yield greater distribution.

Our first idea for a hash function was to take advantage of our proposed evaluation function structure. We had planned to evaluate based on 12 features, and the hash key would be 12 concatenated bits where bit$_i$ is determined by whether feature$_i$ is above or below a certain threshold.

This hash function would be expensive to implement, however, because the evaluation function would have to be executed for every lookup.

In searching for alternatives, we found an approach used in chess called the Zobrist Key (Moreland 2001). The Zobrist Key keeps a three-dimensional array where each <board row, board column, player> tuple has a corresponding randomly generated 32-bit integer. A key is generated by beginning with 0 and XORing the random number corresponding to every piece on the board as described by the above tuple.

This key algorithm has several strengths. Because it is based on random numbers and the XOR operator, it achieves a broad distribution and the hash keys are unrelated characteristics inherent to the board. Also, XOR is a fast operation.

Also, while a naïve implementation would generate a key for each board by going through each space and accumulating the key, an efficient implementation can exploit the fact that (a ^ b) ^ b = a so that each board lookup only requires a single XOR. At the beginning of the Othello game, we generate a key for the given board start configuration with four pieces and pass that initial key to the search function. The search function simply XORs the appropriate random number r_1 for the piece added to the successor board before looking up the successor. In order to generate a key for the next successor, simply XOR r_1 again to "remove" the piece addition just explored and XOR the next appropriate random value to lookup the board state with the next possible key addition.

We used Quantify to profile our code and found that replacing the naïve implementation of key generation with the incremental method made one of our more time consuming methods one practically inconsequential.

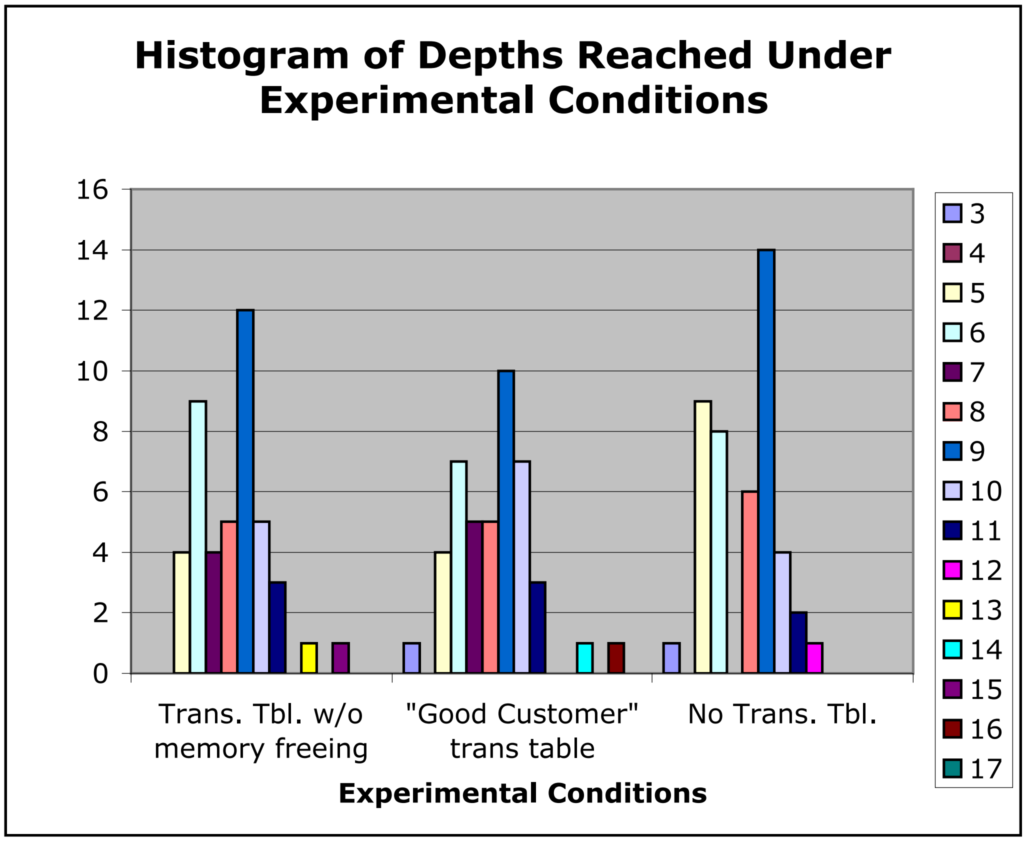
1.3.2 Significance of Transposition Table with Iterative Deepening

Because iterative deepening makes successively deeper searches, and we just restart each search instead of storing a fringe, we have to clear the transposition table after every increment in the iterative deepening search. Otherwise, the transposition table will provide the evaluation function value for a board instead of taking the opportunity to search further and obtain a more accurate estimation.

1.3.3 Systems Notes

We tried to implement the hash table as leanly as possible. We used STL and linked in a C++ module, having great faith in the powers of Bjarne Stroustrup. We tried to implementation strategies. The first strategy depended heavily on copying information off of the stack and involved very little allocation that wasn't already done by DoMove. Our hope was that this strategy would reduce the need for heap allocation calls and thus demand less run-time cycles and prevent memory fragmentation. This implementation also made tracking and freeing boards very difficult, however, and we hemmoraged memory

By profiling in quantify, we found clearing the board to be of significant overhead so we experimented with various memory management policies.



We tested the transposition table both with and without freeing memory. (We dub the memory freeing approach as the "good customer" policy.) We found that the transposition table that did not free memory reached an average depth of 8.205 on epic machines. Freeing memory we obtained average depth of 8.25 on epic and without a transposition table, we obtained an average depth of 7.5, gaining us about 1-ply of depth on epic. Epic machines are among the slowest in Sweet Hall so the benefit of transposition tables would likely increase with machine speed.

Our fear that freeing and allocating memory would come at the cost of search depth was false. Not freeing did not incur a significant difference, though we feared that it would cause paging and therefore a slowdown in the system. The client has about a 47 MB memory footprint on a machine with at least 256 MB RAM, so memory is not scarce.

1.4 Other Search Enhancements

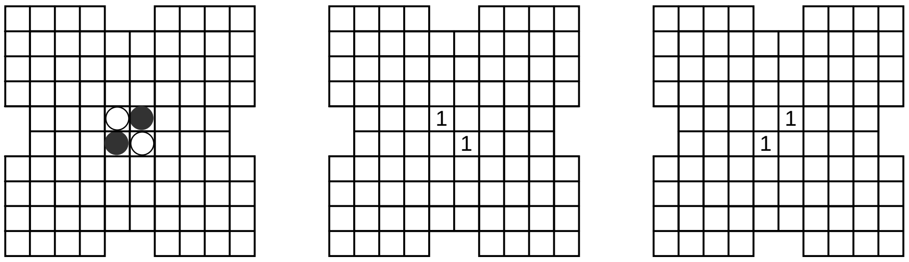
The search enhancements described above do not change the result of a board exploration, but instead only speed up search without *adding* uncertainty to our result. We explored another approach to pruning by Michael Buro (1995a and 1997b) called Multi-Prob Cut. ProbCut is based on the principle that, given some non-zero choice of a reasonable margin of error, one can use linear regression to find models predicting the value of a game state should it be searched to depth h by using the game state value at some depth $d < h$. Subtrees which appear to be bad enough that, with some probability, we would not choose them should we search them entirely are then probabilistically pruned.

This method posed a practical challenge because running a linear regression to discover suitable models requires a stable, reliable evaluation function. Though we researched this method in depth, we failed to stabilize our evaluation function soon enough to run a regression and test the method with our implementation.

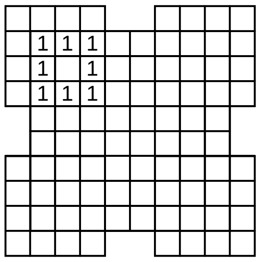
1.4 BitBoards and Optimization

The stock method of searching for valid board states (iterating over every square on the board, passing it to IsLegalMove) is fairly computationally expensive. Moreover, it overlooks a simple fact: all legal moves are in empty squares adjacent to one of the opponents' discs.

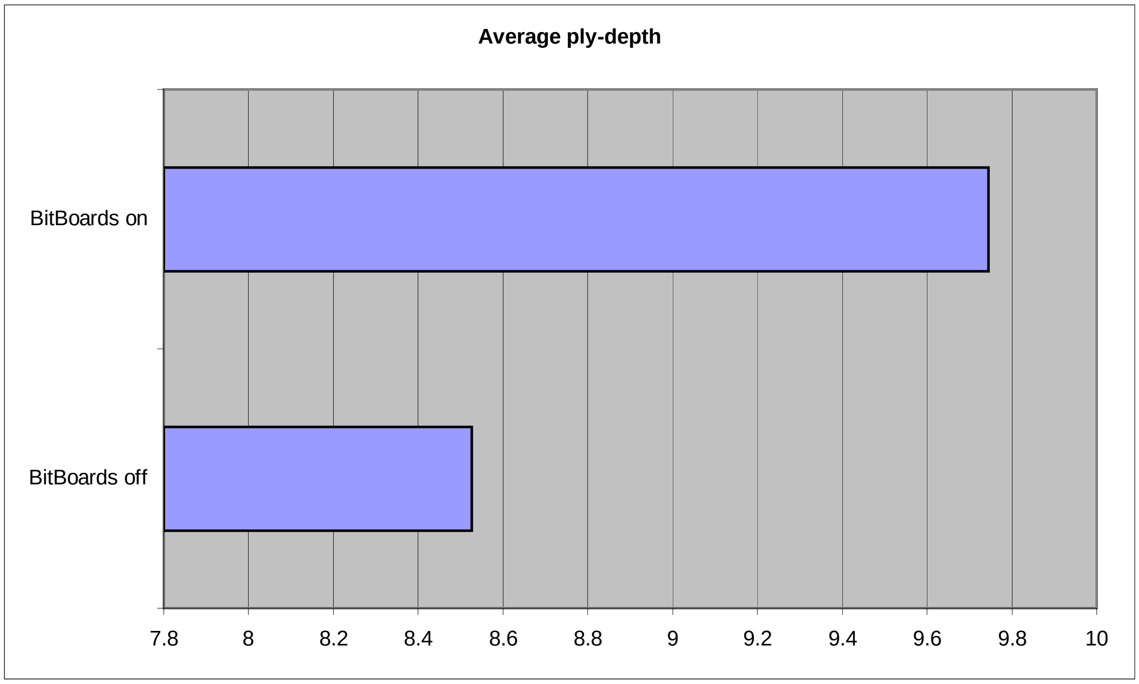
Thus, we implemented the BitBoard approach described in Cheung, Chi and Pang (2001), in which each board state is represented by two BitBoard structures (internally stored as arrays of 13 byte-length characters). Below, the board on the left would be represented by the two BitBoards on the right, representing the white and black discs, respectively. All other spaces are 0.



In keeping with Cheung, Chi, and Pang, we also pre-compute an array of AdjacencyTables, which store BitBoards of the adjacent squares to each square on the board. For example, the following is the AdjacencyTable for (2, 2):



Thus, the potentially legal moves are all to be found on the BitBoard represented by the union of AdjacencyTable(x, y) $\wedge$ $\neg$(RedBitBoard(x, y) $\vee$ WhiteBitBoard(x, y)) over all x and y. Not only does this speed up the search so as to allow over one ply extra search (see figure, below), but it is useful as an evaluation function metric in itself (described in section 2.2).



2 Evaluation Features

Given the search algorithm described above, what is the search searching for? Certainly *end*-game states should prefer high disc differentials (i.e. the "greedy" heuristic), as that is how one wins, but such a heuristic does not (necessarily) correspond with good game states in the early stages of the game, or even the mid-game. In fact, many human Othello strategy guides suggest that one should *minimize* the number of one's pieces early in the game.

2.1 Othello Evaluation Features for Humans

To get a sense for what game states are preferable in Othello, we consulted a number of strategy guides intended for human players (Feinstein, Lazard 1993, le Comte 2000, Mendelson 2001, Othello University). Each of them stressed three primary goals (beyond disc difference at the end of the game):

1. Mobility – One should maximize the number of moves one has, and minimize the number of moves available to one's opponent. Some of the guides stress using the "frontier" or "fringe" discs (i.e. discs that are next to empty squares) as an easy metric for mobility, as each move must be on an empty square next to an opponent's disc.

2. Stability – One should aim for "stable" (i.e. un-flippable) discs. The corners are the most obvious stable positions, upon which one can establish rows, columns, and/or diagonals of likewise stable discs.

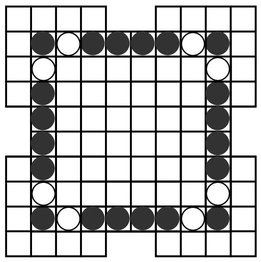
3. Parity – As the last player to put down a disc (both in a closed region and overall) flips last, one should favor situations in which one is the last to flip.

2.2 Implemented "Complex" Features

The submitted version of DESDEMONA includes eight "complex" features, each designed to model one or more of the human metrics (mobility, stability, and parity). We called these human-specified features "complex" to distinguish them from "simple" features, which are based on machine-learned patterns. As we went through several iterations of "simple" features, and they do not use the standard learning algorithm, they shall be described later (in Section 4).

1. Corners heuristic – We used a slightly more advanced version than merely weighting the corners; instead, we used the Cheung, Chi and Pang (2001) technique of summing the square of the number of corners in each quadrant. This is based on the assumption that it is better to completely own a quadrant's corners than to own disparate corners throughout the board—making this a good stability metric.

2. "Dumb" X Squares – X squares in Othello are those that are next to corners but not on an edge. In traditional Othello, thus, there are only four X squares; in our modified version, it depends on how one counts. We used the same X squares as Cheung, Chi and Pang (2001), that is, the black discs in the figure below.

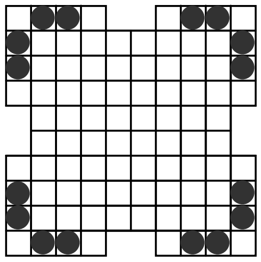


   In hindsight, we should have probably also included the white squares as well; however, by the strict definition of X square, the squares next to the "cutouts" (and thus, at least technically, on the "edge") perhaps should not be counted. We shall take up this dilemma again in section 4.

   We call this feature "Dumb" X Squares in keeping with Wang, Eisenberg and Vadera, as the metric is the number of X squares for which one does not already have the adjacent corner. Since playing in X squares when one does not own the corner usually allows the opponent to take that corner, this is usually weighted negatively, as owning the corner is usually good for stability.

3. "Smart" X Squares – This is essentially the same as the previously described metric, except that square are counted on if one *does* own the adjacent corner. This is generally a positive metric (as it enhances stability).

4. C Squares – C squares are squares adjacent to corners which are on the edge (see figure below). Because edge squares are often useful in stability in their own right, these squares intuitively are less dangerous than X squares, although they do endanger the corners.



Once again, there is some question about what should count as a C versus X square in our altered board, but note that in hindsight, whether one should play in a C square or not depends not on the stage of the game (or even if one owns the corner) but rather on the entire edge situation, making this a less-than-optimal metric. We shall return to this dillema in section 4.

5. Stability Edge Heuristic – In attempt to compensate for potentially poor edge play, we adapted a technique from McAlister and Wright (2000) in which one sums up the number of discs one has in a row in each edge, and adds the number of discs in a row in the adjacent rows/columns. This did not seem to vastly improve edge play, we believe largely because of the dilemmas mentioned regarding the X and C squares.

6. Potential Mobility – The "potential legal moves" speedup described in section 1.4 allows one a very simple metric for measuring each side's potential mobility: simply calculate the potential legal moves (i.e. how many blank squares there are next to the opposite side's squares), and consider how many there are. Thus, potential mobility is |whitePotentialLegalMoves| – |redPotentialLegalMoves|, where each is a BitBoard. This is relatively fast, and banks on calculations we are already doing in order to speed up the search algorithm.

7. Current Side – This function returns a 1 if it is white's turn and -1 if it is red's turn. We learned that it is much more important for it to be your turn later in the game than in earlier stages. Thus, this—very roughly—approximates parity, although ideally we would consider parity within "closed" regions, and also account for players passing turns.

8. Disc Difference – At the end of the game, the only important thing is who has more discs: thus, we keep the greedy calculation in, as it has increasing importance (up until the end, when it is the only factor).

3 Machine Learning

How do we combine these evaluation features into one evaluation function? For each feature, we used machine learning to determine how much relative weight it should hold.

3.1 Learning Method

After researching the different types of reinforced learning, we decided to implement an online stochastic gradient descent algorithm. As an extension to the basic method, we choose to use exponential summing.

In the early stages of our planning, we considered using a batch training algorithm to adjust our weights. This method minimizes the total error of the training set and would allow different processes of our program to train on different states in parallel (since it is only after all the states of a stage are processed—and their net error calculated—do we adjust the weights for that iteration).

In comparison to batch gradient descent, online stochastic gradient descent, which adjusts the weights after each state it sees, has several compelling advantages.

First, because online stochastic gradient descent adjusts the weights after each state it trains on, the algorithm has significantly more reasonable weights after seeing some reasonable portion of the training data. Thus, since we adjust more often and the date from each training state has the benefit of being compared to our best weight data at the time, convergence has the possibility of occurring sooner with less iterations.

Second, since we do alter our weights after each individual state and each state provides a gradient that is in a slightly different direction than the last, we avoid taking large steps in a single direction and falling victim to a local maxima or minima as we would in batch gradient descent. Since each state takes us in a slightly different direction, we are less likely to move directly towards a local extrema and hence less like to get "stuck."

Our base calculation for each game state that we trained on was:

$$w_{i\,+=}\,\lambda(v' - \mathrm{V}(s))f_{\mathrm{i}}$$

where:

w_{i} – weight of feature i

λ – learning rate (around the magnitude of 0.5)

v'- value of the state after having taken some action a and the opponent following by taking action a'

V(s)– value given by the evaluation function on state s

f_{i}– value of feature i in state s

Note that v' was taken by determining what action (via the minmax tree) should be taken at state s (where the end state values were the piece differential) and then what reactionary step our opponent would take. The value of the resulting state was v'.

This resembles METHOD 1 of the handout. After trying METHOD 2 on a few stages without any significant gain, we decided to use METHOD 1 to train.

The method described above trains off the immediate states following s. This is reasonable since one can argue that immediate gains and loses are felt more quickly and can more quickly determine the direction (and eventual success) in a game. However, it is also important to take into consideration the value of game states much further into game (since these more accurately reflect the value of the end states). To achieve a blend of these in our final training implementation, we choose to incorporate exponential summing of the states on the path going down the tree 10 of our moves:

$$w_{i\,+=}\,\lambda(v' - \mathrm{V}(s))f_{\mathrm{i}} + \lambda^{2}(v'' - \mathrm{V}(s))f_{\mathrm{i}} + \ldots + \lambda^{10}(v'''''''''' - \mathrm{V}(s))f_{\mathrm{i}}$$

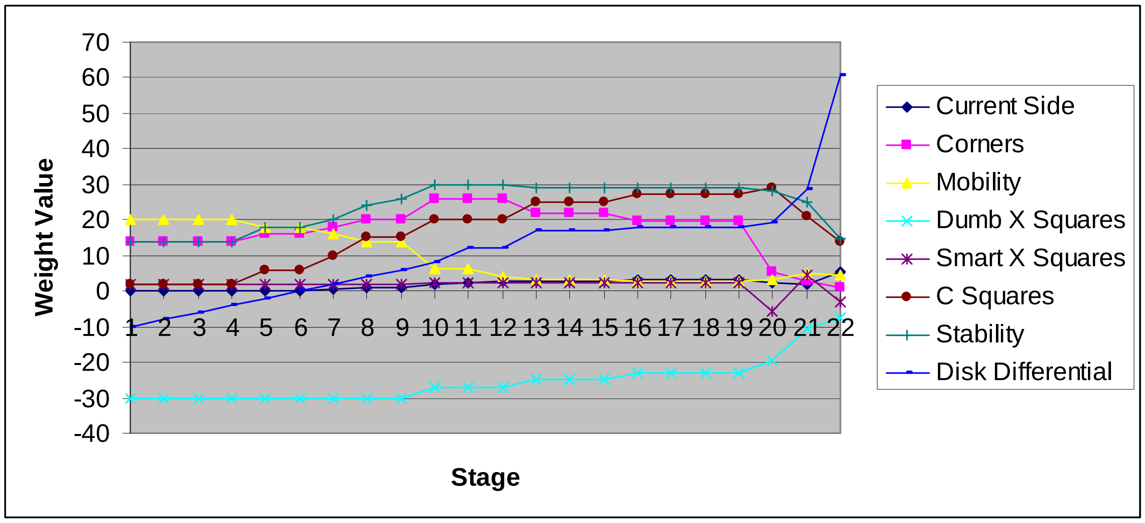
3.2 The Stages and Training

We split the game into 22 stages where each stage was 4 moves long. To start the training, we trained the end game stages (the last two) and used a lookahead that allowed the search to go all the way to the end game nodes. For these stages, we used random initial weights and with a relatively large lambda (0.6). We gradually decreased the lambda to 0.1. The rate at which we decreased lambda was determined by experimentation for each stage.

We then choose initial weights for the earlier stages based on the ones in the stage just trained in order to aid convergence. (We initialized stage 18 with stage 19's converged weights). In these stages we used a lookahead of 4 (since that was enough for us to get to a new, already calculated, stage).

The training set of stages that we trained off of were 600 games saved at different points and played from start to finish where each player took a "smart" move 50% of the time (using the smart evaluation function from the milestone) and a random move the other 50% of the time. We then started at the appropriate state into the game

Because of the AFS crash the night before the program was due, the weights of the earlier stages were hand tweaked to estimate convergence.



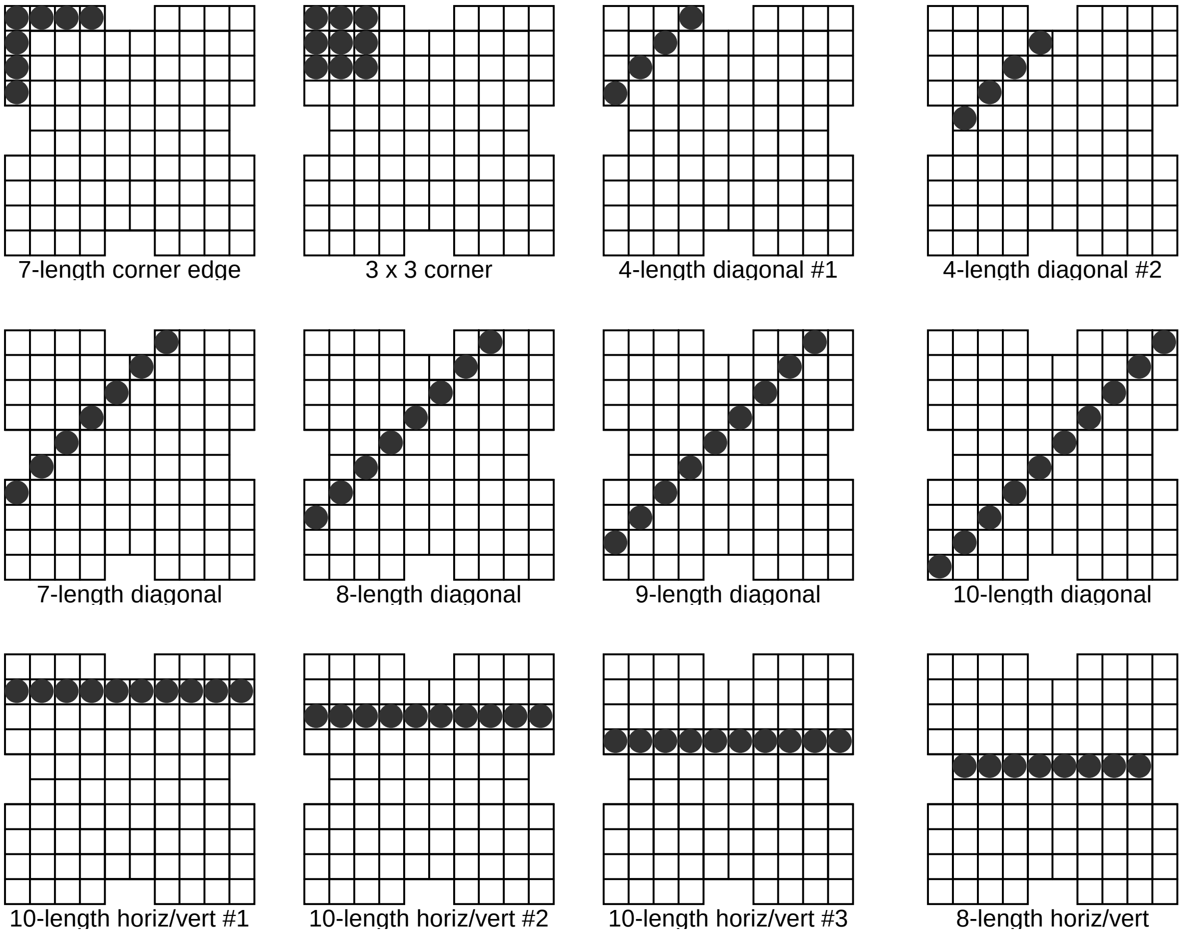
4 Simple Features

In various papers, Buro (1995b, 1997a) presents a convincing argument for using a collection of "simple" features—essentially, combinations of 4 to 8 disc patterns—rather than more complex, calculated features. In more recent papers (notably, Buro 1997b), Buro even advocates that such features can *replace*, not merely augment, complex features, since most complex features are based on simpler patterns. In essence, the "simple feature" technique not only produces faster results than complex features (as it requires relatively little computation), but it also can find relevant patterns that go beyond simple calculations such as mobility and board squares.

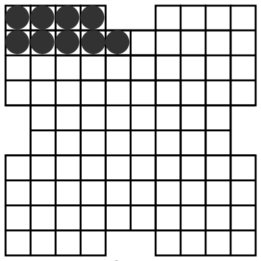
Our earlier dilemma, that of various corner, X, and C square metrics depending on edge configurations, can perhaps be solved using simple features: One could consider each possible combination of the edge squares, and thus make intelligent choices about each kind of square, depending upon which of its neighbors belong to whom.

4.1 Our Simple Feature Space

Inspired by LOGISTELLO's various features for the standard 8 x 8 Othello board (described in various Buro papers) as well as other programs such as KEYANO (Brockington 1997) and ZEBRA (Anderson 2002), we arrived at the following twelve simple features:



Note that, in hindsight, we might have included another feature, which would aid more readily in the X/C square dillema (as it expressly includes all the X and C squares on one edge):


new feature?

4.1.2 Assigning Values to Features

In order for these features to contribute to an evaluation function, however, we would need to assign values to specific configurations of features. We experimented with a number of ways of doing so, as well as a number of ways of combining these values once they are calculated.

4.1.2.1 Arbitrary Assignment

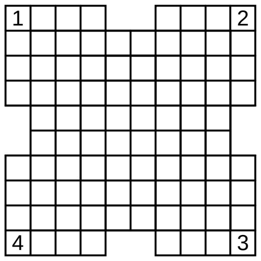
The simplest technique would be to assign each instance to a number; for example, one could use the simple technique (Buro 1997a) of assigning a 0, 1, or 2 to each square (for white, red, and empty) and take the base-3 sum of the squares. Thus, each configuration would correspond to a unique number.

However, this does not take symmetry into account; for example, in the 4-length diagonals, the arrangement [WHITE, blank, blank, blank] should be considered the same as [blank, blank, blank, WHITE]. We came up with an algorithm that assigned a unique number to each *symmetrically unique* configuration. As each of the simple features would likely interact with each other, we considered combining them in a neural net (whose hidden layer would allow weights not just for individual features, but for combinations of features).

Nevertheless, if each instance of the pattern is assigned to an arbitrary index, then there will be no consistency between those indices. Put another way, the Hamming distance (i.e. numbers of bits that differ) will not correspond to any real factor, and thus it will be difficult (if not impossible) for the neural net to find appropriate patterns.

4.1.2.2 Disc-Level Symmetry

We next experimented with (and actually coded) a version that assigned numbers based on individual disc symmetry; thus, each feature had a certain number of "sub-features" based on the number of symmetric and non-symmetric discs. For example, one of the sub-features for the 3 x 3 corner is the following:



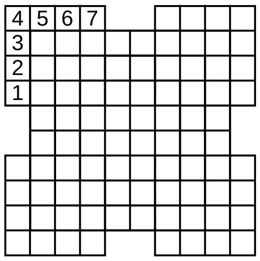
While this may seem like simple square weighting, it is somewhat more complicated, in that the values of the squares labeled 1, 2, 3, and 4 were summed up using a lookup table, in order to distinguish the symmetric pairs of owning {(1 and 3), (2 and 4)} from {(1 and 2), (2 and 3), (3 and 4), (4 and 1)}. We fed each adjusted disc sum into a neural net, and trained it based on calculated disc differentials (in a technique similar to that described in Buro 1997b).

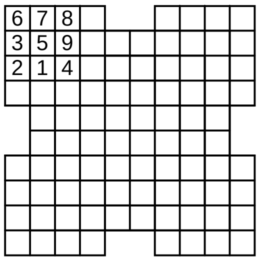
However, this technique did not yield good results: Partly, this is because there are too many inputs, which do not vary considerably. Moreover, in our attempt to allow for interactions between features, we broke down the "walls" that define the features to begin with, thus making the metric similar to square weighting, even if it is more nuanced.

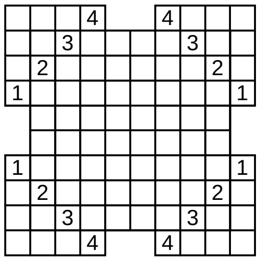
4.1.2.3 Individual Feature Values

We finally settled on calculating value metrics for individual features, inspired by the KEYANO program (Brockington 1997), which used a software package of Adaptive Logic Networks. The technique is simple: feed disc information about only *one feature* into the neural network, and train on the disc difference, as before. If the feature is good, then disc difference will correlate with certain configurations, which the network can learn.

What should one feed this specialized neural network? One can either choose the arbitrary disc-by-disc LOGISTELLO method, or one involving symmetries. We ultimately used the disc-by-disc method for speed, but we also implemented a function which would take an ordered list of squares, such as those below, and calculate the number of symmetric and non-symmetric squares

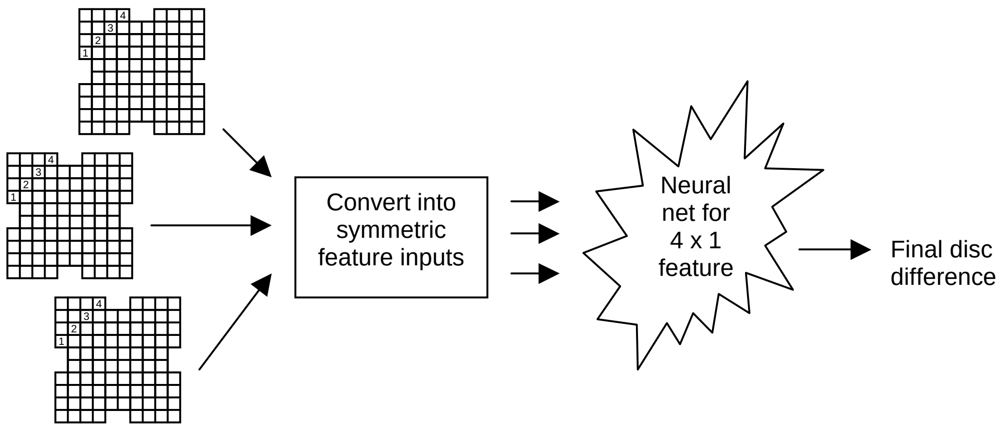






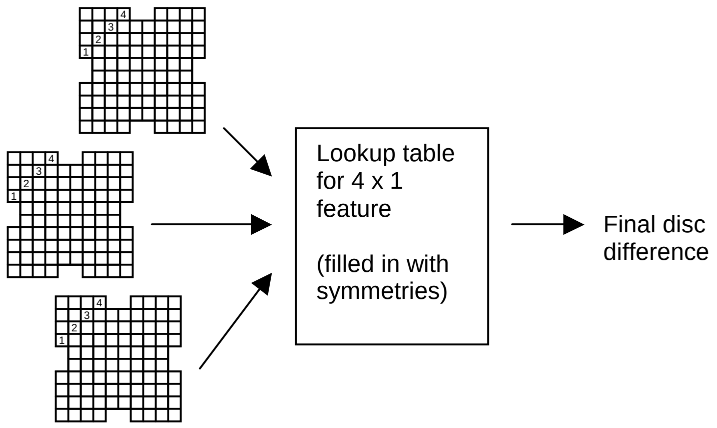
Symmetric squares would be added up, thus ensuring that 4-length diagonals [WRRR] and [RRRW], for example, would both total to the same values (1 W on the outside, 1 R on the outside, and two reds on the inside). In order to distinguish non-symmetric configurations which have the same "symmetric sums," such as [WRWR] and [WRRW], we also calculated the number of runs and/or the length of the longest run (where a "run" is an uninterrupted string of discs of one color or another). Thus, we achieve Hamming distances between configurations that correspond to actual symmetries and meaningful configurations! (However, as noted earlier, the submitted version of DESDEMONA does not use these symmetric calculations, due to time constraints…note also that it would be possible to *train* on the symmetric calculations but *store* the disc-by-disc sum, thus having the best of both worlds, but this is not yet implemented.)

The alternate version (not in the submitted program) is illustrated below:


Convert into symmetric feature inputs
Neural net for 4 x 1 feature
Final disc difference

4.2 Table Lookup

While we experimented with neural nets, both for ease of training as well as for speed in lookup, we used a simple table for the submitted version of DESDEMONA. The structure of the table is illustrated below.



How did we train the table values? For each configuration of a simple feature, we calculate the expected disc differential in the end of the game. For example, one possible configuration of the length 4 diagonal from (0, 3) to (3, 0) is RRWR. Using a generated database of Othello games, we calculate the expected final disc differential given that the diagonal is RRWR. This computation is done prior to the game, and thus finding expected disc differential during play only requires a simple lookup.

Ideally, one would calculate the expectations based on "good" games, ones typical of tournament Othello play. Here, a chicken and egg problem arises: we need a good Othello program to generate a database of good games, and we need a database of good games to get a good Othello program. Also, we need a large game database to make sure that each possible configuration appears in the games. If a configuration does not appear, we have no way of calculating an expectation for it, and we will not know what to do when the configuration arises in tournament play. Because of this, and because of time constraints, we chose to train on 10,000 "random" games. At every stage in the game, each player chose a move at random from all legal moves. The random database was fast and easy to generate, and it contained instances of most feature configurations.

We took full advantage of the inherent symmetries of the Othello board when calculating the expected disc differentials. From each board state, we generated 16 total board states using the 8 symmetries of the board, along with the states obtained by switching white discs with red discs. So, for a database of 10,000 games, each of which had about 88 moves, we were able to generate 10,000 * 88 * 16 = 14,080,000 game states with final disc counts. The expected disc differentials for a configuration were calculated by taking the average of the final disc counts of the game states in which the configuration appeared. In addition to the benefit of more game states, adding the symmetries of the board guaranteed that symmetric states had the same expected disc differential.

Of course, once we were able to calculate expected disc differential from a game state, we needed a way to access these values. Feature lookup was performed by treating each configuration as a ternary number. If a feature had n squares, we chose an ordering of the squares, and then assigned WHITE a value of 0, RED a value of 1, and EMPTY a value of 2. In this way, a unique number could be computed for each configuration, and the expected disc-differential lookup could be done by indexing into an array containing $3^{n+1} – 1$ expectations. In practice, to speed up computation, we treated each configuration as a quaternary number, and thus could calculate the index using only bit operations and no multiplications.

4.3 Combination and Integration with "Complex" Features

How to combine these individual feature values? In one paper (1997b), Michael Buro describes combining simple features like DESDEMONA's into one feature using a technique similar to a neural network. The neural network has the advantage of taking into account complex feature interactions. Unfortunately, due to a file-server crash, we were unable to train a network, so we were forced to come up with a hand-tuned solution. We chose to average our 10 simple features to get an estimate of final disc differential. In testing, this proved to be a better feature in the early stages of a game than the greedy heuristic.

While the "simple" features were designed to replace some of the "complex" features, the "complex" features were left in the final submitted program, and a simple weight of 80% complex and 20% simple features was decided upon, given the nascent stage of the simple features metrics. With more training (perhaps on a neural net) and tweaking, likely the simple features could contribute more to the final evaluation function.

6 Performance and Future Research

As noted in the abstract, our program performed at a mediocre level in the tournament, losing three games and winning one in the first round. While the program seemed to be searching fairly deep through the tree, many of its choices seemed poor, likely due to our inability to fully train the program on all game stages. For future research, we of course would like to try completing the training process! We would have also liked to implement an "opening book," found in many strong programs (e.g. Buro 1999, Cheung, Chi and Pang 2001), to essentially cache a large database containing the first few moves of the game, such that one can spend nearly no time on the first few moves, yet still search fairly deep through the tree. Naturally, such functionality would depend on a strong, stable evaluation function, which was not available at the time (even though all the transposition table mechanism for implementing it was in place). Finally, it seems that a more thorough approach to the "simple" features, perhaps augmented by training on neural nets (even if they are not used for actual lookup of feature values), would be able to replace all of our search features, except perhaps the Potential Mobility heuristic (which is useful in speeding up search anyway).

Works Cited/Consulted

Anderson, G. 2002. The Inner Workings of Strong Othello Programs. http://www.nada.kth.se/~gunnar/howto.html

Buro, M. 1995a. ProbCut: An Effective Selective Extension of the Alpha-Beta Algorithm. *ICCA Journal* 18(2), 71-7. http://www.cs.ualberta.ca/~mburo/publications.html

Buro, M. 1995b. Statistical Feature Combination for the Evaluation of Game Positions. *Journal of Artificial Intelligence Research* 3, 373-382.

Buro, M. 1997a. An Evaluation Function for Othello Based on Statistics. NECI Technical Report #31.

Buro, M. 1997b. Experiments with Multi-ProbCut and a New High-Quality Evaluation Function for Othello. *Workshop on game-tree search*, NECI, August 1997.

Buro, M. 1998. From Simple Features to Sophisticated Evaluation Functions. *The First International Conference on Computers and Games* (CG '98), Tsukuba, Japan.

Buro, M. 1999. Toward Opening Book Learning. *ICCA Journal* 22(2), 98-102, reprinted in: *Games in AI Research*, H.J. van den Herik, H. Iida (ed.), 2000, and in: *Machines That Learn to Play Games*, J. Fürnkranz and M. Kubat (ed.), 2001.

Buro, M. 2002. The Evolution of Strong Othello Programs. *Proceedings of the IWEC-2002 Workshop on Entertainment Computing*, Makuhari, Japan

Brockington, M. G. 1997. KEYANO Unplugged – The Construction of an Othello Program. *Technical Report 97-05*, Department of Computing Science, University of Alberta. http://www.cs.ualberta.ca/~games/keyano/

Cheung, A., Chi, A. and Pang, J. 2001. CS221 Othello Project Report: Lap Fung the Tortoise. http://www.stanford.edu/~hcpang/othello.html

Feinstein, J. Othello Pages. http://www.maths.nott.ac.uk/othello/index.html

le Comte, M. 2000. The strategy to winning Othello. *Introduction to Othello.* Nederlandse Othello Vereniging (The Dutch Othello Federation) http://www.othello.nl/guides/comteguide/strategy.html

Lazard, E. 1993. Othello Strategy Guide. Translated by C. Springer. Federation Française d'Othello (French Othello Federation. http://homepages.compuserve.de/othelloclub/strategy1.html [among other locations]

Lee, K. F. and Mahajan, S. 1990. The Development of a World Class Othello Program. *Artificial Intelligence* 36, 1-25.

McAlister, J. and Wright, D. 2000. Rocket Monkey Deathmobile. [CS 221 Writeup.] http://bigmac.stanford.edu/

Mendelson, J. 2001. Jonathan's Reversi Page http://www.mathjmendl.org/reversi.html

Moreland, B. 2001. Zobrist Keys. http://www.seanet.com/~brucemo/topics/zobrist.htm

Othello University. Othello Strategy. http://home.nc.rr.com/othello/strategy/

Russell, S. J. and Norvig, P. 2002. *Artificial Intelligence: A Modern Approach*, Second Edition. Pearson Education, Inc.

Scapel, N. and Mazzella, F. 1999. Programming Assigment #2 – Othello: The French Connection. http://movement.stanford.edu/nico/othello/Othello.html

Sweetkind-Singer, J. A. 2000. Combining Reinforcement Learning With Genetic Algorithms To Produce An Othello-Playing Program. http://www.stanford.edu/~singer/Papers/papers.html

Wang, P., Eisenberg, L., and Vadera, K. Perversi: CS 221 Othello Project. http://www.google.com/search?sourceid=navclient-menuext&q=cache:http%3A//www.stanford.edu/~pxwang/OthelloWriteUp.pdf

[Note: our team's "working bibliography" is online at http://xenon.stanford.edu/~lswartz/cs221/ and includes basic notes that we used when developing our program. We'd also like to thank the teaching staff, especially Mykel Kochenderfer, for their help throughout the project.]