CS 221 Programming Assignment

Othello: The Moors of Venice

a report in seven parts

Rion Snow rlsnow@stanford.edu
Kayur Patel kdpatel@stanford.edu
Jared Jacobs jmjacobs@stanford.edu

Abstract: Here we present the algorithms and extensions that produced our successful Othello-playing program, which finished among the top five programs of this year's Othello tournament. Specifically, we discuss a variety of search algorithms and extensions, as well as our algorithms for adaptive time-management, our efficient 'bit board' data structure, our features comprising our evaluation function, our two learning methods, our opening book, and our end-game solver. Throughout the report we discuss the experiments and evaluations that led us to our particular design decisions, including comparison of the efficiencies of search strategies, performance analysis of our data structures, and testing of different feature sets.

1 Search Algorithms and Extensions

We began our study of search algorithms with the implementation of standard Minimax search with alpha-beta pruning. We quickly discovered that our initial implementation of search was far too slow to win the tournament, which inspired us to explore a variety of means by which to reduce the number of nodes we search and thereby improve the speed of our algorithm. Most important of these were implementations of a transposition table, an efficient move-ordering scheme, and explorations of the advanced search algorithms NegaScout and MTD(*f*).

1.1 Minimax

Our initial Minimax search algorithm with alpha-beta pruning was implemented in the standard way, with some minor exceptions to deal with specific game-related subtleties. For example, we implemented a subtle, but important change in dealing with the problem of 'passing' in Othello. To refresh, there are instances in Othello where a particular player has no legal move and therefore must "pass", allowing the opponent to move twice (or more). In the event of such a pass, it is important that we decrease the number of nodes we look ahead by one, as the determination of 'who moved last' will necessarily alter any evaluation function depending on the piece differential, and thereby skew the leaves towards the player who moved lass. To avoid this pitfall, we decrement the lookahead in the case of a pass.

1.2 NegaScout

NegaScout is a refined version of the Principal Variation Search (PVS). The first search is performed with a wide alpha-beta window, specifically $(\alpha = \infty, \beta = -\infty)$. When the

last level is reached (as specified in the depth parameter), the best leaf is returned and set to as 'principal variation'. The game tree is then searched again with a smaller alpha-beta window, in effect implementing the hypothesis that this search cannot do better than the principal variation already obtained. To test this hypothesis the search takes place with a "zero window" where alpha is the PV and beta is the PV plus a small epsilon.

The notion of the zero window introduces two more concepts: that of "failing high" and that of "failing low". Since the first search is made with a zero window, the result, *r*, could either be that we were correct and the values were below the PV (fail low), or we were incorrect and the values were above the PV (fail high). If we were incorrect then we re-search, with a new more realistic window of $(\alpha_{t+1} = r, \beta_{t+1} = \beta_t)$.

NegaScout is widely used in the creation of good Othello and chess programs. Pseudocode and a lengthy discussion of NegaScout may be found at [4].

1.3 MTD(*f*)

The MTD(*f*), or Memory-enhanced Test Driver with value *f*, algorithm works on the same notion of a "zero window" search as in NegaScout. MTD(*f*) performs repeated zero window searches and keeps bounds for the Minimax value. These bounds are initialized to between negative infinity and infinity. As before, the search can also fail high or fail low of the window; if the value fails high, the lower bound is moved up to the value, and if the value fails low, then the upper bound moves down to the value. When the upper bound is lower then the lower bound, the correct move has been found.

The repeated searches are started with an initial guess of the zero window. This initial guess is very important. The closer the value is to the actual value of the node, the fewer searches need to be made. Repeated searches on the same game tree may seem expensive, however, when a lookup table is used, repeated states do not necessarily have to be re-searched. In fact MTD(*f*) heavily relies on lookup tables in order to make it worthwhile.

Many modern board game programs use MTD(*f*). Though the code is simple, it does take longer to implement because of the transposition table code. Pseudocode and a lengthy discussion of MTD(*f*) may be found at [5].

1.4 Transposition table

A transposition table is a hash table cache that stores the best move found so far for each board position encountered. It improves search performance because expanding the best value (or a close approximation) for a variable first causes the alpha-beta window to shrink faster and thus more search nodes get pruned. Using a transposition table helps during a single-move search in Othello because different move sequences can often result in the same board position. It is also helpful between different move searches because many of the same board positions must be considered for two consecutive moves. In our experiments using both the greedy and smart evaluation functions, the transposition table decreased the total number of nodes expanded in a game by an average of 16% (used in conjunction with minimax search with alpha-beta pruning).

The hash function we implemented comes from Robert Jenkins [3]. It results in very few collisions because it eliminates funnels; that is, it has the property that a change in any k input bits can affect at least k bits in the hash value when $k < v$ (the number of bits in the hash value), and all bits in the hash value when $k \geq v$. This means that any change to a board position—even a single piece—will usually change its hash value. Computation of the hash value for our 40-byte bit-board is relatively cheap at just 76 machine instructions.

For ease of implementation, we store only the most recently encountered board in any given hash table bucket. It is unlikely that the cache miss rate would decrease significantly with more entries per hash table bucket (and fewer buckets) since our hash function provides such well distributed hash values. Since each hash table entry requires 64 bytes, our transposition table contains 2^{20} entries for a total memory footprint of 64 MB.

1.5 Move Ordering Heuristic

In addition to implementing the alternate search algorithms and transposition table, we extended our system with a move ordering heuristic. The tables for MTD(*f*) lend themselves to storing extra data about the moves, in particular, the best moves from previous searches of the board. When we come across a board, we check to see if the board and its previous best move are already in the table. If they are, then we expand the previous best move first. By finding the best move as early as possible, we benefit from additional pruning and therefore faster search time. Since the best move is stored in the transposition table, it is a very quick lookup. A strategy we considered but later decided against was that of sorting our moves for each board using our evaluation function; since our evaluation function is rather costly to compute, this strategy would ultimately be counterproductive.

1.6 Experiment: Comparison of Search Strategies

Finally, we performed a comparison of the algorithms before implementing our smart search strategy. In the graph below we chart the number of nodes searched vs. the number of pieces currently on the board, measuring search efficiency as the number of nodes searched. As expected, NegaScout generally outperforms Minimax, and MTD(*f*) generally outperforms NegaScout. Also, as expected, the addition of the transposition table and move ordering heuristic improved performance. In fact, the addition of these two extensions improved our performance to the point that Minimax became approximately on par in efficiency with the other algorithms. Due to this, coupled with a lingering fear of possible bugs in our zero window searches, we chose the simple Minimax algorithm learning extended by the additions of our transposition table and our move-ordering heuristic, which proved to be very fast.

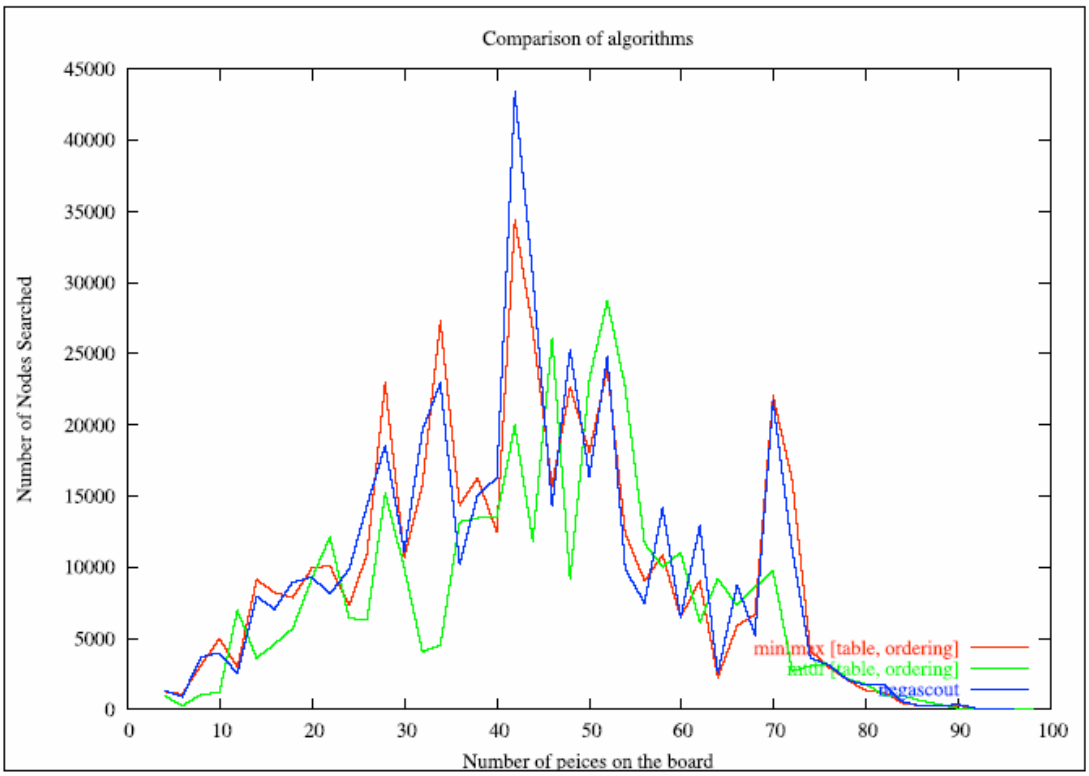

Figure 1: The final versions of the searches are above. These searches were tested against a greedy player playing at depth 4. We played a greedy strategy at depth 6. We evaluate search efficiency according to the fewest nodes expanded. The Minimax with tables and ordering performs worse than MTD*f*. The tables, however, do make Minimax on par with NegaScout.

2 Time Management

In consideration of time management we implemented two basic strategies. The first relied on an implementation of iterative deepening search; this was later abandoned in favor of the faster fixed depth search with adaptive depth approximation.

2.1 Iterative Deepening with Adaptive Time Management

With iterative deepening we search only every two-ply, so as to compare node scores always as the same player. This reduces much of the 'noise' present in many of the features, caused by the alternate switching of players. To manage our time we essentially calculate a specific allocation of time t_a for this move, which is performed by dividing the total time remaining T_r by the maximum number of remaining moves m_r to arrive at a first-guess time allocation estimate t_a, i.e. $t_a = \frac{T_r}{m_r}$. Due to the nature of iterative deepening, we may simply check after each successive deepening whether we have exceeded our allocated time, or, more specifically, whether we would expect to exceed

our allocated time by searching the next depth (estimating the amount of time required to search the next depth as simply the time we took to search the immediately previous depth). If we expect to exceed or have already exceeded our allocation, we simply return the result of our last, deepest search.

2.2 Branching Factor Estimation and Adaptive Depth Approximation

We eschewed iterative deepening in favor of a faster depth-first search; however, this requires that we estimate our depth ahead of time based on our previous searches. We were inspired to make many of our approximations based on those by Alvin Cheung et al. in [1]. In order to make an approximation of the depth of our optimal search, we first allocate a specific amount of time for our search; this is the same t_a calculated in the previous subsection. For the first two moves of the search after our opening book, we search to a fixed depth (with default of eight). For subsequent moves, we use the number of nodes searched and the depth reached during the previous two searches to calculate the effective branching factor b and time required per node t_n. The branching factor for our present search b_s is calculated as the depth of the last search d_{s-1}'th root of the total number of nodes searched in the last search n_{s-1}, i.e.

$$b_s = n_{s-1}^{-d_{s-1}}.$$

Next, the average time required per node is simply the number of nodes searched in the last search n_{s-1} divided by the actual amount of time used during the last search t_{s-1}, i.e.

$$t_n = \frac{n_{s-1}}{t_{s-1}}.$$

We average over the last two searches instead of just the previous one to reduce the impact of noisy statistics from atypical searches. These figures allow us to estimate the lookahead depth d we can afford for the current search. The specific equation for this is

$$d = \left\lfloor 0.7 + \log_b \left(\frac{t_a}{t_n} \right) \right\rfloor$$

This expression effectively rounds depth estimates up to the nearest integer from 3/10 or above, then rounds odd depths down one to the nearest even depth. We toyed with our rounding method until we were happy with the average time left over at the end of a game (usually 20-40 seconds). Finally, we bound the search lookahead depth to the range [4,16], to prevent oddly calculated branching factors.

Once we have calculated a lookahead depth, we commence the search. If it takes more than twice the allocated time, we abort the search, recalculate allotted time t_a, then recommence the search. Use of a transposition table makes the cost of restarting an aborted search very low. We often found that searches aborted after about 6.5 seconds completed in 0.7 seconds after being restarted. One more safeguard we implement is ensuring that we do not try searching to the same depth for any one move more than twice (unless that depth is 4).

When only 14 or fewer plies remain in the game, our smart agent does an exhaustive search using just the score feature (discussed in Section 7, the end-game solver).

3 Bit-board Implementation

The two primary motivations we had for implementing a more compact representation of the Othello board were (1) to minimize the amount of memory required for each transposition table entry and (2) to improve running time for feature evaluation (details in section 5).

Our bit-board implementation includes one 10x10 bitmap to indicate which cells are occupied and another to discriminate between red/white or empty/illegal. Bitwise operators make isolating the white, red, or empty cells trivial. We use the 20 lower-order bits in five words of memory to store each bitmap. This makes the calculation of the word index for a given board cell (x,y) as simple as shifting x right one bit. We cache bit masks for isolating the particular bit in a word of memory that corresponds to a cell (x,y) in a static 10x10 array indexed by x and y. Our bit-board class also stores how many red and white pieces are on the board since those values are queried so frequently.

Compared with the provided board implementation (a 10x10 char array), which requires 108 bytes per instance, our bit-board implementation is much more compact (42 bytes per instance). Single-cell queries on our bit-board require 12 arithmetic instructions instead of just 5, but our bit-board has the advantage that queries on two entire rows can be done in one step. More importantly, the efficiency gains our bit-board implementation can achieve by exploiting parallelism in our feature evaluation algorithms make the choice between the two board representations a no-brainer.

4 Evaluation Function

4.1 Score Feature

This feature is the only one that the greedy evaluation function uses. Its value is simply the piece differential—that is, the number of white pieces on the board minus the number of red pieces. We use it in our smart evaluation function because it is clearly the most important feature when the game ends: a positive value corresponds precisely to a win for white, a negative value to a red victory, and zero to a draw. In addition, we found from experience that the side that maintains a piece deficit during the first half of the game often ends up winning.

Querying this feature's value for a given board position is a constant-time operation for our board implementation since the piece counts are updated each time either side plays a piece.

4.2 Mobility Feature

This feature is the number of possible moves white has in a given board position minus those red has. Intuitively, the more options a side has, the higher the chances are that one

of them will be good. In addition, high mobility means that a side does not have to forfeit a turn and will likely not have to in the near future.

To determine the potential moves each side has given some board position, we use a bit-board shifting algorithm based on the one that Chuong Do et al. describe in [2]. The pseudocode below demonstrates how to calculate mobility for white. For counting the number of one bits in a given bitmap, we use a 1024-entry lookup table indexed by a ten-bit number. For a 10x10 bitmap stored in 5 words of memory, the entire counting operation requires only 10 array lookups, 9 additions, and 5 bit shifts.

```
Bitmap vacant := empty cells
Bitmap thisside := white cells
Bitmap opponent := red cells
Bitmap mobility := all zeros

for each Direction dir
      possible := shift(thisside, dir) AND opponent
      while possible is not empty
            possible := shift(possible, dir)
            mobility := mobility OR (possible AND vacant)
            possible := possible AND opponent
```

4.3 Corners Feature

The corners on an Othello board are key because they are the only inherently stable cells on the board; that is, the opponent has no way to recapture them. What's more, they serve as a stabilizing anchor from which stable regions can extend. This feature, keeping with the tradition of the others, is the number of corners white holds minus the number red occupies.

4.4 Frontier Feature

The frontier is the set of cells a side occupies that are adjacent in at least one of the eight directions to an empty cell. Frontier cells are often vulnerable, so minimizing how many you have can be an important strategy throughout the game. We compute this feature as the number of white frontier cells minus the number of red ones. It might also have been interesting to compute what percentage of the cells each side occupies are frontier cells. Figure 2 shows how we isolate the white frontier cells in parallel using bit-board shifting.

```
Bitmap vacant := empty cells
Bitmap frontier := white cells

for each Direction dir
      frontier := frontier AND NOT shift(vacant, dir)
```

4.5 Stability Feature

Even though the notion of a stable cell is simple, the algorithm for determining whether some occupied cell on a board is stable is surprising complex. While stable regions tend to grow out from the corners, it is also possible for certain cells to become stable before any corners have been claimed. When we realized that the algorithm could not be

reformulated to exploit parallelism using bit-board shifting, we decided to try implementing a simple stability heuristic with comparable running time first. Our heuristic just counts the pieces belonging to each side that the opponent could not capture in one move. Using profiler, we found that our client spent over half of its total execution time evaluating this one feature. For that reason, we scrapped it.

4.6 Global Parity Feature

This feature evaluates to 1 if the white player will play last and -1 otherwise, assuming that the game will not end with any empty squares and neither player will have to forfeit a turn. The intuition behind this feature is quite simple: since the score differential changes by at least 3 (in favor of the side playing) each time a piece is played, the player who plays last will clearly have at least a small advantage. Given the total number of pieces on a board and whose turn it is to move, this feature can be evaluated in constant time. Note that this feature always has the value -1 at the start of a game, and its sign flips whenever a side forfeits a turn.

4.7 Access Feature

This feature is the difference between the numbers of board regions into which each side can play. It is analogous to mobility, but on a different level. The justification is that a side cannot control parity (i.e. who gets the last move) in a region if it is unable to play in that region. We define an empty-cell region by stating that two empty cells A and B are in the same region if there exists a path consisting of only empty cells from A to B such that consecutive cells in the path are adjacent in one of the eight compass directions. Our algorithm for identifying board regions requires only a single pass through the cells in the board in row-major order. A region label is assigned to each cell based only on the labels of its previously visited adjacent neighbors. When two regions must be merged, backtracking for relabeling never exceeds ten cells.

4.8 Local Parity Feature

This feature is closely related to both global parity and access. We loosely define it as the difference between the percentages of regions in which each player is confident he can make the last move. When there are an even number of regions on the board, neither player can expect to play last in more regions that the other. On the other hand, when there are an odd number of regions on the board, the side with global parity can expect to play last in one more region than the other side. This advantage is greater when the total number of regions is smaller; hence the percentage in the calculation.

5 Learning methods

5.1 The Square Learner Architecture

Since we were working with an augmented board, we could not use some of the conventional Othello heuristics. We did not know how important non-corner squares were to the outcome of the game, so we decided to attempt to learn a quantitative scoring of the importance of each square.

Our square leaner used a passive reinforcement learning algorithm; it would simply watch games and attempt to figure out which squares it should add as features. Since the learner did not actually change the outcome of the game, we decided to save game logs and learn using reinforcement learning. The logs were created by randomly switching between our smart and greedy strategy. By saving games we controlled which games were in our training set, and thus were able to train different learners efficiently on the same training set, without having to run the games over again. The final logs contained 704 games, of which there were 254 white wins, 445 red wins, and 5 ties. Our final training set contained 250 white wins and 250 red wins.

We took advantage of the symmetry of Othello boards by parameterizing the squares. Squares that are in the same parameter have the same number on the board below

	0	1	2	3	4	5	6	7	8	9
0	9	8	7	6	5	4	3	2	1	0
1	8	17	16	15	14	13	12	11	10	1
2	7	16		21	20	19	18		11	2
3	6	15	21	22	23	24	25	18	12	3
4	5	14	20	23	[27] W	[26] R	24	19	13	4
5	4	13	19	24	R [26]	W [27]	23	20	14	5
6	3	12	18	25	24	23	22	21	15	6
7	2	11		18	19	20	21		16	7
8	1	10	11	12	13	14	15	16	17	8
9	0	1	2	3	4	5	6	7	8	9

Figure 2: Parameterization of Board Squares

We found, in Othello, at certain points in the game it is better to have certain square than another. At the end of the game it is important to have as many squares as possible, however it is not obvious what to have at other stages in the game. We, therefore, had our learner partition the game into stages. The stages were determined by how many pieces there were on the board. We experimented which different values for the number of boards in a stage, however the final version of our leaner contains a total of 92 stages. The first stage is the initial board and the last stage is the board with 95 pieces. We only use final board in a game to determine the winner. Since the last board with 96 pieces will always be a final board, we do not have a stage for it.

5.2 Learning Functions for Square Importance

5.2.1 Reward Function

Our first method was to propagate a score from the end game to all the squares that we had. Let $V_{i,s}$ equal the value of the parameter *i* at stage *s*, $N_{i,s}$ equal the number squares in parameter *i* was in our position in stage *s*, R_g be the reward of the game *g* in the training set *G*, and α be the value of the learning rate. We initialize all our $V_{i,s}$ to one. The update rule for a game is as follows:

$$V_{i,s} \leftarrow V_{i,s} * N_{i,s} * \alpha * R_g .$$

We experimented with different values for R_g and α. The results were hard to interpret. Values for parameters would have a very high range. Also, since we did not do a batch update, the first few games had a large influence on results.

5.2.2 Information gain of squares

We decided to abandon the idea of a reward function, and looked at the Russell-Norvig book for inspiration. The idea of information gain seemed like it would help us understand how important a square was. Let $Gain_{i,s}$ be the gain of knowing parameter *i* at stage *s*, $I(p)$ be the same as $I(p,1-p)$ and $P_s(win)$ be the probability of winning at a stage. Also let $P_s(win \mid have_{i,s})$ mean the probability of winning given that we have the parameter *i* at stage *s* and let $P_s(win \mid empty_{i,s})$ and $P_s(win \mid opponent_{i,s})$ be defined similarly. We go through each of the games and create a count of wins and losses given the status of a parameter for a stage, and from these we may obtain the probabilities. The $Gain_{i,s}$ is calculated by the following formula:

$$\begin{aligned} Gain_{i,s} = I(P_s(win)) - [P_s(win \mid have_{i,s}) I(P_s(win \mid have_{i,s})) + P_s(win \mid empty_{i,s}) \\ + I(P_s(win \mid empty_{i,s})) - P_s(win \mid opponent_{i,s}) I(P_s(win \mid opponent_{i,s}))] \end{aligned}$$

We are trying to calculate how much knowing who has a piece at a certain stage of the game can help determine who won the game. We found that there was a high information gain for squares along the edge [0-9 in the picture] and for the X squares [10 and 17]. Our data also suggested the there was a difference in which corner we had [0 as opposed to 9]. We added new features for edge possession as well as for each X square and we also split the corners and started another set of learning processes. Unfortunately we were not able to train the weights before the tournament. When the weights finished we tested them against our tournament player and our new weights lost. The tournament player was created using different learning parameters, so the results are not conclusive.

5.3 Online reinforcement learner

Our reinforcement learner uses different weights for every two-ply stage of the game (a total of 46 weights for each feature). We decided on this granularity because we realized that the importance of various features altered dramatically during the game, and because sharing weights between two consecutive plies eliminates the huge disparity between the number of training examples in which a side moves at even vs. odd plies.

Our general strategy was to initialize all weight values to zero and then train several two-ply stages at a time, starting with the endgame (the last 14 plies), where an exhaustive search to the end of the game is feasible and V* can be computed directly, and working backwards to the beginning of the game, training weights for k plies at a time using a search lookahead depth of k until all weights have been trained. This approach ensures that the evaluation function always uses weights whose training has completed. In a sense, it propagates the truth backwards from the endgame to the opening, losing a degree of precision with each training phase.

For the j^{th} weight update step in a training phase, we use the following rule:

$$w_{i,t} \leftarrow w_{i,t} + \alpha \cdot \gamma^{j} \cdot f_i(s) \cdot \left[V_{t+k}(s^*) - V_t(s) \right].$$

In our notation, s is the current game state, $w_{i,t}$ is the weight getting updated, i is the feature index, t is the total number of pieces on the board in state s, α is the learning rate, γ^j is an exponential decay factor that guarantees convergence, $f_i(s)$ evaluates the i^{th} feature for state s, $V_t(s)$ denotes the evaluation function's estimate of the value of a state s that has t pieces on the board, and s^* is the state a minimax search from s with lookahead k returns. We chose γ such that γ^j would reach 0.1 as j reached its maximum value (500), but it turns out that our agent performed better using weights trained with $\gamma = 1$ (i.e. no convergence fudge factor). We also considered using an exponentially decayed average of the evaluations of the states at the nodes between s and s^* in place of $V_{t+k}(s^*)$ but decided not to because it would allow weights-in-training to heavily influence the update step, and so convergence would likely require many more training examples.

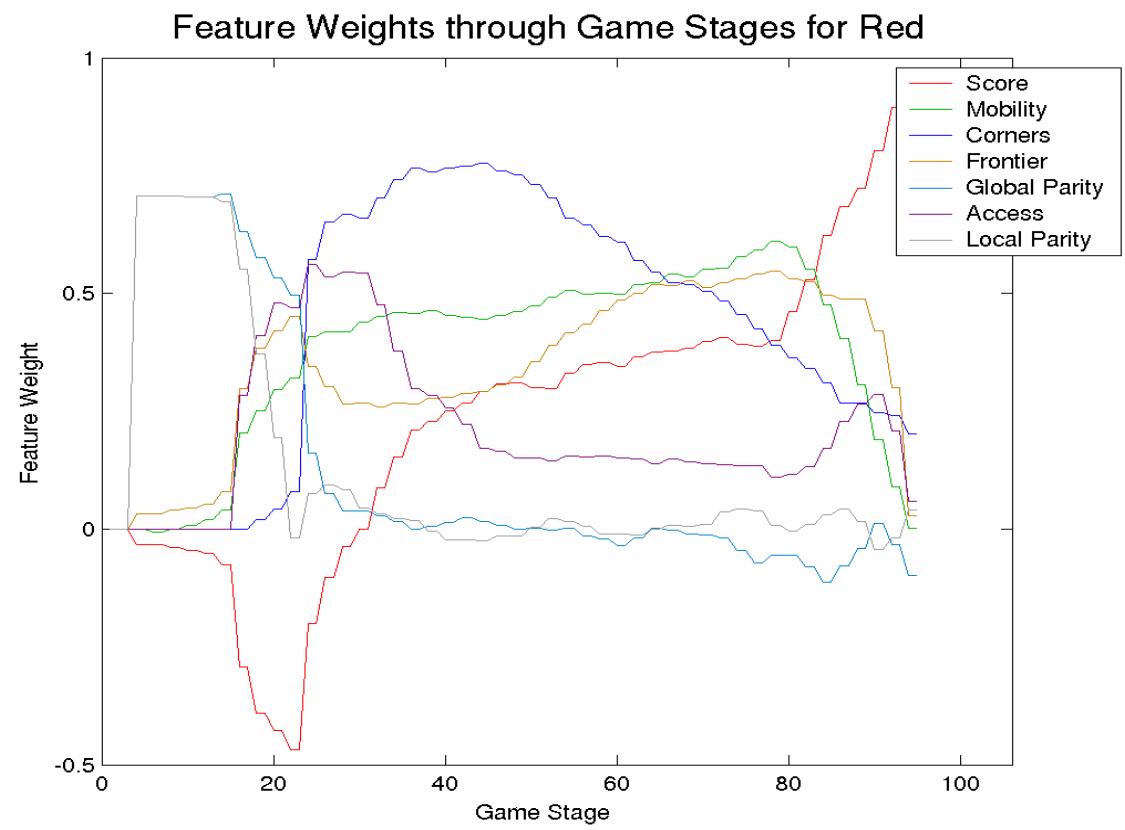

Figure 3: Fully-trained weights of stage-dependent features

We trained separate weights for the white and red players simultaneously by playing them against each other. To ensure diversity in the training examples, we have the agents take random moves 10% of the time, greedy moves 20% of the time, and smart moves utilizing uniform, untrained weights the remaining 70% of the time for moves earlier in the game than the training phase. We tried averaging the white and red weights as a post-processing step, but both players fared better using their custom weights. We also tried applying a low-pass filter to the weights to make their transitions smoother, but that too had a negative impact on game play.

When deciding on a value of k, the number of plies in each training phase, we wanted to maximize k in order reduce the number of training phases, or steps away from V*. On the other hand, minimizing k makes for quicker Minimax searches during training and brings s* closer to s, reducing our degree of reliance on the assumption that the opponent will play as our evaluation function predicts. We ended up training with $k = 8$ and $k = 10$, but only the $k = 8$ training session completed in time for the tournament.

6 Opening Book

Most good game playing programs have opening books and end game databases. To compute an end game database you must evaluate all possible positions of white and red pieces at the end of the game. The number of possible end game positions is explosive (exponential in the number of squares, 96 in our case), so that even if we had the time to

create such an end game database, it would be quite impossible to store it in the space allotted.

An opening book, on the other hand, is much easier to build and to store. After deciding on our features, and training and freezing our weights, we create the opening book by performing deep searches (depth 12 in our case) for the first few moves of the game (up to 12 pieces on the board, in our case). This is most useful in timed games when moving quickly in the beginning leaves you more time to search during the middle and end game. To create the opening book we wrote a small program that searches for a best board for a player. It takes that move and then calculates the board for all possible opponent moves, and then does another search. The book essentially consists of a hash table in which a particular board position is mapped to the move discovered by this algorithm; for the specific implementation we used a hash function from (3) and the <hash_map> hash table implementation from the SGI STL library[6] (we cannot use the same hash table as in our transition table, as we wish to store all boards and not replace any keys). This repeats until we reach the desired size of the opening book. With each further move, the opening book grows by a large degree; however, it similarly saves us more time.

Upon evaluation of our final opening book, we found the speedup to be somewhat minimal, given that we were usually finishing long ahead of time anyway. Fearing that we might run into an untested scenario during the tournament, we guarded against a potential technical loss by deciding against the inclusion of the opening book in our tournament-ready final program.

7 End Game Solver

As a game in Othello nears its end, the branching factor drops dramatically, and if there are only *n* squares remaining empty, a search of depth *n* will fully solve this game. We implement this strategy, choosing to go into 'solver-mode' as soon as 82 pieces have been played, i.e. when there are fourteen empty squares (this specific depth was initially inspired by [4]). In this mode we change our evaluation function to operate solely on our *score* feature, as our win and possible tie-breaker depends solely on finishing with the greatest margin of score. We have found that having an end-game solver with the maximal depth possible was crucial in determining the outcome of our games, and that having a solver of a greater depth than an opponent program, even if only by a single unit of lookahead, could result in a smashing victory, all other qualities remaining equal.

References

[1] Cheung, Alvin et al. "Lap Fung the Tortoise." 28 Nov 2001. http://www-cs-students.stanford.edu/~lswartz/cs221/jimmypang.pdf.

[2] Do, Chuong et al. "Demostheses." 27 Nov 2002. http://www.stanford.edu/~chuongdo/report.pdf.

[3] Jenkins Jr., Robert J. “Hash Functions for Hash Table Lookup.” 1997. http://burtleburtle.net/bob/hash/index.html.

[4] Reinefeld, Alexander. “NegaScout – A Minimax Algorithm faster than AlphaBeta.” 1989. http://www.zib.de/reinefeld/Research/nsc.html.

[5] Plaat, Aske. “MTD(*f*): A Minimax Algorithm faster than NegaScout.” 3 Dec 1997. http://www.cs.vu.nl/~aske/mtdf.html.

[6] SGI STL Library, “Hash_Map.” 2003. http://www.sgi.com/tech/stl/hash_map.html.