Perversi

Philip Wang pxwang@stanford.edu
Louis Eisenberg tarheel@stanford.edu
Kabir Vadera kvadera@stanford.edu

1 Abstract

In this programming project we designed and implemented an Othello playing program that learns from its own experience. The board we used has two major changes to the common version of Othello: first, we use a 10x10 board rather than the traditional 8x8 board; second, we use a board from which the middle 2 squares of each side have been removed. A time limit is placed on the total amount of time our program can use in making all of its legal moves. Since there is not enough time for our program to search the entire search tree for a best move, the program we created searches for a move it estimates to be best to play by using alpha-beta search with an evaluation function. In helping us to produce a good evaluation function, we utilized reinforcement learning.

For the majority of the project, we followed the hints and recommendations suggested by the handout. We implemented standard Minimax Search with Alpha-Beta Pruning. Also, in the reinforcement learning portion, we chose to utilize Method 2. What sets *Perversi* apart from other Othello programs, however, is the many interesting extensions we chose to explore. Such optimizations include data structures like a fringe and hashtable and algorithms and procedures such as dynamic variable ordering and purely static memory manipulation. All of these extensions are discussed in greater detail further on.

Finally, we all studied the tactics and strategems of the game of Othello. As a result, we put forth a great deal of effort in choosing quality feature heuristics. Also, and just as importantly, we carefully chose initial weights for the training of our program. Since our weights are broken down into stages, we had to decide the importance of each heuristic feature for each particular stage. Again, Othello research and various websites proved invaluable in this endeavor.

2 Data Structures

2.1 The Fringe

We included a fringe data structure inside of the board structure. The fringe is an array of coordinates that represent all of the empty squares that are adjacent to occupied squares. The fringe is inexpensive to maintain and offers significant benefits: it limits the domain of possible moves for any instance of ExploreTree; it makes the calculation of the mobility heuristic much faster; and most importantly, it enables dynamic successor ordering (discussed below).

2.2 The Hashtable

During each search, our program stores the value of all non-leaf nodes in a hashtable. Entries to the hashtable have a board position as their key and a heuristic value as their value. Later during that same search – NOT during deeper searches on the same turn, however – we check the hashtable to see if a given board position has already been solved; if it has, we can save significant time by halting any further exploration from that position and returning the already-determined value.

3 Algorithms and Procedures

3.1 Alpha-Beta Pruning

Alpha-beta pruning is a standard technique to reduce the search space. We implemented this technique based on the algorithm in the course leader. We calculated our estimated speedup to be around 10-20%. The speedup was further increased by sorting the fringe cells, making the search space more suitable for alpha-beta pruning.

3.2 Dynamic Variable Ordering

Using the fringe data structure, each game tree performs a quicksort on the fringe of the root node before returning. Specifically, each game tree puts the successors of the root node (i.e. the possible moves for this turn) in order from best to worst. When the next game tree is called in the iterative deepening process, the root's successors are in the ideal order to maximize the benefit of alpha-beta pruning.

3.3 Memory Optimization

To avoid the expense of dynamic memory allocation, our program operates without a single call to malloc() (excluding the allocations for the HashTables and DArrays on which they are based). In order to use stack memory with ExploreTree, we implemented an UndoMove function that we call immediately after making a move and calling a new recursion of ExploreTree on the resulting board. UndoMove removes the piece played, flips back all flipped pieces, and reinstates the original fringe array.

We also optimize memory use by employing low-level memory functions such as memcmp() and memcpy(). These functions offer a faster method of performing operations such as copying or comparing board positions.

Finally, we turn on gcc compiler optimizations (-O3) for a further improvement in speed and performance.

4 Feature Heuristics

4.1 Pieces

Simply the number of Pieces White has MORE than Red. Our learning demonstrated that this feature is insignificant in the beginning and middle game, but increases very much in the endgame. It is normalized by dividing by the total number of pieces. Almost instantaneous to calculate, as the number of pieces for each side is built into the board data structure.

4.2 Mobility

Mobility in a position is the total number of legal moves available. Again, features must be relative, so the heuristic returns White's mobility minus Red's. All legal moves must occur in the *fringe* (See Section 2, Data Structures and Procedures) so we call *IsLegalMove()* for both White and Red on all squares currently in the fringe. The *IsLegalMove()* routine is expensive, so not having to iterate through every square on the board is much more efficient. Nevertheless, this is still probably the most time-consuming of our feature heuristics. And of course, we normalize, based on the size of the fringe.

4.3 Corner Control

There are a total of 12 corners in this game of Othello. The importance of corners is obvious – they cannot be flipped and help ensure the stability of entire regions of the board. A player is said to control a corner if the corner is occupied by a piece of her color. We count the total number of corners occupied by White compared to the total corners occupied by Red. Then we normalize by dividing over the total number of corners, 12.

4.4 Dumb Corner Occupancy

Based on a hint in the *Othello* handout, we wanted to explore the idea of "negative features." Part of playing quality Othello is not only recognizing good moves but also avoiding mistakes. We asked ourselves what is the single biggest mistake a beginning Othello player (or program!) will make. The answer is giving up a corner unnecessarily. But what sort of mistakes can lead to a corner being lost? Well, it is impossible for the opponent to capture a corner if you have no pieces adjacent to an empty corner! Thus, a common mistake in Othello would be to place a piece adjacent to an empty corner.

Thus, the concept of *Dumb Corner Occupancy* is born. If a corner square is unoccupied, then occupancy of any adjacent square is undesirable. We iterate through each corner, and check the status of its adjacent squares. The outside corners have three adjacencies, but most have four, so the total number of possible dumb occupancy squares is 48, which is what we normalize by.

4.4 Smart Corner Occupancy

Similar to the previous feature heuristic, Dumb Corner Occupancy. This time, it is advantageous to occupy a corner adjacent square if the corner is already occupied by a piece of your color. Although this feature is somewhat similar to stability, we felt is was important on its own, as it would improve corner play dramatically. Again, we normalize by dividing by 48.

4.5 Stability

The Othello board is divided into four *corner regions*. We measure the notion of *stability* as the number of consecutive pieces each side has in a corner region. For each *corner region*, we start at the "true corner," (0,0) is one of the four true corners, for example. From here, we count the number of consecutive pieces going along the edge row/column, and also the adjacent edge column. This gives us a good notion of "stability" in that respective corner. We repeat this process for all four corners. Finally, we normalize the result before returning.

4.6 Side

Simply whoever's turn it is to play. Side is 1 if it is White to play, while -1 if it is Red.

4.7 Parity

One of the most subtle stratagems in competitive Othello is the schematic strategy of *Parity*. Simply put, whoever gets the last laugh wins. It is a huge advantage to play the last move in an Othello game, because your opponent doesn't have the opportunity to recapture the pieces you just won. So Parity returns 1 if the last move is White's, while -1 if Red gets the last laugh.

5 Training

5.1 Random Boards

We divided the game into 23 stages, with each stage corresponding to a set of four moves. Consequently, we had separate weights for each stage.

In order for our weights to converge for a sequence of just four moves, we had to generate a large number of boards. The problem of generating *quality* random boards is very important in the entire training process. Certainly, the random boards should appear from positions of high quality games. So the natural choice would be to have our own smart program play itself, and take boards out of each position occurred. However, the problem is one of determinism.

Our program would keep playing the same moves, so there is no way we could generate enough test boards just from these games.

Therefore, we added an element of randomness, in order to get the desired effect. We turned the lookAhead to something very shallow, and had the program move randomly 15% of the time. As a result, a set of high-quality test boards can be benerated.

For the task of board generation, we utilized a perl script that generated almost 300 boards per game stage, for a total of 7000 boards.

5.2 Learning

Once the boards were ready, the training started. To automate training, we used yet another perl script. This script would iterate and test each stage. However, as recommended in the handout, we went in backwards order starting with Stage 23, and finishing with Stage 1. At each stage, it would train the program on all 300 boards before continuing. Each stage took approximately 45 mins to complete.

One must wonder how we managed to train 300 test boards for low stages where the position is very close to the initial starting point. Wouldn't it take forever to run a thousand games from start to end?

A key point we realized was that the purpose of testing each phase, is to *only* improve the weights for that particular phase. So if we are testing Stage 1, we only need 4 moves before the learning updates the correct weights. And since we are training in backwards order, all weights for later stages have already converged anyway.

So, to stop a game (and make training on thousands of boards a possiblity) we simply programmed the client to send an illegal move after about six moves are made.

5.2 Weight Update Algorithm

This is not too complicated. Our update algorithm corresponds to Method 2 as suggested in the Othello handout. We are at a stage s. We use minimax to make an optimal move. We wait for a opponent to make a move. We are now in state s". The update algorithm for the individual weights is:

$$w(i) \leftarrow w(i) + \gamma f_i(s)(v(s'') - v(s))$$

After careful deliberation and experimentation we chose gamma (our learning rate) to be 0.04.

We ran our scripts over all possible game boards during the training stage and stored the resulting weights in the weights.net file. Overall we attained a good degree of convergence.

5.2 Choice of Initial Weights

The choice of initial weights was mostly, as Daphne likes to refer to it, a "black art." Aided by numerous Othello strategy books as well as helpful websites, we gained the following insights:

1. Mobility is important throughout the entire game
2. Corners are key, especially in the middle game.
3. Pieces are disadvantageous early, but essential at the end.

Based on these axioms, we formulated some reasonable weight heuristics. For example, pieces are weighted by far the most at the final stage of the game, because it is the number of pieces that decide who the victor is. On the other hand, pieces were given a negative weight early in the opening and middlegame.

Also, Mobility is given a high weight throughout, and whose value doesn't really change.

Finally, corners were given a huge weight during the middlegame, because that is when they are key.

6 Time Management

Our program manages time by employing iterative deepening. Each turn begins searching at a specified minimum lookahead and continues making deeper searches until its allotted time is exhausted. The best move returned by the deepest completed search is used. However, if the final, incomplete search returns a move that appears more attractive than the previous search's move, we choose that one instead. Hence the final search, though aborted, is not always a waste of time. To allocate time, we use a simple downward-sloping line formula that results in much higher time allotments at the beginning of the game, and tiny allotments at the end. The justification for this is that by the end of the game, the program needs little time to do fairly deep searches; furthermore, we believe that the outcome of the game is usually decided well before the endgame – when one side establishes corner dominance, for example.

7 Experiments and Empirical Data

7.1 Extensions and Optimizations

We conducted numerous experiments simply in the course of testing the various optimizations that we implemented or attempted to implement. A summary of the more interesting results:

-With our iterative deepening strategy, we found that over 30% of our program's 150-second time limit is typically wasted on fruitless searches (i.e. aborted searches that do not return a better value than the search preceding them.) We also tried a variety of strategies for iterative deepening that involved

varying the starting depth, the maximum depth, and the increment. Somewhat surprisingly, none of these factors (when applied reasonably) have a significant impact on program performance: shallow iterations are relatively inexpensive (so varying the minimum depth is not very important), and running extra iterations (with a lower increment) is not necessarily wasteful because it aids dynamic ordering. One valuable finding is that the increment should not be higher than one when the lookaheads reach a depth of significant expense. For example, if the deepening goes from six to eight, it frequently fails to complete eight when it could have easily completed seven.

-Before we added iterative deepening, our program performed a shallow search to sort the fringe and then performed a deep search to find a move. We found that including the shallow search increased the speed of the deep search by between 10 and 20%.

-The hashtable was also a source for considerable experimentation. The primary question is which nodes to consider in the hashtable. There is a cost associated with entering a node into the table, so it should only be done if the expected benefit (i.e. the chances of a successful lookup multiplied by the savings of a successful lookup) outweighs that cost. For leaf nodes, this is clearly not the case. After trying various numbers, our tentative conclusion is that the optimal policy is to limit the hashtable to nodes with lookahead of two (or maybe three) or higher. With this setting, our program can perform a typical deep search anywhere from 10-40% faster, depending largely on which stage of the game we're playing in.

7.2 Othello Time Limits

We ran tests of our program Othello playing itself with different time limits, as well as on various machines with different load values. The empirical results are as follows.

Time Limit 150

fable17	*Red 67*	*White 25*
fable17	*Red 67*	*White 25*
junior	*Red 59*	*White 33*
junior	*Red 59*	*White 33*

Time Limit 100

fable17	*Red 36*	*White 56*
fable17	*Red 36*	*White 56*
junior	*Red 35*	*White 57*
junior	*Red 35*	*White 57*

Time Limit 75

fable17	*Red 31*	*White 61*
fable17	*Red 31*	*White 61*
junior	*Red 12*	*White 80*
junior	*Red 12*	*White 80*

Time Limit 50

fable17	*Red 49*	*White 43*
fable17	*Red 49*	*White 43*
junior	*Red 51*	*White 41*
junior	*Red 51*	*White 41*

Thus, for a large value of time, Red consistently seemed to defeat white, suggesting that if one can achieve a high lookahead, then playing second is definitely an advantage. However, White started to turn the tables as we reduced the time value. Finally though, Red started winning again at the lowest playing time, albeit with a small victory margin.

As the lookahead decreases, the machine with more computing power (junior) leads to a larger disparity in the final scores (case in point: the 80-12 thrashing of red for time = 75). However, for large lookahead (corresponding to time = 150), the greater CPU, led to both sides playing well, causing the game to be move balanced.

7.3 Is It Better to Play White or Red?

The answer, somewhat surprisingly, is that it is more advantageous to be Red! Unlike most competitive board games such as chess, checkers, and go, in Othello, it is not an advantage to go first.

Much of our empirical evidence supports this conclusion. Time and time again, Red seemed to have the upper hand. Whenever we have Greedy vs Greedy, Random vs Random, or even our program playing against itself, Red scores significantly better. This also holds true when different programs play each other. In our own tests against other groups, it was always more difficult to win with White than with Red.

There is one concept that explains why Red has the advantage in Othello. It is the notion of *Parity*, and it is captured in one of our feature heuristics. If neither player passes his turn during the game, there will be an even number of empty squares whenever White moves, and an odd number of empty squares whenever Red moves. From this we can conclude that red will play the final move of the game and may possess a slight advantage, since the disc which he places and those which he flips are clearly stable.

The strategy behind Parity is to maneuver so that the last move of the game is yours – an advantage that Red already has from the outset.