University of Alberta

KEYANO Unplugged – The Construction of an Othello Program

by

Mark G. Brockington

Technical Report TR 97–05
June 1997

DEPARTMENT OF COMPUTING SCIENCE
The University of Alberta
Edmonton, Alberta, Canada

KEYANO Unplugged - The Construction of an Othello Program *

Mark G. Brockington, *brock@cs.ualberta.ca*
Department of Computing Science
University of Alberta
Edmonton, Alberta T6G 2H1
Canada

July 18, 1997

Abstract

This paper describes the inner workings of KEYANO, a competitive Othello program that has achieved many top-three finishes in tournament play over the last five years. The unique features of KEYANO's midgame search routine, evaluation function and opening book are described in this paper.

1 Introduction

Othello[1] programming is a very interesting field. It attracts many programmers interested in game-tree search because the game has very simple rules and evaluation functions are relatively easy to construct. Furthermore, it is easy to design a program that will defeat the programmer.

However, most first attempts at writing an Othello program are significantly flawed. Commonly-held misconceptions include that opening knowledge is unimportant, endgame solving is vital, and the midgame evaluation can be taken care of by assigning weights to each type of square on the board (for example, corners are good, and squares beside the corner are bad).

Opening knowledge is vital for competing in the upper echelons of computer Othello. Most Othello programs have extensive opening books, allowing the program to play a large number of their 30 moves without using any time on the clock. However, simply having a large database of games is insufficient. The games in the database must be good, and the games should be examined, analyzed for mistakes and corrected before they are placed in the book for the program to use.

Using the midgame evaluation described above limits the strength of the program. No matter how well tuned the square evaluations are, the program will lose to a reasonable

*This paper has been accepted for presentation at the "Game Tree Search in the Past, Present and in the Future" Workshop at the NEC Research Institute, Princeton, NJ, August 1997.

[1] Othello is a registered trademark of Tsukuda Original, licensed by Anjar Co.

evaluation function consisting of a mobility measure and tables of pre-computed pattern values.

Endgame solving is somewhat important, but only if there is a vast disparity between when the two players solve the endgame. In top-level play, most programs solve within one or two moves of one another. At current computer speeds, 24-ply to 26-ply win/loss/draw solves are not uncommon. Getting to a position where the computer can solve for a win is much more important than having a fast endgame solver.

Finally, a database of Othello games played by strong players is vital for training and testing all phases of the program: the opening, the midgame and the endgame.

This paper will describe the key components of KEYANO[2], the author's Othello program. It has routinely placed in the top six in 21 computer Othello tournaments over the last five years. This includes 2nd, 5th, 3rd and 4th place finishes in the four Paderborn Computer Othello tournaments, believed to be the toughest field of Othello-playing computer programs ever assembled.

Section 2 describes the routine used for searching midgame positions, including KEYANO's independent implementation of multiple-level ProbCut. Section 3 describes the evaluation function in detail, including detailed descriptions of KEYANO's complex mobility function and parity approximation. Furthermore, the method of training pattern databases using adaptive logic networks is described in detail, along with KEYANO's method of generating evaluation function coefficients. Section 4 describes the opening book design used by KEYANO. Finally, Section 5 gives some games where KEYANO plays well against LOGISTELLO and HANNIBAL, the top Othello-playing programs in the world at the time of writing.

2 Midgame Searching

Othello and chess belong in the same abstract class of games: two-player zero-sum games with perfect information. Thus, it is not surprising to discover that the construction of an Othello search routine is very similar to a chess search routine.

The basis of the search routine is an iteratively-deepened $\alpha\beta$ search. Variants of the $\alpha\beta$ search routine work better in practice, both in terms of time spent executing the search and the number of nodes explored. NegaScout, as proposed by Reinefeld [12], is employed in KEYANO, since $\alpha\beta$ searches 7% more nodes than NegaScout in the current version of the program.

2.1 Move Ordering

The move ordering within KEYANO is led by three heuristics: the transposition table, killer moves [15], and the history heuristic [14]. Each of these heuristics are commonly used in chess programs, and all of them work well in KEYANO. We will briefly mention the key points where KEYANO's heuristics differ from the common implementation found in other game-playing programs.

Most programs generate a 64-bit number by combining predetermined random numbers for the game pieces and their locations on the board. A portion of this 64-bit number is used

[2]Keyano was the mascot of the 1978 Commonwealth Games, held in Edmonton.

Trans. Table Size	Avg. Tree Size	Size Increase over 2^{20} Entry Result
2^{20}	974614	0.00%
2^{18}	978746	0.42%
2^{16}	1001342	2.74%
2^{14}	1058428	8.60%
2^{10}	1272702	30.59%

Table 1: KEYANO, Transposition Table Size and Effect on 10-ply Fixed-Depth Searches

as the hash key for the transposition table. The rest of the number is used as a hash lock to determine if the position in the transposition table is the same as the current position in the game tree. Unlike other programs, KEYANO stores the entire position into the hash table as the hash lock. The board representation within KEYANO is only 128 bits. The author did not feel that it was worthwhile to map the board representation into 64 or 32 bits, since this mapping allows for identification of dissimilar positions as identical positions on rare occasions. The drawback of storing the full board is that the size of each transposition entry grows from 8 bytes to 20 bytes.

The transposition table in KEYANO normally has 2^{20} entries. For searches that take 60 seconds on an SGI Challenge (the computer KEYANO usually runs on), this size of transposition table is sufficient to guarantee the majority of the benefits. We see the results of varying the transposition table size in Table 1, where we have searched 20 typical mid-game positions to a fixed depth of 10 ply.

The transposition table is not subdivided into a two-level table as advocated by other researchers. The benefits of using a two-level transposition table are less than 1% for KEYANO, in terms of nodes searched.

Although the killer moves are used in the standard way, the history heuristic information is used to determine a static move ordering for a given tree search. Only between iterations of iterative deepening are the move lists sorted by the history heuristic information.

If we use KEYANO to explore the aforementioned series of fixed-depth 10-ply game trees, we can get some measures of how well the move ordering heuristics combine with one another. At a Knuth type-2 (CUT) node, KEYANO searches an average of 1.10 nodes over a series of fixed-depth 10-ply searches. Furthermore, the best move is searched first at any node within the game tree 89%-93% of the time. Thus, we feel that the move ordering within the game tree is reasonable.

2.2 Search Extensions and Reductions

Most methods of search extensions and reductions that work in other domains do not work in Othello. For example, null moves do not work directly in Othello because it is often preferable to pass in a given position. Fortunately, there are other methods which do work in Othello.

ProbCut [4] is the selective reduction algorithm of choice for the game of Othello, and has dramatically improved the search depth reached along main lines in KEYANO. The general idea behind ProbCut is that we wish to determine if a d-ply search will fail high or low, by approximating it with a d'-ply search that has a wider search window where $d' < d$. If the d'-ply search fails low or fails high, then the d-ply search will behave the same way with high probability, and the appropriate window bound (α or β) is returned back up the tree. The original implementation of ProbCut used $d = 8$ and $d' = 4$. However, it is clear that this method can be generalized, and used at many levels within the search tree.

In KEYANO, we attempt to approximate a d-ply search with a d'-ply search where $d' = \lfloor d/2 \rfloor$. Any non-ProbCut search, from $d = 4$ up to $d = 10$ ply, can be considered for approximation by ProbCut. Although it is possible to change how the window of the d'-ply search is computed at various stages throughout the game, the window sizing parameters in KEYANO are independent of the stage of the game. The window used for the d'-ply search is translated first by the mean of the difference between the d' and d-ply values (as done in single-level ProbCut) and then widened by 1.2 times the standard deviation of the difference. The factor of 1.2 was chosen because it yielded the best result in self-play matches against a non-ProbCut version of KEYANO. We should note that this factor yields tighter search windows than Buro's original bound of 1.5 in LOGISTELLO.

The mean and standard deviation are determined from the search of 1000 distinct positions from various stages within the game (moves 5 to 45 in 5 move increments). The 1000 positions come from a random sampling of a database of top-level computer Othello games. Each position is searched to 10 ply, allowing the determination of all necessary statistics for any selection of d and d' up to 10 ply.

The implementation of multiple-level ProbCut dramatically improves KEYANO's search depth. Although the maximum pruning possible from the current implementation is five ply, KEYANO routinely searches four ply further ahead when multiple-level ProbCut is used during the search. However, this is somewhat misleading. For example, when KEYANO reports it is searching 16-ply, it is really searching all variations to 11 ply, while extending important lines up to 5 ply further.

We would also like to know whether ProbCut increases the playing strength of Keyano. To measure this, KEYANO played 70-game self-play matches against other versions of itself on a private version of the Internet Othello Server (IOS) that has been set up at the University of Alberta. The openings in the match are taken from the 35 starting positions presented in Buro's Ph.D. thesis [3]. To ensure that the experiment is repeatable and not based on varying processing loads, the time control is based on the number of nodes searched. If the program is not playing to a fixed depth, each program must immediately terminate a search once 512000 nodes have been evaluated, and announce their move choice.

Table 2 summarizes the results of two self-play matches between versions of fixed-depth KEYANO and KEYANO with single-level ProbCut versus KEYANO with multiple-level ProbCut. As we can see, there is a significant advantage to be obtained from using multiple-level ProbCut over both single-level ProbCut and fixed-depth searches. We can compare these results to fixed-depth self-play matches (Table 3). A comparison shows that the benefit from implementing multiple-level ProbCut is equivalent to approximately 1.5 ply of additional search.

Similar methods of implementing multiple-level ProbCut have been developed independently by Michael Buro, the author of LOGISTELLO, and Martin Piotte and Louis Geoffroy, the authors of HANNIBAL.

Winner Of Match	Loser Of Match	Score For Winner (Avg. Disc Differential)
multi-ProbCut	no ProbCut	67.2% (34.7 - 29.3)
multi-ProbCut	ProbCut	64.2% (33.8 - 30.2)

Table 2: KEYANO, Results of ProbCut Self-Play Matches

Winner Of Match	Loser Of Match	Score For Winner (Avg. Disc Differential)
11-ply	10-ply	62.1% (33.2 - 30.8)
12-ply	11-ply	62.8% (33.6 - 30.4)
11-ply	9-ply	70.4% (36.0 - 28.0)
12-ply	10-ply	72.1% (34.7 - 29.3)

Table 3: KEYANO, Results of Fixed-Depth Self-Play Matches

2.3 Parallel Search

The author has been investigating methods of parallel search in $\alpha\beta$-based game-playing programs for his Ph.D. research. A natural test bed for these approaches is KEYANO.

Preliminary versions of APHID [2], a portable parallel game-tree search library, have been used in Keyano's tournament appearances over the last two years. APHID is an asynchronous game-tree search algorithm. Unlike most other published approaches to game-tree search, APHID does not impose any global synchronization points over the course of a search.

APHID defines a frontier a fixed number of moves away from the root of the search tree. A "manager" process sends all frontier nodes to "worker" processes to be evaluated. Each worker process is assigned an equal number of frontier nodes to search. The workers continually search their frontier nodes deeper and deeper, reporting results to their manager. The manager process repeatedly searches to the frontier nodes to retrieve the latest search results. After each pass of the tree, the manager reports any changes in the work lists to the workers. For both manager processes and workers, there is effectively no idle time; inefficiencies are primarily due to search overhead. APHID's performance does not rely on the implementation of a global shared memory or a fast interconnection network between the processes, which makes the algorithm suitable for loosely-coupled architectures (such as a network of workstations), as well as tightly-coupled architectures. APHID uses PVM [8] as a message passing interface to allow for the maximum portability among available hardware.

APHID is successful in KEYANO, because the top of the tree (explored by the manager) is stable between iterations. In recent testing, the fixed-depth version of KEYANO has achieved speedups of 13.5 on a 16-processor SGI Origin 2000 system. This compares favourably to Young Brothers Wait [7], which achieves a speedup of 10.5 over the same set of tree searches. The ProbCut-enhanced version of KEYANO has achieved speedups between 10 (when using local transposition tables) and 12 (when using a shared-memory transposition table accessible by all processors).

3 Evaluation Functions

In a minimax-based search algorithm, one needs an evaluation function to map a position into a value that can be manipulated to determine the minimax value of the game tree.

What is the general structure of this evaluation function? In most programs, the evaluation function follows a linear model. We take a series of k features from the position, assign values to each of the features, and multiply them by predefined weights to achieve the evaluation. If $\mathrm{f}_i(p)$ is the numerical value of the ith feature in position p, and w_i is the weight associated with the ith feature, the formula for evaluating a position p is $\mathrm{eval(p)} = \sum(\mathrm{f}_i(p) \times \mathrm{w}_i)$.

In general, an evaluation function should have a maximal value (such as $+\infty$) to represent a winning position, while a minimal value (such as $-\infty$) represents a lost position. However, these maximal and minimal values are not absolute. For example, LOGISTELLO's former evaluation function attempts to determine the probability of winning from a position [5]. This implies that the evaluation of a position varies between 0 and 1[3]. Logistello's new evaluation function and the evaluation function within KEYANO attempt to approximate the final outcome of the game, returning a value between -64 and $+64$.

3.1 Features of the Evaluation Function

Before we discuss how to generate the feature weights, we must first define the features KEYANO uses in the evaluation function. There are many samples of features for an Othello evaluation function in the literature. One of the obvious features is the number and location of discs on the board. For example, a beginner quickly ascertains that corners are good, and an easy feature to implement is the number of corners the player owns minus the number of corners the opponent owns. This can be generalized to other types of squares on the board, and has been called a *weighted squares* evaluation in the literature. However, the game of Othello is a lot more complicated than this simplistic model. Although the weighted square model is a reasonable heuristic, there are sufficient occasions in the typical Othello game where this generalization will yield poor moves. Rosenbloom's paper on IAGO [13], Lee and Mahajan's work on BILL [11], and Kierulf's work on PEER GYNT [9] illustrate many different features that can be used.

KEYANO has only two types of features. The first is a measure of the mobility in a given position. The second is a series of patterns from areas around the edge of the board.

3.1.1 Mobility

There are many different ways of computing a mobility feature in an Othello position. Each of the previously mentioned programs proposed a different method for generating the feature. IAGO [13] used a non-linear combination of the number of moves each player has at a leaf node to generate the mobility score in a given situation. BILL [11] optimized the computation of mobility at a leaf node by pre-computing arrays that determined the possibility of flipping discs along any of the 38 rows, columns or diagonals. The indices to access the correct

[3] To avoid using floating-point numbers, the internal representation of the evaluation function is an integer encoding of the logarithm of the winning probability over the losing probability: $\log(p/(1-p))$. This internal representation can be translated to the probability of winning when presented to the user.

elements in the 38 arrays are generated at the root of the game tree, and are updated for each disc placed or flipped on the board. Thus, when we arrive at a leaf node, the mobility computation at a leaf is a sum of 38 values retrieved from arrays at the pre-computed indices. This makes the mobility computation extremely fast, but does not return the exact mobility score. Instead, this only returns the number of directions that discs can be flipped; playing to one square may flip discs in all four main compass directions. The number of disc-flipping directions, also known as BILL-mobility, is an approximation which is only slightly worse than using exact mobility, but is significantly faster to compute.

PEER GYNT [9] used a knowledge-based approach to the game, as opposed to the brute-force methods advocated by the authors of IAGO and BILL. *Parity* is introduced in PEER GYNT as part of the mobility calculation. In brief, the concept of parity is to know who will likely be the last player to move in the game. White has parity at the start of the game, since Black must move first and there are an even number of squares to be filled in the game. The advantage of having parity is very important in Othello, and is widely believed to a decisive advantage in White's favour. By removing the ability to play to a region of the board, one may be able to force parity to swap to the other player. In human Othello matches, the typical scenario is that a weak player loses access to a region of the board early in the game, and the stronger player can wait patiently until a pass is forced by avoiding to play into that region. It is important to note that parity cannot be determined by deep searches near the beginning of the game; evaluation function knowledge is required to determine the parity.

How exactly is the mobility term computed in KEYANO? We take the number of disc-flipping directions for each player and transform them to a base feature value. This base feature value is augmented by the result of evaluating the disc-flipping directions by board region. Finally, the computation of the number of parity-losing regions is added to generate the complete mobility feature.

We shall start by discussing how the number of disc-flipping directions for the full board are transformed into evaluations that can be used to create the base feature value. Using sample positions, we can determine the value of having k disc-flipping directions over the entire board, and compute its correlation to the disc differential at the end of the game. This gives a curve that looks logarithmic when viewed with the number of flipping directions along the x-axis and the evaluation of that number of flipping directions on the y-axis, as illustrated by the full board line in Figure 1. The shape and logarithmic curve of the full board line is intuitive, since it is more important to increase the number of options available when you have very little mobility than when you have many moves to choose from.

The full board BILL-mobility evaluations can be stored within the program, and used to scale both the number of disc-flipping regions for the player and the opponent. Subtracting the two values from one another gives us the base mobility feature value.

However, this does not do a reasonable job of approximating the location of the moves over the entire board. If the locations where discs can be flipped are concentrated in one region of the board, the disc-flipping regions are likely to interfere with one another. Thus, we would like to capture whether the moves are spread out over the entire board.

In KEYANO, we divide the board into 16-square quarters along the vertical and horizontal lines through the middle of the board. We can determine an evaluation for the number of disc-flipping directions in each quarter of the board in exactly the same way that we computed an evaluation of the disc-flipping directions over the full board. The 16-square line in Figure

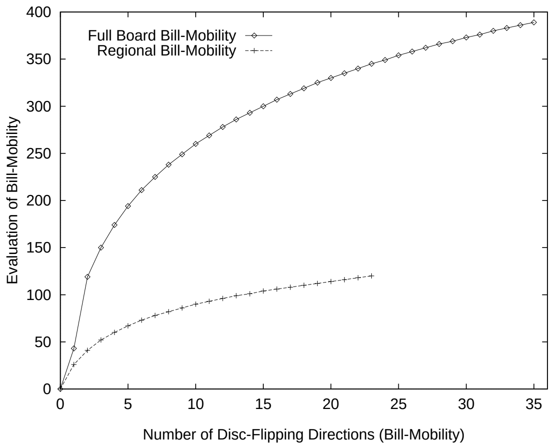

Figure 1: Evaluation of BILL-Mobility for Entire Board and in 16-Square Regions

Mobility Feature Value	Correlation			
	Mv 20	Mv 30	Mv 40	Mv 50
Exact	0.199	0.380	0.575	0.509
Bill	0.181	0.353	0.563	0.516
Bill+Parity	0.186	0.368	0.577	0.515

Table 4: KEYANO, Correlation of Mobility Measures to Final Disc Differential at Various Stages Within an Othello Game

1 illustrates the values determined by a linear regression for the value of regional mobility. Note that since there are four of these regions on the board, the evaluations returned are approximately one-quarter of the values returned for the full board.

The two parts of the evaluation are then added together to generate a new feature value. The full board BILL-mobility is assigned an equal weight as the regional BILL-mobility measure, since experimental evidence showed that any other weighting led to a lower correlation of the combined feature evaluation with the final disc differential.

We can follow the same methodology to generate the values for the exact mobility feature to see whether the correlation to the final disc differential is sufficient to ignore the speed benefits of the BILL-mobility measure. The first and second lines of Table 4 give us the correlation of the exact mobility and Bill-mobility measures (including the combination of the full board and regional evaluations) over various stages within the game. As we can see, exact mobility has a marginally better correlation to the final disc differential. In practice, the difference is not sufficient to prevent KEYANO from using the faster BILL-mobility measure.

KEYANO also has an approximation of parity within the evaluation. We first determine the regions on the board by using a space-filling algorithm; any empty square that is attached in any of the eight compass directions to an empty square belong in the same region as one another. Until the empty squares are physically separated, this results in one region at the beginning of the game. However, the board rapidly separates into many smaller regions. After these regions are determined, we use the exact mobility generation to determine where each side is able to play. The algorithm determines whether by playing two consecutive moves, we can play to new squares that are not in the original mobility list. If a region of the board can not be played to by one side, nor can that side force access into the region, this is considered as a potential parity-losing region.

The computation of potential parity-losing regions is extremely expensive. Thus, the determination of parity-losing regions is done 4 ply away from the leaves. Even moving the computation this far away from the leaves slows down Keyano's nodes evaluated per second by 15%. The computation of potential parity-losing regions is really an approximation, since the situation may change within the final 4 ply of the game tree.

The optimal weight for the addition of parity to the Bill-mobility measure was determined by the value which led to the best correlation to the final disc differential. In KEYANO, 64 is taken away from the mobility evaluation feature when the player to move has lost access to a potential parity-losing region. The third line of Table 4 give us the correlation of the BILL-mobility feature when combined with the parity computation. When BILL-mobility is combined with parity, we see that the correlation to the final disc differential improves slightly.

The addition of the parity feature to the Bill-mobility (computed for both the entire board and for regions) to generate a single mobility feature gives a significant boost to the overall strength of KEYANO. We ran a 70-game match between KEYANO without parity and KEYANO with parity in June 1996. Since the program with parity takes about 15% longer to search 512000 nodes, the program without parity was allowed to search 15% more nodes than the program with parity. Even with the additional nodes given to KEYANO without parity, the match result was 40.5-29.5 in favour of the version of KEYANO with the parity adjustment enabled. Thus, the author feels that the significant costs associated with the computation of parity are justified.

3.1.2 Patterns

Buro took the concept and implementation of the vectors employed by BILL into a generalized framework of patterns to generate features for LOGISTELLO [3]. In Buro's scheme, the patterns do not necessarily represent simple vectors on the board; the region may be a 3 by 3 square surrounding a corner, a 2 by 4 region near a corner, a group of 8 squares near the centre of the board, *et cetera*. The patterns selected for inclusion in the evaluation function were the ones that did the best job of discriminating between won and lost positions.

KEYANO uses a series of vectors as patterns. The 8-disc vector along the edge of the board, the 8-disc vector one away from the edge of the board, and all of the 7 and 8-disc diagonals are relatively important at some stage of the game. Thus, they are all used by the evaluation function.

KEYANO's set of patterns includes a 3 by 3 square around a corner. KEYANO also uses one of two 10-square edge patterns. Figure 2a illustrates the squares examined if none of the squares marked N have discs on them, while Figure 2b illustrates the squares examined if one of the squares marked N from Figure 2a does have a disc on it. This approach of discriminating between the two types of edges was first implemented by Colin Springer for ECLIPSE. The switch from the first to the second pattern takes place during the midgame search when a square marked N for that edge is occupied.

In the general case, switching features during the middle of a search can be dangerous. However, allowing the 10-square pattern to change depending on whether the squares marked

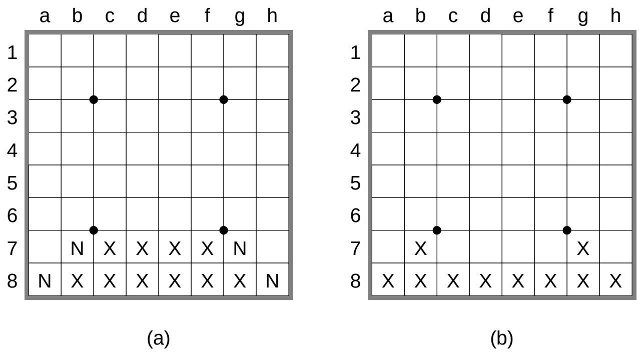

Figure 2: Two Different Edge Patterns Used

N are occupied is relatively safe in KEYANO. In essence, the first 10-square pattern can be considered as a 14-square pattern, because it can only be used when the four squares marked N are blank. The second 10-square pattern is the important subset of the full 14-square pattern. By leaving out c7, d7, e7 and f7 in Figure 2a, the value of the edge pattern will not change dramatically for the majority of possible edge configurations. Only in a few key situations will the value change dramatically. Fortunately, the majority of these exceptions occur when none of the squares marked N are occupied (which is evaluated by the first 10-square pattern) or involve 1 or 2-square holes along an edge (which are handled by the parity approximation in the mobility feature).

Many other patterns have been tried, including a 2 by 5 edge pattern extending from a corner, but only the vectors, 3 by 3 corners and the 10 disc edge patterns have been successful additions to KEYANO's evaluation function.

Once we have defined a pattern, one must determine how good every possible configuration of the pattern is. One method of doing this is by examining a large database of games to look for samples of these configurations, and compute statistics on how often each of the configurations appeared and correlate this to the probability of winning or the average disc differential.

What sort of database of games would one like to examine? If we use a database that contains games of suspect quality, the statistics garnered from those games may not be reliable. In games between poor opponents, the winner is usually the person who makes the second-to-last game-theoretic mistake. Thus, the ability to determine that a specific configuration in a pattern leads to a winning position or a good disc differential is flawed. If we only use a database of tournament games, we only see a small sample of the number of possible patterns. For example, we may only see situations where playing to an X-square (a square one away from the corner along a main diagonal) yields reasonable results, because the good players know how to sacrifice X-squares correctly. We may never see the converse situation where an X-square is played incorrectly and the player is immediately and swiftly punished. Thus, the ideal database of games contains a wide variety of random openings and midgames played by good Othello players. Since it is generally believed that computers are playing Othello better than most humans, it is better to take a database of good computer Othello games than human Othello games for this purpose.

When we have more than 100 samples for a pattern configuration, KEYANO uses the average disc differential as the configuration's value in the evaluation function. However, even a large database of games may be insufficient to give 100 samples for a specific pattern configuration. The problem is to determine a value for a pattern configuration that does not appear frequently in the database.

Buro uses the probability of winning for all pattern configurations, and an evaluation of 0.5 for any configuration that does not appear in the database. However, using only a relatively small number of samples to determine an evaluation of a pattern configuration is hazardous. Patterns that have not been seen in regular play may not show up in the database, but it may be critical to evaluate these patterns correctly in an important tournament game. No programmer has the time to examine thousands of pattern configurations and hand-code values for each one. Thus, we need an intelligent classifier to determine an evaluation of these unseen or rarely-seen pattern configurations.

In KEYANO, we use neural networks to assist in determining the value of pattern configurations that do not appear 100 times within the database. A neural net software package using adaptive logic networks (ALNs) [1] was used to facilitate the process.

The pattern configurations that occurred over 100 times were used as the training data for the ALNs. This yielded about 2000 distinct pattern configurations for each of the 8, 9 and 10-square patterns during the initial development of KEYANO's evaluation function. Each pattern configuration appeared only once in the training data, and was not replicated based on its frequency.

To use a neural network to determine the rest of the pattern configurations, we first use the training data to teach the neural network to determine the values of the pattern configurations in the training data. We can then feed the under-determined pattern configurations into the neural network, and use the output to substitute for the unavailable samples from the database.

For example, a given pattern configuration has occurred 20 times and yields an average disc differential of -4.0. After training the neural network, we give the pattern configuration to the network and it determines that the value of the configuration should be -8.0. Thus, we take the weighted average of the two $(20 \times -4.0 + 80 \times -8.0)$ to determine an evaluation for the pattern configuration of -7.2. If the pattern configuration did not appear in the database of games, it would be assigned a value of -8.0.

One of the interesting curiosities of the ALN package used to determine the values is that each ALN only returned a single binary output. It was decided that instead of having only one ALN, there would be 100 ALNs to be trained. Each randomly-generated ALN would attempt to discriminate between the pieces of training data that won with an average disc differential of x, as x increased from -49 to 50. Once all of the ALNs were trained, each pattern was run through the 100 ALNs, and the sum of the binary outputs of all of the ALNs, minus 50, became the value used to replace the missing samples for the under-determined pattern configurations.

There are three main pattern features that are used in the evaluation function: the 7-disc and 8-disc vectors, the 9-disc corner pattern, and the 10-disc edges. The value of all of the 7-disc and 8-disc vectors are added together to make a single feature of the evaluation function. This is due to the method of coefficient generation, as we shall see in the next section.

3.2 Coefficient Generation

Now that we have discussed each of the features that KEYANO examines in an Othello position, and how each of these features can be turned into a numerical evaluation, we must now generate an evaluation function from these features by generating multiplicative constants or weights.

Othello is a wonderful domain for studying how to combine merits and generate an evaluation function, since the tactics of games such as chess and checkers often tend to hide the benefits and drawbacks of these methods. BILL used a quadratic discriminant function [10] in an attempt to improve upon the linear evaluation function. Buro later showed data that Fisher's linear discriminant and the quadratic discriminant function are both weaker than a linear evaluation function determined by logistic regression [5].

Despite the advantages of these approaches, KEYANO has always used a linear regression to determine the constants to be used in the evaluation function. The reason behind this is that the author believes that the expected disc count at the end of the game is a natural metric for success, rather than the probability of winning, which is encoded by the logistic regression and the quadratic discriminant function.

The training data for KEYANO is generated by a sample from a large database of games. For generating coefficients after k moves have been played, representative positions where $k-2$ to $k+2$ moves have been played are taken from each game[4]. The primary reason for taking positions other than at k moves within the game is to smooth out the curves and yield evaluations that do not leap radically as one increases the depth of search.

One problem with completely automatic weight generation is that linear regressions can yield negative weights for heuristics that are positively correlated with the expected outcome or probability of winning. This can occur when two features are correlated. If one has a much stronger correlation to the expected value than the other feature, the first feature may be overemphasized while the second feature may be given a negative weight. Thus, the final phase of the computation is a check by the author to see whether the generated weights are all positive. Negative weights are hand-tuned by the author. In general, the weights are positive, since each of the features are relatively independent from the other features.

The ability of the program to determine negative weights for features that are positively correlated with the disc differential is disturbing. This problem has driven the design of the four main features currently used in KEYANO. For example, the parity, the full-board mobility and regional mobility features are each correlated to one another and to the final disc differential. Thus, they are all combined into one feature to prevent giving one of the features a negative weight. A similar problem forced the author to combine all 7 and 8-disc vectors into one feature.

Figure 3 gives the relative strength of the coefficients for the mobility, vector patterns, 3x3 corner pattern, and the 10-disc edge patterns used within KEYANO. All of the lines have been scaled so that the graph shows how each feature is weighted over each stage of the game, with respect to its maximal weight. As we can see, mobility rapidly increases in

[4] The reader may wonder whether taking the representative positions in this asymmetric manner might be the cause of the odd/even effect in Keyano's evaluation function. The author has experimented with many different ways of taking the samples from the database, and every method yields an odd/even effect when used within KEYANO.

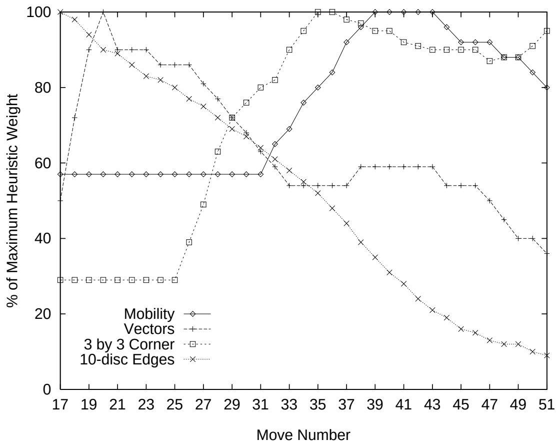

Figure 3: KEYANO, Comparison of Coefficient Weights Over Various Stages in an Othello Game

importance as the game progresses, while both the vector and 10-disc edge patterns are much more important near the beginning of the game. The flat lines on the left side of Figure 3 represent values that were fixed by the author during the final stage of generating weights for the coefficients. Finally, the graph ends at move 51 because any position at nine empty squares or less is handled by a special-purpose endgame solver that does not use KEYANO's evaluation function.

4 Opening Book

KEYANO's opening book is based on the method used in SPOCK. The method described by Delteil [6] is based on a permanent transposition table structure which contains positions, best moves and current minimax values. Unlike a standard transposition table, the work begins when a position is added onto one of the branches in the database. Every time a new position is added into the database, the parents of that position must be examined to determine if any of the move choices or minimax values change. This propagation continues all the way up the tree to the root if necessary. This would be very simple if there was only a single path up the tree. However, one must follow every path from move transpositions up the tree. In some positions of the Tiger, the preferred opening in computer Othello play, there are over 30 separate paths to the root from often-played positions.

Delteil notes that the only time we need to do any additional search is when the current best move from a node drops in value. At that point, we must see if there are any additional moves (that we have not yet considered) in between the new best value (α) and the old best value (β). All of the other changes are simple changes to the best move and/or best score.

How does KEYANO use Delteil's method to generate an opening database? A database of good Othello games is required, and each of the games should have the endings corrected to a suitable depth. Each game from the game database can be added to the opening database. For any position not currently in the database, best move alternatives are examined to ensure that the moves played are reasonable. This prevents mistakes from entering the database where a player reaches a crushing position and then fails to play optimally. The search that is completed when examining best move alternatives is relatively short: KEYANO takes the average of the values returned by 11-ply and 12-ply searches as the evaluation of a position. The level chosen is not larger because the book was started before the implementation of multiple-level ProbCut within KEYANO. KEYANO often takes a long time to test numerous move alternatives to see if they are worthwhile at 12-ply. If the computation of the opening book was restarted today, KEYANO would use 15-ply and 16-ply multiple-level ProbCut searches to determine the best move alternatives, since that is the search depth usually obtained during the search.

The database is stored as a large hash table on disk, with each entry containing 40 bytes. To avoid long delays when loading data, KEYANO partitions the database into a series of buffers, each of which can get no larger than 1000 positions. This allows KEYANO to load only the specific part of the opening database that is required. The current opening database consists of 1024 buffers, with over 800,000 positions in total. An index file, which is loaded whenever KEYANO is started, lists the physical location and the range of hash values that each buffer contains.

5 Selected Games

LOGISTELLO, by Michael Buro, has been the top-performing Othello program for the last four years, amassing a record of 15 first place and 5 second place finishes in 21 computer Othello tournaments. Although KEYANO has achieved many draws against LOGISTELLO, a game from the 1996 Paderborn Computer Othello tournament (shown in Figure 4) represents the first time that KEYANO has achieved a won endgame against LOGISTELLO. Unfortunately, KEYANO did not win the game due to an unfortunate blunder on move 42, caused by a typographical error in the interface between the APHID library and the Othello program. This mistake allowed LOGISTELLO to win the 1996 Paderborn tournament over HANNIBAL by a half-point. If KEYANO had not blundered, HANNIBAL would have won the tournament.

HANNIBAL, by Martin Piotte and Louis Geoffroy, is a new program by the authors of BRUTUS that is set to challenge LOGISTELLO's supremacy as the top Othello program in the world. HANNIBAL has won or finished second in the last few tournaments it has entered, and games between LOGISTELLO and HANNIBAL are normally decided by parity; White normally wins. However, KEYANO has been moderately successful against HANNIBAL. KEYANO knocked HANNIBAL out of first place in the April 1997 IOS Open tournament with two draws in the final round. Although one of the games has been shown to be a won endgame for HANNIBAL, the game in Figure 5 has, to the author's knowledge, not been proven to be a win for either player. The two draws allowed LOGISTELLO to overtake HANNIBAL in the final round of the tournament.

6 Conclusions

In this paper, we have outlined some of the concepts and design of KEYANO. As alluded to in the introduction, it is the author's hope that this paper inspires other programmers to build world-class Othello-playing programs, and build upon the ideas and concepts presented here.

	a	b	c	d	e	f	g	h
1	44	45	34	35	42	51	54	47
2	39	43	31	28	33	49	46	40
3	36	25	24	1	6	21	27	26
4	32	29	10	○	●	15	23	22
5	38	30	2	●	○	4	14	48
6	37	41	11	7	3	5	9	17
7	50	58	18	13	12	8	55	52
8	59	60	16	19	20	57	56	53

Figure 4: Logistello (Black) 33-31 Keyano (White), October 1996 Paderborn Computer Othello Tournament

	a	b	c	d	e	f	g	h
1	59	60	45	44	49	48	47	58
2	46	50	43	37	28	18	55	41
3	31	35	5	3	24	11	16	40
4	30	26	4	○	●	2	34	39
5	29	17	19	●	○	9	10	27
6	42	23	8	6	1	7	14	25
7	53	38	22	32	12	13	57	56
8	51	36	52	33	15	20	21	54

Figure 5: Hannibal (Black) 32-32 Keyano (White), April 1997 IOS Open

It is very interesting to note that the opening book, the search routine and the midgame evaluation function each used a database of games to train and/or tune parameters. Training from top-quality Othello games, is vital to the success of any Othello program. This is in contrast to chess, where very little training based on grand-master games is done. The author currently has a database of 36,000 games played on the Internet Othello Server or extracted from the Thor database. The database currently includes games played by humans that have been awarded the title of World Othello Champion, along with games played by strong Othello programs. All of the games have been back-solved or corrected to 22 empty squares or more. Generating or acquiring a database of this type will assist interested readers in developing a world-class Othello program.

Although programs are still a long way from completely solving the game of Othello, the best programs are believed to be significantly stronger than World-Champion calibre tournament Othello players. The Logistello-Murakami match will show whether this commonly-held belief is true.

7 Acknowledgements

Many people have contributed and assisted with KEYANO over the years. The other members of the University of Alberta GAMES group have been a valuable resource for lively discussions and many improvements to KEYANO: Yngvi Björnsson, Yaoqing Gao, Andreas Junghanns, Tony Marsland, Denis Papp, Aske Plaat and Jonathan Schaeffer.

I would also like to thank other programmers and members of the Othello community for their hospitality, games and conversations: Bill Armstrong, Bruno de la Boisserie, Michael Buro, Jean Delteil, Igor Durdanovic, Louis Geoffroy, Paul Hsieh, David Parsons, Martin Piotte, Brian Rose, Colin Springer, Vince Sempronio, Marc Tastet, and Jean-Christophe Weill.

Last, but not least, I would like to thank Jonathan Schaeffer, Jennifer Walchuk and the anonymous referees for suggesting improvements to the presentation of this paper.

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

References

[1] W. Armstrong and J. Gecsei. Adaptation Algorithms for Binary Tree Networks. *IEEE Transactions on Systems, Man and Cybernetics*, 9:276–285, 1979.

[2] M. G. Brockington and J. Schaeffer. The APHID Parallel $\alpha\beta$ Search Algorithm. Technical Report 96-07, University of Alberta, Department of Computing Science, Edmonton, Canada, August 1996.

[3] M. Buro. *Techniken für die Bewertung von Spielsituationen anhand von Beispielen.* PhD thesis, University of Paderborn, Paderborn, Germany, October 1994. In German.

[4] M. Buro. ProbCut: An Effective Selective Extension of the Alpha-Beta Algorithm. *ICCA Journal*, 18(2):71–76, 1995.

[5] M. Buro. Statistical Feature Combination for the Evaluation of Game Positions. *Journal of Artificial Intelligence Research*, 3:373–382, 1995.

[6] J. Delteil. A Propos des Bibliotheques d'Ouvertures. *FFORUM*, 29:18–19, 1993. In French.

[7] R. Feldmann. *Spielbaumsuche auf Massiv Parallelen Systemen.* PhD thesis, University of Paderborn, Paderborn, Germany, May 1993. English translation available: Game Tree Search on Massively Parallel Systems.

[8] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, B. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine – A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, 1994.

[9] A. Kierulf. *Smart Game Board: a Workbench for Game-Playing Programs, with Go and Othello as Case Studies.* PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 1990.

[10] K.-F. Lee and S. Mahajan. A Pattern Classification Approach to Evaluation Function Learning. *Artificial Intelligence*, 36:1–25, 1988.

[11] K.-F. Lee and S. Mahajan. The Development of a World Class Othello Program. *Artificial Intelligence*, 43(1):21–36, 1990.

[12] A. Reinefeld. An Improvement to the Scout Tree-Search Algorithm. *ICCA Journal*, 6(4):4–14, 1983.

[13] P. S. Rosenbloom. A World-Championship-Level Othello Program. *Artificial Intelligence*, 19:279–320, 1982.

[14] J. Schaeffer. The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-11(11):1203–1212, 1989.

[15] D. J. Slate and L. R. Atkin. Chess 4.5 - The Northwestern University Chess Program. In P.W. Frey, editor, *Chess Skill in Man and Machine*, pages 82–118. Springer-Verlag, New York, 1977.