The Development of a World Class Othello Program*

Kai-Fu Lee and Sanjoy Mahajan
School of Computer Science, Carnegie-Mellon University, Pittsburgh, PA 15213-3890, USA

ABSTRACT

In this paper we describe an Othello program, BILL, that has far surpassed the generation of Othello programs represented by IAGO. Its performance is due to a combination of factors. First, a wide variety of searching and timing techniques are used in order to increase its search depth. Furthermore, BILL efficiently uses a large amount of knowledge in its evaluation function. This efficiency is achieved through the use of pre-computed tables that can recognize hundreds of thousands of patterns in constant time. Finally, we applied Bayesian learning to combine features in BILL's evaluation function. This algorithm is automatic and optimal. It encapsulates inter-feature correlations, and directly estimates the probability of winning. These techniques are instrumental to BILL's playing strength, and we believe that they are generalizable to other domains.

1. Introduction

Computers have always excelled in Othello because average human players cannot envision the drastic board changes caused by moves. However, few programs played at an advanced level until Paul Rosenbloom created IAGO [8]. By quantifying Othello maxims into efficient algorithms, and by using AI search techniques, Rosenbloom made IAGO into a formidable World Class Othello player.

In this paper, we present another Othello program, BILL, that plays at an even more advanced level. Its strength can be directly traced to several factors. First, BILL uses a number of state-of-the-art searching and timing algorithms. These techniques enable deep searches to be conducted even under tournament time controls.

BILL efficiently recognizes and integrates an enormous amount of knowledge in its static evaluation function. In a knowledge-intensive program like IAGO,

*This research was partly sponsored by a National Science Foundation Graduate Fellowship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the National Science Foundation or the US Government.

Artificial Intelligence **43** (1990) 21–36
0004-3702/90/$3.50

the use of knowledge deprives the program of several plies of search. BILL avoids this problem by using a pre-compiled knowledge base that enables the recognition of complex patterns using only table lookups. Each table lookup evaluates a certain *feature* of the position.

Instead of the usual linear combination of these features, we introduce an algorithm based on Bayesian learning [3] that automatically and optimally combines features using a quadratic polynomial. This learning procedure provides a discriminant evaluation function that maximally separates winning positions from losing ones. In this method, an evaluation of a position directly measures *the probability of winning*.

An early version of BILL, BILL 1.0 captured first place in the 1985 Waterloo Computer Othello Tournament, and second place in the 1986 North American Computer Othello Championship. Moreover, BILL 1.0 consistently defeated IAGO, the program that inspired BILL. A full description of BILL 1.0 can be found in [5]. We have since incorporated more accurate evaluation tables and Bayesian learning into BILL 3.0. Results showed that Bayesian learning improved BILL's play by two plies of search. BILL 3.0 captured first place in the 1989 North American Computer Othello Championship. In a match against Brian Rose, the highest rated American Othello player, BILL won 56–8. Therefore, we believe that BILL 3.0 is one of the best, if not the best, Othello player in the world.

In this paper, we first briefly describe the game of Othello. In the subsequent sections, we discuss the three major contributions to BILL's strength: searching techniques, table-based evaluation, and Bayesian learning of evaluation function. Finally, we present the results and some concluding remarks.

2. The Game of Othello

For those readers not familiar with Othello, a brief description of the rules and square naming conventions is provided. The rules of Othello are quite simple. The game is played on an 8×8 board, initially set up as in Fig. 1(a). Each player, starting with Black, takes a turn by placing a piece of his color on the board, flipping to his own color any of the opponent's pieces that are *bracketed* along a line. There are two restrictions however: (1) one of the bracketing pieces must be the piece just placed on the board and (2) a move must flip at least one piece. Figure 1(b) contains a board with legal moves for Black to c6, d6, d2, e6, and g2. Figure 1(c) shows the board after Black moves to e6. When a player does not have any legal moves, he must pass his turn; when neither player has a move, the game is over and the player with the most discs is declared the winner. The game usually ends when all sixty-four squares are occupied, but this is not a requirement.

Standard Othello notation consists of naming a square by a letter–number combination. The letter (a–h) indicates the column, and the number (1–8)

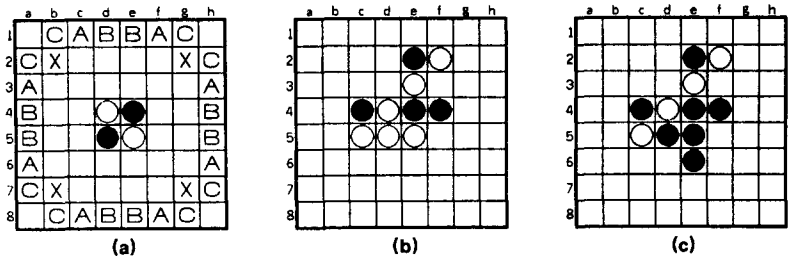


Fig. 1. (a) shows the initial Othello board setup and the standard names of the squares; (b) shows a sample board with legal moves (for Black) to c6, d6, d2, e6, and g2; (c) shows the board after Black plays to e6.

indicates the row. For example, the lower left corner is named a8. Some of the more important square types are also given designations. For example, the square on the edge that is next to a corner is called a C-square, while the square diagonally adjacent from a corner is termed an X-square. Figure 1(a) shows the standard names of the Othello squares.

3. Searching Algorithms

Before discussing BILL's evaluation function, a brief description of its various searching and timing algorithms is in order. The interested reader is referred to [5] for a more detailed discussion. Like most game-playing programs, BILL uses a full-width alpha-beta search to determine its move. Since the efficiency of an alpha-beta search depends heavily on the successor ordering, much effort was invested in this area. The three techniques used are described below.

3.1. Iterative deepening

In an iteratively deepened search, a full N-ply search is performed before attempting an $(N + 1)$-ply search (Slate and Atkin [11]). Although this may result in some repetition, the inexpensive, early searches place ordering information in the hash and killer tables (see Section 3.2) that greatly decrease the time in the later, more expensive searches. Overall, iterative deepening produces large savings in time, or, equivalently, substantial increases in search depth.

3.2. Hash and killer tables

Iterative deepening would be of no benefit without some method of saving the ordering information from previous searches. The hash and killer tables serve this purpose. Both tables increase the probability that the best descendants are examined first, thereby increasing the number of alpha-beta cutoffs.

The hash table contains a record of positions encountered, along with what BILL considered to be the best move. When a position is reached again, in a

later search or a later move, BILL expands the stored move first. Because the hash table contains exact information from previous searches, it is our most reliable source of ordering information.

While the hash table provides *one* good move, we introduce a novel killer table architecture that provides ordering information for *all* moves. An entry in our killer table is a linked list of all possible (not necessarily legal) responses to each move. Initially, these responses are ordered heuristically. As information is gained from searching, the ordering of the responses is dynamically updated by moving good moves toward the front of the linked list. While conventional killer tables suggest only the top few best moves (Slate and Atkin [11], Rosenbloom [8]), our table provides a sorted list of all moves. Moreover, by testing moves in the order suggested by the killer table for legality, *incremental move generation* is possible.

Together, the killer and hash tables speed up an eight-ply search by a factor of seven and reduce the effective branching factor from 4.26 to 3.60 [5]. This reduction in branching factor significantly improves BILL's play.

3.3. Zero-window search

The third technique used to increase the number of cutoffs is known as zero-window searching (Pearl [7]), a modified form of the iterative-deepening alpha-beta search. Standard alpha-beta searches initializes alpha and beta at $+\infty$ and $-\infty$. The zero-window search searches the first descendant with a narrow alpha-beta window to get an exact value, and searches the remaining moves with a *zero-window* around this value. In other words, we are only interested in whether a move is better than the current best move, and not in the exact difference. If no move was found to be better, then the current best move is the best move. If one move was found to be better, then that is the best move. If several moves were found to be better, then it is necessary to re-search all of them. Clearly, with poor move ordering, many moves have to be re-searched. However, because our hash and killer tables provide excellent ordering information, zero-window search requires only 63% of the time of a normal alpha-beta search [5].

3.4. Endgame search

Since there are at most 60 non-pass moves in an Othello game, it is possible to determine the game-theoretic value of a position sufficiently close to the end of the game. An endgame search expands each position into terminal positions. We use two possible evaluation functions:

(1) a simple win/loss/draw function; the search which uses this function is called an *outcome* search;
(2) a disc count differential function; the search which uses this function is called an *exact* search.

The outcome search can usually be done with 15 to 18 empty squares, while the exact search can usually be done with 13 to 15 empty squares. The exact search ensures that BILL achieves the optimal final disc count, while the outcome search ensures that BILL does not lose a won game by making a plausible, but inferior move. Although our use of a two-phase endgame search was taken from IAGO, our addition of zero-windowing and ordering mechanisms improves the endgame search significantly.

3.4.1. *Search performance*

Because of the techniques discussed above, and because our evaluation function is extremely efficient,[1] BILL can attain an average search depth of 8.5 plies under tournament time controls. This figure is a weighted average between an average depth of 8 plies for normal searches, and 16 plies for the first endgame search.

3.5. Timing

Because BILL was designed to play in a competitive environment, we need a time management algorithm that maximally utilizes, yet never exceeds, the allocated time. The algorithm BILL uses is heavily influenced by the timing algorithm of HITECH (Berliner [2]), the winner of the 1985 ACM Computer Chess Championship. International Othello rules allow 30 minutes for a player to make all of his moves. The time allocated for a single move is determined by how much time is left on BILL's clock and by the move number. In general, BILL allocates more time for the later moves. Whenever BILL completes one iteration of search, it checks the following conditions:

- If less than 40% of the time allocation is used, begin the next iteration.
- If the time elapsed is between 40% and 100% of the time allocation, continue only if the last two levels of search disagree as to the best move. This condition ensures that BILL does not waste time searching "obvious" moves.
- If more than the time allocation is used up, stop.

As a final precaution, if BILL has exceeded its time allocation by more than 8%, an alarm terminates the search, and the best move from the previous iteration is selected. This timing algorithm has proven very effective in managing BILL's time.

4. Table-Based Evaluation Features

BILL's knowledge-intensive evaluation function is crucial to its success. This evaluation is composed of four features: edge, current mobility, potential

[1] BILL can search 1100 nodes per second on a VAX 11/785.

mobility, and sequence penalty. Since run time computation of these features are extremely expensive, we have developed a number of algorithms that pre-compute these features and store them as tables. In an actual game, each feature can be computed as a sequence of table lookups. In this section, we will discuss how complex feature evaluation can be encoded as tables.

4.1. Edge stability

The edges are the most important set of squares on the Othello board, because stable edge discs cannot be flipped and can be used as anchors to gain stable internal discs. Control of stable edge discs virtually guarantees victory. For this reason, BILL must have a thorough understanding of edge play. However, edge play is fraught with traps that may take many moves to recognize [5, 8]. Such an analysis would be inordinately expensive at run time.

Instead, by using a pre-compiled edge table, the evaluation of any edge position can be done almost instantly. Since there are 8 squares on each edge, a possible table could contain 3^8 (6561) entries. However, from our experiments we found that the state of the X-squares has a prominent effect on the value of an edge. Thus, BILL's edge table includes the X-squares, and so contains 3^{10} (59049) entries. We now describe how the table is generated.

The generation is done in two phases. First, each edge position is assigned a *static value* which is the value of the position, assuming it doesn't change. Then, a variant of the minimax search algorithm is applied to each possible edge position. The outline of the search is as follows:

(1) All completely filled positions have accurate static values, so they are marked as having *converged*. All other positions are marked as *not converged*.
(2) Fill in the *not converged* positions for each color to move by recursively computing the value for the positions after:
    (a) *legal moves*: moves that flip edge discs; *pass* is considered a legal edge move.
    (b) *possible moves*: moves that flip no edge discs.
    After all recursive calls return, the values of the children nodes are negated and combined into the value for the parent position (the combining algorithm will be described later). The parent position is then marked as having *converged*.
(3) The above procedure is called with the empty edge. This call will recursively fill in values for all other positions.

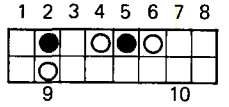
Clearly, the standard minimax backup method does not apply because of the possibility of *possible* moves which do not flip any edge discs, yet may be legal. In order to properly consider the possible moves, we associate each such move with a probability. These probabilities were empirically computed from a very

large number of sample games. To combine the probabilities and scores of all the children of an edge position, we introduce the following algorithm:

(1) Find the best *legal* move, L, with probability 1.0, and score $S(L)$. All other *legal* moves can be ignored because L is *always* a better move.
(2) Initialize the value of the edge to 0, and the remaining-probability, R, to 1.0.
(3) Sort all *possible* moves, and loop through them from best to worst: For each *possible* move M_i, with probability $P(M_i)$, and score $S(M_i)$:
    (a) If $S(M_i)$ is worse than $S(L)$, quit the loop.
    (b) Otherwise, increment the value of the edge by $P(M_i) \times R \times S(M_i)$.
    (c) Decrement R by $P(M_i)$.
(4) Increment the value of the edge by $R \times S(L)$.

Figure 2(a) shows an example where Black has three *legal* moves (to squares 3, 7, and NOMOVE), and three *possible* moves (to squares 1, 8, and 10). Figure 2(b) shows the scores and probabilities of each move as backed up by searching. The value of this position with Black to move is computed as follows: 92% of the time, Black will be able to move to square 1 (the best *possible* move), obtaining a partial score of $450 \times 0.92 = 414$. In the event that move 1 is illegal (8%), Black would move to square 8 (the second best *possible* move) 2% of the time, for a partial score of $400 \times 0.02 \times 0.08 = 0.64$. The final *possible* move (square 10) is inferior to the best *legal* move (NOMOVE), so it is not considered because *even if the move to square 10 were legal, we are better off making NOMOVE*. So NOMOVE is the course of action in the event that neither of the best two *possible* moves is legal (8% × 98%, or 7.85%), for a partial score of $200 \times 0.0784 = 15.67$. Therefore, the evaluation of the position shown is $414 + 0.63 + 15.67 = 430.30$.

a



b

	Score	Prob
Legals moves		
Square 3	-600	1.00
7	-500	1.00
NO MOVE	200	1.00
Possible moves		
Square 1	450	0.92
8	400	0.02
10	-100	0.80

Fig. 2. A node in the edge table generation. Numbered squares are part of the edge table; unnumbered squares need not be empty. The table shows the values (range is [−1000, 1000]) returned by recursive probabilistic minimax for the *legal* and *possible* moves.

The resulting edge table gives BILL an extensive understanding of edge play, with virtually no run time cost. Although the underlying principles of our edge table were inspired by IAGO, our edge table is more robust because it includes the two X-squares. Moreover, our novel probabilistic minimax search and score combination are both mathematically and intuitively plausible.

4.2. Mobility

Expert Othello players usually do not yield stable edge discs without being forced to do so. Similarly, the edge table serves to prevent BILL from unnecessarily yielding stable edge discs, and to capitalize on *forced* edge-yielding moves by the opponent. To force such a move, BILL's play relies on *mobility optimization*. The object of this strategy is to reduce the opponent's options until he has only moves that yield stable edge positions. Figure 3 shows BILL utilizing this strategy in its tournament games. In both positions, BILL's overwhelming mobility advantage results in subsequent control of most of the edges.

To measure mobility, BILL uses three features:

(1) current mobility,
(2) potential mobility,
(3) sequence penalty.

These features are described in the subsequent sections.

4.2.1. *Current mobility*

Current mobility is a function of the available moves. However, not all moves were created equal. In general, moves that surrender corners, such as X-square moves, are worthless. Conversely, moves that capture corners are usually very valuable. Besides the effect on the corners, each move has an effect on future mobility that must also be considered. Flipping many discs next to empty squares (*frontier discs*) will give the opponent many possible moves. In Fig. 4,

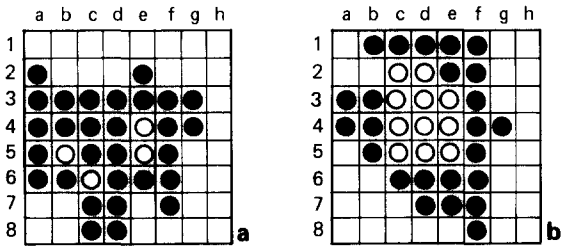


Fig. 3. Examples of the danger caused by lack of mobility. In (a), from OTHELLO^0 (B-10) versus BILL (W-54), Black has one legal move, while White (to move) has 12; in (b), from GRAY BLITZ (B-4) versus BILL (W-60), Black has 2 moves, while White (to move) has 15. In both positions, BILL's opponent is forced into yielding the edges.

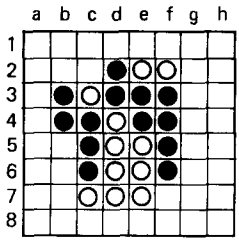


Fig. 4. If White plays to f7, he flips f2–f5. This gives Black four additional moves.

by moving to f7, white flips four frontier discs, thereby giving black *four* additional moves. On the other hand, if black were to move to f7, he would flip no frontier discs, and would give white only *one* additional move.

Based on the above examples, we see that the following considerations affect value of a move:

- Does it capture or surrender a corner?
- How many discs does it flip, and in how many directions?
- Are those discs internal or on the frontier?

Because an evaluation that processes the entire board searching for these features would be prohibitively expensive, we use a highly accurate approximation algorithm that exploits the fact that *all of these factors can be measured or approximated by examining relationships between sequences of adjacent squares.*

To evaluate current mobility for a position, all the legal moves for the side to move are first counted, and the *unadjusted current mobility* is computed from these moves by assigning a bonus for each move according to its location. The *unadjusted current mobility* represents the highest value that the available moves could attain. Then, a move penalty is calculated to penalize moves that adversely affect the three abovementioned considerations.

Both move finding and penalty calculation can be extremely expensive. However, through clever use of tables, we are able to compute both efficiently. To find moves using tables, we represent a board as 38 numbers, where each number is an index into a table that represents a horizontal, vertical, or diagonal line[2] on the board. For example, a vertical line with 8 squares can take on one of 3^8 values. It is easy to efficiently find all the legal moves given these indices.

Move penalty is also calculated by a series of table lookups. Each table lookup attempts to adjust the mobility evaluation for a move on a horizontal, vertical, or diagonal line on the board. A table is constructed for each line, and contains a penalty for every possible move on the line given every possible

[2]There are 8 horizontal lines, 8 vertical lines, 2 main diagonals, and 20 minor diagonals of length 3 or longer, for a total of 38 lines. Although there are 38 distinct lines on the board, only 9 tables are needed due to symmetry and compression.

configuration of that line. The penalties are computed by considering the factors listed above. For example, if a move flips many discs along one line, it will be penalized very heavily by one table. If it flips a disc in 3 directions, it will be penalized moderately by each of the 3 tables, again resulting in a severe penalty. All the 38 penalties are summed, scaled, and then subtracted from the unadjusted current mobility for the side to move.

4.2.2. *Potential mobility*

Because mobility is so important in Othello, it is important to play into positions that are likely to yield moves in the future. Potential mobility captures this likelihood. This term can be measured by examining the adjacency between empty squares and pieces, or by counting frontier discs. We generate tables that evaluate each of the 38 lines. In these tables, a bonus is given for each of the opponent's internal discs next to an empty square because this empty square may become a legal move in the future. The size of the bonus naturally depends on the desirability of the potential move. For example, a small bonus is given for a potential X-square move, whereas a large one is given for a potential corner move.

4.2.3. *Sequence penalty*

At almost all times, it is prudent to avoid long sequences of one's own discs. These sequences often hinder one's mobility, especially if they are on the frontier. BILL recognizes and penalizes strings of discs in the sequence penalty component of its evaluation function. These penalties are pre-computed for each configuration of each of the 38 lines by adding the penalties for each sequence of discs (including a single disc) according to its location and length. For example, a long string of discs in the b-column table is given a large penalty, because they are most likely frontier discs.

4.2.4. *Parallel mobility feature computation*

Since move penalty, potential mobility bonus, and sequence penalty all require a lookup for each of the 38 lines on the board, it is possible to compute them in parallel by concatenating the tables. In the current implementation, each table entry is a 32-bit word: 16 bits are used for move penalty, and 8 bits are used for potential mobility bonus and for sequence penalty. For each evaluation, by adding together the 32-bit words, one from each of the 38 lines, BILL simultaneously obtains the move penalty, potential mobility, and sequence penalty for the side to move. This storage technique allows BILL to compute most of its evaluation in parallel.

5. Learning

Once the four features have been computed, they must be combined to yield the final static evaluation. The standard linear feature combination has two

serious shortcomings. First, a linear polynomial is ignorant of feature interaction. Samuel's work on checkers [10] demonstrated that some nonlinearity provides a significant improvement over a linear polynomial. Second, the implementator usually must guess the optimal polynomial coefficients, which is a difficult task even for an expert. It is further complicated by inter-feature correlations and nonlinearity. Clearly, some automatic method of determining the coefficients is desirable. In this section, we present an algorithm based on Bayesian learning that optimally combines features using a quadratic polynomial.

5.1. Bayesian learning

Bayesian learning of discriminant functions is a standard pattern recognition technique. Typically, it is used to determine, say, if a particular image matches the letter "A" or "B," based on features extracted from the *image*. In Othello, we use Bayesian learning to decide whether a particular position is a *win* or *loss*, based on the features extracted from the *board*. The algorithm consists of four steps:

(1) Generate a large database of training positions.
(2) Label these positions as winning or losing.
(3) Compute a *discriminant function* from the labeled data. This function attempts to recognize feature patterns that represent winning or losing positions. Given the feature vector for a position, it assigns a probability that the position is a winning one.
(4) Build different classifiers for different stages of the game.

We first discuss how the training data was generated and labeled. Then, we describe the discriminant function. Finally, we compare our learning algorithm against other algorithms.

5.1.1. *The training stage*

The training data was taken from actual games. Each position of the losing player is marked as a losing position, and each position of the winning player is marked as a winning one. Draws are discarded. While this method is simple and consistent, it has a serious problem: a winning position could be blundered away by a poor move, and would then be mislabeled as a losing position. We dealt with this problem by only using expert games. Such an expert was readily available in an older version of BILL. BILL 2.0, which uses a manually tuned linear evaluation function, already played at the World Championship level. We simply used it to generate the training positions. BILL 2.0 played itself under the following conditions: the first 20 half-moves are made randomly, and the next 25 half-moves are played in 20 minutes. When there are 15 half-moves remaining, an outcome endgame search (see Section 3.4) is performed to

determine the winner, assuming perfect play on both sides. The game is terminated, and the database of winning and losing positions is updated.

It is well known that different strategies are needed for different stages of the game. Therefore, we generated a discriminant function for each stage. We define a stage as the number of discs on the board. Since there are almost always 60 moves per game in Othello, disc count provides a reliable estimate of the stage of the game. The function for N discs was generated from training positions with $N-2$, $N-1$, N, $N+1$, and $N+2$ discs. By coalescing adjacent data, the discriminant function is slow-varying, similar to the *application coefficients* proposed by Berliner [1].

5.1.2. *Generating the discriminant function*

Once the training positions have been labeled, the feature vector is extracted from each position. Then, the mean feature vector and the covariance matrix between the features are computed for our two classes: Win and Loss. Table 1 shows the mean vector and the correlation matrix for classes of *winning positions* and *losing positions* at $N = 40$.

For normally distributed features, the optimal quadratic discriminant functions for the two classes are [3]:

$$g_{\text{Win}}(\boldsymbol{x}) = -\tfrac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_{\text{Win}})^{\text{T}}\Sigma_{\text{Win}}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_{\text{Win}}) - \tfrac{1}{2}N\log 2\pi - \tfrac{1}{2}\log|\Sigma_{\text{Win}}| + \log P(\text{Win}), \tag{1}$$

$$g_{\text{Loss}}(\boldsymbol{x}) = -\tfrac{1}{2}(\boldsymbol{x}-\boldsymbol{\mu}_{\text{Loss}})^{\text{T}}\Sigma_{\text{Loss}}^{-1}(\boldsymbol{x}-\boldsymbol{\mu}_{\text{Loss}}) - \tfrac{1}{2}N\log 2\pi - \tfrac{1}{2}\log|\Sigma_{\text{Loss}}| + \log P(\text{Loss}), \tag{2}$$

where $\boldsymbol{x}$ is the N-element feature vector, $|\Sigma_c|$ is the determinant of the covariance matrix for class c, Σ_c^{-1} is the inverse of the covariance matrix, and $\boldsymbol{\mu}_c$ is the mean vector for class c.

Table 1
The mean vector and correlation matrix for classes Win and Loss at $N = 40$; Mob: mobility; Pot: potential mobility; Wtd: weighted squares; Edge: edge position

	Win				Loss			
	Mob	Pot	Wtd	Edge	Mob	Pot	Wtd	Edge
Mean	533	266	242	599	−419	−200	−112	−334
Correlation matrix	1.00	0.73	0.65	0.26	1.00	0.74	0.65	0.25
	0.73	1.00	0.64	0.11	0.74	1.00	0.64	0.12
	0.65	0.64	1.00	0.20	0.65	0.64	1.00	0.21
	0.26	0.11	0.20	1.00	0.25	0.12	0.21	1.00

Since the evaluation function should measure some monotonic function of the likelihood that the board position belongs to the class Win, we use

$$g(x) = \frac{P_{\text{Win}}}{P_{\text{Loss}}} = g_{\text{Win}} - g_{\text{Loss}} \cdot \tag{3}$$

Assuming the *a priori* probabilities are equal, our final evaluation function reduces to:

$$\begin{aligned} g(x) = {} & (x - \mu_{\text{Win}})^{\text{T}} \Sigma_{\text{Win}}^{-1} (x - \mu_{\text{Win}}) \\ & - (x - \mu_{\text{Loss}})^{\text{T}} \Sigma_{\text{Loss}}^{-1} (x - \mu_{\text{Loss}}) + \log|\Sigma_{\text{Win}}| - \log|\Sigma_{\text{Loss}}| \,. \end{aligned} \tag{4}$$

Equation (4) is used in BILL 3.0; however, its values are difficult for humans to interpret. Therefore, when the evaluation is reported, BILL 3.0 first converts it to the probability of winning. The probability of winning can be computed from $g(x)$ as shown in (5):

$$\frac{P(\text{Win}|x)}{P(\text{Win}|x) + P(\text{Loss}|x)} = \frac{e^{g(x)}}{e^{g(x)} + 1} \,. \tag{5}$$

5.2. Discussion

This new evaluation learning technique has many advantages compared to other efforts in evaluation function learning [4, 9, 10]. First, Bayesian learning enables nonlinear interaction among features by considering the covariances between every pair of features. This is important because often expert play depends on understanding these interactions.

Learning schemes such as Samuel's signature table algorithm [10] must learn a large number of parameters. In order to control the number of parameter, quantization is often necessary. Unfortunately, this quantization results in the *blemish* effect (Berliner [1]):

> a very small change in the value of some feature could produce a substantial change in the value of the function. When the program has the ability to manipulate such a feature, it will frequently do so to its own detriment.

By contrast, Bayesian learning does not need to quantize the feature values, since its nonlinearity is derived from covariances.

Another advantage of the Bayesian learning approach is its completely automatic nature. All that is needed to apply Bayesian learning to another domain is a set of labeled training data and a feature extractor.

While other learning programs learn to differentiate good features from poor ones (Samuel [9]) or to imitate expert's moves [10], Bayesian learning learns

the optimal concept, namely, "moves that lead to a win." With Bayesian learning, it is theoretically possible for the evaluation (without search) to be superior to the expert who played the training games.

Some results on Bayesian learning will be presented in Section 6.2. The interested reader is directed to [6] for more complete description, analysis and results.

6. Results

6.1. BILL 1.0: Tournament results

BILL 1.0 is a version of BILL that used a simplified version of the evaluation function described in Section 4, and combined the features linearly, rather than using Bayesian learning as described in Section 5. We tuned parameters of BILL 1.0 by playing it against IAGO [5].

BILL 1.0 was entered in the Waterloo Computer Othello Tournament on November 9, 1985. This tournament consisted of 10 programs, most of which were from Canada. BILL won all four games with very large margins, and captured first place.

It was later entered in the North American Computer Othello Championship on February 9, 1986. 11 programs were entered in this tournament. BILL 1.0 won 7 games out of 8, placing second after ALDARON, which accumulated 7 wins and one draw. BILL 1.0's only loss was to ALDARON. This loss was due to the color BILL 1.0 drew in that game.[3] Furthermore, BILL 1.0 also defeated XOANNON, the only program that did not lose to ALDARON during that tournament.

6.2. BILL 3.0: Improvement from Bayesian learning

We modified the evaluation of BILL to that described in Section 4, resulting in BILL 2.0. This version was slightly better than BILL 1.0. Subsequently, we added Bayesian learning, resulting in BILL 3.0.

We evaluated the utility of Bayesian learning by playing BILL 3.0 against BILL 2.0. The games were started from 100 nearly even positions with 20 discs on the board. BILL 3.0 played BILL 2.0 twice from each position, once as black and once as white, with the same amount of time. BILL 3.0 accumulated a record of 139–55–6, and an average score was 37–27 [6]. To determine exactly how significant these figures are, versions of BILL 2.0 that searched to different depths played each other starting from the same 100 positions. The results are summarized in Table 2. Because BILL can search six to eight plies under tournament conditions, we see that a 37–27 disc count is equivalent to

[3] BILL 1.0 preferred to play black, and unfortunately drew white in this game. In an unofficial rematch with the colors reversed, BILL 1.0 defeated ALDARON.

Table 2
Results between two versions of BILL

Players	Win	Tie	Loss	W/L	Average score
7-ply vs. 6-ply	121	7	72	1.68	34.54–29.39
8-ply vs. 7-ply	115	6	79	1.46	35.04–28.89
7-ply vs. 5-ply	141	10	49	2.88	37.03–26.94
8-ply vs. 6-ply	130	13	57	2.28	36.38–27.59
Learn vs. Linear	139	6	55	2.53	36.95–27.03

approximately two plies of searching. With the *effective* branching factor in Othello ranging from about 3.4 to 3.7 two plies translates to a factor of 13 in speed.

6.3. Evaluation of BILL 3.0's strength

From the above record of BILL 1.0's performance, it is clear that BILL 1.0 is already one of the best Othello computer programs. With the addition of an improved evaluation function and Bayesian learning, BILL 3.0 is much stronger. In February 1989, BILL 3.0 was entered in the 1989 North American Computer Othello Championship. It finished first place out of thirteen programs with a record of 7–1. As further evidence, in a match against Brian Rose, the highest rated American Othello player, BILL won with a score of 56–8. In games against IAGO, BILL wins 100% of the games with only 20% as much time. These results indicate that BILL is one of the best, if not the best, Othello player in the world.

7. Conclusion

In this paper, we presented an Othello program, BILL, that plays at World Championship level. Its success can be attributed to three factors: the use of state-of-the-art searching and timing techniques, the use of tables to efficiently encode a large amount of Othello knowledge, and the use of a new learning algorithm to automatically combine evaluation features.

We incorporated many known searching techniques, such as iterative deepening, hash table, killer table, and the zero-window search, and showed how they complement each other. One novel contribution is the use of a linked-list killer table which orders *all* moves, as well as facilitates *incremental move generation*.

We showed that the use of tables could significantly increase the speed of evaluation. In our case, *all* of the knowledge is encoded in a set of tables, and an evaluation consists of a sequence of table lookups. Given the large amount of main memory available on most modern computers, and the time-consuming nature of tree searching, we believe our time-for-space tradeoff is most appropriate.

Due to the efficiency of evaluation and the use of highly optimized state-of-the-art search techniques, BILL is able to search to an average depth of over eight plies under tournament conditions. The large amount of knowledge in its evaluation function, when coupled with this deep a search, resulted in a program that performs at the World Championship level.

Finally, we introduced the Bayesian learning approach for evaluation function learning. This learning algorithm eliminates many of the problems associated with previous algorithms, and has several desirable properties:

(1) completely automatic learning from the training data;
(2) optimal quadratic combination, assuming multivariate normal distribution;
(3) understanding of feature covariances;
(4) evaluation directly estimating the *probability of winning*.

We showed that Bayesian learning improved BILL's play dramatically.

We believe that our rigorous and scientific approach to game playing and learning is responsible for BILL's success. We hope that some of the techniques and approaches presented here will prove useful in other game-playing programs, as well as other domains.

ACKNOWLEDGEMENT

The authors wish to thank Professor Hans Berliner for his encouragement, support, suggestions, and advice. We are also grateful to Gordon Goetsch, who suggested finding moves with tables, and Paul Rosenbloom, who created IAGO, a patient teacher and worthy opponent for BILL.

REFERENCES

1. H. Berliner, On the construction of evaluation functions for large domains, in: *Proceedings IJCAI-79*, Tokyo (1979) 53–55.
2. H. Berliner, Personal communications (1985).
3. R. Duda and P. Hart, *Pattern Classification and Scene Analysis* (Wiley, New York, 1973).
4. A.K. Griffith, A comparison and evaluation of three machine learning procedures as applied to the game of checkers. *Artificial Intelligence* **5** (1974) 137–148.
5. K.-F. Lee, and S. Mahajan, BILL: A table-based knowledge-intensive Othello program, Tech. Rept., Carnegie-Mellon University, Pittsburgh, PA (1986).
6. K.-F. Lee and S. Mahajan, A pattern classification approach to evaluation function learning, *Artificial Intelligence* **36** (1988) 1–25.
7. J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving* (Addison-Wesley, Reading, MA, 1984).
8. P.S. Rosenbloom, A World-Championship-level Othello program, *Artificial Intelligence* **19** (1982) 279–320.
9. A.L. Samuel, Some studies in machine learning using the game of checkers, *IBM J.* **3** (1959) 210–229.
10. A.L. Samuel, Some studies in machine learning using the game of checkers, II, *IBM J.* **11** (1967) 601–617.
11. D.J. Slate and L.R. Atkin, CHESS 4.5: The Northwestern University chess program, in: P. Frey, ed., *Chess Skills in Man and Machine* (Springer, New York, 1977) 82–118.