Experiments with Multi–ProbCut and a New High–Quality Evaluation Function for Othello

Michael Buro

NEC Research Institute
4 Independence Way
Princeton NJ 08540, USA

email: mic@research.nj.nec.com

Abstract

This paper presents ideas concerning game–tree evaluation that recently improved the author's strong Othello program LOGISTELLO considerably. Two main parts of this program have been refined. In the first part of this paper, a new evaluation function for Othello is described. While it is table–based like its predecessors, a novel procedure for estimating table entries now allows modeling pattern interactions which leads to a much higher evaluation quality. The second part introduces MULTI–PROBCUT, which generalizes the selective search procedure PROBCUT. It allows forward cuts at various heights after shallow searches of increasing depths. Combined with the new evaluation function the resulting playing strength increase is equivalent to a speed-up factor of more than ten.

Keywords: selective game–tree search, table based evaluation function, linear regression, Othello

1 Introduction

In today's top game playing programs a variety of techniques are used to optimize the move quality subject to the limited time available in tournament games:

- a high raw search speed is achieved by means of assembler routines or using very fast parallel or special purpose hardware that allows deep game–tree searches and thereby enables playing a strong game even if only poor evaluation functions are used.
- selective searches are performed to follow interesting variations more deeply or to cut off probably irrelevant lines of play early, without missing many decisive variations.
- smart evaluation functions are used which are often automatically tuned.
- large opening books and perfect endgame databases are utilized for improving the move quality in the opening and endgame phase.

The combination of all these techniques is ideal for top–level play. Unfortunately, there are incompatibilities as well as tradeoffs between these techniques. For instance, affordable hardware realizations require a simple structure for both the evaluation function and the selective search mechanism. These restrictions may cause a lower playing strength than expected compared to that of a normal workstation implementation of a smarter search algorithm coupled with a better evaluation function. On the other hand, weaker but faster evaluation functions allow deeper searches which may lead to a better overall performance than the use of smart but slow functions in conjunction with shallower searches. Despite these design problems existing implementations can often be improved by working on *each* of the mentioned topics separately, aiming for the right balance. This is very important, since neglecting one issue can reduce the overall performance considerably.

LOGISTELLO has been one of the top Othello programs ever since its tournament debut in October 1993. It is still a sequential C program running on ordinary hardware. From the beginning the main focus of development has been on deep searches and reasonably good evaluation functions. In what follows the latest program improvements are described. First, a new table estimation technique is presented which significantly improved the evaluation function quality at no additional run time cost. Then the selective search procedure PROBCUT [BURO 1994] is generalized enabling the program to cut off even more variations in advance that probably have no impact on the move decision.

2 LOGISTELLO's previous evaluation function

The details of LOGISTELLO's previous evaluation function recently have been described in [BURO 1997]. In what follows a brief outline of the techniques used is given allowing the comparison of the major aspects with the new method.

The classical approach for constructing evaluation functions for game–playing programs is to combine win correlated evaluation features of the position linearly:

$$f(p) = \sum_{i=1}^{n} w_i f_i(p).$$

This type of evaluation function is chosen very often since the combination overhead is relatively small compared to the time for computing the features and there are efficient methods available for determining the feature weights. When the relative importance of the features or even the feature set varies depending upon the game stage this simple model can be generalized to:

$$f(p) = \sum_{i=1}^{n_s} w_{s,i}\ f_{s,i}(p), \text{ where } s = \text{stage}(p).$$

LOGISTELLO's previous evaluation features fall into two classes, namely mobility measures and patterns. These approximate important concepts in Othello, like striving for stable discs, maximizing the number of moves, and parity. Rosenbloom (1982) and Lee & Mahajan (1990) introduced a table–based evaluation scheme, in which values of all edge configurations were precomputed by (probabilistic) minimax algorithms and stored in a table for a quick evaluation of the edge structure. Furthermore, several local mobility features defined on the lines of the board (horizontals, verticals, and diagonals) were evaluated by fast table accesses. The pattern approach introduced in [Buro 1994,1997] generalized this technique by permitting the automatic evaluation of pattern configurations of any shape. The formerly used evaluation algorithms were tailored for the edge situation and could not be adapted. The current pattern set is shown in Figure 1. Using a large set of about three million example positions, which were labeled with the particular game outcomes, the value for each configuration c was estimated independently by the following relation:

$$V(c) = \frac{Y(c) + 0.5}{N(c) + 1.0},$$

where $N(c)$ = number of positions containing c, and $Y(c)$ = number of positions containing c which are won for Black + $0.5 \cdot$ number of drawn positions containing c. The additive constants 0.5 and 1.0 assure a neutral evaluation (0.5) of pattern instances that do not occur in the training set. Configuration values lie in $(0, 1)$ and model the winning probability for Black conditioned upon the occurrence of configurations on the board.

The second feature subset dealt with mobility and potential mobility. Here, the simplest approach is to count legal or potential moves which — unfortunately — is relatively time consuming compared to the time needed for all other features and making/undoing moves during the game–tree search. In order to speed up the computation the globally defined mobility measures were approximated by the sum of mobilities local to the lines of the board, i.e. the horizontals, verticals, and diagonals. It turned out that in the opening the quality of these approximations is only slightly worse than that of the original mobility measures and later it is almost equal. Thus, the slow mobility features could be safely replaced by their much faster approximations.

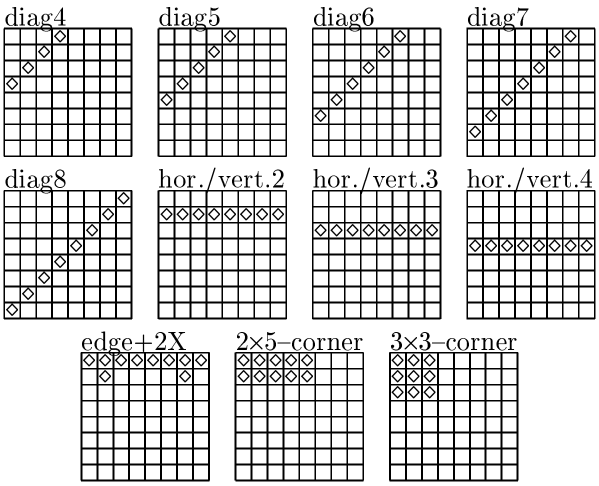

Figure 1: The current pattern set.

Pattern tables were estimated for twelve game stages (disc count 12..15,16..19,...,56..59). Finally, feature weights for each disc count in [12..59] were determined by logistic regression. This generalized linear model describes the winning probability dependent upon the features as follows:

$$\text{Prob}(\text{Win}(p)) = 1/(1 + exp(-\sum_{i=1}^{n_s} w_{s,i} f_{s,i}(p))).$$

The weight vectors that maximize the likelihood of the observed labeled feature vectors can be found by iteratively solving systems of nonlinear equations [Buro 1995b].

3 A generalization of the classical linear evaluation model

Two observations led to an improved evaluation scheme. First, the just described table estimation technique totally ignores the correlation among configuration values because each table entry is determined separately and the tables, as a whole, are weighted afterwards. Secondly, the question arises whether there are better local mobility features, which — for example — assign weights to move squares, or why these mobility features are necessary at all. After all, these approximation features are only defined on the lines of the board for which values are already estimated and stored in tables.

Both problems can be solved simultaneously by generalizing the table–based evaluation approach: Suppose that an evaluation function is to be constructed by combining n discrete features $f_1..f_n$. As described the classical approach assigns weights to each feature and the products are added to form the evaluation function

$$f(p) = \sum_{i=1}^{n} w_i f_i(p). \qquad (1)$$

Of course, when dealing with heuristic evaluations, this simple linear relation is only an approximation in most cases. But fortunately, the expressiveness of this model can be increased easily while still permitting efficient parameter estimations. Let $\{v_{i,1}, ..., v_{i,n_i}\}$ be the image of f_i and let $f_i^{(j)}$, $j \in \{1..n_i\}$ be the indicator variables for feature f_i, i.e.

$$f_i^{(j)}(p) = \begin{cases} 1, & \text{if } f_i(p) = v_{i,j} \\ 0, & \text{otherwise} \end{cases}.$$

Then a natural generalization of (1) is given by

$$f(p) = \sum_{i=1}^{m} \sum_{j=1}^{n_i} w_i^{(j)} f_i^{(j)}(p) + \sum_{i=m+1}^{n} w_i f_i(p) \qquad (2)$$

For the first m features weights are now separately assigned to each feature value rather than to the entire feature as before. It is easy to see that (1) is a special case of (2) by setting $w_i^{(j)} = w_i v_{i,j}$ for all i and j. The described pattern features fit nicely into this model because each pattern configuration represents an indicator variable. Since the weights $w_i^{(j)}$ and w_i can still be estimated by applying the same techniques as for (1) this generalization opens up an alternative way for determining evaluations of pattern configurations.

The goal in Othello is to maximize one's own disc count at the end of the game. Thus, given a position p one natural evaluation model is to approximate the final disc difference $r(p)$ in view of the side to move after optimal play by both sides starting with p. In this model parameters can be estimated by means of linear regression using a large number of examples given in form of labeled feature vectors:

$$(f_1(p_k), ..., f_n(p_k),\ r(p_k)).$$

For the existence of a unique solution the features must be linearly independent. When using indicator variables for more than one feature — say f_i and f_k — this condition is violated because $1 = \sum_{j=1}^{n_i} f_i^{(j)} = \sum_{j=1}^{n_k} f_k^{(j)}$. By introducing the constant feature and omitting one indicator variable of each of the first m features in (2) linear independence can be achieved

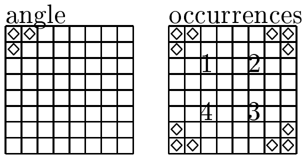

Figure 2: angle pattern that is used in the example and its four occurrences on the board

$$f(p) = w_0 + \sum_{i=1}^{m} \sum_{j=1}^{n_i - 1} w_i^{(j)} f_i^{(j)}(p) + \sum_{i=m+1}^{n} w_i f_i(p). \qquad (3)$$

In order to illustrate the new approach, suppose that each example position is normalized such that it is Black's turn to move. A simple evaluation function is to be constructed by combining the following six features:

$$\begin{aligned} f_1 &= \#\text{empty squares modulo } 2 \\ f_{2.1..2.4} &= \text{pattern features shown in Figure 2} \\ f_3 &= \#\text{moves for Black} - \#\text{moves for White} \end{aligned}$$

Feature f_1 is a crude approximation of the parity concept in Othello. It tries to model the advantage of the player to move when there is an odd number of moves left to make in the game.[1] For simplicity a pattern consisting only of three squares has been chosen. It includes the very important corner square, occurs four times on the board, and maps each of the $3^3 = 27$ corner–angle configurations to real values. Since the angle pattern is symmetrical with respect to one main diagonal only 18 of the 27 configurations have to be distinguished. Feature f_3 measures the mobility advantage of Black.

Modeling f_1 and $f_{2.i}$ by means of indicator variables and using f_3 as is, the evaluation function according to (3) has the following form:

$$f(p) = w_0 + w_1^{(1)} f_1^{(1)}(p) + \sum_{i=1}^{4} \sum_{j=1}^{17} w_{2.i}^{(j)} f_{2.i}^{(j)}(p) + w_3 f_3(p)$$

This representation can be simplified to

$$f(p) = w_0 + w_1^{(1)} f_1^{(1)}(p) + \sum_{j=1}^{17} w_2^{(j)} f_{2.*}^{(j)}(p) + w_3 f_3(p)$$

by setting $f_{2.*}^{(j)} = f_{2.1}^{(j)} + f_{2.2}^{(j)} + f_{2.3}^{(j)} + f_{2.4}^{(j)}$ and $w_{2.i}^{(j)} = w_2^{(j)}$, where $f_{2.*}^{(j)}(p) \in \{0..4\}$ counts the number of occurrences of pattern configuration j in position p and $w_2^{(j)}$ is its evaluation. After generating a training set of labeled feature vectors the weights $w_0, w_1^{(1)}, w_2^{(j)}$, and w_3 can be estimated by means of linear regression.

[1] This player usually also makes the last move in the game giving him at least two discs.

4 Dealing with large tables

The most important patterns are those which can quickly approximate the major Othello concepts: corner–possession and –threats, mobility, and parity. With regard to this, patterns of length three — like the corner–angle used in the example — are not expressive enough. Figure 1 gives an overview of the patterns that are currently used in LOGISTELLO's evaluation function. These patterns have been chosen manually taking into account their evaluation–quality and –speed. The first eight patterns deal with features that can be approximated locally on the lines of the board such as mobility. The three remaining patterns cover edge tactics as well as access and parity issues of small corner regions.

For this set of patterns the total number of variables in the proposed linear model is about $n = 110{,}000$ when taking symmetries into account. This large number of variables prevents solving the linear regression by means of algorithms that perform inversions of $(n \times n)$–matrices. But linear regressions with such a large number of variables can be solved iteratively by updating the weight vector in the direction of the negated current gradient of the sum of squared errors. While there might be faster methods (such as conjugate gradient algorithms [Press et al. (1992)]) the procedure described below — known as "backpropagation" in the artificial neural network community — performs sufficiently well and can be implemented very quickly.

Let $r(p) \in [-64..64]$ be the game result in view of the side to move after optimal play by both players and $\{p_k\}_{k=1}^N$ the set of training examples. The objective of linear regression is to minimize the mean squared error, i.e. to find a weight vector $\boldsymbol{w}_0$ that minimizes the error function

$$E(\boldsymbol{w}) = \frac{1}{N}\sum_{k=1}^{N}\Delta_k(\boldsymbol{w})^2, \quad \Delta_k(\boldsymbol{w}) = r(p_k) - \sum_{i=1}^{n} w_i f_i(p_k).$$

Starting with an initial guess $\boldsymbol{w}^{(0)}$ for the weights, in each step the weight vector is updated according to

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} - \beta \cdot (\mathbf{grad}_{\boldsymbol{w}} E)(\boldsymbol{w}^{(t)}),$$

where $\beta > 0$ is the learning rate and $\mathbf{grad}_{\boldsymbol{w}} E$ is the vector consisting of E's partial derivatives $\frac{\partial E}{\partial w_i}$. This update scheme changes the weights in direction of the error function's steepest descent and is widely used for training artificial neural networks.

Here, the partial derivatives have a simple form since E is quadratic in w_i:

$$\frac{\partial E}{\partial w_i}(\boldsymbol{w}) = \frac{1}{N}\sum_{k=1}^{N}\frac{\partial \Delta_k(\boldsymbol{w})^2}{\partial w_i} = -\frac{2}{N}\sum_{k=1}^{N}\Delta_k(\boldsymbol{w}) f_i(p_k).$$

Thus, the steepest descent update for the i–th weight is

$$w_i^{(t+1)} = w_i^{(t)} + \frac{2\beta}{N}\sum_{k=1}^{N}\Delta_k(\boldsymbol{w}^{(t)}) f_i(p_k),$$

which can be computed simultaneously for all weights in one pass through the examples as follows: For each weight w_i there is a variable s_i that holds $\sum_{k=1}^{s}\Delta_k(\boldsymbol{w}^{(t)}) f_i(p_k)$ for $s = 1..N$. At example k, first $\Delta_k(\boldsymbol{w}^{(t)})$ is determined. This usually takes linear time depending on the number of variables. But in case of pattern features, for which many values $f_i(p_k)$ are 0, the running time can be reduced to a constant factor times the number of occurring configurations on the board by adding $w_i^{(t)} \cdot f_i(p_k)$ only for those i with $f_i(p_k) \neq 0$ which can be easily found. Analogously thereafter $\Delta_k(\boldsymbol{w}^{(t)}) f_i(p_k)$ is only added to s_i if $f_i(p_k) \neq 0$ holds. After N steps s_i contains $\sum_{k=1}^{N}\Delta_k(\boldsymbol{w}^{(t)}) f_i(p_k)$ and w_i is updated by $2\beta s_i/N$.

Since the number of occurrence varies largely among pattern configurations the just described update scheme changes weights at different speeds. Furthermore, weight estimates for configurations that occur seldomly have a high variance which may introduce large evaluation errors later in actual game–tree search. The current implementation deals with these problems by starting with $\boldsymbol{w}^{(0)} = \mathbf{0}$ and updating weight w_i by $2\,\beta\, s_i \min\{1, N_i/50\}/N_i$, where N_i is the number of examples with $f_i(p_k) \neq 0$. Dividing s_i by N_i instead of N normalizes the update speed dependent upon the number of configuration occurrences, whereas the factor $\min\{1, N_i/50\}$ introduces a controllable estimate muting for rare configurations. It is clear that regardless these constant factors E will still be minimized in the limit, since E is convex and the gradient vector still converges to $\mathbf{0}$. In practice, however, only a relatively small number of iterations will be performed (due to the large number of examples used) so that the muting factor is effective.

5 The new evaluation function

As before the new evaluation function is dependent upon game stage. In Othello the number of discs on the board is a reasonable measure. Thirteen stages were chosen, namely 13–16 discs, 17–20 discs, ..., 61–64 discs. For parameter estimation the same set of examples is used as before. It consists of ca. three million Othello positions stemming from about 60,000 games played between early versions of Igor Đurđanović's program KITTY and LOGISTELLO and 20,000 additional games that were generated by LOGISTELLO while extending its opening book. All positions were labeled by negamaxing the final game results in the tree built from all games. This procedure labels endgame positions accurately since the example games are played perfectly in this stage, whereas labels assigned to opening and middle–game positions are only approximations.

In addition to the pattern features shown in Figure 1 the phase dependent version of the example's parity feature f_1 is used. Thus, the new evaluation function has the following form:

$$
\begin{aligned}
f(p) = (&\\
&[f_{\mathrm{d4},s.1} + \dots + f_{\mathrm{d4},s.4}] + [f_{\mathrm{d5},s.1} + \dots + f_{\mathrm{d5},s.4}] +\\
&[f_{\mathrm{d6},s.1} + \dots + f_{\mathrm{d6},s.4}] + [f_{\mathrm{d7},s.1} + \dots + f_{\mathrm{d7},s.4}] +\\
&[f_{\mathrm{d8},s.1} + f_{\mathrm{d8},s.2}] + [f_{\mathrm{hv2},s.1} + \dots + f_{\mathrm{hv2},s.4}] +\\
&[f_{\mathrm{hv3},s.1} + \dots + f_{\mathrm{hv3},s.4}] + [f_{\mathrm{hv4},s.1} + \dots + f_{\mathrm{hv4},s.4}] +\\
&[f_{\mathrm{edge+2X},s.1} + \dots + f_{\mathrm{edge+2X},s.4}] +\\
&[f_{2\times5,s.1} + \dots + f_{2\times5,s.8}] +\\
&[f_{3\times3,s.1} + \dots + f_{3\times3,s.4}] + f_{\mathrm{parity},s})(p)
\end{aligned}
$$

where $s = \mathrm{stage}(p) := \max\{0, \lfloor(\#\mathrm{discs}(p) - 13)/4\rfloor\} \in \{0..12\}$ and $f_{x,s.i}$ evaluates the ith occurrence of pattern x on boards at game stage s.

In order to smooth the parameter estimates among game stages each example position p does not only contribute to stage $s = \mathrm{stage}(p)$ but also to stages $s\pm1, s\pm2$. Furthermore, in case of insufficient data parameter estimates are extrapolated resp. interpolated among game phases as follows: If a configuration does not occur in the example positions for stage s, the first stages before and after s are determined for which examples exist. Then the estimate for stage s is set to the linear interpolation of the estimates for both end points. If no examples exist in one direction, 0 is used as the end point parameter estimate.

Each iteration of the described steepest descent algorithm for parameter estimation needs about four minutes of CPU time on a PentiumPro/200 machine. In this time the entire set of about three million compressed example positions is read from disc (ca. 54 MBytes) and uncompressed. For each position p_k 46 table indices for the occurring pattern configurations are determined, and $\Delta_k(\boldsymbol{w}_{s'}^{(t)})$ is computed for five stages $s' = s, s\pm1, s\pm2$, where $s = \mathrm{stage}(p_k)$, for updating the summation variables of each involved pattern configuration. After this scan through all examples, the weights are updated and the next iteration begins. Figures 3 shows the average absolute prediction error of the evaluation function dependent on the game stage and the number of iterations for $\beta = 1$. Apparently the prediction quality increases with the number of discs on the board. Possible causes for this behavior are the decrease of labeling errors and the choice of patterns — like 2×5– and 3×3–corner — which show their best performance in late game stages. Of course, the good evaluation quality near the end of the game impacts on much earlier move decisions because typical selective searches in the middle–game already visit endgame positions. In Figure 4 graphs of the maximum and average absolute weight alterations dependent on the number of iterations are shown. The optimization process was stopped after 300 iterations and

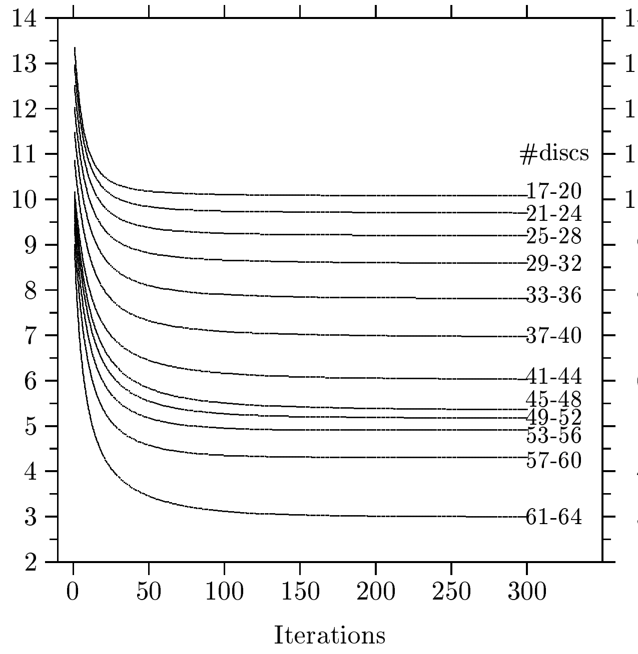

Figure 3: Average absolute prediction error dependent on game stage and iteration number

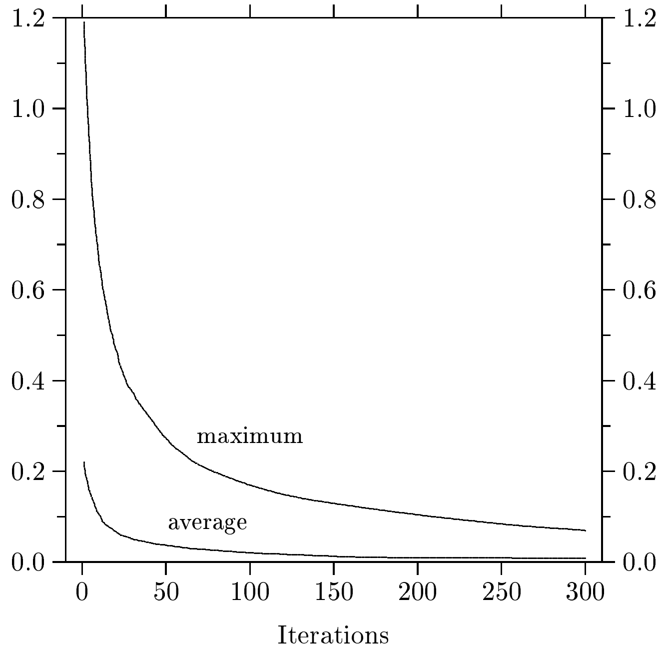

Figure 4: Average and maximum absolute weight alteration in each iteration

d	1	2	3	4	5	6	7	8	9
+0	33.6	32.1	31.8	35.0	34.3	30.7	26.1	26.4	31.8
+1	58.9	56.1	46.4	56.8	50.4	41.4	44.3	40.4	37.9
+2	82.5	74.6	70.0	62.1	61.4	59.3	47.9	50.7	53.6

Table 1: Results of several tournaments between fixed depth versions of LOGISTELLO using different evaluation functions and depths. Given are winning percentages of the player using the previous evaluation function searching at depths $d, d+1$, and $d+2$ against the player with the new function looking d plies ahead.

ca. 20h CPU time where the absolute weight alteration reached a maximum of 0.07 and its average dropped to 0.008.

6 Performance

In order to compare the quality of the previous and new evaluation function several tournaments were played between fixed depth versions of LOGISTELLO using both evaluation functions. In the tournaments each game and its return game with colors reversed were played starting with 70 nearly even opening positions with fourteen discs selected from LOGISTELLO's opening book. Sixteen plies before the end of the game all games were solved perfectly in order to focus on middle–game evaluation quality which could be spoiled by blunders in tactical endgames.[2] The results summarized in Table 1 indicate that the strength increase from using the new evaluation function under tournament conditions is comparable to that of two additional plies of brute–force search or, equivalently, to a speed–up factor of about 10 which is otherwise only achievable by parallelization.

For the construction of the new evaluation function the same patterns and training examples were used and even the mobility features were omitted in the new function. Thus, the considerable playing strength increase is surprising. However, the crucial difference between the new and the previous evaluation function is that values of pattern configurations are no longer estimated independently. The previous approach neglected correlations among configuration values and seemed to compensate for this in part by assigning considerable weights to mobility approximations which already could have been modeled only by means of line patterns. The new method on the other hand takes correlations into account and allows for more accurate modeling.

7 ProbCut

Human players are able to find good moves without searching the game–tree in its full width. Using their experience they are able to prune unpromising variations in advance. The resulting game–trees are narrow and might be rather deep. By contrast the original minimax algorithm searches the entire game–tree up to a certain depth and even its efficient improvement — the $\alpha\beta$ algorithm — is only allowed to prune backwards because it has to compute the correct minimax value. The selective search procedure PROBCUT presented in [BURO 1995a] permits pruning of subtrees that are unlikely to affect the minimax value and uses the time saved for analysis of more probably relevant variations. The idea is to take advantage of the fact that values returned by minimax searches of different depths are highly correlated provided that a reasonably good evaluation function and — if necessary — a quiescence search is used. In this case one expects that at height h (Figure 5) the result v_d of a shallow search of depth $d < h$ is a good predictor for the true minimax value v_h. Based on this estimation it is possible to determine whether v_h lies outside the current $\alpha\beta$ window with a prescribed likelihood. If so, the position does not have to be searched more deeply (since the deep search result will unlikely change root's minimax value) and the appropriate window bound is returned. Otherwise, the deep search is performed yielding the true value. Here, a shallow search has been invested but relative to the deep search the effort involved is negligible.

A natural way to express v_h by means of v_d is to use a linear model of the form $v_h = a \cdot v_d + b + e$ where $a, b \in \mathbb{R}$ and e is a normally distributed error variable having mean 0 and variance σ^2. After height h and check depth d have been chosen parameters a, b and σ can be estimated using linear regression applied to a large number of examples $(v_d(p_i), v_h(p_i))$ which have been generated by actual brute–force game–tree searches. In (Buro 1995a) regression coefficients of determination greater than 0.96 are reported which clearly indicates that the simple linear model is suitable at least for Othello and

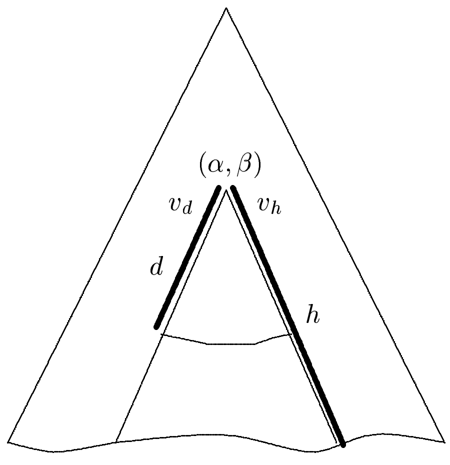

Figure 5: Forward cut scenario

[2] Today's Othello programs handle endgame positions separately by calling special endgame solvers in which the heuristic evaluation is only used for move ordering.

the chosen evaluation function.

Once all model parameters have been estimated the ProbCut procedure can test the cut conditions efficiently: $v_h \geq \beta$ holds with probability at least p if and only if $(\hat{v}_h - \beta)/\sigma \geq \Phi^{-1}(p)$ is true, where $\hat{v}_h = a \cdot v_d + b$ and Φ denotes the distribution function of a normally distributed random variable with mean 0 and variance 1. This condition is equivalent to $v_d \geq (\Phi^{-1}(p) \cdot \sigma + \beta - b)/a$. Analogously, it can be shown that $v_h \leq \alpha$ holds with probability of at least p iff $v_d \leq (-\Phi^{-1}(p) \cdot \sigma + \alpha - b)/a$. If one of these conditions is met during the game–tree search the current position will not be searched to depth h. In this way large subtrees can be cut in order to save time for more relevant lines.

It remains to choose the cut threshold $t = \Phi^{-1}(p)$. For this purpose tournaments between the non–selective and the selective program versions can be played using different thresholds in order to find the value that results in the greatest playing strength.

In the first ProbCut implementation used in LOGISTELLO[3] $h = 8$ and $d = 4$ were determined experimentally, resulting in the best performance of the program running at a node rate about eight times slower than today's. Model–parameters a, b, and σ were estimated separately for several game phases for which disc count is an adequate measure in Othello. $t = 1.5$ was empirically found to be the best cut threshold choice. For this parameter constellation the winning percentage of the ProbCut–enhanced version of LOGISTELLO playing against the brute–force version was 74% in a 70–game tournament. In this tournament both versions used a simple quiescence search and iterative deepening.

8 Multi–ProbCut

When thinking about the weaknesses of ProbCut and its simplifying assumptions the following potential improvements came to mind:

1. ProbCut detects bad moves or good refutations only at one specific height and proceeds in brute–force manner without any further pruning opportunity if no cut occurs. However, the ProbCut idea also applies to these brute–force subtrees. This observation suggests to allow recursive forward pruning at several heights (Figure 6a) in order to cut more probably irrelevant lines.

2. In extremely unbalanced positions very shallow check searches are usually sufficient for suggesting a cut. Thus, performing several check searches of increasing depth until a cut condition is met may save additional time (Figure 6b).

[3]Detailed descriptions of all components of this strong Othello program can be found at http://www.neci.nj.nec.com/homepages/mic/publications.html

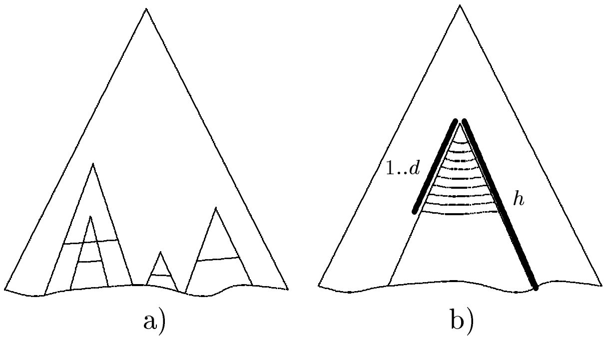

Figure 6: Illustrations of the first two ProbCut generalizations: a) allowing forward cuts at several heights and b) performing a sequence of check searches of increasing depth.

3. ProbCut uses a single cut threshold which simplifies the threshold optimization process. This appears to be a reasonable approach since stage dependencies are already dealt with by the stage dependent parameter estimations. Furthermore, in both extreme cases ($t \to 0, t \to \infty$) ProbCut performs almost as well as the plain $\alpha\beta$ algorithm which gives hope that for some $t > 0$ its performance is better. This approach only yields an approximation to the optimal parameter vector because it reduces a multi–dimensional parameter space to a straight line. As a slight generalization, cut thresholds can be optimized separately for each game stage.

4. Replacing the simple linear opinion change model by a more accurate one that makes use of additional tactical or positional features may yield a better performance. In theory this approach sounds promising since it could result in a considerable variance reduction in both quiet and tactical positions. In practice, however, it has turned out that for Othello and chess quickly computable features for the more accurate modeling of opinion changes are hard to find especially if quiescence search is used which already deals with the most important (tactical) features. Further investigations in this direction are necessary.

The ProbCut procedure refined by 1.–3. is called Multi–ProbCut (or MPC for short) indicating the pruning of more probably irrelevant subtrees by means of additional checks and cut thresholds. Figure 7 outlines a straight forward C implementation of MPC which is built upon a negamax version of the $\alpha\beta$–algorithm. Several code optimizations can be applied as well as the usual $\alpha\beta$ refinements like quiescence search, iterative deepening, and NegaScout (Reinefeld 1983). In all tournaments reported later the brute–force version and the various (Multi–)ProbCut versions of LOGISTELLO featured the same $\alpha\beta$ related enhancements. For the sake of clarity

```
const int MAX_STAGE  = 64; // e.g. disc number
const int MAX_HEIGHT = 13; // max. check height
const int NUM_TRY    = 2;  // max. number of checks

// ProbCut parameter sets for each stage and height

struct Param {
  int   d;        // check depth
  float t;        // cut threshold
  float a, b, s; // slope, offset, std.-dev.

} param[MAX_STAGE+1][MAX_HEIGHT+1][NUM_TRY];


Position pos;

int MPC(int height, int alpha, int beta)
{
  int i, max, val;
  PosDelta delta; // contains undo information

  if (height == 0) return pos.eval(); // leaf

  // check part:

  if (height <= MAX_HEIGHT) {

    for (i=0; i < NUM_TRY; i++) {
      int bound;
      Param &pa = param[pos.stage][height][i];

      if (pa.d < 0) break; // end-marker reached?

      // is v_height >= beta likely?

      bound = round((pa.t*pa.s+beta-pa.b)/pa.a);
      if (AlphaBeta(pa.d, bound-1, bound) >= bound)
        return beta; // yes => cutoff

      // is v_height <= alpha likely?

      bound = round((-pa.t*pa.s+alpha-pa.b)/pa.a);
      if (AlphaBeta(pa.d, bound, bound+1) <= bound)
        return alpha; // yes => cutoff
    }
  }

  // the remainder of the alpha-beta algorithm:

  max = alpha;
  for (i=0; i < pos.move_num; i++) {
    pos.make_move(i, delta);
    val = -MPC(height-1, -beta, -max);
    pos.undo_move(delta);
    if (val > max) {
      if (val >= beta) return val;
      max = val;
    }
  }
  return max;
}
```

Figure 7: A C implementation of negamax MPC.

they have been omitted from the given code fragment.

Central to this implementation is array `param` in which parameter sets for each game phase and search height are stored. The first `for` loop performs ProbCut checks of several depths in the form of zero window $\alpha\beta$ searches until either a cut condition is met or no more checks are left for the current game stage and node height. In the check part MPC does not call itself recursively to avoid a search depth degeneration. The original ProbCut implementation did not have to worry about recursive calls since forward cuts occurred only in nodes at one specific height. However, the recursive call of `MPC` in the $\alpha\beta$ part now causes an estimation inaccuracy: if all checks fail in a node MPC does not necessarily continue with a brute–force search like ProbCut does. Instead, cuts may occur in the subtree beneath the node. Thus, estimating the ProbCut parameters a, b, and σ by means of brute–force evaluations of example positions is no longer accurate since MPC uses shallow brute–force results to predict deep MPC results. Ad hoc parameter optimization is no longer feasible in such a general model because the parameter space dimension is large and evaluation of one data point already takes considerable CPU time. So, the proposed approximation approach described below is similar to that used in ProbCut and just ignores this subtle difference.

9 Parameter Determination

Table 5 lists the heights and check depths that are currently used by LOGISTELLO. They were determined in four steps. First, ProbCut parameters a, b and σ were estimated for each disc number, search height $h \in \{2..13\}$, and all check depths $d < h$ by linear regression using the brute–force evaluations of thousands of example positions up to depth 13 which at this time marks the maximum manageable depth. Thereafter the first check depth sequence was specified. The difference $h - d_1$ gets larger for increasing heights, which allows pruning of larger and larger subtrees in the iterative deepening process. The maximum search depth reached is now $13 - 5 = 8$ plies deeper than the brute–force part of the tree whereas the original ProbCut implementation allowed only $8 - 4 = 4$ ply extensions. In the third step, additional check depths were introduced in order to minimize the total running time of selective searches in a couple of test positions. In this process we also at-

h	3	4	5	6	7	8	9	10	11	12	13
d_1	1	2	1	2	3	4	3	4	3	4	5
d_2	–	–	–	–	–	–	5	6	5	–	–

Table 2: Currently used check depths d_1, d_2 for different heights h.

tempted to use just the evaluation function value as a predictor for deep search results (i.e. $d = 0$) — but these attempts failed due to increased running times.

After determining the check depths for each height cut thresholds were specified for different game stages by the following iterative procedure. First, tournaments were played to find an optimal cut threshold for the last game stage. For this task, starting positions were selected in such a way that ProbCut checks only occur at last stage positions. For example, if the last stage consists of positions with ≥ 36 discs, starting positions with 26 are chosen because under tournament conditions the brute–force part of the selective search tree, in which no probability cuts occur, usually reaches at least depth 10. Thereafter, the remaining thresholds were obtained analogously by keeping the already determined thresholds and optimizing the next threshold. LOGISTELLO currently distinguishes only two game phases with respect to cut thresholds, namely positions with < 36 or ≥ 36 discs. The two cut thresholds were determined by playing two sets of tournaments using starting positions with 26 and 14 discs, respectively. Optimizing the threshold in increments of 0.1 the first set of tournaments led to an "optimal" near–endgame value of 1.4. Thereafter, in a second set of tournaments, 1.0 was determined for the opening and middle–game cut threshold.

10 Performance

In order to gauge the performance of MPC relative to ProbCut and the brute–force $\alpha\beta$–algorithm, both fixed time and fixed depth tournaments were played between several versions of LOGISTELLO, which is one of today's top Othello programs. Its current pattern based evaluation function is described in (Buro 1997). Each program version ran on a Pentium–Pro CPU at 200 MHz (on which LOGISTELLO achieves a speed of about 160k nodes/second in the middle game), used iterative deepening and NegaScout, and played perfectly in the endgame when there were 20 or less moves left in the game. All tournaments consisted of 2 × 70 games starting with 70 even opening positions selected from LOGISTELLO's opening book.

For the current evaluation function the ProbCut(4,8) threshold optimization again yielded a value of 1.5, similar to the evaluation function used in (Buro 1995a). The results of the fixed time tournaments reported in Table 3 indicate that MPC LOGISTELLO is considerably stronger than its ProbCut and $\alpha\beta$ competitors. A conservative statistical test shows that all results of 140 game tournaments stating a winning percentage of at least 61% are statistically significant at the 95% level.[4] The effect of optimizing a second cut threshold is measurable but not very significant.

Pairing	Result
ProbCut(4,8) ($t = 1.5$) vs. brute–force	70%
MPC ($t = 1.0, 1.4$) vs. brute–force	80%
MPC ($t = 1.0, 1.4$) vs. ProbCut(4,8) ($t = 1.5$)	72%
MPC ($t = 1.0, 1.4$) vs. MPC ($t = 1.1$)	56%

Table 3: Results of 140 2×30 minutes game tournaments between the brute–force, ProbCut(4,8), and MPC versions of LOGISTELLO. t denotes the cut threshold(s) used. The result is the rounded winning percentage of the first player (draws count half a point).

The statistics for fixed depth tournaments given in Table 4 indicate that MPC LOGISTELLO is significantly stronger than the brute–force version reaching depth ≥ 10 even at 1:8 time odds. Moreover, already at depth $d_{\mathrm{BF}} + 1$ (time odds up to 1:25) it appears to be as strong as brute–force LOGISTELLO. Finally, at equal search times MPC looks 5 to 7 plies further ahead in selected lines and achieves a winning percentage of about 80%.

For Othello and the chosen evaluation function Multi–ProbCut significantly outperforms ProbCut as well as brute–force $\alpha\beta$ search. The amazing performance of MPC demonstrates that the $\alpha\beta$ algorithm wastes most of its time by analysing irrelevant variations. MPC, on the other hand, detects potential bad moves very early and postpones their further investigation to the next iteration. In this way, it concentrates on probably relevant lines of play without overlooking crucial tactical variations near the root. It remains to be shown whether

[4]This test is based on the fact that a sum of multinomially distributed error variables is asymptotically normally distributed. The given lower bound can be improved to 58% by using a resampling procedure.

d_{MPC}	d_{BF} 8		9		10		11		12	
d_{BF}	52	0.23	42	0.10	44	0.05	41	0.03	42	0.03
+1	58	0.22	54	0.15	51	0.08	56	0.05	55	0.04
+2	62	0.35	64	0.22	68	0.12	58	0.07	52	0.05
+3	74	0.58	73	0.31	78	0.19	60	0.11	62	0.09
+4	75	0.83	73	0.47	75	0.31	68	0.18	67	0.16
+5	84	1.36	83	0.77	82	0.50	78	0.31	77	0.30
+6	83	2.17	84	1.23	80	0.92	77	0.64	76	0.67
+7	89	3.47	89	2.25	81	1.78	84	1.33	79	1.42
+8	90	6.13	84	4.52	89	3.88	83	2.77	82	2.43
$\bar{t}_{\mathrm{BF}}$	12.5 s		28.4 s		66.7 s		171.4 s		361.8 s	

Table 4: Statistics of several 140 game tournaments between fixed depth brute–force and MPC LOGISTELLO. d_{BF} denotes the brute–force search depth ranging from 8 to 12. The MPC search depth d_{MPC} lies in $\{d_{\mathrm{BF}}, .., d_{\mathrm{BF}} + 8\}$. Reported are the rounded winning percentage of the MPC version and its relative running time compared to that of the brute–force program. The last line states the average time brute–force LOGISTELLO needed for one game.

MPC can be successfully applied to other games. Since it coexists with most of the $\alpha\beta$ enhancements currently used in chess programs, MPC may improve these programs, too.

11 Conclusion

In this paper considerable improvements of a strong Othello program have been presented. Generalizing the previously used evaluation model and selective search technique caused a playing strength increase equivalent to a speed–up factor of more than ten. This result encourages further investigations in these directions also for other games.

12 Acknowledgements

The author would like to thank Warren Smith, Mark Brockington, and David Waltz for their helpful remarks on an earlier version of this article.

References

[Buro 1994] M. Buro. *Techniken für die Bewertung von Spielsituationen anhand von Beispielen*, Ph.D. thesis (in German), University of Paderborn, Germany.

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

[Buro 1995b] M. Buro. *Statistical Feature Combination for the Evaluation of Game Positions*, JAIR 3, 373–382.

[Buro 1997] M. Buro. *An Evaluation Function for Othello Based on Statistics*, NEC Research Institute Technical Report #31.

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

[Lee & Mahajan 1990] K.F.Lee & S.Mahajan. *The Development of a World Class Othello Program*, Artificial Intelligence 43, 21–36.

[Press et al. 1992] W.H.Press, S.A.Teukolsky, W.T.Vetterling, B.P.Flannery. *Numerical Recipes*, Cambridge University Press, 2nd edition.

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

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