

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 already strong Othello program LOGISTELLO considerably. Two main ingredients of this program have been refined: First, a new evaluation function for Othello is described. While it is still table-based (and therefore fast) like its predecessors a novel procedure for estimating the table entries now allows modeling pattern interactions which leads to much higher evaluation quality. Second, MULTI-PROBCUT is introduced generalizing the formerly used selective search procedure PROBCUT in that it allows forward cuts at various heights after shallow searches of increasing depths.

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

1 Introduction

The secrets of today's best programs for unsolved perfect information games like Othello and chess are

- achieving a high raw search speed by means of assembler routines or even very fast parallel/special hardware that allows deep game-tree searches and thereby enables to play a strong game even if only poor evaluation functions are used.
- performing selective searches to follow interesting variations more deeply or to cut off probable irrelevant lines of play early, without missing many decisive variations.
- using smart evaluation functions which are often hard to construct or very time consuming.

Ideal for top-level play is the combination of all these approaches. Unfortunately, there are incompatibilities among them as well as tradeoffs. For instance, affordable

hardware realizations require a simple structure of 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

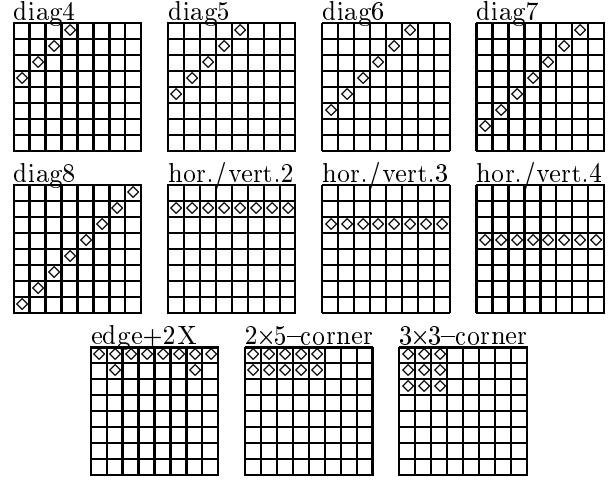


Figure 1: The current pattern set.

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.

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 1995].

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

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

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 $\{v_{i,1}, \dots, v_{i,n_i}\}$ 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

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

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:

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

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

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

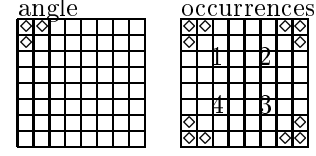


Figure 2: angle pattern that is used in the example and its four occurrences on the board

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:

$$f_1 = \# \text{empty squares modulo } 2$$

$$f_{2.1..2.4} = \text{pattern features shown in Figure 2}$$

$$f_3 = \# \text{moves for Black} - \# \text{moves for White}$$

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.

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.

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

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 \mathbf{w}_0 that minimizes the error function

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

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

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

where $\beta > 0$ is the learning rate and $\mathbf{grad}_{\mathbf{w}} E$ 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 :

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

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

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

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 $\sum_{k=1}^s \Delta_k(\mathbf{w}^{(t)}) f_i(p_k)$ for $s = 1..N$. At example k , first $\Delta_k(\mathbf{w}^{(t)})$ 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 $\Delta_k(\mathbf{w}^{(t)}) f_i(p_k)$ is only added to s_i if $f_i(p_k) \neq 0$ holds. After N steps s_i contains $\sum_{k=1}^N \Delta_k(\mathbf{w}^{(t)}) f_i(p_k)$ 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 $\mathbf{w}^{(0)} = \mathbf{0}$ 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:

$$f(p) = ([f_{d4,s,1} + \dots + f_{d4,s,4}] + [f_{d5,s,1} + \dots + f_{d5,s,4}] + [f_{d6,s,1} + \dots + f_{d6,s,4}] + [f_{d7,s,1} + \dots + f_{d7,s,4}] + [f_{d8,s,1} + f_{d8,s,2}] + [f_{hv2,s,1} + \dots + f_{hv2,s,4}] + [f_{hv3,s,1} + \dots + f_{hv3,s,4}] + [f_{hv4,s,1} + \dots + f_{hv4,s,4}] + [f_{edge+2X,s,1} + \dots + f_{edge+2X,s,4}] + [f_{2 \times 5,s,1} + \dots + f_{2 \times 5,s,8}] + [f_{3 \times 3,s,1} + \dots + f_{3 \times 3,s,4}] + f_{\text{parity},s})(p)$$

where $s = \text{stage}(p) := \max\{0, \lfloor (\# \text{discs}(p) - 13)/4 \rfloor\} \in \{0..12\}$ and $f_{x,s,i}$ evaluates the i th 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 $s = \text{stage}(p)$ but also to stages $s \pm 1, s \pm 2$. 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 $\Delta_k(w_{s'}^{(t)})$ is computed for five stages $s' = s, s \pm 1, s \pm 2$, where $s = \text{stage}(p_k)$, 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 ca. 20h CPU time where the absolute weight alteration reached a maximum of 0.07 and its average dropped to 0.008.

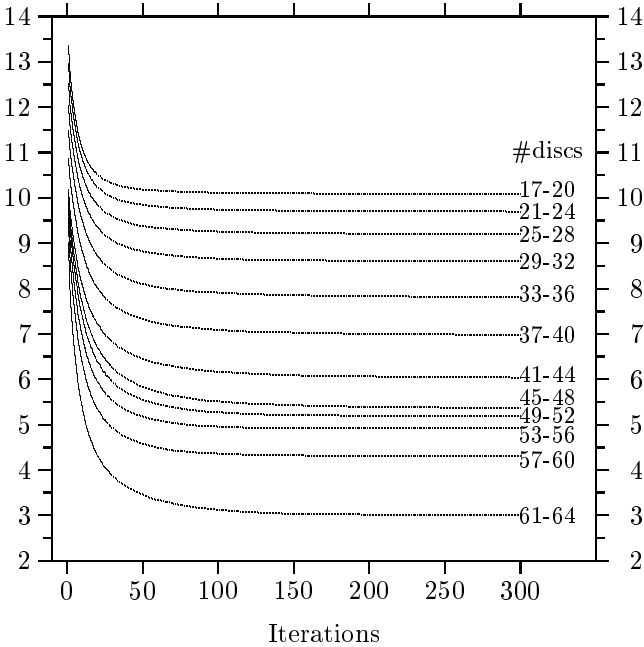


Figure 3: Average absolute prediction error dependent on game stage and iteration number

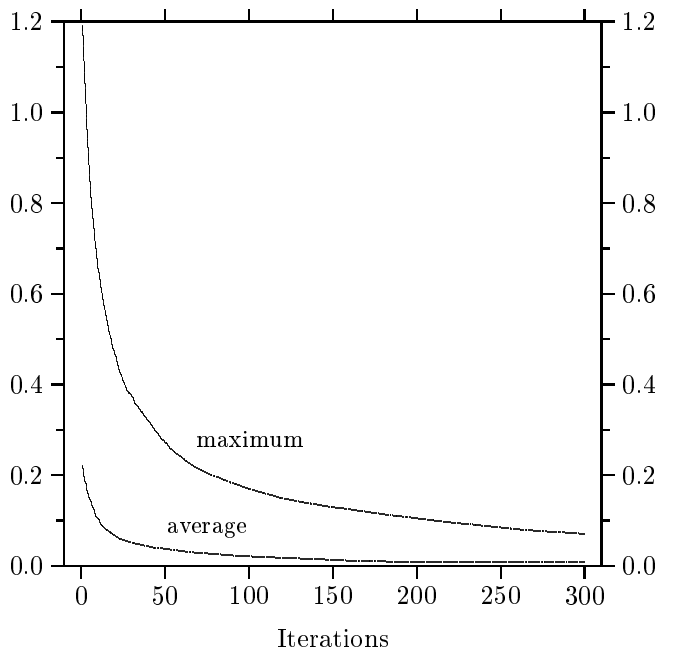


Figure 4: Average and maximum absolute weight alteration in each iteration

| 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.

6 Performance

The performance gained by the novel table estimation technique was greater than expected. 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.

One wonders why the playing strength increased dramatically since 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. The crucial difference between the new and the previous evaluation function is that values of pattern configurations are no longer estimated independently. The former evaluation 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

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

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 crucial variations. The idea is to take advantage of the fact that values returned by minimax searches of different depths are highly correlated. In order to evaluate a position at height h , it can first be examined by a shallow search of depth $d < h$. The result v_d is then used for estimating the true value v_h and to decide with a prescribed likelihood whether v_h lies outside the current $\alpha\beta$ window (Figure 5). If so, the position is not searched more deeply 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 $v_h = a \cdot v_d + b + e$ with $a, b \in \mathbb{R}$ and a normally distributed error variable e with mean 0 and variance σ^2 . After choosing height h and check depth d the parameters a, b and σ can be estimated using linear regression applied to a large number of examples $(v_d(p_i), v_h(p_i))$. Now it is possible to test the cut conditions probabilistically: $v_d \geq \beta$ 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

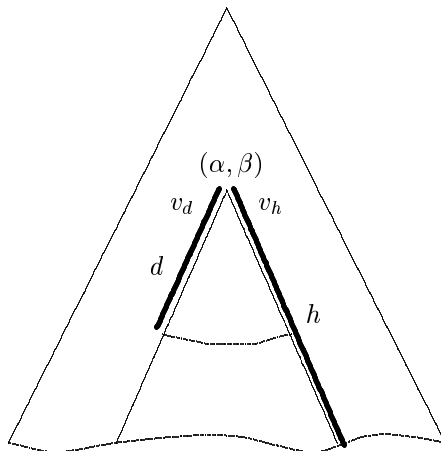


Figure 5: Forward cut scenario

will not be searched to depth h . In this way large subtrees can be cut in order to save time for the relevant lines.

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

In the first PROBCUT implementation of LOGISTELLO $h = 8$ and $d = 4$ were chosen, and a, b , and σ were estimated separately for each game phase. When $\Phi^{-1}(p) = 1.5$ the winning percentage of the PROBCUT-enhanced version of LOGISTELLO playing against the brute-force version was 74.2% in a 70-game tournament.

Recently this selective search procedure has been improved.

8 Generalizations

PROBCUT can be generalized in at least four ways by

1. allowing forward pruning at different heights. In this way bad moves — which exist in almost any position — or very good refutations can be detected earlier and more time can be saved for relevant lines.
2. performing several check searches of increasing depth until a cut condition is met. This procedure saves time in very unbalanced positions.
3. using different cut thresholds for each game stage.
4. replacing the simple linear model for the deep evaluation by a more accurate one that makes use of additional tactical or positional features for opinion change prediction. In theory this approach sounds promising since it should result in a considerable variance reduction in both quiet and tactical positions. In practice, however, it turned out that for Othello and chess quickly computable features for the more accurate modeling of opinion changes are hard to find. Further investigations in this direction are necessary.

Experiments showed that by applying generalizations 1.–3. the playing strength can be increased considerably even without the ambition to find an *optimal* parameter setting. Table 5 lists the heights and check depths that are currently used by LOGISTELLO. They were determined in four steps. First of all, the 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 that time marked 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 larger and larger subtrees. The

| 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 .

maximum search depth reached is now $13 - 5 = 8$ plies deeper than the brute-force part of the tree. The original PROBCUT implementation allowed only $8 - 4 = 4$ ply extensions. In the third step, additional check depths were selected in order to minimize the total running time of selective searches in a couple of example positions. In this process it was attempted to use just the evaluation function value as a predictor for deep search values (i.e. $d = 0$) — but these attempts failed due to increased running times.

After determining the check depths for each height cut thresholds can be specified for different game stages by the following iterative procedure. Tournaments are played as before beginning with starting positions selected from late middle-games in order to find an optimal cut threshold for this game stage. Thereafter, the remaining thresholds are obtained analogously by keeping the already determined thresholds and optimizing the next threshold by playing tournaments in which starting positions of the current game stage are used.

LOGISTELLO currently distinguishes only two game phases with respect to cut thresholds, namely positions with < 36 resp. ≥ 36 discs. The cut thresholds were determined by playing two sets of tournaments using starting positions with 26 and 14 discs, respectively. Disc number 26 was selected 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. 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 middle-game cut threshold.

9 Multi-ProbCut: Implementation and Performance

The just described generalization of PROBCUT is called MULTI-PROBCUT indicating the more accurate pruning of more probably irrelevant subtrees by means of additional checks and cut thresholds. Figure 6 outlines a straight forward implementation of MULTI-PROBCUT which is built upon a negamax version of the $\alpha\beta$ -algorithm. Central to this implementation is array `para` in which parameter sets for each game phase and search height are stored. The first `for` loop performs PROB-

```
// NUM_TRY ProbCut parameter sets for each disc number
// and each distance to the search horizon
```

```
struct Para {
    int d;          // check depth
    float t;        // cut threshold
    float a, b, s;  // slope, offset, std.-dev.
} para[64][MAX_HEIGHT+1][NUM_TRY];

Position pos;

int MultiProbCut(int height, int alpha, int beta)
{
    int i, max, val;
    PosDelta delta;

    if (IsLeaf(pos, height)) return Eval(pos);

    // check part:

    if (height <= MAX_HEIGHT) {
        for (i=0; i < NUM_TRY; i++) {
            int bound;
            Para pa = para[DiscNum(pos)][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.movenum; i++) {
        Move(pos, pos.move[i], &delta);
        val = -MultiProbCut(height-1, -beta, -max);
        Undo(pos, delta);
        if (val > max) {
            if (val >= beta) return val;
            max = val;
        }
    }
    return max;
}
```

Figure 6: A negamax implementation of MULTI-PROBCUT.

| Pairing | Result |
|--|--------|
| 4-8 ($t = 1.5$) vs. brute-force | 68.6% |
| multi ($t = 1.0, 1.4$) vs. brute-force | 80.0% |
| multi ($t = 1.0, 1.4$) vs. 4-8 ($t = 1.5$) | 71.8% |
| multi ($t = 1.0, 1.4$) vs. multi ($t = 1.1$) | 55.7% |

Table 3: Results of 140 game tournaments between the brute-force, 4-8-PROBCUT, and MULTI-PROBCUT versions of LOGISTELLO. t denotes the cut threshold(s) used. In case of 4-8-PROBCUT threshold optimization again yielded a value of 1.5 like for the previous evaluation function. The results, given in form of the winning percentage of the first player, indicate a considerable playing strength gain when using MULTI-PROBCUT. Moreover, the last result shows that the effect of multiple cut thresholds is measurable but not very significant.

CUT checks of several depths in 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 MULTI-PROBCUT 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. On the other hand the recursive call of MultiProbCut in the $\alpha\beta$ part now causes an estimation inaccuracy: If in a node all checks fail MULTI-PROBCUT does not necessarily continue with a brute-force search like PROBCUT does. Instead, in the subtree beneath the node cuts may occur. Thus, estimating the PROBCUT parameters a, b , and σ by means of brute-force evaluations of example positions is no longer accurate. In fact, the underlying model has changed by allowing cuts at various depth in that now values of MULTI-PROBCUT searches have to be approximated by shallow searches rather than brute-force values — as before. Exact parameter estimation in such a general model is no longer feasible because it has to go hand in hand with cut threshold optimization. Therefore, it was decided to apply the previous estimation technique — with success as the tournament results listed in Table 3 show. To increase the performance of PROBCUT based selective search routines even further future research could focus on more accurate opinion change models.

10 Conclusion

In this paper considerable improvements of an already 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 10. This amazing result encourages further investigations in these directions also for other games. It also shows the bene-

fits of trying approaches that look extreme in theory but may be effective in practice.

Acknowledgements

Without the competition with other strong Othello programs and the discussions with the program authors LOGISTELLO would not be as strong as it is today. In particular I would like to thank Igor Đurđanović, who pointed out improvements of the steepest descent procedure for table value estimation to me. I am also grateful to Warren Smith and Mark Brockington 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.
- [ROSENBLOOM 1982] P.S. Rosenbloom. *A World-Championship-Level Othello Program*, Artificial Intelligence 19: 279–320.