Improving Heuristic Mini-Max Search by Supervised Learning

Michael Buro

NEC Research Institute, Princeton NJ 08540, USA [1]

Abstract

This article surveys three techniques for enhancing heuristic game-tree search pioneered in the author's Othello program LOGISTELLO, which dominated the computer Othello scene for several years and won against the human World-champion 6-0 in 1997. First, a generalized linear evaluation model (GLEM) is described that combines conjunctions of Boolean features linearly. This approach allows an automatic, data driven exploration of the feature space. Combined with efficient least squares weight fitting, GLEM greatly eases the programmer's task of finding significant features and assigning weights to them. Second, the selective search heuristic PROBCUT and its enhancements are discussed. Based on evaluation correlations PROBCUT can prune probably irrelevant sub-trees with a prescribed confidence. Tournament results indicate a considerable playing strength improvement compared to full-width α-β search. Third, an opening book framework is presented that enables programs to improve upon previous play and to explore new opening lines by constructing and searching a game-tree based on evaluations of played variations. These general methods represent the state-of-the-art in computer Othello programming and begin to attract researchers in related fields.

Key words: Selective Game-Tree Search, Evaluation Function, Feature Construction, Opening Book Learning, GLEM, PROBCUT, LOGISTELLO

> "I consider the most important trend was that computers got considerably faster in these last 50 years. In this process, we found that many things for which we had at best anthropomorphic solutions, which in many cases failed to capture the real gist of a human's method, could be done by more brute-forcish methods that merely enumerated until a satisfactory solution was found. If this is heresy, so be it." – Hans Berliner on AI trends [4]

1 Introduction

The achievements of AI research over the past decades have been tremendous and a front runner in this area — right from the start in the 1950s — has been computer games research. Driven by the desire to build machines capable of beating the best humans at chess, remarkable discoveries and technological advances have paved the way to Deep Blue's narrow victory over Garry Kasparov

[1] E-mail: mic@research.nj.nec.com

in 1997. While an old AI dream had become true, the means to its accomplishment had not been envisioned by the AI pioneers who thought that beating the chess World-champion would require human-like reasoning abilities. Instead, after the discovery of α-β search and the success of Chess 4.5 in the 1970s, it became clear that the future of computer chess lies in full-width – or so called *brute-force* – searching on fast hardware. Steady hardware improvements in conjunction with an efficient parallelization of α-β search finally led to IBM's Deep Blue, a 64-node SMP machine equipped with 512 special purpose chess chips, which are capable of searching 200 million chess positions a second. Kasparov, on the other hand, searches 1-2 nodes a second and still is regarded the better chess player by many. The main AI lesson learned from this event is that in some domains human creativity, intuition, and reasoning ability can be compensated for or even be surpassed by brute-force search requiring only simple evaluation functions. Unfortunately, this insight does not help much when it comes to solving much harder decision problems than chess for which full-width search is infeasible or simple heuristics do not work. Clearly, there is no hope of finding a practical general purpose problem solver because basic decision problems already belong to PSPACE or EXPTIME, or are even undecidable. Therefore, AI research is focusing on real world problems that humans routinely solve but even the fastest parallel machines currently can not handle.

Three months after Garry Kasparov lost 2.5–3.5 against Deep Blue, a similar event was organized at the NEC Research Institute in Princeton. This time, the then human Othello World-champion Takeshi Murakami played six long-timed games against LOGISTELLO – a program running on an ordinary PC – and lost 0–6, without getting the program into trouble a single time. While LOGISTELLO can be regarded a classic two-person game program, all of its move decision components are automatically tuned by machine learning techniques. This sets it apart from other programs that mostly rely on manual tuning and, after TD-Gammon reaching master level [20], marks the second breakthrough of machine learning applied to games. In this article the evaluation, search, and opening book learning techniques pioneered in LOGISTELLO are surveyed. We first describe a novel evaluation function model and its application to Othello. The resulting evaluation function is pattern based, accurate, and very fast, and outperforms all other approaches tried so far. All of its more than a million parameters have been fitted by linear regression applied to a large training set. We then move on discussing PROBCUT – a domain independent selective search heuristic based on evaluation correlations that considerably improves α-β search. The parameters of the underlying linear model have been estimated by linear regression as well. In the last part we present an opening book framework that allows programs to learn from past games in order to avoid losing games the same way and to explore new openings automatically.

2 Evaluation Function Learning

Many AI systems use evaluation functions for guiding search tasks. In the context of strategy games they usually map game positions into the real numbers for estimating the winning chance for the player to move. Decades of research has shown how hard a problem evaluation function construction is, even when focusing only on particular applications. To simplify the construction task, the notion of evaluation features emerged. This notion assumes that there exist reasonable approximations of the perfect evaluation function in the form of combinations of a few distinct numerical properties of the state - called features. Given this, evaluation functions can be constructed in two phases by selecting features and combining them. Selecting features is one of the most important and difficult subtasks in the construction of game playing programs. It requires both domain-specific knowledge and programming skills because of the well-known tradeoff between speed and knowledge in look-ahead search. A couple of years ago the authors of the best programs still picked not only features but also their weights in course of a tedious optimization process. This is somewhat surprising, because already in the 1950s Samuel proposed a way for automatically tuning weights [19] similar to TD-learning. Least squares fitting a large number of parameters in a (linear) model is well understood and a computationally feasible problem. However, our understanding of how to generate features for building fast and accurate evaluation functions is limited. What makes the latter problem much harder than least squares parameter fitting, is the much larger search space and the lack of nice analytic properties that can guide the search algorithm to find (local) extrema quickly. The standard approach to attacking this hard combinatorial problem is to generate functions with respect to a particular model and to keep the best one encountered so far. A prominent example is Genetic Programming (GP). GP is very general and has produced quite a number of good solutions to small sized problems [11]. However, because the methods used for generating offspring make little use of available domain knowledge, GP currently does not scale well to larger problem sizes. Another disadvantage of GP is its lack of efficient numerical parameter optimization. Alternative techniques, such as adapting the topology and edge weights of feed forward networks (e.g. meiosis networks [12], node splitting [22]), Morph [14], or ELF [21], are promising but face similar problems and have not yet led to high performance applications.

2.1 GLEM

In what follows we discuss GLEM – a generalized linear evaluation model that efficiently combines GP-like automatic feature space exploration with fast numerical parameter tuning. Basically, GLEM evaluation functions can be regarded as two level feed forward networks with binary inputs: one level that and-combines inputs and one node that applies a squashing function to the sum of weighted results:

$$e(p) = g(\sum_{i=1}^{n} w_i \cdot c_i(p)),$$

where each *configuration* c_i is a conjunction of given Boolean (so called *atomic*) features, each $w_i \in \mathbb{R}$ is a weight, and $g : \mathbb{R} \to \mathbb{R}$ is an increasing and differentiable link function. In this context Boolean values are treated as 0 or 1. The weights are subject to the usual least-squares optimization. That is, given a set of configurations $c_1, \ldots, c_n$, a link function g, and a sequence of scored training instances $((p_i, r_i) \mid i = 1 \ldots N)$, the weights are chosen such that the total squared error

$$E(w) := \sum_{i=1}^{N} (r_i - e_w(p_i))^2$$

is minimized. This simple model has several desirable properties:

- Atomic features are the building blocks of more sophisticated ones. This allows the automated discovery of new important features by systematic combination.
- There is no need for tuning parameters of analytic functions (e.g. neural networks) to approximate simple non-linear relations. GLEM models non-linear effects quite naturally by Boolean combinations.
- Combining features linearly keeps the evaluation time overhead low. Actually, not even multiplications with weights are necessary because $c_i(p)$ is either 0 or 1. Moreover, the simple form of the evaluation function allows a fast approximation of optimal weights, even in large systems.
- In order to deal with saturation effects an increasing non-linear link function, such as $g(x) = 1/(1 + exp(-x))$, can be used without decreasing the search speed. Because g is monotone it suffices to compare g's arguments ($g(x_1) > g(x_2) \iff x_1 > x_2$).

Before applying GLEM several practical and theoretical issues have to be resolved, including how to generate a (potentially large) set of labelled training examples, how to select atomic features, how to generate relevant configurations from data, and how to fit a large number of parameters. Practical answers to those and other important questions are given in [8]. In this context we only mention two of the most important techniques proposed: pre-selecting configurations depending on their training set match count to fight over-fitting and combining configurations to patterns to greatly speed-up weight fitting and evaluation. Patterns play a central role in the application of GLEM reported in the next subsection.

GLEM allows program authors to concentrate on the part of evaluation function construction, where humans excel: the discovery of fundamental features by reasoning about the problem. GLEM simplifies this task because the exact feature formulation is no longer needed. The system is able to approximate complex features by combining atomic fragments. In this way it is possible for the programmer to speculate about feature building blocks and to leave the details of creating features and assigning weights to them to the system.

2.2 Application to Othello

The presented general framework for the construction of evaluation functions has been inspired by the work on our Othello program LOGISTELLO. Besides the progress in selective search and automated opening book construction – which is reported later – the application of GLEM has greatly improved the playing strength of this program. LOGISTELLO is able to beat the best human Othello players handily, even when running only on ordinary hardware [7].

Othello is a popular Japanese board game, played by two players on an 8x8-board using 64 two-colored discs. Moves consist of placing one disc on an empty square and turning all bracketed opponent's discs over. Fig. 5 shows an example. The game ends when neither player has a legal move, in which case the player with the most discs on the board has won.

The most important positional features in Othello are disc stability, mobility, and parity. In particular:

- Stable discs can not be flipped by the opponent. Therefore, they directly contribute to the final score. The most prominent stable discs are occupied corners, which can be used as anchors for creating more stable discs.
- Having fewer move options than the opponent is dangerous, because it increases the chance of losing a corner in the near future.
- Making the last move in an Othello game is advantageous, since it increases one's own disc count while decreasing the number of opponent's discs. Parity generalizes this observation by considering last move opportunities for every empty board region.

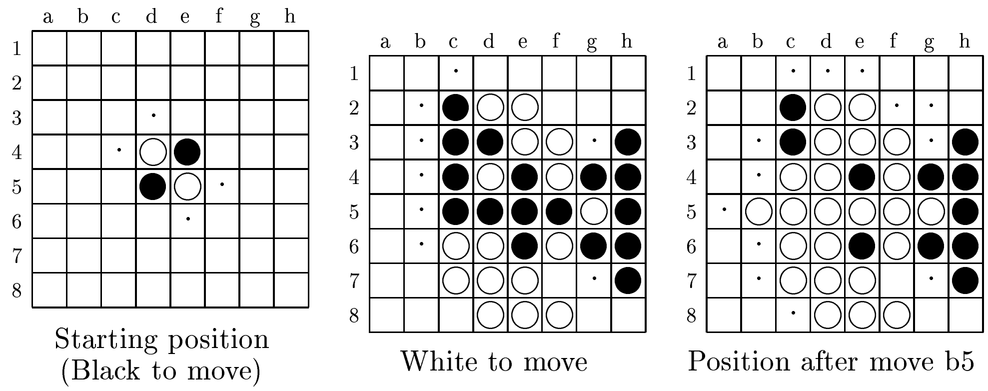


Fig. 1. Example positions. Legal moves are marked with a dot.

In [18] and [13] table-based evaluation schemes were introduced, in which values of all edge configurations were precomputed by (probabilistic) mini-max algorithms and stored in a table for a quick evaluation of the edge structure. GLEM generalizes this technique by allowing configurations of arbitrary shape and replacing the ad-hoc weight assignment by a least squares fit. LOGISTELLO's current pattern set is shown in Figure 2. Important positional features of Othello can be quickly approximated by those patterns, which are built upon the raw board representation: horizontal, vertical, and diagonal lines cover mobility and the remaining patterns deal with corner- and edge-tactics and parity issues. In the GLEM context patterns are defined as complete configuration sets spanned by discrete features. E.g., the integer valued features "contents of square a1", ... , "contents of square a8" define a pattern which covers all $3^8 = 6561$ disc configurations along the western edge of an Othello position. Using patterns greatly increases the speed of weight fitting and evaluation at the expense of evaluation accuracy, because the number of configurations to be evaluated equals the number of patterns. This allows for a very efficient table-based evaluation (Figure 3) and still offers an expressiveness matched only by large neural networks. It should be noted that patterns only define supersets of the configurations actually used. The final decision to add specific pattern configurations to the model depends on statistical considerations such as the configuration's training set match count (to counter over-fitting) and correlation with the game outcome. The given pattern set is the result of many experiments which involved evaluation accuracy and speed

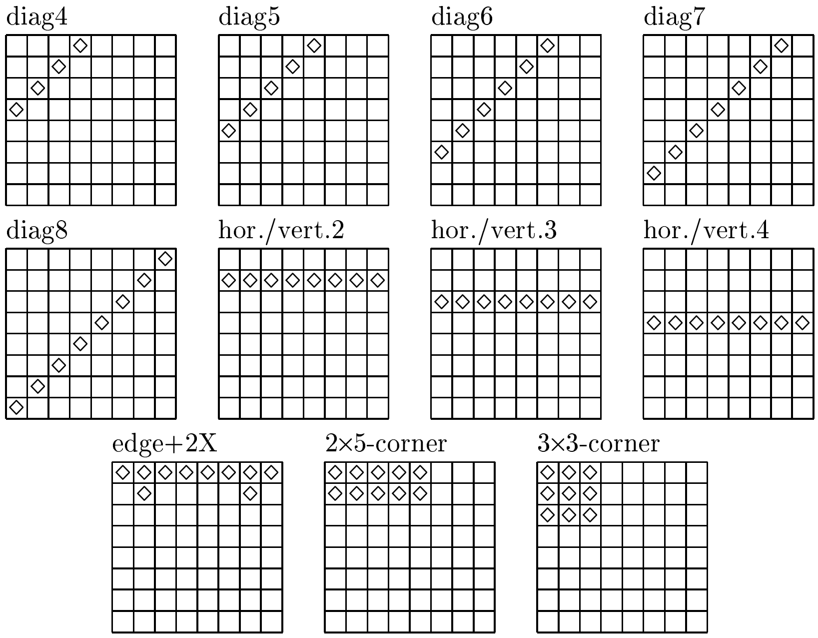


Fig. 2. LOGISTELLO's pattern set. Patterns that can be obtained by rotating and mirroring the board have been omitted. Each diamond represents a discrete feature f with range $\{0, 1, 2\}$. $f(p)$ is defined by the particular square contents (i.e. white disc $\mapsto 0$, empty $\mapsto 1$, black disc $\mapsto 2$).

considerations. Ongoing research is dealing with automatic pattern generation. A detailed discussion of these and related issues can be found in [8].

LOGISTELLO's evaluation function distinguishes 13 game stages, depending on the number of discs on the board. In addition to the patterns shown in Figure 2 a simple parity (pattern) feature is used which deals with the last move advantage globally by considering the number of empty squares modulo 2. LOGISTELLO's evaluation function has the following form:

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

where $s = \mathrm{stage}(p)$ and $f_{x,s.i}$ evaluates the i-th occurrence of pattern x on boards at game stage s (e.g. $f_{3\mathrm{x}3,s.1} + \ldots + f_{3\mathrm{x}3,s.4}$ determines the evaluation for the whole corner structure by adding up table values for each of the four

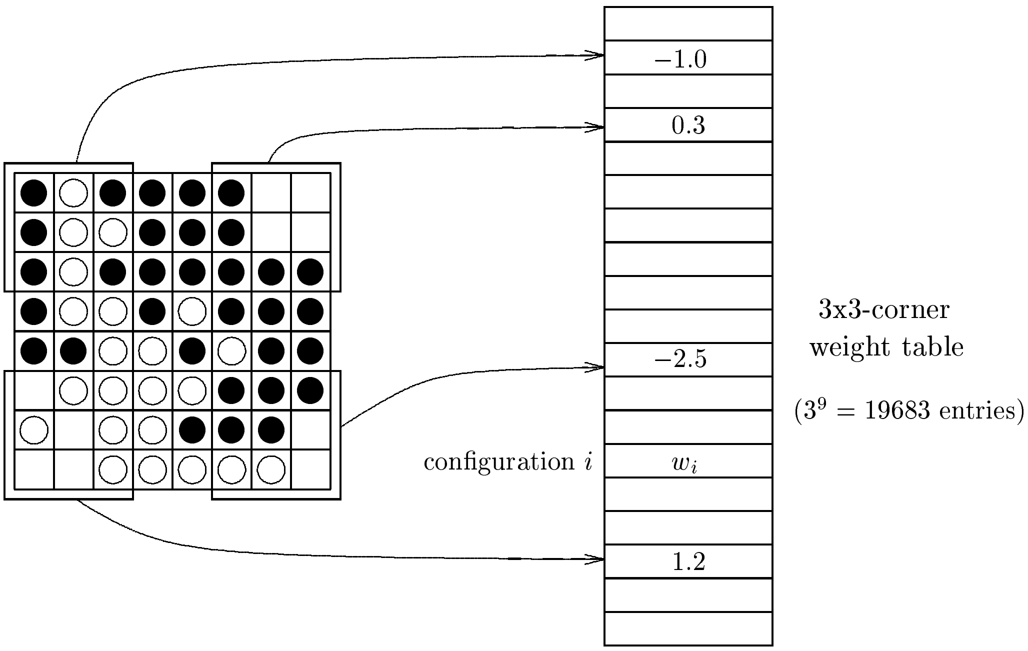


Fig. 3. Fast table-based pattern evaluation.

corners, Figure 3). Several million training positions labelled with either the true mini-max value or an approximation of it were generated from self-played games to fit approximately 1.2 million weights. This figure takes weight sharing among symmetric configurations into account. Equipped with this evaluation function and running on a PentiumPro/200 Linux PC, on which it achieves a search speed of 160k nodes/second, LOGISTELLO beat the human Othello World-champion 6–0 in August 1997.

Over the years, LOGISTELLO's evaluation function changed considerably: from a classic form – featuring only a handfull manually weighted features, over a version that estimated configuration values using the naive Bayes approach and weighted whole patterns by logistic regression [5], to its current form utilizing approximately 100,000 binary features in conjunction with over 1.2 million automatically tuned parameters. In each step the evaluation accuracy *and* speed was increased significantly. Table 1 shows experimental evidence for the considerable accuracy gain obtained when moving from weight assignment based on naive Bayes combined with logistic regression to fitting a large sparse linear regression system. The strength increase is comparable to that of two additional plies of full-width search or, equivalently, to a speed-up factor of about ten, which is otherwise only achievable by parallelization. For the construction of the GLEM based evaluation function the same patterns and training examples were used and even the previously utilized mobility features were omitted. The significant playing strength increase is therefore surprising. However, the crucial difference between the new and the previous evaluation model is that values of pattern configurations in GLEM 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 by means of line patterns alone. GLEM, on the other hand, takes feature correlations into account.

Observing that short Boolean combinations of simple binary features can approximate important Othello concepts combined with the "mechanical" anal-

Table 1
Results of several 140 game tournaments between fixed depth versions of LOGISTELLO using different evaluation functions and depths. Given are the rounded winning percentages of the player using the previous evaluation function searching at depths $d, d+1$, and $d+2$ against the GLEM version looking d plies ahead.

d	1	2	3	4	5	6	7	8	9
+0	34	32	32	35	34	31	26	26	32
+1	59	56	46	57	50	41	44	40	38
+2	83	75	70	62	61	59	48	51	54

ysis of millions of training positions has produced an expert program capable of beating any human player. Interestingly, the game knowledge encoded in the set of over a million configuration weights goes far beyond the mobility features we intended the system to approximate in the first place.

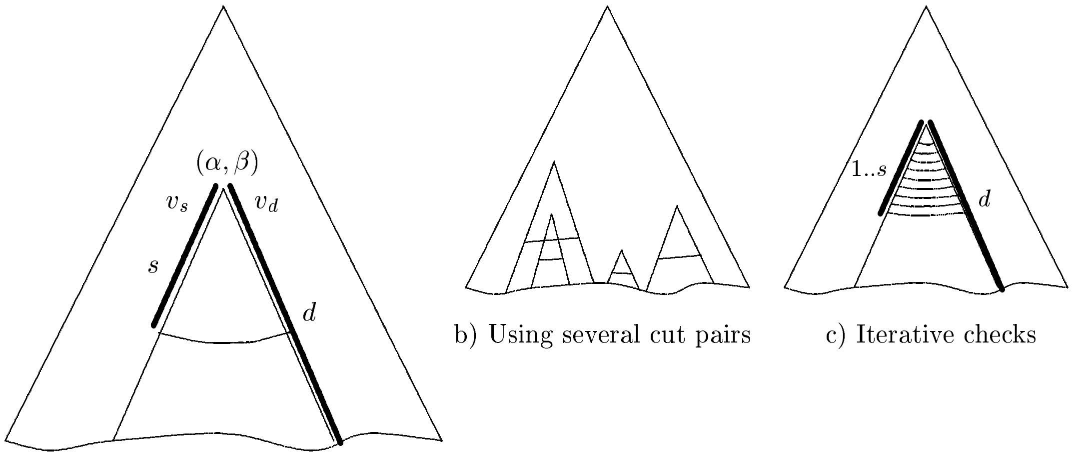
This result encourages the application of GLEM to other games and decision problems in other domains. Attractive candidates are chess and Go because both games are very popular and well analyzed. And yet, for chess, hardware roughly equivalent to 2,000 ordinary PCs is currently needed to compete with the human World-champion. For Go, the status is even worse because full-width search is infeasible due to the large branching factor. Because a good evaluation function is not known either, amateurs can still beat the best Go programs handily. The key to better chess and Go programs lies in improved evaluation functions. A starting point could be the analysis of known features with regard to their approximation by simple Boolean functions as proposed by GLEM.

3 Selective Search Based on Evaluation Correlation

Human players can find good moves without searching the game-tree in its full width. Using their experience, they can prune unpromising variations in advance. The resulting game-trees are narrow and might be rather deep. By contrast, the original mini-max algorithm searches the entire game-tree up to a certain depth and even its efficient improvement – the α-β algorithm – may only prune backwards because its purpose is to compute the correct mini-max value. In what follows, the forward pruning heuristic ProbCut [6] is discussed which aims at focusing the look-ahead search to relevant variations and thus makes more efficient use of the allocated time. Several approaches to selective search have been studied in the past. Besides selective quiescence-search techniques, such as the null-move heuristic [3] which is quite effective in non-zugzwang games, other algorithms have been proposed ([15] [16] [17] [2]) that search the game tree in best-first manner, but use an amount of memory roughly on the order of number of nodes searched. Currently, this is not practical for programs with fast evaluation functions running on conventional hardware with limited memory. ProbCut needs no additional memory and its application is not limited to quiescence search. Moreover, it is effective not only in tactical positions where one move is clearly superior to all others, such as the "singular extensions" introduced in [1].

3.1 ProbCut

The selective search heuristic ProbCut permits pruning of subtrees that are unlikely to affect the mini-max value and uses the time saved for analysis of



b) Using several cut pairs

c) Iterative checks

a) Forward cut scenario

Fig. 4. a) ProbCut uses v_s to decide whether v_d lies outside (α, β) with a prescribed likelihood. b) & c) ProbCut enhancements.

probably more relevant variations. This approach capitalizes on the fact that values returned by mini-max 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, a shallow search result v_s is a good predictor for the deep mini-max value v_d. Based on this estimation, we can determine whether the deep mini-max value lies outside the current α-β window with a prescribed likelihood. If so, the position need not be searched more deeply because the deep search result will unlikely change the root's mini-max value. 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, due to the exponential tree growth (Figure 4a).

A natural way to express the relationship between search results of different depth is a linear model of the form $v_d = a \cdot v_s + b + e$ where a, b are real constants and e is a normally distributed error variable having mean 0 and variance σ^2. Once all model parameters are estimated by linear regression applied to a large number of training pairs $(v_d(p_i), v_s(p_i))$, ProbCut can test the cut conditions $v_d \leq \alpha$ and $v_d \geq \beta$ efficiently during game-tree search: after computing the shallow search result v_s, the search is terminated in the current position iff $a \cdot v_s + b$, which is an unbiased estimator for v_d, lies outside of $[\alpha - t \cdot \sigma, \beta + t \cdot \sigma]$. Here, t is an adjustable confidence parameter that can be optimized by means of tournaments.

In the first ProbCut implementation used in Logistello (outlined in Figure 5), s=4 and d=8 were chosen and t=1.5 was empirically found to be the best cut threshold. For this parameter constellation the winning percentage of the ProbCut-enhanced version of Logistello playing against the full-width version was 74% in a 70-game tournament.

```
const float t     = 1.5;      // confidence level
const int   s     = 4;        // depth of shallow search
const int   h     = 8;        // check height
const float a     = ...;      // regression slope
const float b     = ...;      // regression bias
const float sigma = ...;      // regression standard deviation

int AlphaBeta(int height, int alpha, int beta)
{
  if (height == 0) return eval(pos);

  // ProbCut heuristic:

  if (height == h) {
    int bound;

    // v_h >= beta likely? yes => cutoff

    bound = round((+t * sigma + beta - b) / a);
    if (AlphaBeta(s, bound-1, bound) >= bound) return beta;

    // v_h <= alpha likely? yes => cutoff

    bound = round((-t * sigma + alpha - b) / a);
    if (AlphaBeta(s, bound, bound+1) <= bound) return alpha;
  }

  ... remainder: do/undo moves and call AlphaBeta recursively
}
```

Fig. 5. C implementation of the ProbCut heuristic

3.2 Multi-ProbCut *and* EndCut

Although ProbCut already marks a big and game independent improvement over full-width α-β search, it can easily be refined in several ways: Multi-ProbCut (MPC, [10]) allows for pruning at different search heights, uses game-stage dependent cut thresholds, and conducts shallow check searches using iterative deepening. The latter improvement detects extreme positions much earlier. Incorporated in Logistello, MPC featuring up to $(s = 5,\ d = 17)$ cuts and two cut thresholds (for the opening and middle game) beats regular $(s = 4,\ d = 8)$ ProbCut about 72% in a 140 game tournament. At equal search times MPC looks 5 to 7 plies further ahead in selected lines compared with full-width α-β search and achieves a winning percentage of about 80%. Both program versions are equally strong if the MPC time gets reduced by a factor of 25.

The ProbCut approach also applies to Othello endgames in which computers usually play perfectly due to exhaustive searches of the remaining game-tree. Solving a position for win/draw/loss or maximizing the score earlier than the opponent is a big advantage. Thus, authors of good programs spend considerable time on optimizing their endgame search. The tricks of the trade include avoiding last move disc flips (just counting them suffices), using a bit-board representation for fast move generation, and sorting moves using the middle-game evaluation to increase the number of α-β cut-offs. Recently, significant endgame speed improvements have been reported by Gunnar Andersson, who is using a mixture of fastest-first and best-first search in his strong program ZEBRA, and by Jan C. de Graaf whose clever if-less endgame code at least doubles the search speed on Intel processors. The fastest endgame searchers running on ordinary PCs are currently able to solve for win/draw/loss in a matter of minutes when there are around 26 moves left in the game. At this game stage programs decide to switch from middle-game to endgame search taking into account the remaining time and simple measures of the position's search complexity. When starting the exact endgame search early, it often happens that the search runs out of time before completion in which case it may miss a winning move. In order to bridge the gap between heuristic middle-game and perfect endgame search with regard to both search time and accuracy, good programs use selective endgame searches based on the PROBCUT idea. In its simplest form, the so called ENDCUT procedure performs shallow middle-game searches when reaching positions with a specific number of discs. Depending on the search result it then decides whether the true mini-max endgame result of the subtree beneath the node will fall outside the current search window with a prescribed likelihood. If so, the subtree gets pruned and the search proceeds with more relevant paths. Embedded in an iterative framework, which increases the confidence level stepwise, ENDCUT allows a smooth transition from heuristic middle-game to exact endgame search that is able to find best moves in a limited time more often than the classic approach.

In summary, for Othello and the chosen evaluation function a search based on PROBCUT significantly outperforms full-width α-β search. MPC's amazing performance demonstrates that the α-β algorithm wastes most of its time by analyzing irrelevant variations. MPC, on the other hand, detects potential bad moves early and postpones their further investigation. In this way, it concentrates on probably relevant lines of play without overlooking crucial tactical variations near the root position. It remains to be shown whether MPC can be successfully applied to other games. Because it coexists with most of the α-β enhancements currently used in chess programs, MPC might improve these programs, too.

4 Opening Book Construction

In spite of many evaluation and search improvements, programs still show weaknesses in the opening phase stemming from a lack of strategic planning. To mitigate this problem, programs utilize opening books in which move sequences or positions together with moves are stored. Their automatic generation was of little interest in the past, because move sequences could be taken from the literature, suited to one's own requirements – such as the striving for tactical complications – and manually updated if necessary. Today, however, many game-playing programs are attached to servers, playing against human players and other programs 24 hours a day. In order to prevent repeated losses it therefore has become necessary for programs to update their opening books automatically without human intervention.

In multi-game matches players are facing simple but effective playing strategies that cannot be met by the well-known game-tree search techniques alone. Perhaps the most obvious and simple one is the following: "If you have won a game, try it the same way next time." A player with no learning mechanism and no move randomization follows this strategy, but is also a victim of it, because he does not deviate and therefore can lose games twice in the same way. To avoid this, the player must find reasonable move alternatives. He can do so passively by copying opponent's moves when colors are reversed. This elegant method lets the opponent show you your own errors, so you can play the opponent's winning moves next time by yourself. In this way, even an otherwise stronger opponent can be compromised, because – roughly speaking – eventually he is playing against himself. Thus, copying moves makes it necessary to come up with good move alternatives actively. To do so, a player must understand his winning chances after deviations from known lines.

These basic requirements of a skilled match strategy lead directly to an algorithm for guiding opening book play based on mini-max search [9]. The procedure builds a game-tree from played variations – starting with the initial game position – and labels the leaves depending on the particular game outcomes. Moreover, in each interior node the algorithm evaluates all moves not played so far and adds the edge and node corresponding to the heuristically best move together with its evaluation to the tree. Given such a tree, it is easy to guide the opening book play – that avoids losing the same way and explores new variations – by propagating leaf evaluations to the root using the mini-max algorithm and extending lines by expanding mini-max leaves.

All good Othello programs now use variations of this opening book algorithm. Surprises in tournament games caused by blindly following non-evaluated opening lines are a thing of the past, programs connected to the Othello server (telnet:external.nj.nec.com:5000) are improving their books autonomously, and extensive parallel book extension by self-play has revealed flaws in some commonly played openings.

5 Outlook

The application of the described machine learning approaches for tuning the core components of programs for two-person perfect information games has spawned a new generation of Othello programs much stronger than before. After four years of successful tournament play and beating the human World-champion, LOGISTELLO ended its career with a straight 22-win victory in its last computer Othello tournament in October 1997.

Although the techniques utilized by game programs and human players are still quite different, the proposed methods for improving evaluation, search, and post-mortem game analysis aim at closing the gap:

- cognitive research shows that good human chess and Go players have access to a large number of patterns associated with plans on how to proceed in the game. GLEM allows to generate pattern configurations from data and approximates winning plans by assigning (optimal) weights to configurations, which are then used by a crude planning algorithm – α-β search. As the GLEM approach is quite general it will be interesting to see how it performs in other applications.
- good human players are conducting a highly selective look-ahead search in which they only rarely miss decisive variations. On the other hand, the original mini-max algorithms waste most of their time by analyzing irrelevant lines. In the presence of good evaluation functions, selective α-β searches based on PROBCUT can approximate the very focussed human search behavior. However, programs still have to search much more nodes in order to come up with decisions of comparable quality. Future research on better evaluation schemes, selective search, and planning will benefit from machine learning advances and certainly result in a decreased search effort.

References

[1] T. Anantharaman, M.S. Campbell, and F.H. Hsu. Singular extensions: Adding selectivity to brute-force searching. *Artificial Intelligence*, 43:99–109, 1990.

[2] E.B. Baum and W.D. Smith. A Bayesian approach to relevance in game playing. *Artificial Intelligence*, 97(1/2):195–, 1997.

[3] D.F. Beal. A generalized quiescence search algorithm. *Artificial Intelligence*, 43:85–98, 1990.

[4] H. Berliner. AI's greatest trends and controversies. *IEEE Intelligent Systems*, page 9, January/February 2000.

[5] M. Buro. Statistical feature combination for the evaluation of game positions. *JAIR*, 3:373–382, 1995 [2].

[6] M. Buro. ProbCut: An effective selective extension of the alpha-beta algorithm. *ICCA Journal*, 18(2):71–76, 1995 [2].

[7] M. Buro. The Othello match of the year: Takeshi Murakami vs. Logistello. *ICCA Journal*, 20(3):189–193, 1997 [2].

[8] M. Buro. From simple features to sophisticated evaluation functions. In H.J. van den Herik and H. Iida, editors, *Computers and Games, Proceedings of CG98, LNCS 1558*, pages 126–145. Springer Verlag, 1999 [2].

[9] M. Buro. Toward opening book learning. *ICCA Journal*, 22(2):98–102, 1999 [2].

[10] M. Buro. Experiments with Multi-Probcut and a new high-quality evaluation function for Othello. In H.J. van den Herik and H. Iida, editors, *Games in AI Research, Proceedings of a workshop on game-tree search held in 1997 at NECI in Princeton, NJ*, pages 77–96. Universiteit Maastricht, The Netherlands, 2000 [2].

[11] J.R. Koza et al. *Genetic Programming III: Darwinian Invention and Problem Solving.* Morgan Kaufmann, San Francisco, 1999.

[12] S.J. Hanson. Meiosis networks. *Advances in Neural Information Processing Systems*, pages 553–541, 1990.

[13] K.F. Lee and S. Mahajan. The development of a World-class Othello program. *Artificial Intelligence*, 43:21–36, 1990.

[14] R.A. Levinson and R. Snyder. Adaptive pattern–oriented chess. In L. Birnbaum and G. Collins, editors, *Proceedings of the 8th International Workshop on Machine Learning*, pages 85–89, 1991.

[15] D.A. McAllester. Conspiracy numbers for minmax search. *Artificial Intelligence*, 35:287–310, 1988.

[16] A.J. Palay. *Searching with Probabilities.* Pitman Publishing, 1985.

[17] R.L. Rivest. Game tree searching by minmax approximation. *Artificial Intelligence*, 34(1):77–96, 1988.

[18] P.S. Rosenbloom. A World–championship–level Othello program. *Artificial Intelligence*, 19:279–320, 1982.

[19] A.L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):211–229, 1959.

[20] G. Tesauro. Temporal difference learning and TD–Gammon. *Communications of the ACM*, 38(3):58–68, 1995.

[21] P.E. Utgoff. Constructive function approximation. Technical Report 97–4, Univ. of Mass., 1997.

[22] M. Wynne-Jones. Node splitting: A constructive algorithm for feed–forward neural networks. *Neural Computing and Applications*, 1(1):17–22, 1993.

[2] The author's articles can be downloaded for personal use from
http://www.neci.nj.nec.com/homepages/mic/publications.html