From Simple Features to Sophisticated Evaluation Functions

Michael Buro

NEC Research Institute
4 Independence Way
Princeton NJ 08540, USA

Abstract. This paper discusses a practical framework for the semi–automatic construction of evaluation functions for games. Based on a structured evaluation function representation, a procedure for exploring the feature space is presented that is able to discover new features in a computational feasible way. Besides the theoretical aspects, related practical issues such as the generation of training positions, feature selection, and weight fitting in large linear systems are discussed. Finally, we present experimental results for Othello, which demonstrate the potential of the described approach.

Keywords: automatic feature construction, GLEM, Othello

1 Introduction

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 on particular games. In order to simplify the construction task, the notion of *evaluation features* was introduced. The underlying assumption is that there exist reasonable approximations of the perfect evaluation function in the form of combinations of a few distinct numerical properties of the position — called features. Given this, evaluation functions can be constructed in two phases by 1) selecting features and 2) combining them.

Selecting features is one of the most important and difficult sub–tasks in the construction of a game playing program. It requires both domain specific knowledge and programming skills because of the well known tradeoff between speed and knowledge in game–tree search. A couple of years ago, the authors of the best game playing programs still picked not only features but also their weights in course of a tedious optimization process. This is somewhat surprising, since already in [7] Samuel proposed ways for automatically tuning weights. While selecting features is difficult for a machine, fitting even a large number of weights given a set of training positions is not. Research focused on the latter topic produced TD–Gammon, a world–class backgammon–program [8, 9], and contributed to Deep Blue's victory over Kasparov in 1997 [4].

In this article we go a step further towards the ultimate goal of automatic evaluation function construction. First, a generalized linear evaluation model is presented. It restricts evaluation features to boolean combinations of given atomic functions. The model parameters can be tailored such that an automatic feature space exploration becomes feasible. The following sections cover all aspects of evaluation function construction — from generating training positions over feature selection to weight estimation — with respect to the new model and emphasis on efficient implementation. Finally, we show how the presented techniques can be applied to the game of Othello and discuss the new approach with regard to related work.

2 Evaluation Model

We first give a definition of the evaluation model we are proposing and discuss its properties. In what follows, $\mathcal{P}$ denotes the set of all legal game positions[1], and $\mathrm{I\!R}$ the set of real numbers. Let A be a finite set of integer valued — so called *atomic* — features and $R_A := \{\ (f(\cdot) = k) \mid f \in A,\ k \text{ is an integer}\}$ the set of relations over A that compare feature values with integer constants. *Configurations* are conjunctions of relations in R_A. For a position $p \in \mathcal{P}$ and a configuration $c = r_1 \wedge \ldots \wedge r_l$ we define

$$\mathrm{val}(c(p)) := \begin{cases} 1, & \text{if } r_1(p) \wedge \ldots \wedge r_l(p) = \text{true} \\ 0, & \text{otherwise} \end{cases}.$$

A configuration c is called *active* in a position p, iff $c(p) = $ true.

With this notation we can now define the *Generalized Linear Evaluation Model* — GLEM($\mathcal{P}, A, g$) for short. Evaluation functions in this model have the following form:

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

where $c_1, ..., c_n$ are configurations over R_A, $w_1, ..., w_n \in \mathrm{I\!R}$ are weights, and $g : \mathrm{I\!R} \to \mathrm{I\!R}$ is an increasing and differentiable link function.

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 positions $\big((p_i, r_i) \mid i = 1 \ldots N\big)$, 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 model has several desirable properties:

[1] W.l.o.g. it is assumed that game positions in $\mathcal{P}$ are normalized in such a way that a fixed player is to move.

- Atomic features are the building blocks of more sophisticated ones. This, in principle, allows the automated discovery of new important features by systematic combination.
- If necessary, complex features can be added to A. Thus, "atomic" is not necessarily a synonym for "simple".
- When evaluating a position, features are combined linearly. This keeps the time overhead low. Actually, not even a multiplication with the weight is necessary since $\mathrm{val}(c_i(p))$ is either 0 or 1.
- Non–linear effects can be approximated by using configurations that consist of several relations.
- In order to deal with saturation an increasing non–linear link function, such as $g(x) = 1/(1 + exp(-x))$, can be used without increasing the run time during minimax search. There is no need to compute g, because $g(x_1) > g(x_2) \iff x_1 > x_2$.
- The simple linear core of the evaluation function allows an efficient approximation of optimal weights, even for large systems. In the application reported later, more than a million weights were fitted to a training set consisting of eleven million scored positions in a reasonable period of time.

At this point GLEM should be moved into the right perspective: in the stated form it is neither a new revolutionary evaluation approach, nor does it ease the task of automatic evaluation function exploration. This is because the model is built upon well known linear evaluation functions and does not impose a severe restriction on the structure of functions it includes. E.g., for any atomic feature set A, which is capable of distinguishing any two different positions (including game history if the game result depends on it) via conjunctions over R_A, GLEM covers *all* evaluation functions over $\mathcal{P}$. A trivial example for such a complete atomic feature set for board games without position repetition is

$$A = \left\{ f_s \mid \begin{array}{l} f_s(p) = \text{contents of square } s \text{ in position } p, \\ s \text{ is a square} \end{array} \right\},$$

where the contents of a square is considered to be an integer value.

However, GLEM allows one to define a hierarchy of submodels in a natural way, which reflects different levels of computational complexity and the expressive power of the covered evaluation functions. By restricting the size of A, the number of configurations, or their structure, an automated search for new features becomes feasible. In the application discussed later, evaluation functions based on GLEM outperformed the best known functions so far. In this respect, GLEM breaks new ground.

Good evaluation functions accurately estimate the winning chances in positions visited during game–tree search and are optimized for speed. Therefore, the following topics have to be borne in mind when using scored training positions for tuning configuration weights:

- The training positions have to be representative of the positions that will be evaluated later in actual game–tree search.

– Training positions must be scored accurately.
– The selected configurations and their combination must have the expressive power to explain the data reasonably well while avoiding over–fitting. Given the flat evaluation function representation in GLEM, meeting this condition may require a large number of configurations. Their automatic construction is therefore of great interest.
– Evaluation speed is important.
– While computing weights is an off–line process, its memory and time consumption should still be subject to optimization. The reason is that in the feature selection phase usually many evaluation function versions have to be compared. Moreover, without optimization the current solver might not be able to handle the number of features one would like to use.

In the following sections these topics are discussed in detail in the context of GLEM.

3 Training Positions

A theory of how to generate good training sets in the context of evaluation function tuning has not been developed yet. In this section practical ideas are discussed which may become the seed for further investigations.

Training positions can be generated and scored in several ways. If the considered game has a long tradition and is quite popular, many games may be available in electronic format. The simplest scoring procedure assigns the final game result (depending on the side to move) to all positions occurring in a game. Obviously, this ad hoc procedure has limitations, since it does not ensure accurate scoring. Selecting games between good players alleviates this problem. But this approach leads to high–quality games, in which hardly any catastrophe takes place, such as losing material in chess or a corner in Othello without compensation. The reason for this is obvious: good players know the important evaluation features and keep them mostly balanced in their games. What we (and machines) can learn from such games are the finer points of play, which make the difference between good and the best players. However, an evaluation function must also be aware of the most important features. Thus, our training set should also contain games in which at one point a player makes a serious mistake that is rigorously exploited by the opponent. In summary, a reasonable strategy for generating training positions from a game database is to select games played by at least one good player and to score game positions according to the final game result. This procedure is efficient and its output can serve as the basis for tuning the first evaluation function version.

Besides the still present potential mis–scoring problem, the question arises, whether the so generated training set is representative to positions encountered in game–tree search. This question is of importance, since the weight fit for a linear evaluation function is influenced by the correlation among features in the training set. The answer obviously depends on the type of game–tree search we are conducting: in a highly selective search evaluated positions are in the vicinity of principal variations, whereas in brute–force searches many ridiculous positions are evaluated, which one would never encounter in actual games. It seems natural to let the search algorithm generate the training positions by itself. For instance, starting searches with positions from played games, a random subset of evaluated positions can be saved in a file and serve as the training set after scoring. In this way, the generated positions are surely a representative sample of the positions encountered in game–tree searches. It remains to assign accurate scores to the positions. This task can be accomplished again by game–tree searches, which normally return more reliable results than the evaluation function itself. In particular, in many games endgame positions can be evaluated perfectly — or at least more accurately than middle–game or opening positions — in a reasonable amount of time. In this case, a game–stage dependent evaluation function can be improved iteratively by first tuning the endgame weights. Thereafter, training positions from the previous game stage are evaluated by a game–tree search, which utilizes the just tuned evaluation function, and so on. The next step would be to generate even positions and those with a narrow advantage for one side. Similar to considering games between good players mentioned above, these positions are useful for tuning weights of minor features or revealing possible tradeoffs between major features (e.g. material vs. king safety in chess or corner possession vs. mobility in Othello).

If training positions are selected randomly during minimax–based searches, one soon discovers that the winning chance in such positions is biased towards the player to move. This phenomenon is easy to explain, given the fact that in typical positions the majority of searched moves lose. Its undesirable effect on fitted weights is an artificial bonus for the player to move. This, in turn, leads to unstable evaluations, which compromise comparing evaluations backed–up from depths of odd difference during selective search. Because the proposed generation procedure labels positions with search results, a simple cure for this problem is to add the principal variation successor positions to the training set after labelling them with the negated search result.

4 Selecting Configurations

GLEM proposes a new perspective on how to look at evaluation features. In the classical approach a couple of complex features are combined linearly. Weights were mostly hand–tuned. Later, the study of neural networks opened up a practical way of combining features non–linearly. Application of the well known gradient descent procedure (in this context called "back–propagation") makes it possible to automatically tune a large number of network parameters. A prominent and very successful example is Tesauro's backgammon network which, in its strongest version, makes use of hand–crafted features in addition to a raw board representation. GLEM uses a different approach. Instead of modelling non–linear effects by applying parameterized analytical functions to features, GLEM handles non–linearities *directly* by assigning values to boolean feature combinations, called configurations. In this way, distinct cases can be handled naturally, without the detour over non–linear analytical functions. The design of neural networks corresponds to configuration selection in GLEM, which is the topic of this section. After stating basic requirements for the atomic features, we will present an algorithm for generating configurations by analyzing training positions, and discuss several optimizations.

4.1 Atomic Features

Atomic features are the building blocks for configurations. As the scope of automatic configuration selection is limited by its time and space complexity, choosing the right abstraction level for atomic features is crucial. In Othello, configurations based upon the raw board representation are sufficient for building good evaluation functions — as we shall see later. The reason is that many relevant features in this game can be expressed by local board configurations of small cardinality. Other games may require a greater abstraction level. For instance, the relation "piece A attacks piece B" in chess has a long description length when using raw board representation languages. Since many important features, such as forks and pins, are based on those attack features, they certainly should be included in the atomic feature set. In general, candidates for atomic features are common parts of relevant features, that — combined in novel ways — may lead to new important features. Obviously, this selection task is beyond current program abilities.

Not all atomic features have to be useful for building other features. Limitations of the configuration generator may suggest the inclusion of complex features that can not be expressed or well approximated by restricted combinations of other members of the atomic feature set.

Moreover, GLEM generalizes the classical use of features — $w \cdot f(p)$ — because $(w \cdot k)$ in

$$w \cdot f(p) = \sum_k (w \cdot k) \cdot \mathrm{val}(f(p) = k).$$

specializes the weight of $\mathrm{val}(f(p) = k)$. This generalization is only meaningful if f has a small range. In case one likes to incorporate a feature f having a large range, GLEM can be easily extended by allowing summation terms of the form $w \cdot f(p)$.

4.2 Generating Configurations

In a balanced evaluation function design the number of features can be increased up to a point where either 1) adding additional knowledge is compensated for by a decreased evaluation speed or 2) over–fitting becomes a problem. Since configurations can be computed quickly, once the atomic features have been evaluated, GLEM encourages to use many configurations rather than a few complex features. Our chief concern is therefore over–fitting.

We will first present an algorithm for generating a configuration set that does not suffer from over–fitting. Thereafter, we will discuss how to deal with a possibly unacceptably long run time for the configuration generator, for weight fitting, or for the configuration value look–up during game–tree search.

Configurations have to cover positions that occur in game–tree search while avoiding over–fitting when optimizing weights. Both requirements can be met by using a large set of training positions — generated as described in the previous section — and selecting configurations that match a sufficiently large number of these positions. Fig. 1 shows a straight forward algorithm for this task. Given a set of atomic features A, training positions E, and a minimal match count n, it computes all *valid* configurations over A that occur in at least n positions in E. Beginning with all valid configurations of length one, the algorithm iteratively builds larger configurations by specializing previously generated configurations, until the matches count drops below n. The algorithm certainly halts, since the set of valid configurations is finite. Its correctness can be shown by induction using the fact, that for $k > 1$, valid configurations of length k have valid sub-configurations of length $k - 1$.

The run time of the algorithm is $O(|C| \cdot |R_A|^2 \cdot |E|)$, where C is the computed set of valid configurations and E the set of training examples. The most time–consuming part is computing the match counts in the inner loop. Since in the beginning the number of checked configurations grows exponentially in

Function GenConf

Input: atomic feature set A, training position set E, minimal match count n
Output: configurations over A that are active in at least n positions of E

```
    R := {{f(·) = k} | f ∈ A, k ∈ range(f), #match({f(·) = k}, E) ≥ n}
    C := R     ; collects all valid configurations
    N := R     ; set of configurations created in previous iteration

    while N ≠ ∅ do
      M := ∅                     ; set of valid configurations in current iteration
(*)   foreach c ∈ N, d ∈ R do
        e := c ∪ {d}             ; specialize configuration c
        if #match(e, E) ≥ n then
          M := M ∪ {e}           ; append if valid
        endif
      endfor
      N := M                     ; next configurations to specialize
      C := C ∪ N                 ; add valid configurations
    endwhile
    return C
```

Fig. 1. Pseudo code for generating the set of configurations that occur in at least n training positions. The function iteratively specializes configurations, which are implemented as sets of relations, until the number of matching positions ($\#\text{match}(e, E)$) drops below n.

each iteration, it is crucial to optimize the match computations, especially if the number of positions is large. The following optimizations speed up a naive implementation considerably:

- Due to the commutativity of $\wedge$, valid configurations of length k may have several valid subconfigurations of length $k-1$. This observation suggests that we should check whether a given specialization has been tested before in the current iteration, in order to avoid repeated match computations. An even better solution is to generate specializations in an ordered fashion by defining a total order over R and replacing line (*) by

$$\textbf{foreach } c \in N,\ d \in R \text{ with } d > \max_{d' \in c} d' \textbf{ do}$$

  It is not hard to show that after applying this time–saving modification the algorithm still generates all valid configurations.
- A naive algorithm for deciding $\#\text{match}(e, E) \geq n$ evaluates the relations in e for every member of E. The computation time of this algorithm can be reduced by preprocessing and parallelizing computations. The idea is to compute, for each $r \in R$, a sequence of bits $(b_i)_{i=1}^{\#E}$ defined by $b_i := \text{val}(r(p_i))$, where $p_i \in E$ is the i–th training position. After this preprocessing step, the actual features and positions are no longer needed. The match count computation reduces to and–combining the bit sequences of the involved relations and counting set bits in the result sequence. Modern CPUs allow

Function MatchHeuristic

<u>Input:</u> configuration e, chunk size s, random partition $E_1, ..., E_m$ of position set E as described in the text, confidence level $t > 0$
<u>Output:</u> true, if $\#\text{match}(e, E) \geq n$ is likely; false, otherwise

```
q := n/#E                          ; match count fraction aimed for
d := 0                             ; number of elements checked
u := 0                             ; current match count
for i := 1 to m − 1 do
    u := u+#match(e, E_i)          ; update counts
    d := d + s
    if u ≥ dq + t√(dq(1 − q)) then
      return true                  ; #match(e, E) ≥ n is likely
    endif
    if u < dq − t√(dq(1 − q)) then
      return false                 ; #match(e, E) < n is likely
    endif
endfor
return u+#match(e, E_m) ≥ n
```

Fig. 2. A fast procedure for testing the hypothesis $\#\text{match}(e, E) \geq n$

a very efficient implementation of the and–part by handling 32 or even 64 bits in parallel. Iterating $x := x \wedge (x - 1)$, which clears the rightmost one in the binary representation of x, allows us to count set bits quickly. In this application table–based techniques for counting bits are inferior because the number of set bits is decreasing rapidly due to specialization.

– Replacing the condition $\#\text{match}(e, E) \geq n$ by a sequential statistical test procedure speeds up the computation further. This optimization can be motivated by an intuitive example: if among the first 100 randomly selected bits of 1000 there is only a single one, it is very unlikely that the total number of ones exceeds 500. More formally, we propose the following heuristic function, which quickly checks whether $\#\text{match}(e, E) \geq n$ holds with a prescribed likelihood. In a preprocessing step, E is randomly partitioned into chunks $E_1, ..., E_m$ of size s (E_m might have less elements). For a given configuration e, the function then iteratively computes the match counts for increasing subsets beginning with E_1. If the match count fraction at one point significantly differs from the one we aim for, the function returns the likely truth value of $\#\text{match}(e, E) \geq n$ early. The pseudo code implementation shown in Fig. 2 makes use of the fact that the expected number of ones in a sequence of d randomly generated bits is dq, if $\text{Prob}\{1\} = q$, while its standard deviation is $\sqrt{dq(1-q)}$. The behaviour of this function is controlled by confidence level t. For large values of t, hardly any break condition will be met — the function will be slow, and almost always return the correct result. If t is small, the function is quick, but it also returns unreliable results. Experiments can tell how to choose t depending on the speed/reliability one likes to achieve.

4.3 Finding Active Configurations

During weight fitting and position evaluation the set of active configurations has to be computed quickly for a large number of positions. For this purpose, we represent the set of all configurations over R_A by a DAG G. Nodes in G correspond to configurations, and arcs mark direct specializations. A detailed example is shown in Fig. 3a. The just described selection algorithm computes all configurations that occur at least n–times in a set of training positions. This set of valid configurations induces a sub–DAG G' of G. Given a position, all active configurations can be found by a depth–first search in G' starting at its root. During search, all visited configurations are marked and their active status is determined. The search stops in nodes that have been visited before or have been found inactive. This algorithm quickly finds all active configurations. However, the only relevant active configurations for evaluation purposes are those without active specializations, because generalizations are redundant. It is easy to extend the described algorithm accordingly by restricting its output to leaves of the active configuration sub–DAG. Fig. 4 illustrates the entire procedure.

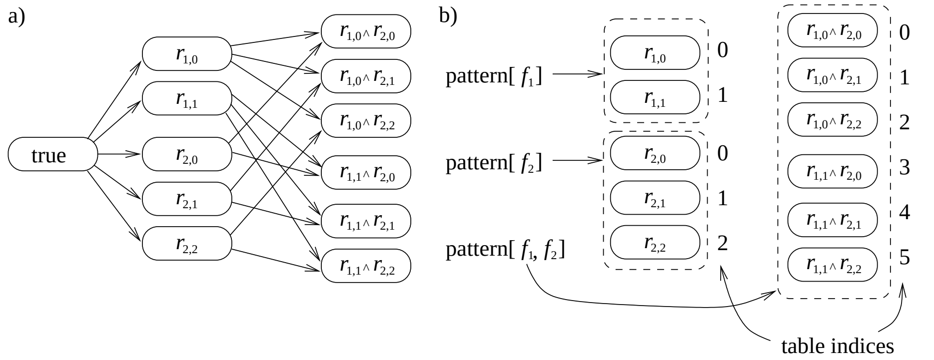

Fig. 3. a) Configuration DAG for two features f_1, f_2 with $\text{range}(f_1) = \{0, 1\}$ and $\text{range}(f_2) = \{0, 1, 2\}$. $r_{i,k}$ denotes the relation $f_i(\cdot) = k$. b) Configurations belonging to patterns over f_1 and f_2.

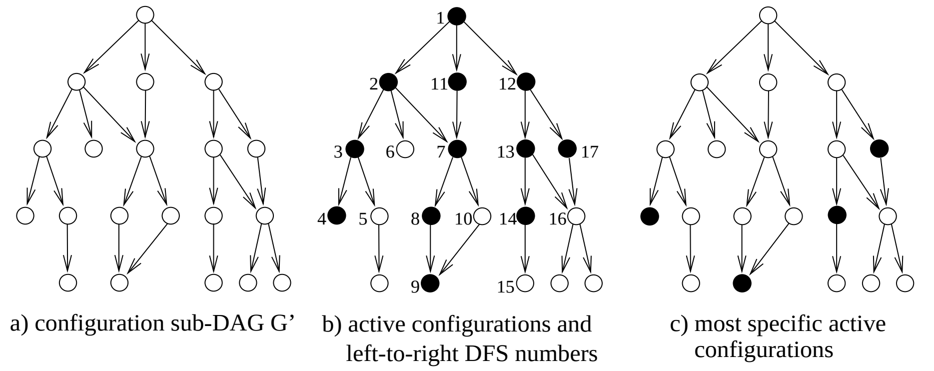

Fig. 4. Finding the most specific active configurations by depth–first search in the configuration DAG

4.4 Reducing Complexity: Patterns

So far, our focus has been on efficient ways for generating configurations and computing active configurations. Despite the optimization efforts, GenConf may still not be able to generate all valid configurations due to time or space limitations. Furthermore, a large number of generated configurations might prevent an efficient position evaluation, because too many configurations are active, or the configuration data needs too much memory.

One solution to these problems is to increase the minimal match count n, until the number of generated configurations is manageable. This approach, however, narrows the evaluation function's view by focusing it on the most common phenomena. A compromise is to generate all valid configurations choosing n high enough to avoid over–fitting, and to reduce their number afterwards by looking at their statistical significance with regard to winning chance prediction.[2] Another option for reducing the number of configurations is to limit their size or to choose subsets of the atomic feature set as the base for generating configurations.

Finally, considering sets of mutual exclusive configurations helps to reduce the number of active configurations in order to speed up the evaluation considerably. Let G be the complete configuration DAG for $\{f_1, ..., f_m\} \subset A$ (Fig. 3a), and let $r_{\min}$ and $r_{\max}$ denote the minimum/maximum range cardinality of the features. Then the number of nodes in G is bounded by $(1 + r_{\min})^m$ and $(1 + r_{\max})^m$, and for any position the number of active configurations is 2^m. [3] Thus, in case of complete configuration DAGs the DFS algorithm presented in the last subsection seems to waste time by searching a large number of nodes before it eventually returns the single active configuration we are interested in. This observation motivates looking for a more efficient data structure. For $\{f_1, ..., f_m\} \subset A$ we collect all possible most specific configurations in a set called $\text{pattern}[f_1, ..., f_m]$, i.e.

$$\text{pattern}[f_1, ..., f_m] := \{r_{1,l_1} \wedge ... \wedge r_{m,l_m} \mid r_{i,l_i} = (f_i(\cdot) = l_i),\ l_i \in \text{range}(f_i)\}$$

Configurations in $\text{pattern}[f_1, ..., f_m]$ correspond to leaves of the complete configuration DAG (Fig. 3b). Data related to these configurations can therefore be stored in a table addressed by feature values. For instance, in Fig. 3b the table index for $\text{pattern}[f_1, f_2]$ with regard to position p is simply $3 \cdot f_1(p) + f_2(p)$. Checking whether a pattern configuration is valid only requires incrementing a match counter stored in a table whenever a configuration is active, and comparing the result with the minimal match count. Detecting whether a pattern configuration is active during weight fitting or positional evaluation is a matter of a fast index computation and one table access. Incremental updates of only those indices which are influenced by moves speeds up game–tree search further. In summary, the flat table is the data structure of choice for storing information regarding small and medium sized complete configuration sets. The fast access encourages to restrict configuration sets to patterns.

Large patterns require a more memory efficient representation. In order to avoid over–fitting, we are still only interested in configurations that match several training positions. Consequently, large patterns are sparse. Fig. 5 outlines a very fast and — to our knowledge — novel technique for accessing sparse data which trades memory for speed. It is based on representing valid configurations as index tuples (i_1, i_2). For a given position and pattern, i_1 and i_2 are computed by splitting the pattern's feature set into two parts and performing the index calculations described above separately for each subset. Both indices are then used for accessing a hash–table, in which data regarding configuration (i_1, i_2)

[2] The general problem of deciding the relevance of variables in a multivariate regression model in advance is hard. Nevertheless, simple statistics like the feature's correlation with the training position scores can serve as a reasonable first approximation.

[3] These numbers can be derived by adding lower/upper bounds for the number of nodes/active configurations for each depth and applying the identity $\sum_{i=0}^{m} \binom{m}{i} x^i = (1 + x)^m$.

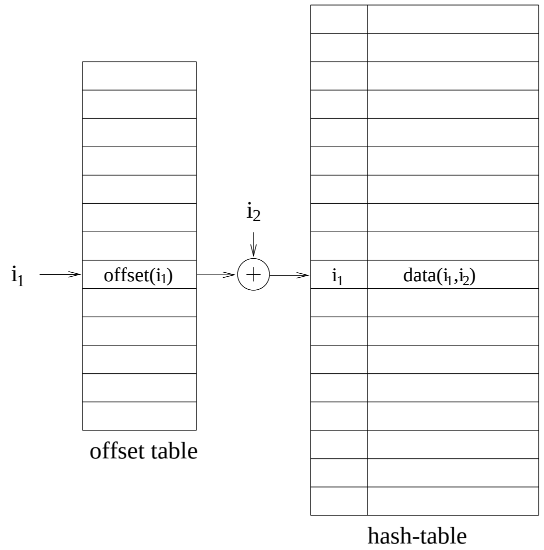

Fig. 5. Fast sparse data access. Data regarding a configuration represented by two indices i_1 and i_2 can be accessed quickly in two steps.

is stored. First, an offset is looked–up in a table using index i_1. Then, this offset, incremented by i_2, is used to access the hash–table. For the algorithm to be correct, 1) unique hash–table entries have to be assigned to valid index tuples, and 2) invalid index tuples must be detected. The first condition can be met by choosing suitable offsets and a sufficiently large hash–table. In practice, the following greedy algorithm for constructing collision–free hash–tables has produced reasonable results: beginning with the most frequent i_1–values, offsets are assigned to them in first–fit manner. That is, whenever a collision occurs when attempting to occupy the hash–table entry $\text{offset}(i_1) + i_2$, all i_1 entries claimed so far are erased and $\text{offset}(i_1)$ is incremented before restarting. The hash–table size must be greater than the sum of the maximal offset and maximal *possible* value of i_2, in order to avoid accesses beyond table end. A simple way for meeting condition 2) is to add the lock i_1 to hash entries for all valid tuples (i_1, i_2) and to reject tuples (i_1, i_2), for which the lock stored in the accessed hash entry does not match i_1. Locks of unused hash entries must be initialized with a value different from any *possible* i_1 (e.g. -1). Finally, offsets for all i_1, which are not the first component of any valid index tuple, can be safely set to 0, since all locks in the hash–table are different from those i_1 values.

Patterns may outperform configuration sets constructed by GenConf due to a much faster generation and evaluation of configurations. However, patterns suffer from their limited scope because patterns may miss essential generalizations. This observation suggests building a hierarchy of patterns in order to quickly cover both general and specific position aspects. Since this approach also increases the evaluation time, experiments have to tell, which is the better strategy for a given application.

5 Weight Fitting

The previous sections discussed the generation of scored training positions and the selection of configurations. In order to conclude the evaluation function construction, we must show how to assign weights to configurations.

If the number of weights is large or non–linear models are used, direct weight computation is no longer feasible. Instead, iterative methods have to be used for weight fitting, which are usually based on variations of the gradient decent procedure. In each step, this procedure updates the current weight vector in direction of the negated gradient of the error function. If features are highly correlated, this simple algorithm is known to converge slowly. Faster conjugate gradient algorithms have been developed [6], that do not suffer from this problem. However, because the basic algorithm works sufficiently well in practice and is easier to implement, its application will be discussed in more detail in the remainder of this section.

5.1 Basic Considerations

In games, the purpose of evaluation functions is to estimate the winning chance for the player to move. This goal can be accomplished literally by constructing functions that map positions into $[0, 1]$. Alternatively, the game may provide a numerical scoring of terminal positions reflecting the win size. In this case, a reasonable evaluation objective is to estimate the final game score. In either case, experiments should be conducted to find a suitable link function g. The most commonly used candidates are the identity function and sigmoid functions of the form $g(x) = 2C/(1+exp(-x))-C$. For instance, for modeling the winning chance an S–shaped link function $g : \mathbb{R} \to [0, 1]$ can be used in order to deal with saturation. In this regard, $g(x) = 1/(1 + exp(-x))$ is of special interest, because the weight fitting process benefits from a quickly computable derivative of g, which in this case is $g(x)(1 - g(x))$. A straight forward scoring scheme for terminal positions in this model assigns 0.9 to won positions, 0.5 to draws, and 0.1 to lost positions for the player to move. It is important to realize that an optimal weight vector may not exist if the extreme values 1.0 and 0.0 are chosen.

Given a sequence of scored training positions $((p_i, r_i))_{i=1}^N$ the objective is to find a weight vector $\boldsymbol{w}_0$ which minimizes the error function

$$E(\boldsymbol{w}) = \frac{1}{N} \sum_{k=1}^{N} \Delta_k(\boldsymbol{w})^2,$$

where

$$\Delta_k(\boldsymbol{w}) := r_k - g\Big(\sum_{i=1}^{n} w_i h_{i,k}\Big) \text{ and } h_{i,k} := \mathrm{val}(c_i(p_k)).$$

Starting with an initial guess $\boldsymbol{w}^{(0)}$, in each step the basic gradient descent procedure updates the weight vector according to

$$\boldsymbol{\delta}^{(t)} = -\alpha \cdot (\mathbf{grad}_{\boldsymbol{w}} E)(\boldsymbol{w}^{(t)})\ ^{4}$$

$$\boldsymbol{w}^{(t+1)} = \boldsymbol{w}^{(t)} + \boldsymbol{\delta}^{(t)}.$$

$\alpha > 0$ is the step size 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.

In this application, the partial derivatives have a simple form due to GLEM's flat evaluation structure:

$$\frac{\partial E}{\partial w_i}(\boldsymbol{w}) = -\frac{2}{N} \sum_{k=1}^{N} g'\Big(\sum_{i=1}^{n} w_i h_{i,k}\Big) \Delta_k(\boldsymbol{w}) h_{i,k}. \tag{2}$$

If g is the identity function, this expression reduces to

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

Thus, steepest descent updates for all weights can be computed efficiently in a single pass through the training data. It is worth noting, that the computation of (2) can be arranged in such a way that its run time depends on the number of $h_{i,k}$ different from 0, rather than on N. Especially when using patterns, the savings thus achieved are significant.

Since the configuration match count may vary by large factors, the described update step changes weights at very different speeds. This is undesirable, because at one point the iteration process has to be stopped, and by then, weights of rare but important configurations might not have reached a proper level yet. A simple way to deal with this problem is to normalize the updates by dividing the sum by the number of $h_{i,k} \neq 0$ instead of N.

5.2 Position Type Dependent Weights

The evaluation of configurations may depend on the game stage or, more generally, on the particular type of the position. For instance, centralizing the king in chess openings is considered suicide, whereas his activation is crucial in many endgames. It may therefore be worthwhile to partition the training set according to position type, and to select configurations and fit weights separately for each set. In order to avoid big evaluation jumps when crossing type boundaries, which can cause undesired artifacts in game–tree search, it is helpful to define fine grained position types and to smooth evaluations across adjacent types. Fitting

[4] adding $\beta \cdot \boldsymbol{\delta}^{(t-1)}$ — known as "momentum" — can improve the convergence in case of correlated features.

weights for many position types, however, requires a large number of training positions, provided the minimal match count is maintained in order to eliminate over–fitting. Globally lowering the match count is therefore not an option. Instead, a more local view can help to reduce the number of needed positions. One suggestion when fitting weights for a particular position type, is to consider the training positions from adjacent types as well. This method increases the number of positions for any single position type and weights are smoothed automatically. The second option is to fit position type dependent weights in a more flexible manner. For this purpose, valid configurations are generated by considering *all* training positions. The weight fitting process then decides, how to compute the configuration weights separately for each type of position. For any type, for which the particular configuration match count is sufficiently high (say ≥ 20), it is safe to fit the according weight as described in the previous subsection. If the count is small (say ≤ 4), over–fitting is likely and the configuration should be treated as if there is no information available, i.e. the weight is set to 0. Cases in between can be handled by merging adjacent position types, until the total match number allows a robust weight fit. Here, the alternatives are to have only a single weight for all involved types or, if there are enough positions available, to fit a parameterized weight model. An example for such a model is $w(k) = a \cdot k + b,\ k_0 \leq k \leq k_1$, which states a linear relationship between the weight and the position type k — coded as an integer — in $[k_0, ..., k_1]$. Of course, this kind of model is only meaningful for position types that can be totally ordered, such as opening, middle–game, and endgame. Incorporating the update of parameters a and b in the gradient descent procedure is not hard.

This technique allows a flexible and robust fitting of position type dependent weights. After generating training positions and selecting configurations, this concludes the evaluation function construction.

6 Application: 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, the application of the techniques discussed has contributed to the considerable playing strength of this program. LOGISTELLO is able to beat the best human Othello players handily, even when running only on ordinary hardware [2]. The details of LOGISTELLO's evaluation function already have been discussed in [1]. We will therefore only give a short overview and concentrate on its recent improvement, which is based on the sparse pattern approach presented above.

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.

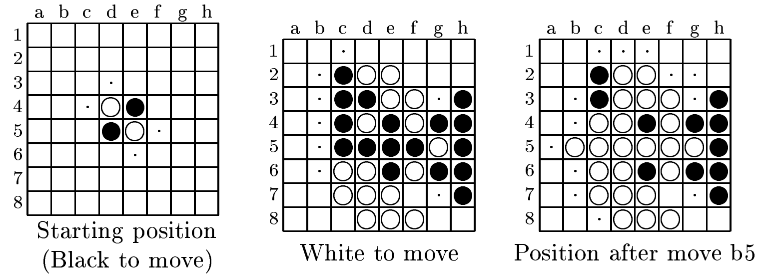

Fig. 6. Example positions. Legal moves are marked with a dot.

The most important concepts 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.

In [1] it has been shown, that all of these features can be quickly approximated by pattern configurations built upon a raw board representation. The chosen patterns are shown in Fig. 7. Horizontal, vertical, and diagonal lines of length ≥ 4 are included for covering mobility. The remaining patterns deal with the important corner regions and edges. The evaluation function distinguishes 13 game stages, depending on the number of discs on the board. Applying the techniques described in the previous sections, about eleven million scored training positions were generated to fit approximately 1.5 million weights. This figure takes weight sharing among symmetrical configurations into account. Starting with $\boldsymbol{w}^{(0)} = \mathbf{0}$, the weight fitting procedure took a Pentium II/333 CPU about 30 hours to reach an acceptable accuracy level after 250 iterations. Equipped with an evaluation function very similar to that we have just described, LOGISTELLO beat the human Othello World–champion 6–0 in August 1997 [2]. After four years of successful tournament play, LOGISTELLO ended its career in October 1997 with a straight 22–win victory in its last computer Othello tournament.

Recently, the incorporation of larger patterns has improved the evaluation performance. In the current implementation, configuration weights are represented as 16 bit integers. Storing weights for 10–square patterns in 13 flat tables thus requires $3^{10} \cdot 2 \cdot 13 \approx 1.5$ million bytes. Using the same approach for storing weights for much larger patterns is therefore out of the question. The first experiments with several sparse data access schemes based on binary search were

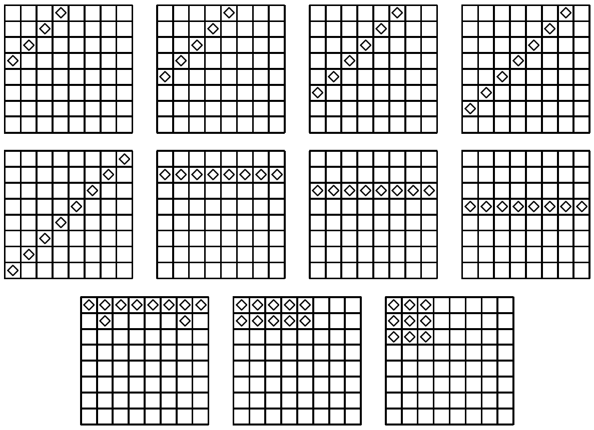

Fig. 7. LOGISTELLO's previous pattern set. Patterns that can be obtained by rotating and mirroring the board have been omitted. Each diamond represents an atomic feature f with range $\{0, 1, 2\}$. $f(p)$ is defined by the particular square contents (e.g. white disc $\mapsto 0$, empty $\mapsto 1$, black disc $\mapsto 2$).

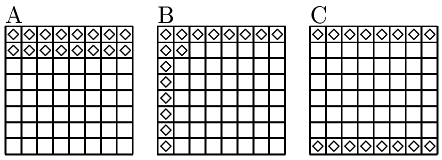

Fig. 8. Large patterns tested. For each of these patterns the simplified pattern version of GenConf generated about 88,000 valid configurations (#E $\approx$ 11 million, $n = 75$). All configuration sets fit in hash–tables with about 310 thousand entries.

disappointing. Increasing the program's knowledge by adding the patterns shown in Fig. 8 could not compensate for a slowdown of about 45%. Only after utilizing the fast hash–table access scheme and adding just one of the three features, the program achieved its best performance so far. Table 1 summarizes the results of all tournaments that have been played to evaluate each version. All games were played by brute–force versions of LOGISTELLO running on Pentium II/333 PCs. On this hardware LOGISTELLO achieves a middle–game speed of approximately 270K nodes/sec when the patterns shown in Fig. 7 are used. This speed enables the program to look 12–14 ply ahead in the opening and middle part of ten minutes games.

The patterns presented in Fig. 8 were chosen based on both game and evaluation speed considerations. Human players frequently make use of their abilities to evaluate large disc formations which are not covered by the basic patterns. Of special interest are edge interactions and 2×8–corner configurations. On the other hand, it is preferable to add patterns for which the index computation can make use of already determined indices. The chosen 16–square patterns meet this preference. Nevertheless, the results show, that the combined knowledge coded

Table 1. Tournament results. LOGISTELLO using the basic patterns played 434–game tournaments against several versions that — in addition — employed the large patterns shown in Fig. 8. The results indicate that speed matters. The strongest versions are those that only use either pattern A or B. They beat the previous version significantly, although they are 11% slower. When playing at equal strength the best version only needs to search about 2/3 of the nodes — as the results of the time–handicap tournaments indicate.

opponent	time/game (minutes)	#nodes (fraction)	opponent results wins	draws	losses	winning percentage
A	10-10	0.89	213	58	163	55.8
B	10-10	0.89	211	60	163	55.5
AB	10-10	0.83	203	60	171	53.7
ABC	10-10	0.8	211	49	174	54.3
A	6-10	0.51	172	59	203	46.4
A	7-10	0.62	183	55	196	48.5
A	8-10	0.71	195	63	176	52.2

in the new patterns does not compensate for the speed drop. This finding indicates that a significant improvement of a sequential program may not be possible by adding further patterns based on the raw board representation. However, a more effective atomic features might exist which in combination outperform the current evaluation function.

7 Summary and Discussion

In this paper a practical framework for the semi–automatic construction of evaluation functions has been presented. Based on a generalized linear evaluation model — called GLEM — efficient procedures have been developed for generating training positions, exploring the feature space, and fitting feature weights. Rather than combining a few features by using complicated non–linear functions, we propose to construct evaluation functions by combining many — possibly more than hundred thousand — features, which are boolean combinations of atomic relations. This approach allows us to model non–linear effects directly, without the detour over analytic functions, and opens up practical ways for generating features automatically. GLEM allows the program author to concentrate on the part of evaluation function construction, where humans excel: the discovery of fundamental positional features by *reasoning* about the game. 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 now possible for the programmer to speculate about feature building blocks and to leave the creation of actually used features as well as assigning weights to them to the system. One example for this strategy has been presented in this paper: the observation that configurations can approximate important Othello concepts combined with the “mechanical” analysis of millions of training positions has produced an expert program capable of beating any human player. An interesting fact is that the game knowledge encoded by the set of over a million configuration weights goes far beyond the features we intended the system to approximate in the first place [1]. This result encourages the application of GLEM to other games or even to search or decision problems in other domains. Attractive candidates are chess and Go since both games are very popular and well analyzed. And yet, for chess, hardware roughly equivalent to four thousand ordinary PCs is currently needed[5] to compete with the human World–champion. For Go the status is even worse because brute–force search is not feasible due to the large branching factor. Since a good evaluation function is not known either amateurs are still able to beat the best Go programs. It is our opinion that the key to better chess and Go programs lies in improved evaluation functions. A starting point is the analysis of known features with regard to their approximation by weighted configurations as proposed by GLEM.

The automatic construction of features has been studied by several authors. Utgoff [10] proposes a general evaluation function learner, called ELF, which combines the processes of constructing boolean feature combinations and weight fitting. This approach has been shown to be effective in small artificial problems, but could not convince in its application to checkers. The main problem of ELF is its low speed. Taking into account the large number of features needed for an adequate evaluation in complex domains, and the resulting considerable effort for optimizing weights, it seems hopeless to combine feature construction and weight fitting. Other approaches for constructing features or adapting the combination function while fitting weights (e.g. Morph [5], meiosis networks [3], node splitting [11]), face similar complexity problems. Our solution is to separate these tasks in order to speed–up the process and to give many opportunities for optimization.

References

1. M. Buro. Experiments with Multi–Probcut and a new high–quality evaluation function for Othello. *Workshop on Game–Tree Search, NEC Research Institute*, 1997.
2. M. Buro. The Othello match of the year: Takeshi Murakami vs. Logistello. *ICCA Journal*, 20(3):189–193, 1997.
3. S.J. Hanson. Meiosis networks. *Advances in Neural Information Processing Systems*, pages 553–541, 1990.
4. F. Hsu, S. Anantharaman, M.S. Campbell, and A. Nowatzyk. Deep Thought. In T.A. Marsland and J. Schaeffer, editors, *Computer, Chess, and Cognition*, pages 55–78. Springer Verlag, 1990.
5. 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.

[5] Deep Blue searched around 200 million nodes per second in the 1997 match with Kasparov. Assuming a speed–up of four gained by using special purpose evaluation hardware and a speed of 200K nodes/sec of a state–of–the–art PC chess program leads to the given speed factor estimate.

6. W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes, 2nd edition*. Cambridge University Press, 1992.
7. A.L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 3(3):211–229, 1959.
8. G. Tesauro. TD–Gammon, a self–teaching backgammon program, reaches master–level play. *Neural Computation*, 6(2):215–219, 1994.
9. G. Tesauro. Temporal difference learning and TD–Gammon. *Communications of the ACM*, 38(3):58–68, 1995.
10. P.E. Utgoff. Constructive function approximation. Technical Report 97–4, Univ. of Mass., 1997.
11. M. Wynne-Jones. Node splitting: A constructive algorithm for feed–forward neural networks. *Neural Computing and Applications*, 1(1):17–22, 1993.