Evaluation-Function Factors

T.A. Marsland

Computing Science Department
University of Alberta
Edmonton
Canada T6G 2H1

ABSTRACT

The heart of a chess program is its evaluation function, since it is this component which characterizes the style of play. A typical program evaluates a move by computing a weighted sum of the features that it considers. It is important to make the best selection of the relative weights of these features. They may be used not only to assess horizon nodes in the game tree, but also to order moves at the other nodes so that alpha-beta searching efficiency is improved. Standard optimization techniques can be used to find the weights, provided a suitable cost function can be found. This paper assesses the properties of several cost functions, and presents a method for finding optimum weightings for any set of features.

Introduction

Some modern chess programs combine knowledge in complex ways; others follow Shannon's suggestion [5] and evaluate material and strategic factors by computing a weighted sum of the "values" of such features as are recognized. To do so one must first tabulate all the relevant features in chess. This is difficult enough in itself, but determining their relative importance is even harder. Often the resulting decision is arbitrary and subject to the programmer's biases. In principle it is possible to determine those relative weightings mathematically. In practice this is not only computationally expensive, but also fraught with difficulty because what passes for chess expertise is sometimes inconsistent. We are all aware of arguments about the preferred placement of pieces (and yet have no difficulty finding exceptions) and of rules about the advantages of bishops over knights, on which there is little agreement. Similarly, while doubled pawns are usually weak, there are often compensating advantages such as a half open file or control of a key square. Most of these apparent contradictions occur because of interactions between knowledge. Thus not all features in a chess position are independent; some are important only if others are present. We must, therefore, seek relative weights for the features, but these can only be correct in the statistical sense (more often right than wrong).

In an attempt to find the relative importance of chess features in an early chess program, a set of about 1000 chess positions were put together. These positions, referred to as the NY1924 data set [3] and taken from games played in the famous Grand Master event in New York [1], were used to measure the quality of move ordering mechanisms in WITA [4] (an early 1970's chess program and forerunner of Awit) and TECH [2]. It is customary and possible to base move ordering on a weighted sum of features. Let us assume that N features, such as various pawn formations, doubled rooks and king safety, are to be considered. Associated with the i^{th} feature is a weight, w_i. Let there be M legal moves in any given chess position; we can then build a feature matrix, F, having N rows and M columns. The feature matrix is produced by the chess program itself. The j^{th} column in the matrix contains entries that indicate whether the given features are present in the position which results from the j^{th} move out of M. These entries may also measure how often (or how much of) this feature exists after the move, should it be selected. For example, how many doubled pawns or how much piece mobility would result. Note: our use of indices i and j is the reverse of the normal mathematical convention. Finally we use the knowledge of a chess expert to identify the most desirable move, d. Typically this would be the move actually played by a Grand Master in some tournament.

Notation

$$f_{ij} = \text{measure of feature } i \text{ in move } j \text{ for any position.}$$

$$d = \text{desired (master) move for a position, where } 1 \le d \le M.$$

$$s_j = \sum_{i=1}^{N} w_i \, f_{ij} = \text{score for move } j$$

$$w_i = \text{relative weight of feature } i\text{, where } 1 \le i \le N.$$

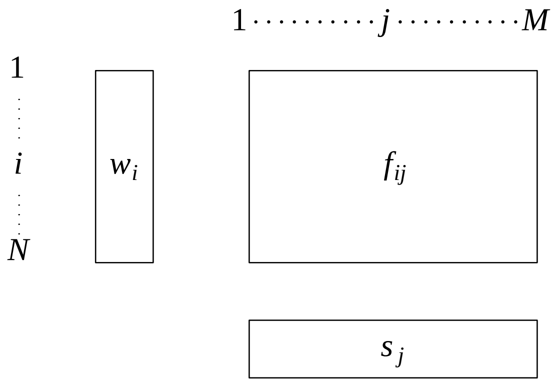

Figure 1: Relationship between features, weights and scores.

Problem Statement

It is unlikely that any given set of weights will be optimum for even a single position. What constitutes an optimum may also depend on the intended use of the evaluation function. For a horizon (terminal) node evaluator, optimum would mean that the score of the desired move is greatest. For preliminary ordering of moves at the root or interior nodes of the game tree, optimum would mean that the desired move would be rated so highly, that no other weighting of the features would give the desired move a higher rating in a move list ordered by score. More formally, we wish to find the $\{w_i\}$ such that

$$C(w_i, d, f_{ij})$$

is minimized over i, where C is some cost function whose properties are to be specified. For example, if the score for the j^{th} move is s_j, then a simple cost function to count how many moves have a score greater than the desired move, d, would be

$$C_1(w_i, d, f_{ij}) = \sum_{j=1}^{M} (if\ \ s_d \ge s_j\ \ then\ 0\ else\ 1).$$

The $\{w_i\}$ which minimizes C_1 has the effect of minimizing the rank-number of the desired move in a move list ordered by score. This simple scheme has a major flaw in that as much worth may be given to pushing a move from location 33 to 32 in a move list, as for changing from location 3 to 2. Generally speaking the latter is much harder to achieve and is far more beneficial in a root or interior node evaluator, since it improves the alpha-beta search efficiency, and does so more spectacularly as the node in question is nearer to the root.

A. First Remedy: Improve the value of the desired move

If we choose a cost function which is well behaved (continuous, differentiable around its single minimum), then we can apply conventional optimization techniques to find $\{w_i\}$. For these reasons consider a cost function based on the following:

$$C_2 = \sum_{j=1}^{M} [Max\{0, (s_j - s_d)\}]^2,$$

which minimizes the difference between the score for the desired move and the score for those above it. Strange though this function might appear, it is differentiable and its partial derivatives are given by:

$$\frac{\partial C_2}{\partial w_i} = 2 \sum_{j=1}^{M} \mathrm{Max}\ \{0, (s_j - s_d)\}\ \ (f_{ij} - f_{id}).$$

Unfortunately, C_2 tries even harder to push upwards the moves near the bottom of the list, at the expense of those near the desirable top, since there is more scope for improving the score of bottom moves. This problem might be circumvented by striking out all moves with sufficiently poor scores, since they cannot be assessed by the feature matrix.

One other major disadvantage of C_2 is that it tends to force all the w_i to zero; the reason is clear: one way to achieve a zero value for the set

$$\{ \frac{\partial C_2}{\partial w_i} \}$$

is to insert zeros into all w_i, in turn forcing all the s_j to zero in C_2. If we assume that we know which are beneficial and which are detrimental features, then we can assign negative values to the detrimental ones. The weights, w_i, can now all be positive, and we can ensure that they will not be forced to zero by applying the restriction that $w_i \geq 1.0$ for all i. This is appropriate since the ordering of the moves is insensitive to a uniform constant of proportionality - even if the scores s_j are not. However, there is still a tendency to reduce as far as possible the weight corresponding to the most prominent feature, namely the one given by:

$$\underset{i}{Max} \sum_{j=1}^{M} f_{ij}$$

Typically, this most prominent feature represents space and mobility [6].

The problems of restricting the w_i can be handled in several ways, including either use of a penalty function or scaling the smallest weight to 1. A slightly simpler version of C_2 is worth considering

$$C_3 = [Max \{ 0, (s^* - s_d) \}]^2$$

where $s^* = \underset{j}{Max} \{ s_j \}$ is the highest score for any move in the current position. Clearly C_2 and C_3 have the same problem: undue influence by poorly rated desired moves (i.e., moves based on features which are not present in F). For that reason we propose

$$C_4 = [Min \{ 0, (s^* - \varepsilon - s_d) \}]^2 ,$$

where ε is chosen so that moves which are seriously in error will not be considered. For instance $\varepsilon = s^*/10$ would ensure that only desired moves which have a value within 10% of the highest for the position will affect the weights. C_4 eliminates the influence of poor moves by only considering moves when they are close to the best.

Sometimes d, the desired move, is not characterized by the features in the matrix, so that the desired move always has a poor score. Also the features chosen may not discriminate well between the desired move and several others, giving them all a high score. For these reasons, one might be content with a function which maximizes the number of desired moves in the top t of the move list. With $t = 1$ one would produce weights for a horizon node evaluator, and for wider values of t a move ordering mechanism for a selective search program.

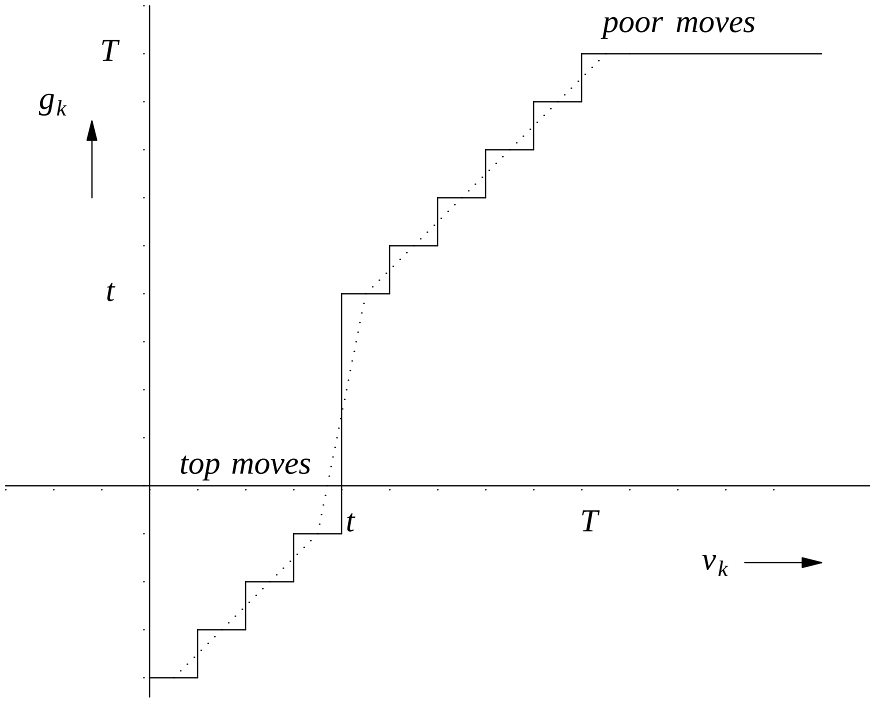

Figure 2: Incremental Cost Function with Bonus.

B. Second Remedy: Improve the ranking of the desired move

Of course it makes no sense to improve weights using data from a single position only. Use therefore a set of P positions, like the 1000 positions of the NY1924 data set, for which the k^{th} position has M_k moves. Nothing changes, except that our previous cost functions must contain sums over all P positions, and we now must consider the consequences of trade-offs between different desired moves. Consider

$$v_k = \sum_{j=1}^{M_k} (\text{ if } \ s_{dk} \geq s_{jk} \ \text{ then 0 else 1})$$

and

$$g_k = \text{if } \ (v_k < t) \ \text{ then } v_k - t$$

$$\text{else } v_k \, .$$

If we minimize

$$\sum_{k=1}^{P} g_k$$

then a bonus will be given to any desired move as it passes into the top few of the move list, as Figure 2 illustrates. The main reason for this bonus is to retain desired moves in the top t, and not to drop those in favour of minor improvements in the evaluation of positions lacking known features. Of course, g_k shows step discontinuities, and the most appropriate form of the derivative is ambiguous, there being a difference on whether one takes the derivative coming from the left or from the right. Moreover, it likely that the cost function is constant over small perturbation of weights (but should vary continuously if the derivatives are to be estimated numerically), quite apart from the fundamental problem of the influence of desired moves that are located far from the top of the move list.

There are at least two ways of dealing with desired moves which lack features:

(a) Construct a window, based on t and T, within which changes in the ranking of desired moves have no effect, as illustrated in Figure 2. For example, such a window might change g_k to

$$\begin{aligned} g_k = \text{if } (v_k < t) \ &\textit{then } (v_k - t) \\ &\textit{else} \text{ if } (v_k > T) \ \textit{then } T \\ &\textit{else } v_k . \end{aligned}$$

This cost function also minimizes the average distance of the desired from the top of the move list, but does not allow poor moves to affect the weights. Its derivative is not continuous, so the gradient following method to be described later may not work well. However, variations of that technique, or integer programming methods, may be used [7].

(b) Construct a cost function in which costs alter sharply when moves are near the top of the list, and more slowly in proportion to their distance from the top. This method is attractive and will be dealt with in more detail.

C. Third Remedy: Promote moves in reverse proportion to their rank

One final class of cost function is

$$\sum_{k=1}^{P} \frac{1}{R_k}$$

where R_k is the relative ranking of the desired move in the ordered move list, and P is the number of positions in the data set. R_k is given by

$$R_k = \sum_{j=1}^{M_k} (\textit{if} \quad s_{dk} \geq s_{jk} \quad \textit{then } 0 \textit{ else } 1).$$

By this means, changing the ranking of a move which is near the bottom of the list has significantly less effect than moving one near the top. For example, changing a desired move from location 33 to 32 gives an incremental change in cost of 1/32 - 1/33, while changing from location 3 to 2 gives an incremental change of 1/2 - 1/3. Some of the earlier objections can be overcome by this means, and R_k can be further refined so that greater importance is given to rating the desired moves in comparison to the best, rather than to force moves to the top of the list. Although R_k can be approximated by a continuous function, Figure 3, it is of course only defined at integral values. Convergence problems are to be expected, since small changes in weights may not achieve the desired move locations.

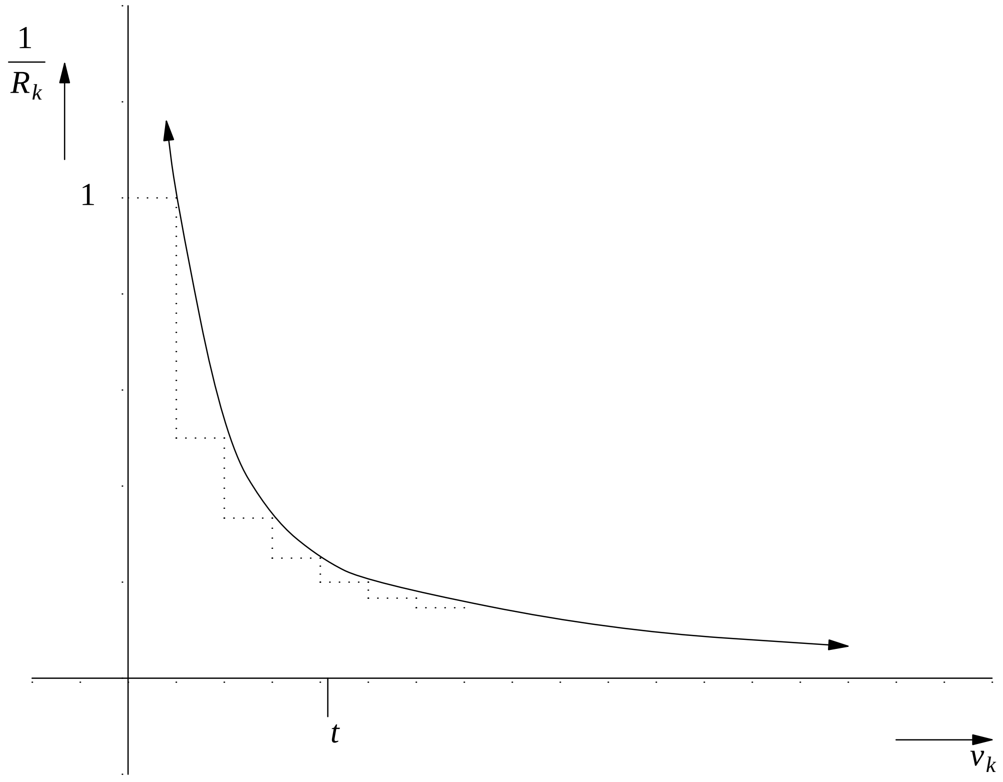

Figure 3: Inverse Cost Function.

Gradient Following Methods

The general problem we have posed is that of finding the set of weights, $\{w_i\}$, which minimizes

$$H(w_i) = C(w_i,\ d_k,\ f_{ijk})$$

subject to the restriction $w_i \geq 1$. When there are no restrictions on the w_i the minimum occurs when the gradient (slope) of H is zero. That is, when

$$\frac{\partial H(w_i)}{\partial w_i} = 0 \ \text{ for all } i,$$

and the minimum can be found in a variety of ways. From any initial set of weights, one general technique perturbs each of the weights by a small amount. Thus the gradient of the function can be estimated, and then by "stepping" in the direction of the negative gradient one can reduce the cost function. This produces a new set of weights w_i^* such that $H(w_i^{*)} < H(w_i)$. The process is applied successively until there is no significant change in the function value (indicating that the gradient is almost zero).

Solving this minimization problem under the linear constraint $w_i \geq 1$ is not difficult. Only when some of the w_i are equal to 1 is special handling required. If the slope component associated with such a variable is negative, as one might reasonably expect, then that component is set to zero. In other words, once a variable attains the constraining value of 1 it will be held at that value while the other weights are altered. The slope components may be estimated by

$$\frac{\partial H(w_i)}{\partial w_i} = \frac{H(w_i + \delta w_i) - H(w_i)}{\delta w_i}$$

Here δw_i represents a small perturbation and will be less than 10% of w_i. The step size may be the same for each slope direction, although this is not the most efficient.

Conclusion

This paper has outlined the issues in designing and selecting a cost function to help the search for the optimum set of weights for chess features. Depending on the cost function selected one can produce weights for a horizon node evaluator, or a root node move ordering mechanism. Our experience has been that it is enough to improve the weights so that they hold the desired moves within a small window, t, at the top of the move list. Further optimizations to force more of the desired moves into the first position, although technically more correct, tend to be counter-productive. The main problem is that the values returned do not assess the merit of the position in a realistic fashion. In a sense the weights have been tuned to the data set of positions, and they no longer measure reliably the quality of chess as a whole.

References

1. A. Alekine and H. H. (editor), The New York International Tournament 1924, Dover, New York, 1961.
2. J. J. Gillogly, Performance Analysis of the Technology Chess Program, CMU, Computer Science Dept., Carnegie-Mellon Univ., Pittsburg, March 1978.
3. T. A. Marsland and P. G. Rushton, "Mechanisms for Comparing Chess Programs," *ACM Annual Conference*, Atlanta, 1973, 202-205.
4. T. A. Marsland and P. G. Rushton, ‘‘A Study of Techniques for Game-playing Programs’’ in J. Rose (ed.), *Advances in Cybernetics and Systems,* vol. 1, Gordon & Breach, 1974, 363-372.
5. C. E. Shannon, ‘‘Programming a Computer for Playing Chess,’’ *Philosophical Magazine* **41**, 256-275 (1950).
6. E. Slater, "Statistics for the Chess Computer and the Factor of Mobility," *Symposium on Information Theory*, Ministry of Supply, London, 1950, 150-152.
7. A. Wouk, A Course of Applied Functional Analysis, Wiley Interscience, 1979.

Appendix

Some typical values for the quantities used in this paper are:

top move window	$t = 5$
poor move window	$T = 15$
Moves in a position	$M_k \leq 80$, with mean of 37
Positions in data set	$100 \leq P \leq 1000$
Features being considered	$1 \leq N \leq 50$

but often fewer than ten primary features.