TUNING EVALUATION FUNCTIONS BY MAXIMIZING CONCORDANCE

D. Gomboc, M. Buro, T. A. Marsland

Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada

{dave, mburo, tony}@cs.ualberta.ca, http://www.cs.ualberta.ca/~games/

Abstract

Heuristic search effectiveness depends directly upon the quality of heuristic evaluations of states in a search space. Given the large amount of research effort devoted to computer chess throughout the past half-century, insufficient attention has been paid to the issue of determining if a proposed change to an evaluation function is beneficial.

We argue that the mapping of an evaluation function from chess positions to heuristic values is of ordinal, but not interval, scale. We identify a robust metric suitable for assessing the quality of an evaluation function, and present a novel method for computing this metric efficiently. Finally, we apply an empirical gradient ascent procedure, also of our design, over this metric to optimize feature weights for the evaluation function of a computer chess program. Our experiments demonstrate that evaluation function weights tuned in this manner give equivalent performance to hand-tuned weights.

Keywords

ordinal correlation, Kendall's τ (tau), heuristic evaluation function, gradient ascent, partial-sum transform, heuristic search, computer chess

1 Introduction

A half-century of research in computer chess and similar two-person, zero-sum, perfect-information games has yielded an array of heuristic search techniques, primarily dealing with how to search game trees efficiently. It is now clear to the AI community that search is an extremely effective way of harnessing computational power. The thrust of research in this area has been guided by the simple observation that as a program searches more deeply, the decisions it makes continue to improve [42].

It is nonetheless surprising how often the role of the static evaluation function is overlooked or ignored. Heuristic search effectiveness depends directly upon the quality of heuristic evaluations of states in the search space. Ultimately, this task falls to the evaluation function to fulfill. It is only in recent years that the scientific community has begun to attack this problem with vigour. Nonetheless, it cannot be said that our understanding of evaluation functions is even nearly as thoroughly developed as that of tree searching. This work is a step towards redressing that deficit.

Inspiration for this research came while reflecting on how evaluation functions for today's computer chess programs are usually developed. Typically, they are refined over many years, based upon careful observation of their performance. During this time, engine authors will tweak feature weights repeatedly by hand in search of proper balance between terms. This ad hoc process is used because the principal way to measure the utility of changes to a program is to play many games against other programs and interpret the results. The process of evaluation function development would be considerably assisted by the presence of a metric that could reliably indicate a tuning improvement.

Consideration was also given to harnessing the great deal of recorded experience of human chess for developing an evaluation function for computer chess. Researchers have tried to make their machines play designated moves from test positions, but we focus on judgements about the relative worth of positions, reasoning that if these are correct then strong moves will be selected by the search as a consequence.

The primary objective of this research is to identify a metric by which the quality of an evaluation function may be directly measured. For this to be convincingly achieved, we must validate this metric by showing that higher values correspond to superior weight vectors. To this end, we introduce an estimated gradient ascent procedure, and apply it to tune eleven important features of a third-party chess evaluation function.

We begin with a summary of prior work in machine learning in chess-like games as it pertains to evaluation function analysis and optimization. This is followed by a brief, but formal, introduction to measurement theory, accompanied by an argument for the application of ordinal methods to the analysis of evaluation functions. A novel, efficient implementation of the ordinal metric known as Kendall's τ is described in §4; also indicated is how to apply it to measure the quality of an evaluation function. Next, we present an estimated gradient ascent algorithm by which feature weights may be optimized, and the infrastructure created to distribute its computation. Issues touching upon our specific experiments are discussed in §6; experimental results demonstrating the utility of ordinal correlation with respect to evaluation function tuning are provided in §7. After drawing some conclusions, we remark upon further plausible investigations.

2 Prior Work on Machine Learning in Chess-Like Games

Here we touch upon specific issues regarding training methods for static evaluation functions. For a fuller review of machine learning in games, the reader is referred to Fürnkranz's survey [12].

2.1 Feature Definition and Selection

The problem of determining which features ought to be available for use within an evaluation function is perhaps the most difficult problem in machine learning in games. Identification and selection of features for use within an evaluation function has primarily been performed by humans. Utgoff is actively involved in researching automated feature construction [11, 47, 48, 49, 51], typically using Tic-Tac-Toe as a test bed. Utgoff [50] discusses in depth the automated construction of features for game playing. He argues for incremental learning of layers of features, the ability to train particular features in isolation, and tools for the specification and refactoring of features.

Hartmann [14, 15, 16] developed the "Dap Tap" to determine the relative influence of various evaluation feature categories, or notions, on the outcome of chess games. Using 62,965 positions from grandmaster tournament and match games, he found that "the most important notions yield a clear difference between winners and losers of the games".

Unsurprisingly, the notion of material was predominant; the combination of other notions contributes roughly the same proportion to the win as material did alone. He further concluded that the threshold for one side to possess a decisive advantage is 1.5 pawns.

Kaneko, Yamaguchi, and Kawai [19] automatically generated and sieved through patterns, given only the logical specifications of Othello and a set of training positions. The resulting evaluation function was comparable in accuracy (though not speed of execution) to that developed by Buro [8].

A common property of Tic-Tac-Toe and Othello that makes them particularly amenable to feature discovery is that conjunctions of adjacent atomic board features frequently yield useful information. This area of research requires further attention.

2.2 Feature Weighting

Given a decision as to what features to include, one must then decide upon their relative priority. We begin with unsupervised learning, following the historical precedent in games, and follow with discussion of supervised learning methods. It is worth noting that Utgoff and Clouse [46] distinguish between state preference and temporal difference learning, and show how to integrate the two into a single learning procedure. They demonstrate using the Towers of Hanoi problem that the combination is more effective than one alone.

2.2.1 Unsupervised Learning

Procedures within the rubric of reinforcement learning rely on the difference between the heuristic value of states and the corresponding (possibly non-heuristic) values of actual or predicted future states to indicate progress. Assuming that any game-terminal positions encountered are properly assessed (trivial in practice), lower differences imply that the evaluation function is more self-consistent, and that the assessments made are more accurate.

The chief issue to be tackled with this approach is the credit assignment problem: what, specifically, is responsible for differences detected? While some blame may lie with the initial heuristic evaluation, blame may also lie with the later, presumably more accurate evaluation. Where more than one ply exists between the two (a common occurrence, as frequently either the result of the game or the heuristic evaluation at the leaf of the principal variation searched is used) the opportunities for error multiply. The TD(λ) technique (Sutton, 1988) solves this problem by distributing the credit assignment amongst states in an exponentially decaying manner. By training iteratively, responsibility ends up distributed where it is thought to belong.

The precursor of modern machine learning in games is the work done by Samuel [30, 31]. By fixing the value for a checker advantage, while letting other weights float, he iteratively tuned the weights of evaluation function features so that the assessments of predecessor positions became more similar to the assessments of successor positions. Samuel's work was pioneering not just in the realm of games but for machine learning as a whole. Decades passed before other game programmers took up the challenge.

Tesauro [40] trained his backgammon evaluator via temporal difference learning. After 300,000 self-play games, the program reached strong amateur level. Subsequent versions also contained hidden units representing specialized backgammon knowledge and used expectimax, a variant of minimax search that includes chance nodes. TD-Gammon is now a world-class backgammon player.

Beal and Smith [4] applied temporal difference learning to the scores backed up to the root by alpha-beta searches to determine piece values for a chess program that included material, but not positional, terms. Program versions using weights resulting from five randomized self-play learning trials each won a match versus a sixth program version that used the conventional weights given in most introductory chess texts. They have since extended their reach to include piece-square tables for chess [5] and piece values for Shogi [6].

Baxter, Tridgell, and Weaver (1998) introduced the name TDLeaf(λ) for Beal and Smith's application of temporal difference learning, and used it to learn feature weights for their program KnightCap. Through online play against humans, KnightCap's skill level improved from beginner to strong master. The authors credit this to: the guidance given to the learner by the varying strength of its pool of opponents, which improved as it did; the exploration of the state space forced by stronger opponents who took advantage of KnightCap's mistakes; the initialization of material values to reasonable settings, locating KnightCap's weight vector "close in parameter space to many far superior parameter settings".

Schaeffer, Hlynka, and Jussila [33] applied temporal difference learning to Chinook, the World Man-Machine Checkers Champion, showing that this algorithm was able to learn feature weights that performed as well as Chinook's hand-tuned weights. They also showed that, to achieve best results, the program should be trained using search effort equal to what the program will typically use during play.

2.2.2 Supervised Learning

Any method of improving an evaluation function must include a way to determine whether progress is being made. In cases where particular moves are sought, a simple enumeration of the number of 'correct' moves selected may be the criterion employed. This metric can be problematic because frequently no single move is clearly best in a position. Nonetheless, variations on this method have been attempted.

Marsland [23] investigated a variety of cost functions suitable for optimizing values and rankings assigned to moves for the purposes of move selection and move ordering. He found tuning to force recommended moves into relatively high positions to be effective, but attempting to force desired moves to be preferred over all alternatives was counterproductive.

Tesauro [39] devised a neural network structure appropriate for learning an evaluation function for backgammon, and compared states resulting from moves actually played against alternative states resulting from moves not chosen. "Comparison training" was shown to outperform an earlier system that compared states before and states after a move was played.

Tesauro [41] applied comparison training to a database of expert positions to demonstrate that king safety terms had been underweighted in Deep Blue's evaluation function in

1996. The weights were raised for the 1997 rematch between Garry Kasparov and Deep Blue. Subsequent analysis demonstrated the importance of this change.

A popular approach is to solve for a regression line across the data that minimizes the least-square error of the data. While this may be done by solving a set of linear equations, typically an iterative process known as gradient descent (a.k.a. hill climbing or gradient ascent) is applied. The partial derivatives with respect to the weights are set to zero, and the parameters are solved for via an iterative procedure. The least-square error function is quadratic, so there will be a single, global minimum to which all gradients will lead. Solving regression curves (non-linear regressions) by gradient descent is also possible, but in these cases, converging to global a minimum may be considerably more difficult.

The Deep Thought (later Deep Blue) team applied least-square fitting to the moves of the winners of 868 grandmaster games to tune their evaluation function parameters as early as 1987 [1. 2, 17, 25]. They found that tuning to maximize agreement between their program's preferred choice of move and the grandmaster's was "not really the same thing" as playing more strongly. Amongst other interesting observations, they discovered that conducting deeper searches while tuning led to superior weight vectors being reached.

Buro [7] estimated feature weights by performing logistic regression on win/loss/draw-classified Othello positions. The underlying log-linear model is well suited for constructing evaluation functions for approximating winning probabilities. In that application, it was also shown that the evaluation function based on logistic regression could perform better than those based on linear and quadratic discriminant functions.

Buro [8] used linear regression and positions labelled with the final disc differential of Othello positions to optimize the weights of thousands of binary pattern features. This approach yields significantly better performance than his 1995 work [7].

2.3 Miscellaneous

Levinson and Snyder [21] created Morph, a chess program designed in a manner consistent with cognitive models, except that look-ahead search was strictly prohibited. It generalizes attack and defence relationships between chess pieces and squares adjacent to the kings, applying various machine learning techniques for pattern generation, deletion, and weight tuning. Morph learned to value material appropriately and to play reasonable moves from the opening position of chess.

Van der Meulen [52] provides algorithms for selecting a weight vector that will yield the desired move, for partitioning a set of positions into groups that share such weight vectors, and for labelling new positions as a member of one of the available groups.

Simulated annealing is distinguished from gradient ascent in that it attempts to step in a random direction. Moves that appear to be beneficial are always accepted, but unlike hill climbing, sometimes moves that appear counterproductive are also accepted. The degree to which such moves will be accepted depends upon what is known as the temperature, which slowly lowers (albeit not monotonically) as the algorithm progresses. However, we are not aware of any tuning procedures for chess-like games that specifically apply this technique.

Evolutionary methods have also been applied to this problem. Kendall and Whitwell [20] evolved intermediate-strength chess players from a population of poor players by applying crossover and mutation operators to generate new weight vectors, while discarding vectors that performed poorly. Usually, a gradient descent approach will converge significantly more quickly than a genetic algorithm or its like would, and therefore these methods are not much in favour. Specifics of our application suggest that such a method may nonetheless be appropriate for our problem. We discuss this issue in §8.1.

3 Measurement Theory

Sarle [32] states: "Mathematical statistics is concerned with the connection between inference and data. Measurement theory is concerned with the connection between data and reality. Both statistical theory and measurement theory are necessary to make inferences about reality." We are endeavouring to identify a direct way to measure the quality of an evaluation function, so a familiarity with measurement theory is important.

3.1 Statistical Scale Classification

The four standard statistical scale classifications in common use today to classify unidimensional metrics originated with Stevens [35, 36]. He defined the following, increasingly restrictive categories of relations:

A relation $R(x, y)$ is *nominal* when R is a one-to-one function. Let function f be such a relation that is defined over the real numbers: $y = f(x)$. Then, f is nominal if and only if

$$\forall i \forall j : i = j \Leftrightarrow f(i) = f(j) \tag{3.1}$$

One common example of a nominal relation is the relationship between individuals of a sports team and their jersey numbers.

Additionally, f is *ordinal* when f is strictly monotonic: as the value of x increases, either y always increases or y always decreases. We will assume (without loss of generality, as we can negate f if necessary) that y always increases. Formally:

$$\forall i \forall j : i < j \Leftrightarrow f(i) < f(j) \tag{3.2}$$

The relationship of the standard five responses "strongly disagree", "disagree", "neutral", "agree", and "strongly agree" to many survey questions is ordinal. The mapping of chess positions to the possible assessments given in Table 1 in §3.3 is also ordinal.

Additionally, f is of *interval* scale when f is affine:

$$\exists c > 0\, \forall i \forall j : i - j = c[f(i) - f(j)] \tag{3.3}$$

The difference between two elements is now meaningful. Hence, subtraction and addition are now well-defined operations. Time is of interval scale.

Additionally, f is of *ratio* scale when the zero point is fixed:

$$\forall i \forall j \neq 0, f(j) \neq 0 : \frac{i}{j} = \frac{f(i)}{f(j)} \tag{3.4}$$

The ratio between two elements is now meaningful. Hence, division and multiplication are now well-defined operations. Temperature is an example of a ratio scale: the zero point is absolute zero, which is 0 kelvins.

These categories are the ones typically covered by introductory statistics texts, but are by no means the only plausible categories. Stevens himself recognized a log-interval scale classification [37], to which the Richter and decibel scales belong. It has a fixed zero, and a log-linear relationship. Every log-interval scale is ordinal, and every ratio scale is log-interval, but interval and log-interval are incompatible. The absolute scale is a specialization of the ratio scale, where not only the zero point but also the one point is fixed. This scale is used for representing probabilities and for counting.

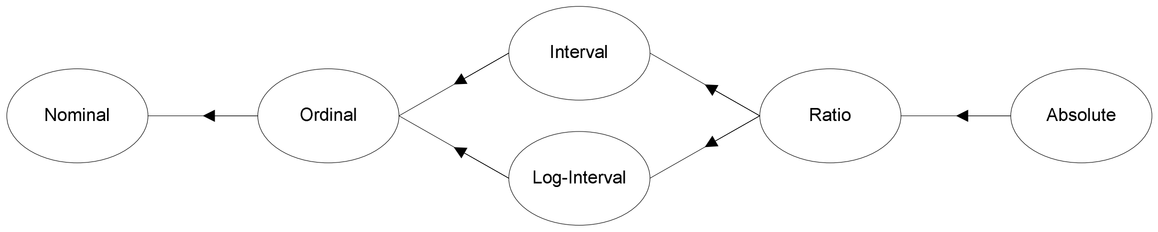

Figure 1: Partial Ordering of Statistical Scale Classifications

Figure 1 demonstrates the inheritance relationships between the various classifications. It is to be understood that this classification system is open to extension, and that it does not encapsulate all interesting properties of functions.

3.2 Admissible Transformations

Transformations are deemed admissible (Stevens used "permissible") when they conserve all relevant information. For example, converting from kelvins to degrees Celsius (°C) by applying $f(x) = x + 273.16$ would not be considered admissible, because the zero point would not be preserved. Such a transformation is not prohibited per se, but it does mean that usages involving ratios would no longer be justified. Three hundred kelvins is actually twice as hot as one hundred fifty kelvins. Clearly, then, 26.84°C cannot be twice as hot as 13.42°C.

The transformations deemed admissible for any category are those that retain the invariants specified by (2.1) through (2.4). A transformation is admissible with respect to the nominal scale if and only if it is injective. For the ordinal scale, the preservation of order is also required. For the interval scale, equivalent differences must remain equivalent, so only affine transformations are permitted ($g(\mathrm{x}) = af(x)+b$, $a>0$). Admissibility with respect to the ratio scale also require the zero point to remain fixed ($b = 0$).

Stevens argued that it is not justified to draw inferences about reality based upon non-admissible transformations of data, because otherwise the conclusions that could be drawn would vary depending upon the particulars of the transformation applied. "In general, the more unrestricted the permissible transformations, the more restricted the statistics. Thus, nearly all statistics are applicable to measurements made on ratio scales, but only a very limited group of statistics may be applied to measurements made on nominal scales" [37].

Much discussion of this highly controversial position has taken place. Tukey opined that in science, the "ultimate standard of validity is an agreed-upon sort of logical consistency and provability" [45], and that this is incompatible with an axiomatic theory of measurement, which is purely mathematical. Velleman and Wilkinson argue that the classifications themselves are misleading [53]. It is important to note that the notion of admissibility is relative to the analysis being performed rather than purely data-centric: it is perfectly valid to convert from kelvins to degrees Celsius, so long as one does not then draw conclusions based upon the ratios of temperatures denoted in degrees Celsius. Stanley's classifications remain in widespread use, though in particular his stricture against drawing conclusions based upon statistical procedures that require interval-scale data applied to data with only ordinal-scale justification is frequently ignored.

3.3 Relevance to Heuristic Evaluation Functions for Minimax Search

Utgoff and Clouse [46] comment: "Whenever one infers, or is informed correctly, that state A is preferable to state B, one has obtained information regarding the slope for part of a correct evaluation function. Any surface that has the correct sign for the slope between every pair of points is a perfect evaluation function. An infinite number of such evaluation functions exist, under the ordinary assumption that state preference is transitive."

Throughout the search process of the minimax algorithm for game-tree search [34], and all its derivatives [22, 27], a single question is repeatedly asked: "Is position A better than position B?" Not "How much better?", but simply "Is it better?". In minimax, instead of propagating values one could propagate the positions instead, and, as humans do, choose between them directly without using values as an intermediary. This shows that we only need pairwise comparisons that tell us whether position A is better than position B.

The essence of a heuristic evaluation function is to make educated guesses regarding the likelihood of successful outcomes from arbitrary states. This is somewhat obscured in chess by the tendency to use a pawn as the unit by which advantage is measured, a habit encouraged by the high correlation between material advantage and likelihood of success inherent in the game.

The implicit mapping between a typical evaluation scoring system (range: [-32767, 32767], where the worth of a pawn is represented by 100) and the probability of success (range: [0, 1]), maintains the ordinal invariant of linear ordering, though not the interval invariant of equality of differences. This is interesting insofar as typically terms will be added and subtracted during the assessment of a position by a computer program, operations that according to Stevens require interval-scale justification. Furthermore, the difference between scores is frequently used as criteria to apply numerous search techniques (e.g. aspiration windows, lazy evaluation, futility pruning). How can we reconcile his theory with practice?

Firstly, a heuristic search is exactly that: heuristic. Many of these techniques encompass a trade-off of accuracy for speed, with the difference threshold set empirically. Thus, to some extent, compensation for the underlying ordinal nature of scores returned by the evaluation function is already present.

Secondly, it can be argued that the region between -200 and 200, where the majority of assessments that will influence the outcome of the game lie, is in practice nearly one of interval scale. This means that decisions made based on a difference threshold while the game is roughly balanced are relatively less likely to be faulty in important situations.

Finally, one may choose to agree with Stevens' detractors, and indeed, many social scientists compute means on ordinal data frequently without much ado.

Does this mean that we should ignore the underlying ordinality of the problem? No. Everything of interval scale is of ordinal scale, while the converse is false. A metric capable of handling ordinal data is readily applicable to data that meets a more stringent qualification. In contrast, the application of a metric suited for interval scale data to data of ordinal scale is, at least to researchers who agree with Stevens, suspect, and requires justification. Such justification might take the form of the ability to predict and validate predictions that depend on the proposed interval nature of the variable.

Let us turn our attention to the application of the assessment categories of Table 1 to chess positions, with the corresponding English definitions as provided by Chess Informant [29]. We have given them in order of White's increasing preference, so the categories are monotonically increasing, however, it would be nonsense to say that the difference between +− and = is three times that of the difference between ± and =.[1] Therefore, they are of ordinal scale. It would be possible to label these with the numbers 1 through 7, and proceed from there as if they were of interval scale, but at least according to Stevens, doing so would require justification. Fortunately, it is unnecessary, as we will show in the next chapter.

Table 1: Symbols for Chess Position Assessment[2]

symbol	meaning
−+	black has a decisive advantage
∓	black has the upper hand
⩱	black stands slightly better
=	even
⩲	white stands slightly better
±	white has the upper hand
+−	white has a decisive advantage

[1] Or, would it? It has been said that it takes three mistakes to lose a chess game. While not literally true, the aphorism is based on experience. On the other hand, it is easier to make a mistake in an already poor position.

[2] Two other assessment symbols, ∞ (the position is unclear) and $\overset{=}{\infty}$ (a player has positional compensation for a material deficit) are also frequently encountered. Unfortunately, the usage of these two symbols is not consistent throughout chess literature. Accordingly, we ignore positions labeled with these assessments.

4 Kendall’s τ

Concordance, or agreement, occurs where items are ranked in the same order. Kendall's τ measures the degree of similarity of two orderings. After defining this metric, we will elaborate on a novel, efficient implementation of τ, provide a worked example and complexity analysis, introduce a generalization of τ, and contrast τ with alternative correlation methods.

4.1 Definition

Given a set of m items $I = (i_1, i_2, \ldots, i_m)$, let $X = (x_1, x_2, \ldots, x_m)$ represent relative preferences for I by analyst a_e, and $Y = (y_1, y_2, \ldots, y_m)$ represent relative preferences for I by analyst a_o. Take any two pairs, (x_i, y_i) and (x_k, y_k), and compare both the x values and the y values. Table 2 defines the relationship between those pairs.

Table 2: Relationships between Ordered Pairs

relationship between x_i and x_k	relationship between y_i and y_k	relationship between (x_i, x_k) and (y_i, y_k)
$x_i < x_k$	$y_i < y_k$	concordant
$x_i < x_k$	$y_i > y_k$	discordant
$x_i > x_k$	$y_i < y_k$	discordant
$x_i > x_k$	$y_i > y_k$	concordant
$x_i = x_k$	$y_i \neq y_k$	extra y-pair
$x_i \neq x_k$	$y_i = y_k$	extra x-pair
$x_i = x_k$	$y_i = y_k$	duplicate pair

n, the total number of distinct pairs of positions, and therefore also the total number of comparisons made, is straightforward to compute:

$$n = \sum_{i=1}^{m-1} \sum_{k=i+1}^{m} 1 = \tfrac{1}{2} m(m-1) \tag{4.1}$$

Let S^+ (“S-positive”) be the number of concordant pairs:

$$S^+ = \sum_{i=1}^{m-1} \sum_{k=i+1}^{m} \begin{cases} 1, x_i < x_k \text{ and } y_i < y_k \\ 1, x_i > x_k \text{ and } y_i > y_k \\ 0, \text{otherwise} \end{cases} \tag{4.2}$$

Let S^- (“S-negative”) be the number of discordant pairs:

$$S^- = \sum_{i=1}^{m-1} \sum_{k=i+1}^{m} \begin{cases} 1, x_i < x_k \text{ and } y_i > y_k \\ 1, x_i > x_k \text{ and } y_i < y_k \\ 0, \text{otherwise} \end{cases} \tag{4.3}$$

Then τ, which we will also refer to as the concordance, is given by:

$$\tau = \frac{S^+ - S^-}{n} \tag{4.4}$$

Possible concordance values range from +1, representing complete agreement in ordering, to -1, representing complete disagreement in ordering. Whenever extra or duplicate pairs exist, the values of +1 and -1 are not achievable.

Cliff (1996) provides a more detailed exposition of Kendall's τ, discussing variations thereof that optionally disregard extra and duplicate pairs. Cliff labels what we call τ as τ_a, and uses it most often, noting that it has the simplest interpretation of the lot.

4.2 Matrix Representation of Preference Data

In typical data sets, pairs of preferences will occur multiple times. We can compact all such pairs together, which allows us to process them in a single step, thereby substantially reducing the work required to compute S^+ and S^-.

Let S_x be the set of preferences $\{x_1, x_2, \ldots, x_m\}$, and S_y be the set of preferences $\{y_1, y_2, \ldots, y_m\}$. Let U = $|S_x|$ and V=$|S_y|$. Let $E = (e_1, e_2, ..., e_U)$ such that $\forall i : e_i \in S_x$ and $e_i < e_{i+1}$. Similarly, let $O = (o_1, o_2, ..., o_V)$ such that $\forall i : o_i \in S_y$ and $o_i < o_{i+1}$. We define matrix A of dimensions U by V such that

$$A_{cr} = \sum_{i=1}^{m} \begin{cases} 1, e_r = x_i \text{ and } e_c = y_i \\ 0, \text{otherwise} \end{cases} \tag{4.5}$$

A's definition ensures that there is at least one non-zero entry in every row and column. Let $A_{(x,y)}^{(z,w)}$ denote the sum of the cells of matrix A within rows x through z and columns y through w:

$$A_{(x,y)}^{(z,w)} = \sum_{i=x}^{z} \sum_{k=y}^{w} A_{ik} \tag{4.6}$$

Then, the contribution towards S^+ of the single cell A_{xy} is

$$\Delta S_{xy}^{+} = A_{(x+1,y+1)}^{(U,V)} A_{xy} \tag{4.7}$$

and S^+ may be reformulated as

$$S^+ = \sum_{x=1}^{U} \sum_{y=1}^{V} \Delta S_{xy}^{+} \tag{4.8}$$

Similarly,

$$\Delta S_{xy}^{-} = A_{(1,y-1)}^{(x-1,V)} A_{xy} \tag{4.9}$$

$$S^{-} = \sum_{x=1}^{U} \sum_{y=1}^{V} \Delta S_{xy}^{-} \tag{4.10}$$

Numerical Recipes in C [28] is a standard reference to this algorithm. However, we have reservations about the manner in which their sample implementation handles duplicate pairs.

4.3 Partial-Sum Transform

The naïve algorithm to compute Kendall's τ computes the number of concordant and discordant pairs for each cell by looping over the preferences data directly; the straightforward algorithm loops over the cells in the matrix. However, because we have ordered the axes monotonically, the computation is replete with overlapping subproblems of optimal substructure. It is this fact that we exploit.

We introduce auxiliary matrix *BR*, which has the same dimensions as *A*.[3] Each cell in *BR* is assigned the value corresponding to the sum of the cells in *A* that are either on, below, or to the right of the corresponding cell in *A*.

$$A_{(x,y)}^{(U,V)} = BR_{xy} = A_{xy} + BR_{x(y+1)} + BR_{(x+1)y} - BR_{(x+1)(y+1)} \tag{4.11}$$

Similarly, auxiliary matrix *BL* holds in each cell the sum of the cells in *A* that are either on, below, or to the left of the corresponding cell in *A*.

$$A_{(1,y)}^{(x,V)} = BL_{xy} = A_{xy} + BL_{x(y+1)} + BL_{(x-1)y} - BL_{(x-1)(y+1)} \tag{4.12}$$

BR and *BL* are both instances of partial-sum transforms of *A*. The sum of an arbitrary rectangular block of cells of the original matrix is conveniently retrieved from such a transformed version.[4] For instance:

$$BL_{xy} = BR_{1y} - BR_{(x+1)y} \tag{4.13}$$

$$A_{xy} = BR_{xy} - BR_{(x+1)y} - BR_{x(y+1)} + BR_{(x+1)(y+1)} \tag{4.14}$$

(4.13) lets us compute *BL* from *BR* in constant time as necessary, so we need not construct both *BR* and *BL*. Additionally, we can construct *BR* in-place by overwriting *A*, even without recourse to (4.14), reducing the storage overhead of computing Kendall's τ in this manner rather than from the original matrix representation to zero.

The single dimension case of the partial-sum transform is well known, e.g. unidimensional versions of (4.11) and (4.14) are provided by the partial_sum and adjacent_diffference algorithms in the C++ Standard Library. The partial-sum transform

[3] For convenience, we define $BR_{(x+1)y}$ to equal 0 when $x = U$. Similarly, $BR_{x(y+1)}$ equals 0 when $y = V$. Nonetheless, space need not be allocated for such cells.

[4] This was brought to our attention in a personal communication by Juraj Pivovarov.

is readily applied to matrices of higher dimension also. However, each additional dimension involved doubles the number of terms required on the right-side expression of (4.20) to maintain the equality.

4.4 Worked Example with Complexity Analysis

Here we provide sample data and work through the mathematics of the previous three subsections.

4.4.1 Definition

We will use 14 states for our example, therefore $m = 14$. X and Y represent the preference data for the 14 states. We could pre-order the positions so that they are ordered by Y, but in the interest of clarity, we will leave this for a later step.

Table 3: Sample Preference Data

$X = (x_1, x_2, \ldots, x_m)$	-0.1	-0.4	0.9	-1.2	0.0	-0.1	0.6	0.0	-0.1	-0.4	2.4	0.0	0.6	0.6
$Y = (y_1, y_2, \ldots, y_m)$	⩱	±	+−	−+	=	∓	⩲	=	+−	⩱	±	∓	−+	⩲

From (4.1), we know that n, the total number of distinct pairs of positions, is 91. We can now loop through the data, accumulating S^+ and S^-.

Table 4: Sample Preference Data: Relationships Between Ordered Pairs

Items Compared	(y_i, x_i)	(y_k, x_k)	y_i R y_k	x_i R x_k	(x_i, x_k) R (y_i, y_k)	ΔS^+	ΔS^-
1 and 2	⩱, -0.1	±, -0.4	<	>	discordant	0	1
1 and 3		+−, 0.9	<	<	concordant	1	0
1 and 4		−+, -1.2	>	>	concordant	1	0
1 and 5		=, 0.0	<	<	concordant	1	0
1 and 6		∓, -0.1	>	=	extra y-pair	0	0
1 and 7		⩲, 0.6	<	<	concordant	1	0
1 and 8		=, 0.0	<	<	concordant	1	0
1 and 9		+−, -0.1	<	=	extra y-pair	0	0
1 and 10		⩱, -0.4	=	>	extra x-pair	0	0
1 and 11		±, 2.4	<	<	concordant	1	0
1 and 12		∓, 0.0	>	<	discordant	0	1
1 and 13		−+, 0.6	>	<	discordant	0	1
1 and 14		⩲, 0.6	<	<	concordant	1	0
2 and 3	±, -0.4	+−, 0.9	<	<	concordant	1	0
2 and 4		−+, -1.2	>	>	concordant	1	0
2 and 5		=, 0.0	>	<	discordant	0	1
2 and 6		∓, -0.1	>	<	discordant	0	1
2 and 7		⩲, 0.6	>	<	discordant	0	1
2 and 8		=, 0.0	>	<	discordant	0	1
2 and 9		+−, -0.1	<	<	concordant	1	0
2 and 10		⩱, -0.4	>	=	extra y-pair	0	0
2 and 11		±, 2.4	=	<	extra x-pair	0	0
2 and 12		∓, 0.0	>	<	discordant	0	1
2 and 13		−+, 0.6	>	<	discordant	0	1
2 and 14		⩲, 0.6	>	<	discordant	0	1
3 and 4	+−, 0.9	−+, -1.2	>	>	concordant	1	0
and so on…							
12 and 13	∓, -0.2	−+, 0.6	>	<	discordant	0	1
12 and 14		⩲, 0.6	<	<	concordant	1	0
13 and 14	−+, 1.0	⩲, 0.6	<	>	discordant	0	1

For this data set, S^+ and S^- are 51 and 25 respectively, and $\tau = 0.2857$. The running time of this algorithm is $\Theta(m^2)$.

4.4.2 Matrix Representation of Preference Data

Based on the definitions of X and Y, S_y = {⩱, ±, +−, −+, =, ∓, ⩲}, S_x = {-0.4, -0.1, 2.4, -1.2, 0.0, 0.6, -0.2 }, and $V = U = 7$. Then O = (−+, ∓, ⩱, =, ⩲, ±, +−) and E = (-1.2, -0.4, -0.1, 0.0, 0.6, 0.9, 2.4). The purpose of this exercise is to sort the preferences in ascending order, with duplicates removed, so that matrix A, as shown in Table 5, has its rows and columns in order of increasing preference, and has no empty rows or columns. If the axes were not sorted, then dynamic programming would not be applicable.

Table 5: Sample Preference Data: (machine, human) assessments

	-1.2	-0.4	-0.1	0.0	0.6	0.9	2.4
−+	1				1		
∓			1	1			
⩱		1	1				
=				2			
⩲					2		
±		1					1
+−			1			1	

In this and subsequent tables, cells that would contain zero have been left blank.

The sorting takes time $\Theta(m \log m)$, while the duplicate entry removal takes time $\Theta(m)$. Zeroing the memory for the matrix takes time $\Theta(UV)$, then populating it takes time $\Theta(m)$. Therefore, the time complexity of constructing A is $\Theta(m \log m + UV)$.

Computing (4.6) takes time $\Theta(UV)$, because each cell in $A_{(x,y)}^{(z,w)}$ is addressed to perform the summations, so computing S^+ and S^- from A via (4.8) and (4.10) takes time $\Theta(U^2V^2)$. Therefore, computing S^+ and S^- from the preference data takes time $\Theta(m \log m + U^2V^2)$.

4.4.3 Partial-Sum Transform

To construct matrix BR, we start with the bottom-right hand corner, and proceed leftwards, than upwards, applying (4.11) at each cell.[5] In the tables that follow, the shaded area represents the portion of the matrix that has already been converted.

[5] One may fill forward in memory by reversing the element order of the table axes.

Table 6: Sample Preference Data:
***BR* being constructed overtop of *A*, at BR_{57}**

	-1.2	-0.4	-0.1	0.0	0.6	0.9	2.4
−+	1				1		
∓			1	1			
⩱		1	1				
=				2			
⩲					2		
±		1					1
+−			1		1	1	

The first cell to change value is BR_{57} – column 5, row 7; see (4.5). We continue sliding left until the entire row is transformed, after which we return to the right edge of the matrix, and continue with the preceding row.

Table 7: Sample Preference Data:
***BR* being constructed overtop of *A*, at BR_{56}**

	-1.2	-0.4	-0.1	0.0	0.6	0.9	2.4
−+	1				1		
∓			1	1			
⩱		1	1				
=				2			
⩲					2		
±		1			2	2	1
+−	2	2	2	1	1	1	

For this sample data, BR_{56} is the first cell for which the wrong value would have been computed if the final term within (4.11) were not present. Both BR_{66} and BR_{57} include BR_{67}, but we want to count it only once, so we must subtract it out.

Table 8: Sample Preference Data:
BR* fully constructed overtop of *A

	-1.2	-0.4	-0.1	0.0	0.6	0.9	2.4
−+	14	13	11	8	5	2	1
∓	12	12	10	7	4	2	1
⩱	10	10	8	6	4	2	1
=	8	8	7	6	4	2	1
⩲	6	6	5	4	4	2	1
±	4	4	3	2	2	2	1
+−	2	2	2	1	1	1	

BR may be constructed from *A* in $\Theta(UV)$ operations via (4.11). Once this has been done, (4.7) can be performed by table lookup, which in turn allows the computation of S^+ via (4.8) given *A* to be performed in $\Theta(UV)$ steps.

Table 9: Sample Preference Data:
BL*, which is deducible from *BR

	-1.2	-0.4	-0.1	0.0	0.6	0.9	2.4
−+	1	3	6	9	12	13	14
∓		2	5	8	10	11	12
⩱		2	4	6	8	9	10
=		1	2	4	6	7	8
⩲		1	2	2	4	5	6
±		1	2	2	2	3	4
+−			1	1	1	2	2

Values are retrieved from *BL*, which is implicitly available from *BR* via (4.13), in constant time, so the computation of S^- via (4.10) given *A* may also be performed in $\Theta(UV)$ steps. Therefore, the time complexity of computing τ via the partial-sum transform is $\Theta(m \log m + UV)$.

What have we achieved? The naïve algorithm runs in time $\Theta(m^2)$, so when *U* and *V* equal *m*, the partial-sum transform gives us no advantage in terms of asymptotic complexity. The work is useful nonetheless because we can reduce *U* and *V* at will by merging nearby preference values together. This enables approximate values for τ to be found quickly even when an exact value would be prohibitively expensive to compute. The improvement from $\Theta(m \log m + U^2V^2)$ to $\Theta(m \log m + UV)$ substantially decreases the amount of approximation required when *m* is large.

That said, most of the time *U* and *V* will be much smaller than *m*. When either *U* or *V* is constant, applying dynamic programming *is* asymptotically more time-efficient than prior methods. For the experiments in §7, *U* is about 0.4*m*, and $V = 7$.

Furthermore, τ is computed many times on similar data. In our application, *Y* remains constant, so, as mentioned in §4.4.1, this sort could be performed only once, ahead of time. *X* does change, but the difference between successive *X* vectors are small. It would be worthwhile to try sorting indirect references to the data, so that successive sorts would be processing data that is already almost sorted: this may yield a performance benefit.

4.5 Weighted Kendall's τ

As described, each position contributes uniformly to Kendall's τ. However, it can be desirable to give differing priority to different examples, for instance, to increase the importance of positions from underrepresented assessment categories, as described in §6.2, or to implement perturbation as described in §8.2.

Weighted Kendall's τ involves, in addition to the preference lists *X* and *Y*, a third list $Z = (z_1, z_2, \ldots, z_m)$ that indicates the importance of each position. The degenerate case where all $z_i = 1$ gives Kendall's τ. Integer z_i have the straightforward interpretation of the position with preferences (x_i, y_i) being included in the data set z_i times, though the use of fractional z_i is possible.

Computing Weighted Kendall's τ is similar to computing the original measure. When populating *A*, instead of adding 1 to the matrix cell corresponding to (x_i, y_i), we add z_i.

Computation of S^+ and S^- remains unchanged. The denominator of (4.4) is adjusted to be the total weight of all positions, rather than the total number of them.

Values for weighted τ will not be limited to the range [-1, 1] if weights less than 1 are used. All example weights can be uniformly scaled, so there is no need to use such weights. For certain applications, such as the optimization procedure described in §5, the denominator of (4.4) is irrelevant, so this guideline need not be strictly adhere d to.

The implementations developed actually compute weighted τ. However, these implementations have not been thoroughly tested with non-uniform weights.

4.6 Application

Earlier, we defined analysts a_e and a_o and tuples E and O without explaining their names. Analyst a_o represents an oracle: ideally, every position is assessed with its game-theoretic value: win, draw, or loss. Analyst a_e represents an imperfect evaluator. Then, the concordance of the assessments E and O is a direct measurement of the quality of the estimates made by analyst a_e.

We usually will not have access to an oracle, but we will have access to values that are superior to what the imperfect evaluator provides. In practice, a_o might be collected human expertise (and depending on the domain, possibly error-checked by a machine), or a second program that is known to be superior to the program being tuned. When neither is available, a_o can be constructed by conducting a look-ahead search using a_e as its evaluation function. In all cases, τ measures the degree of similarity between the two analysts.

4.7 An Alternative Ordinal Metric

Pearson correlation is the most familiar correlation metric. It measures the linear correlation between two interval-scale variables. When the data is not already of interval scale, it is first ranked. In this case, the correlation measure is referred to as Spearman correlation, or Spearman's ρ.

There is a special formula for computing ρ when the data is ranked; however, it is exact only in the absence of ties. In our application, we have seven categories of human assessment, but orders of magnitude more data points, so ties will be abundant. Therefore, it is most appropriate to apply Pearson's formula directly to the ranked data.

$$\rho = \frac{n\sum xy - \sum x \sum y}{\sqrt{[n(\sum x^2) - (\sum x)^2][n(\sum y^2) - (\sum y)^2]}} \tag{4.21}$$

The value computed by this least-square mean error function will be affected greatly by outliers. While ranking the data may reduce the pronouncement of this effect, it will not eliminate it. Changing a data point can change the correlation value computed, but there is no guarantee that an increase in value is meaningful. In contrast, the structure of Kendall's τ is such that no adjustment in the value of a machine assessment yields a higher concordance unless strictly more position pairs are ordered properly than before.

The asymptotic time complexity of computing Spearman's ρ is $\Theta(m \log m)$. As we noted earlier, the time complexity of computing Kendall's τ is $\Theta(m \log m + UV)$. Both algorithms sort their inputs, but afterwards, Spearman's ρ takes time $\Theta(m)$, while Kendall's τ takes time $\Theta(UV)$: for our intended use, these are equivalent. Not only does τ more directly measure what interests us ("for all pairs of positions (A, B), is position B better than position A?"), it is no less efficient to compute than the plausible alternative. Therefore, we use τ in our experiments.

5 Feature Weight Tuning via Kendall's τ

We will now attempt to tune feature weights by maximizing the value of Kendall's τ. After describing the gradient ascent procedure by which new weights are selected, we discuss our distributed implementation of the hill-climber.

The decision to implement a form of gradient ascent was made not because it was thought to be the most efficient method to attempt to learn effective weights, but because the behaviour of gradient ascent is well understood and predictable, allowing inferences from tuning attempts using the proposed optimization metric, Kendall's τ, to be made from the results of hill-climbing. Once some success had been achieved, inertia led to the complete system described in this chapter.

A discussion of problems related to the practical application of gradient ascent for tuning evaluation function weights (notably: performance) is deferred to §8.1.

5.1 Estimated Gradient Ascent

We wish to apply gradient ascent to find weights that maximize Kendall's τ. However, this metric is non-continuous, and so not differentiable. Therefore, we must measure the gradient empirically. The procedure is:

1. determine τ for the current base vector
2. for each weight w in the current base vector
    a. select a value ε by which to perturb the weight w[6]
    b. create two delta weight vectors by replacing the weight w from the current base vector with $w+\varepsilon$ and $w-\varepsilon$, while leaving other weights alone
    c. determine τ for these test weight vectors
    d. determine a new value for w for the base vector of the next iteration based upon the three known values for τ.

In each iteration, the concordance of the current base vector – that is to say, the weight vector currently serving as a base from which to explore – is measured. In addition, for each weight being tuned, we generate two test weight vectors by perturbing that weight

[6] In our most recent implementation, ε begins at 1% of the current value of w, and the percentage used is gradually lowered in successive iterations. However, if a random value between 0 and 1 is larger that ε, it is used instead (for that weight and iteration only). This prevents an inability of a weight to move a meaningful distance whenever w is close to zero.

higher and lower while holding the other weights constant. We refer to these as delta weight vectors because they are only slightly different from the base weight vector.

The two generated weight values are equidistant from the original value. The values of τ at these test vectors are also computed, after which a decision on how to adjust each weight is made. The cases considered are illustrated in Figure 2.

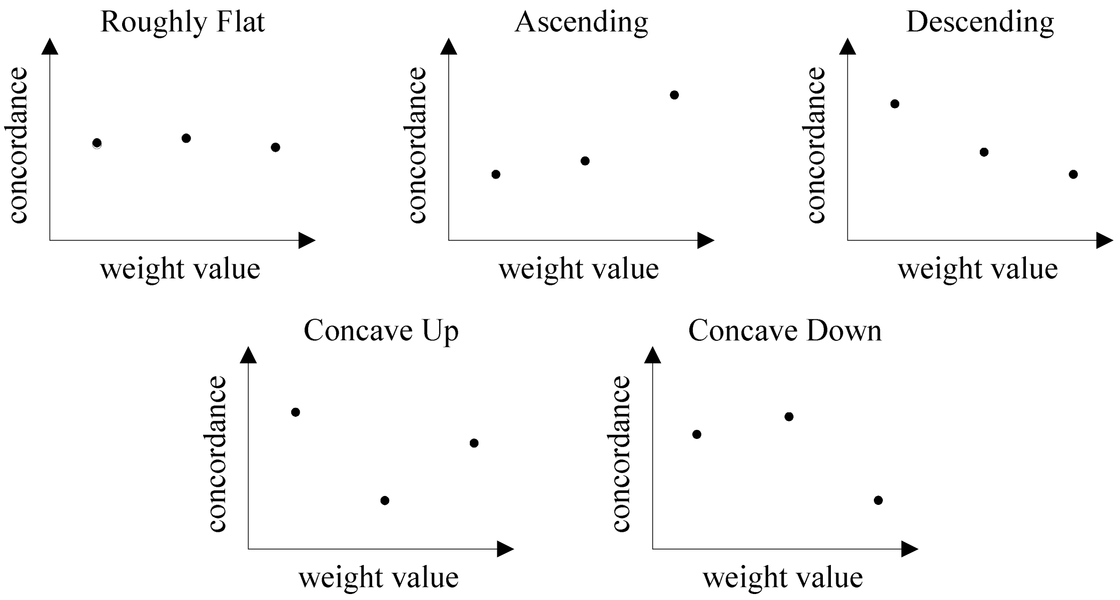

Figure 2: Distinct Cases to be Handled during Estimated Gradient Ascent

If there is no significant change in concordance between the current (base) vector and the test vectors, then the current value of the weight is retained. This precautionary category avoids large moves of the weight where the potential reward does not justify the risk of making a misstep.

When τ either ascends or descends in the region as the weight is increased, the new weight value is interpolated from the slope defined by the sampled points. The maximum change from the previous weight is bounded to 3 times the distance of the test points, to avoid occasional large swings in parameter settings.

When the points are concave up, we adopt the greedy strategy of moving directly to the weight value known to yield the highest concordance. Of course, this must be one of the tested values.

When the points are concave down, we can be even greedier than to remain with the weight yielding the highest measured concordance. Inverse parabolic interpolation is used to select the new weight value at the apex of the parabola that fits through the three points, in the hope that this will lead us to the highest τ in the region.

Once this procedure has been performed for all of the weights being tuned, it is possible to postprocess the weight changes, for instance to normalize them. However, this does not seem to be necessary. The chosen values now become the new base vector for the next iteration.

As with typical gradient ascent algorithms, the step size made along the slope calculated when the concordances are clearly ascending or descending slowly decreases throughout the execution of the algorithm.

It remains to be said that simpler variations of this hill climbing algorithm that sampled only one point per weight or that did not specially handle the roughly flat and concave cases fare more poorly in practice than the one presented here.

5.2 Distributed Computation

The work presented in §4.3 allows us to compute τ from two lists of preferences in negligible time. However, establishing the level of concordance at sampled weight vectors nonetheless dominates the running time of the gradient ascent, because computing the machine assessments for each of the training positions is expensive.

Distributing the work for a single weight vector would lead to a large communication overhead, because all of the preferences must be made available to a single machine to compute τ. Instead, we compute all assessments for a weight vector on a single processor, after which the concordance is immediately computed.

A supervisor process queues work in a database table to be performed by worker processes. It then sleeps, waking up periodically to check if all concordances have been computed. By contrast, worker processes idle until they find work in the queue that they can perform. They reserve it, perform it, store the value of τ computed, and then look for more work to do. Once all concordances have been computed, the supervisor process logs the old base vector and its concordance. Additionally, when any tested weight vector's concordance exceeds the best concordance yet present in the work history, the tested weight vector with maximum concordance is also recorded. The supervisor process then computes the new current base vector, computes the new test weight vectors, and replaces the old material in the work queue with the new.

Occasionally a worker process does not report back, for instance, when the machine it was running on has been rebooted. The supervisor process cancels the work reservation when a worker does not return a result within a reasonable length of time, so that another worker may perform it.

The low coupling between the supervisor and worker processes is based upon the work by Pinchak et al. on placeholder scheduling [26].

6 Application-Specific Issues

Here we detail some points of interest of the experimental design.

6.1 Chess Engine

Many chess programs, or chess engines, exist. Some are commercially available; most are hobbyist. For our work, we selected Crafty, by Robert Hyatt [18] of the University of Alabama. Crafty is the best chess engine choice for our work for several reasons: the source was readily available to us, facilitating experimentation; it is the strongest such open-source engine today; previous research has already been performed using Crafty. Most of our work was performed with version 19.1 of the program.

6.2 Training Data

To assess the correlation of τ with improved play, we used 649,698 positions from Chess Informant 1 through 85 [29]. These volumes cover the important chess games played between January 1966 and September 2002. This data set was selected because it contains a variety of assessed positions from modern grandmaster play, the assessments are made by qualified individuals, it is accessible in a non-proprietary electronic form, and chess players around the world are familiar with it.

The 649,698 positions are distributed amongst the human assessment categories and side to move as shown in Table 10.

Table 10: Distribution of Chess Informant Positions

human assessment	black is to move	white is to move	total positions
+−	153,182	1,720	154,902
±	123,261	6,513	130,134
⩲	65,965	15,543	81,508
=	35,341	72,737	108,078
⩱	5,205	32,775	37,980
∓	2,522	55,742	58,264
−+	889	78,303	79,192

It is evident from the distribution of positions shown that when annotating games, humans are far more likely to make an assessment in favour of one player after that player has just moved. (When the assessment is one of equality, the majority of the time it is given after each player has played an equal number of moves.) It is plausible that a machine-learning algorithm could misinterpret this habit to deduce that it is disadvantageous to be the next to play. We postpone a discussion of the irregular number of positions in each category until §7.2.2.

The "random sample" is a randomly selected 32,768-position subset of the 649,698 positions. The "stratified sample" is a stratified random sample of the 649,698 positions, including 2,341 positions of each category and side to move. In two categories, where 2,341 positions were not available, 4,194,304-node searches were performed from positions with the required assessment but the opposite side to move. The best move found by Crafty was played, and the resulting position was used. It is not guaranteed that the move made was not a mistake that would change the human's assessment of the position. However, it is guaranteed that no position and its computationally generated successor were both used.

There are alternate ways that the positions could have been manipulated to generate the stratified sample, none appearing to have significant benefits over the approach chosen here. However, adjusting the weight assigned to each example in inverse proportion to the frequency of its human assessment, as is possible via the weighting procedure given in §4.5, is a serious alternative. This was not done because it was deemed worthwhile to learn from more positions, and because our code to compute Kendall's τ has not been frequently exercised with non-uniform weights.

6.3 Test Suites

English chess grandmaster John Nunn developed the Nunn and Nunn II test suites [24] of 10 and 20 positions, respectively. They serve as starting positions for matches between computer chess programs, where the experimenter is interested in the engine's playing skill independent of the quality of its opening book. Nunn selected positions that are approximately balanced, commonly occur in human games, and exhibit variety of play. We refer to these collectively as the "Nunn 30".

Don Dailey, known for his work on the computer chess programs StarSocrates and CilkChess, created a collection of two hundred commonly reached positions, all of which are ten ply from the initial position. We refer to these collectively as the "Dailey 200".

6.4 Use of Floating-Point Computation

We modified Crafty so that variables holding machine assessments are declared to be of an aliased type rather than directly as integers. This allows us to choose whether to use floating-point or integer arithmetic via a compilation switch. When compiled to use floating-point values for assessments, Crafty is slower, but only by a factor of two to three on a typical personal computer. Experiments were performed with this modified version: the use of floating-point computation provides a learning environment where small changes in values can be rewarded. However, all test matches were performed with the original, integer-based evaluation implementation (the learned values were rounded to the nearest integer): in computer chess competition, no author would voluntarily take a 2% performance hit, much less one of 200%.

It might strike the reader as odd that we chose to alter Crafty in this manner rather than scaling up all the evaluation function weights. There are significant practical disadvantages to that approach. How would we know that everything had been scaled? It would be easy to miss some value that needed to be changed. How would we identify overflow issues? It might be necessary to switch to a larger integer type. How would we know that we had scaled up the values far enough? It would be frustrating to have to repeat the procedure.

By contrast, the choice of converting to floating-point is safer. Precision and overflow are no longer concerns. Also, by setting the typedef to be a non-arithmetic type we can cause the compiler to emit errors wherever type mismatches exist. Thus, we can be more confident that our experiments rest upon a sound foundation.

6.5 Search Effort Quantum

Traditionally, researchers have used search depth to quantify search effort. For our learning algorithm, doing so would not be appropriate: the amount of effort required to search to a fixed depth varies wildly between positions, and we will be comparing the assessments of these positions. However, because we did not have the dedicated use of computational resources, we could not use search time either. While it is known that chess engines tend to search more nodes per second in the endgame than the middlegame, this difference is insignificant for our short searches because it is dwarfed by the overhead of preparing the engine to search an arbitrary position. Therefore, we chose to quantify search effort by the number of nodes visited.

For the experiments reported here, we instructed Crafty to search either 1024 or 16,384 nodes to assess a position. Early experiments that directly called the static evaluation or quiescence search routines to form assessments were not successful. Results with 1024 nodes per position were historically of mixed quality, but improved as we improved the estimated gradient ascent procedure. It is our belief that with a larger training set, calling the static evaluation function directly will perform acceptably.

There are positions in our data set from which Crafty does not complete a 1-ply search within 16,384 nodes, because its quiescence search explores many sequences of captures. When this occurs, no evaluation score is available to use. Instead of using either zero or the statically computed evaluation (which is not designed to operate without a quiescence search), we chose to throw away the data point for that particular computation of τ, reducing the position count (m). However, the value of τ for similar data of different population sizes is not necessarily constant. As feature weights are changed, the shape of the search tree for positions may also change. This can cause Crafty to not finish a 1-ply search for a position within the node limit where it was previously able to do so, or vice versa. When many transitions in the same direction occur simultaneously, noticeable irregularities are introduced into the learning process. Ignoring the node count limitation until the first ply of search has been completed may be a better strategy.

6.6 Performance

Experiments were first performed using idle time on various machines in our department. In the latter stages of our research, we have had (non-exclusive) access to clusters of personal computer workstations. This is helpful because, as discussed in §5.2, the task of computing τ for distinct weight vectors within an iteration is trivially parallel. Examining 32,768 positions at 1024 nodes per position and computing τ takes about two minutes per weight vector. The cost of computing τ is negligible in comparison, so in the best case, when there are enough nodes available for the concordances of all weight vectors of an iteration to be computed simultaneously, learning proceeds at the rate of 30 iterations per hour.

7 Experimental Results

After demonstrating that concordance between human judgments and machine assessments increases with increasing depth of machine search, we attempt to tune the weights of 11 important features of the chess program Crafty.

7.1 Concordance as Machine Search Effort Increases

In Table 11 we computed τ for depths 1 through 10 for n = 649,698 positions, performing work equivalent to 211 billion (10^9) comparisons at each depth. S+ and S– are reported in billions. As search depth increases, the difference between S+ and S–, and therefore τ, also increases. The sum of S+ and S– is not constant because at different depths different amounts of extra y-pairs and duplicate pairs are encountered.

Table 11: τ Computed for Various Search Depths, n = 649,698

depth	S+ / 10^9	S- / 10^9	τ
1	110.374	65.298	0.2136
2	127.113	48.934	0.3705
3	131.384	45.002	0.4093
4	141.496	36.505	0.4975
5	144.168	34.726	0.5186
6	149.517	30.136	0.5656
7	150.977	29.566	0.5753
8	152.792	22.938	0.6153
9	153.341	22.368	0.6206
10	155.263	20.435	0.6388

It is difficult to predict how close an agreement might be reached using deeper searches. Two effects come into play: diminishing returns from additional search, and diminishing accuracy of human assessments relative to ever more deeply searched machine assessments. Particularly interesting is the odd-even effect on the change in τ as depth increases. It has long been known that searching to the next depth of an alpha-beta search requires relatively much more effort when that next depth is even than when it is odd [22]. Notably, τ tends to increase more in precisely these cases.

This result, combined with knowing that play improves as search depth increases [42], in turn justifies our attempt to use this concordance as a metric to tune selected feature weights of Crafty's static evaluation function. That the concordance increases monotonically with increasing depth lends credibility to our belief that τ is a direct measure of decision quality.

7.2 Tuning of Crafty's Feature Weights

Crafty uses centipawns (hundredths of a pawn) as its evaluation function resolution, so experiments were performed by playing Crafty as distributed versus Crafty with the learned weights rounded to the nearest centipawn. Each program played each position both as White and as Black. The feature weights we tuned are given, along with their default values, in Table 12.

Table 12: Tuned Features, with Crafty's Default Values

feature	default value
king safety scaling factor	100
king safety asymmetry scaling factor	-40
king safety tropism scaling factor	100
blocked pawn scaling factor	100
passed pawn scaling factor	100
pawn structure scaling factor	100
bishop	300
knight	300
rook on the seventh rank	30
rook on an open file	24
rook behind a passed pawn	40

The six selected scaling factors were chosen because they act as control knobs for many subterms. Bishop and knight were included because they participate in the most common piece imbalances. Trading a bishop for a knight is common, so it is important to include both to show that one is not learning to be of a certain weight chiefly because of the weight of the other. We also included three of the most important positional terms involving rooks. Material values for the rook and queen are not included because trials showed that they climbed even more quickly than the bishop and knight do, yielding no new insights. This optimization problem is not linear: dependencies exist between the weights being tuned.

7.2.1 Tuning from Arbitrary Values

Figure 4 illustrates the learning. The 11 parameters were all initialized to 50, where 100 represents both the value of a pawn and the default value of most scaling factors. For ease of interpretation, the legends of Figures 4, 5, and 6 are ordered so that its entries coincide with the intersection of the variable being plotted with the rightmost point on the x-axis. For instance, in Figure 4, bishop is the topmost value, followed by knight, then τ, and so on. τ is measured on the left y-axis in linear scale; weights are measured on the right y-axis in logarithmic scale, for improved visibility of the weight trajectories.

Rapid improvement is made as the bishop and knight weights climb swiftly to about 285, after which τ continues to climb, albeit more slowly. We attribute most of the improvement in τ to the proper determination of weight values for the minor pieces. All the material and positional weights are tuned to reasonable values.

The scaling factors learned are more interesting. The king tropism and pawn structure scaling factors gradually reached, then exceeded Crafty's default values of 100. The scaling factors for blocked pawns, passed pawns, and king safety are lower, but not unreasonably so. However, the king safety asymmetry scaling factor dives quickly and relentlessly. This is unsurprising, as Crafty's default value for this term is –40.

Tables 13 and 14 contain match results of the weight vectors at specified iterations during the learning illustrated in Figure 4. Each side plays each starting position both as White and as Black, so with the Nunn 30 test, 60 games are played, and with the Dailey 200 test, 400 games are played. Games reaching move 121 were declared drawn.

Figure 4: Change in Weights from 50 as τ is Maximized, Random Sample

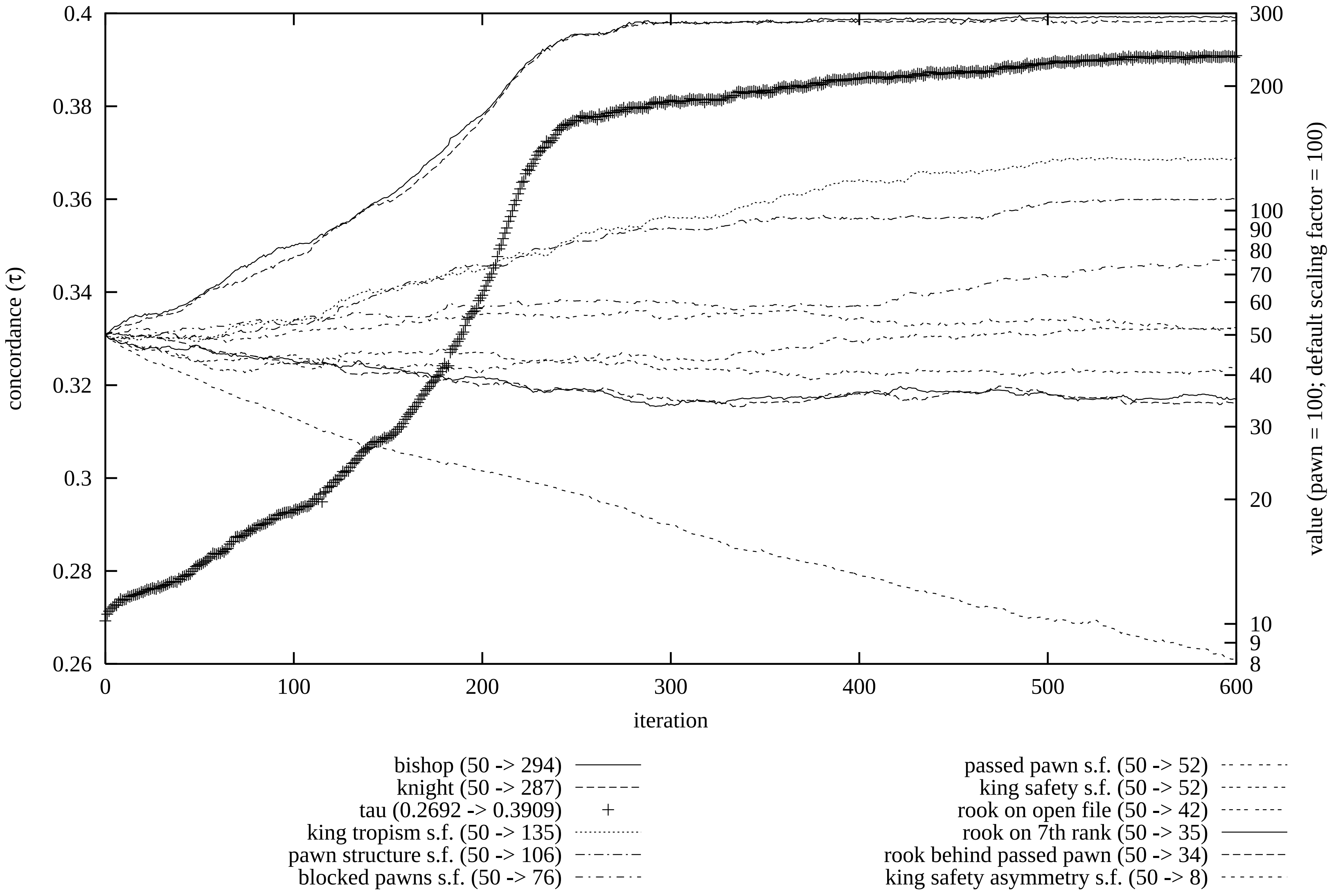

Table 13: Match Results: random sample; 16,384 nodes per assessment; 11 weights tuned from 50 vs. default weights; 5 minutes per game; Nunn 30 test suite.

iteration	wins	draws	losses	percentage score
0	3	1	56	5.83
100	3	9	48	12.50
200	14	21	25	40.83
300	21	26	13	56.67
400	19	28	13	55.00
500	18	26	16	51.67
600	18	23	19	49.17

Table 14: Match Results for random sample; 16,384 nodes per assessment; 11 weights tuned from 50 vs. default weights; 5 minutes per game; Dailey 200 test suite.

iteration	wins	draws	losses	percentage score
0	3	13	384	2.38
100	12	31	357	6.88
200	76	128	196	35.00
300	128	152	120	51.00
400	129	143	128	50.13
500	107	143	150	44.63
600	119	158	123	49.50

The play of the tuned program improves dramatically as learning occurs. Of interest is the apparent gradual decline in percentage score for later iterations on the Nunn 30 test suite. The Deep Thought team [1, 2, 17, 25] found that their best parameter settings were achieved before reaching maximum agreement with GM players. Perhaps we are also experiencing this phenomenon. We used the Dailey 200 test suite to attempt to confirm that this was a real effect, and found that by this measure too, the weight vectors at iterations 300 and 400 were superior to later ones.

We conclude that the learning procedure yielded weight values for the variables tuned that perform comparably to values tuned by hand over years of games versus grandmasters. This equals the performance achieved by Schaeffer et al. [33].

7.2.2 Tuning from Crafty’s Default Values

We repeated the just-discussed experiment with one change: the feature weights start at Crafty’s default values rather than at 50. Figure 5 depicts the learning. Note that we have negated the values of the king safety asymmetry scaling factor in the graph so that we could retain the logarithmic scale on the right y-axis, and for another reason, for which see below.

While most values remain normal, the king safety scaling factor surprisingly rises to almost four times the default value. Meanwhile, the king safety asymmetry scaling factor descends even below -100. The combination indicates a complete lack of regard for the opponent’s king safety, but great regard for its own. Table 15 shows that this conservative strategy is by no means an improvement.

Figure 5: Change in Weights from Crafty's defaults as τ is Maximized, Random Sample

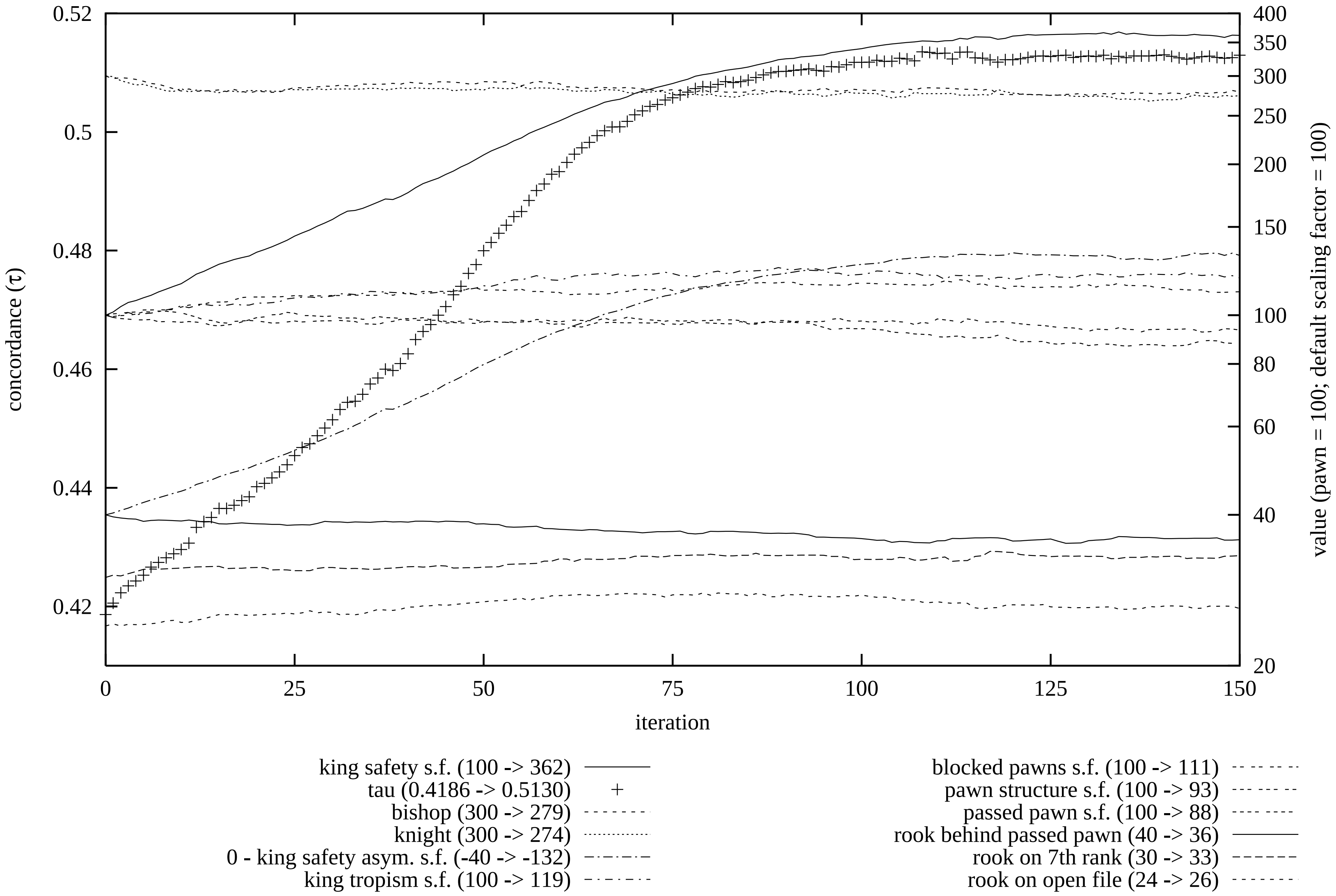

Table 15: Match Results: random sample; 16,384 nodes per assessment; 11 weights tuned from defaults vs. default weights; 5 minutes per game; Nunn 30 test suite.

iteration	wins	draws	losses	percentage score
25	19	23	18	50.83
50	16	31	13	52.50
75	11	32	17	45.00
100	14	28	18	46.67
125	9	23	28	34.17
150	8	35	17	42.50

The most unusual behaviour of the king safety and king safety asymmetry scaling factors deserves specific attention. When the other nine terms are left constant, these two terms behave similarly to how they do when all eleven terms are tuned. In contrast, when these two terms are held constant, no significant performance difference is found between the learned weights and Crafty's default weights. When the values of the king safety asymmetry scaling factor are negated as in Figure 5, it becomes visually clear from their trajectories that the two terms are behaving in a codependent manner.

Having identified this anomalous behaviour, it is worth looking again at Figure 4. The match results suggest that all productive learning occurred by iteration 400 at the latest, after which a small but perceptible decline appears to occur. The undesirable codependency between the king safety and king safety asymmetry scaling factors also appears to be present in the later iterations of the first experiment.

Table 10 in §6.2 shows that there are a widely varying number of positions in each category. Let us consider the effect this has upon our optimization metric. Our procedure maximizes the difference between the number of correct decisions made and the number of incorrect decisions made. Each position is thought to be of equal importance, but there are more positions in certain categories. Our optimization does not take into account that with a lopsided sample, correct decisions involving human assessments that occur less frequently will be sacrificed to achieve a higher number of correct decisions involving those that occur more often. Undesirably, weights will be tuned not in accordance with strong play over the complete range of human assessments, but instead to maximize decision quality amongst the overrepresented human assessments.

Accordingly, we retried tuning weights from their default values, but using the stratified sample so that the learning would be biased appropriately. Additionally, a look-ahead limit of just 1024 nodes was used, trading away machine assessment accuracy to gain an increased number of iterations, because we desire to demonstrate playing strength stability. We can see that the codependency is no longer present in Figure 6. Validating our thought experiment regarding the reason for its previous existence, the match results in Table 16 demonstrate that the learner is now performing acceptably.

Compared against earlier plots, the trajectory of τ has a relatively jagged appearance. Unlike in the other two graphs, here the concordance is not climbing at a significant rate. Consequently, the left y-axis has been set to provide a high level of detail. In addition, because only 1024 nodes of search are being used to form evaluations instead of 16,384, changes in weights are more likely to affect the score at the root of the tree, causing τ to fluctuate more. This effect can be countered by using a larger training set.

Figure 6: Change in Weights from Crafty’s Defaults as τ is Maximized, Stratified Sample

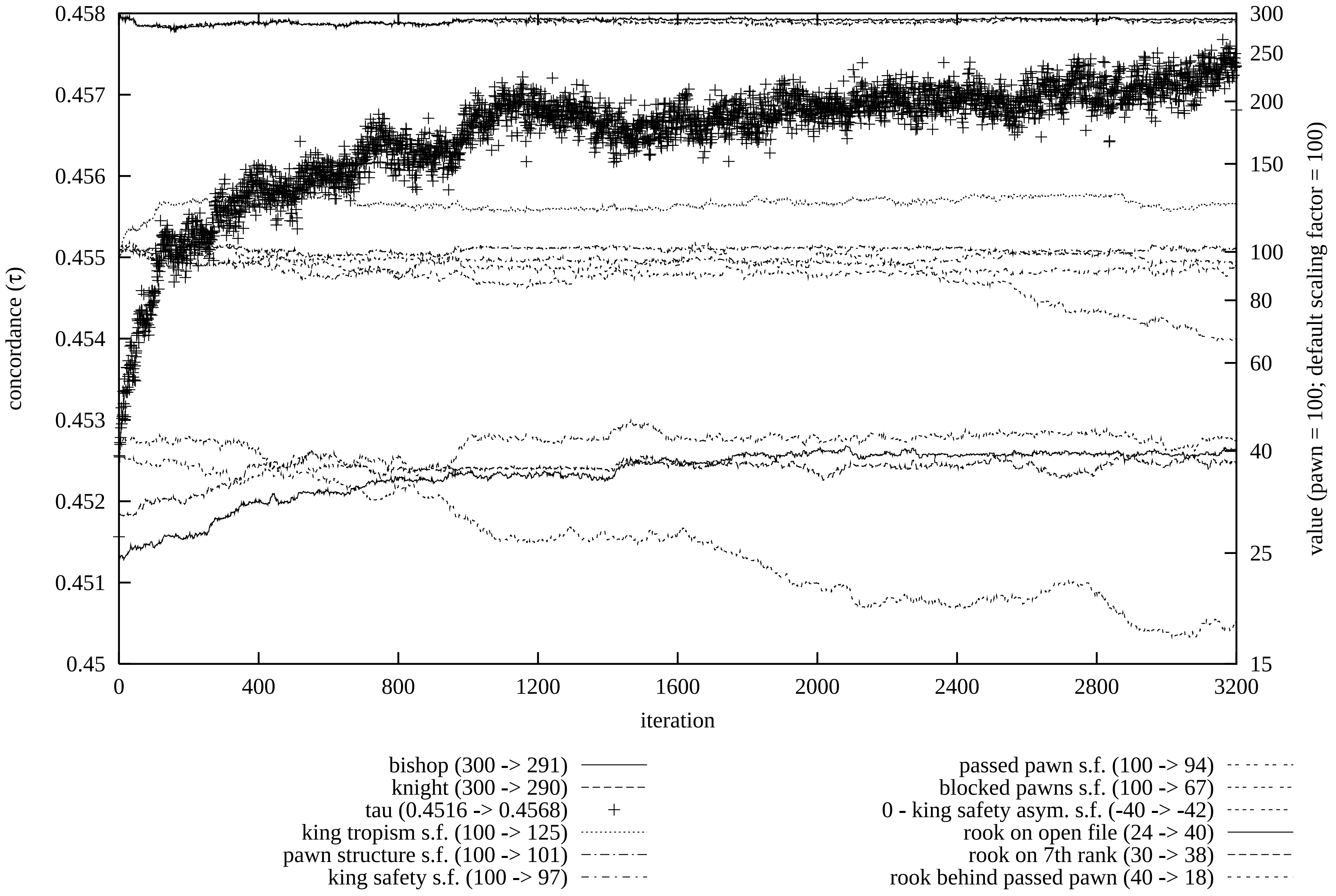

Table 16: Match Results: stratified sample; 1024 nodes per assessment; 11 weights tuned from defaults vs. default weights; 5 minutes per game; Nunn 30 test suite.

iteration	wins	draws	losses	percentage score
0	14	36	10	53.33
200	17	26	17	50.00
400	9	26	25	36.67
600	18	21	21	47.50
800	14	25	21	44.17
1000	17	28	15	51.67
1200	18	26	16	51.67
1400	16	30	14	51.67
1600	21	19	20	50.83
1800	9	34	17	43.33
2000	14	33	13	50.83
2200	13	27	20	44.17
2400	15	30	15	50.00
2600	11	31	18	44.17
2800	14	30	16	48.33
3000	13	33	14	49.17
3200	15	29	16	49.17

Most weights remained relatively constant during the tuning. The king tropism term climbed quickly to the neighbourhood of 125 centipawns, then remained roughly level; this is in line with previous tuning attempts. Also, both placing a rook on an open file and posting a rook upon the seventh rank again reached the region of thirty-five to forty hundredths of a pawn.

The value of placing a rook behind a passed pawn descended considerably over the learning period, which is interesting insofar as this is thought to be an important feature by humans. Probably the reason is that its application is insufficiently specific: there are many positions where it is slightly beneficial, but occasionally it can be a key advantage. Rather than attempt to straddle the two cases, as Crafty's default value for the weight, 40, does, it could be preferable to give a smaller bonus, but include a second, more specific feature that provides an extra reward when its additional criteria are satisfied.

Finally, we are left with the blocked pawns scaling factor. For a long time, this weight remained near its default of 100, but as of approximately iteration 2200 it began to fall, and continued doing so for 1000 iterations thereafter, with no clear indication that it would stop doing so anytime soon. The purpose of this term is for Crafty to penalize itself when there are many pawns that are blocked, so that it avoids such positions when playing humans, who have a comparative skill advantage in such positions.

It is no surprise that when tuning against human assessments, this feature weight would go down, because it is not bad per se for a position to be blocked. Here we have not merely a case where the program author has a different objective than the metric proposed, but a philosophical difference between Dr. Hyatt, the program's author, and ourselves. The author has set his weights to provide the best performance in his test environment, which is playing against humans on internet chess servers. This is an eminently reasonable decision for someone who feels that their time is better expended on improving their program's parallel search than its static evaluation function. We believe that a better long-term strategy is to tackle the evaluation problem head-on: do not penalize playing into perfectly acceptable positions that the machine subsequently misplays. Instead, accept the losses, and implement new evaluation features that shore up its ability to withstand such positions against humans. In the end, the program will be stronger for it.

The match results do not indicate a change in playing strength due to this weight changing, which is unsurprising given that if such positions were reached, neither the standard nor the tuned Crafty would have any advantage over the other in understanding them. Both would be at a relative disadvantage when playing a program that included additional evaluation features as suggested above.

8 Conclusion

We have proposed a new procedure for optimizing static evaluation functions based upon globally ordering a multiplicity of positions in a consistent manner. This application of ordinal correlation is fundamentally different from prior evaluation function tuning techniques that either attempt to select the best moves in certain positions and hope that this skill is generalizable, or to bootstrap from zero information.

If we view the playing of chess as a Markov decision process, we can say that rather than attempting to learn policy (what move to play) directly, we attempt to learn an appropriate value function (how good is the position?), which in turn specifies the policy to be followed when in particular states. Thus, we combine an effective idea behind temporal difference learning with supervised learning.

Alternatively, we can view the preferences over the set of pairs of positions as constraints: we want to maximize the number of inequalities that are satisfied. As shown in §4, our method powerfully leverages m position evaluations into $m(m-1)/2$ constraints.

A cautionary note: we tuned feature weights in accordance with human assessments. Doing so may simply not be optimal for computer play. Nonetheless, it is worth noting that having reduced the playing ability of a grandmaster-level program to candidate master strength by significantly altering several important feature weights, the learning algorithm was able to restore the program to grandmaster strength.

8.1 Reflection

While some weights learned nearly identical values in both experiments, other features exhibited more variance. For cases such as the blocked pawns scaling factor, it appears that comparable performance may be achieved with a relatively wide range of values.

The amount of training data used is small enough that overfitting may be a consideration. The program modification that would most readily allow this to be determined is to have the program compute τ on a second, control set of positions that are used only for testing. If the concordance for the training set continues to climb while the concordance for the control set declines, overfitting will be detected.

The time to perform our experiments was dominated by the search effort required to generate machine assessments. Therefore, there is no significant obstacle to attempting to maximize Spearman's ρ (or perhaps even Pearson correlation, notwithstanding Stevens).

Future learning experiments should ultimately use more positions, because the information gathered to make tuning decisions grows quadratically as the position set grows. We believe that doing this will reduce the search effort required per position to tune weights well. If sufficient positions are present for tuning based only upon the static evaluation can be performed, much time can be saved. For instance, piece square tables for the six chess pieces imply 6 * 64 = 384 features, but for any single position, only a maximum of 32 can have any effect. Furthermore, Crafty's evaluation is actually more dependent upon table lookup than this simple example shows.

Furthermore, it is not necessary to use the entire training set at each iteration of the algorithm. One could select different samples for each iteration, thereby increasing the total number of positions that strong weight vectors are compared against. Alternatively, one can start with very few positions, perhaps 1024, and slowly increase the number of positions under consideration as the number of iterations increases. This would speed the adjustment of important weights that are clearly set to poor values according to relatively few examples before considering more positions for finer tuning. Combining these ideas is also possible.

Precomputing and making a copy of Crafty's internal board representations for each of the test positions could yield a further significant speedup. When computing the concordance for a weight vector, it would be sufficient to perform one large memory copy, then to search each position in turn. If the static evaluation function is called directly, then it is even possible to precompute a symbolic representation of the position. A training time speedup of orders of magnitude should be possible with these changes.

On the other hand, it was not originally planned to attempt to maximize τ only upon assessments at a specific level of search effort. Unfortunately, we encountered implementation difficulties, and so reverted to the approach described herein. We had intended to log the node number or time point along with the new score whenever the evaluation of a position changes. This would have, without the use of excessive storage, provided the precise score at any point throughout the search. We would have tuned to maximize the integral of τ over the period of search effort. Implementation of this algorithm would more explicitly reward reaching better evaluations more quickly, improving the likelihood of tuning feature weights and perhaps even search control parameters effectively. Here too, precomputation would be worthwhile.

The gradient ascent procedure is effective, but convergence acceleration is noticeably lacking. It is somewhat of an arduous climb for parameters to climb from 50 to near 300, not because there was any doubt that they would reach there, but simply because of the number of iterations required for them to move the requisite distance. Fortunately, this is less likely to be a factor in practical use, where previously tuned weights likely would be started near their existing values.

Perhaps the most problematic issue with the gradient ascent implementation occurs when weights are near zero, where it is no longer sufficient to generate a sample point based on multiplying the current weight by a value slightly greater than one. Enforcing a strict minimum distance between the existing weight and its test points is not sufficient: the weight may become trapped. We inject some randomness into the sampling procedure when near zero, but it would be premature to say that this is a complete solution.

It would be worthwhile to attempt approaches other than gradient ascent. The original motivation for implementing the hill-climber was to be able to observe how our fitness metric performed. It was felt that a gradient ascent procedure would provide superior insight into its workings relative to an evolutionary method approach due to its more predictable behaviour. As previously mentioned in §7.2, Crafty uses weights expressed in centipawns, so when we test weights with Crafty, we round them to the nearest centipawn. Sometimes much time is spent tuning a weight to make small changes that will be lost when this truncation of precision occurs. If an alternative method, for instance an evolutionary method, can succeed while operating only at the resolution of centipawns, strong feature weight vectors could be identified more quickly.

Schaeffer et al. [33] found that when using temporal difference learning, it is best to tune weights at the level of search effort that one will be applying later. We believe this is because the objective function is constantly updated as part of that learning algorithm. In contrast, with the procedure presented herein, the objective function is determined separately, in advance. Therefore, if one chooses to use a specific level of search effort to determine the pseudo-oracle values of states as described in §4.6, then directly tuning the static evaluation function should yield values that perform well in practice when the program plays with a similar level of search effort.

It may be argued that deeper searches would lead to different preferred weights because more positions of disparate character will be reached by the search. However, this too can be mimicked by increasing the number of positions in the training set.

A pseudo-oracle is not an oracle, so when the value of τ improves significantly, it makes sense to redetermine the objective function using the new weights that have been learned, and resume the gradient ascent procedure (without resetting the weight values, of course). Temporal difference learning elegantly finesses this issue precisely because the objective function is augmented at each learning step.

8.2 Future Directions

While our experiments used chess assessments from humans, it is possible to use assessments from deeper searches and/or from a stronger engine, or to tune a static evaluation function for a different domain. Depending on the circumstances, merging consecutively ordered fine-grained assessments into fewer, larger categories might be desirable. Doing so could even become necessary should the computation of τ dominate the time per iteration, but this is unlikely unless one uses both an enormous number of positions and negligible time to form machine assessments.

Examining how concordance values change as the accuracy of machine assessments is artificially reduced would provide insight into the amount of precision that the heuristic evaluation function should be permitted to express to the search framework that calls it. Too much precision reduces the frequency of cut-offs, while insufficient precision would result in the search receiving poor guidance.

Elidan et al. [10] found that perturbation of training data could assist in escaping local maxima during learning. Our implementation of τ, designed with this finding in mind, allows non-integer weights to be assigned to each cell. Perturbing the weights in an adversarial manner as local maxima are reached, so that positions are weighted slightly more important when generally discordant, and slightly less important when generally concordant, could allow the learner to continue making progress.

It would also be worthwhile to examine positions of maximum disagreement between human and machine assessments, in the hope that study of the resulting positions will identify new features that are not currently present in Crafty's evaluation. Via this process, a number of labelling errors would be identified and corrected. However, because our metric is robust in the presence of outliers, we do not believe that this would have a large effect on the outcome of the learning process. Improvements in data accuracy could allow quicker learning and a superior resulting weight vector, though we suspect the differences would be small.

Integrating the supervised learning procedure developed here with temporal difference learning, as suggested by Utgoff and Clouse [46], would be interesting.

A popular pastime amongst computer chess hobbyists is to attempt to discover feature weight settings that result in play mimicking their favourite human players. By tuning against appropriate training data, e.g., from opening monographs and analyses published in Chess Informant and elsewhere that are authored by the player to be mimicked, training an evaluation function to assess positions similarly to how a particular player might actually do so should now be possible.

Furthermore, there are human chess annotators known for their diligence and accuracy, and others notorious for their lack thereof. Computing values of τ pitting an individual's annotations against the assessments that a program makes after a reasonably amount of search would provide an objective metric of the quality of their published analysis.

Producers of top computer chess software play many games against their commercial competitors. They could use our method to model their opponent's evaluation function, then use this model in a minimax (no longer negamax) search. Matches then played would be more likely to reach positions where the two evaluation functions differ most, providing improved winning chances for the program whose evaluation function is more accurate, and object lessons for the subsequent improvement of the other. For this particular application, using a least-squares error function rather than Kendall's τ could be appropriate, if the objective is to hone in on the exact weights used by the opponent.[7]

[7] More than one commercial chess software developer was somewhat perturbed by this possibility after the presentation of our earlier paper [13] at the Advances in Computer Games 10 conference, which was held in conjunction with the 2003 World Computer Chess Championship. Newer versions of some programs will be attempting to conceal information that makes such reverse engineering possible. Ideas that were discussed included not outputting the principal variation at low search depths, truncating the principal variation early when it is displayed, declining to address hashing issues that occasionally cause misleading principal variations to be emitted, and mangling the low bits of the evaluation scores reported to users.

Identifying the most realistic mapping of Crafty's machine assessments to the seven human positional assessments is also of interest. This information would allow Crafty (or a graphical user interface connected to Crafty) to present scoring information in a human-friendly format alongside the machine score.

We can measure how correlated a single feature is with success by setting the weight of that feature to unity, setting all other feature weights to zero, and computing Kendall's τ. This creates the possibility of applying τ as a mechanism for feature selection. However, multiple features may interact: when testing the combined effective discriminating power of two or three features it would be necessary to hold one constant and optimize the weights of the others.

The many successes of temporal difference learning have demonstrated that it can tune weights equal to the best hand-tuned weights. We believe that, modulo the current performance issues, the supervised learning approach described here is also this effective. Learning feature weights is something that the AI community knows how to do well. While there are undoubtedly advances still to be made with respect to feature weight tuning, for the present it is time to refocus on automated feature generation and selection, the last pieces of the puzzle for fully automated evaluation function construction.

Acknowledgements

We would like to thank: Yngvi Björnsson, for the use of his automated game-playing software, and for fruitful discussions; Don Dailey, for access to his suite of 200 test positions; Robert Hyatt, for making CRAFTY available, and also answering questions about its implementation; Peter McKenzie, for providing PGN to EPD conversion software; NSERC, for partial financial support [Grant OPG 7902 (Marsland)].

References

[1] Anantharaman, T. S. (1990). A Statistical Study of Selective Min-Max Search in Computer Chess. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA. University Report CMU-CS-90-173.

[2] Anantharaman, T. S. (1997). Evaluation Tuning for Computer Chess: Linear Discriminant Methods. ICCA Journal, Vol. 20, No. 4, pp. 224-242.

[3] Baxter, J., Tridgell, A., and Weaver, L. (1998). KnightCap: A Chess Program that Learns by Combining TD(λ) with Game-tree Search. Proceedings of the Fifteenth International Conference in Machine Learning (IMCL) pp. 28-36, Madison, WI.

[4] Beal, D. F. and Smith, M. C. (1997). Learning Piece Values Using Temporal Differences. ICCA Journal, Vol. 20, No. 3, pp. 147-151.

[5] Beal, D. F. and Smith, M. C. (1999a). Learning Piece-Square Values using Temporal Differences. ICCA Journal, Vol. 22, No. 4, pp. 223-235.

[6] Beal, D. F. and Smith, M. C. (1999b). First Results from Using Temporal Difference Learning in Shogi. Computers and Games (eds. H. J. van den Herik and H. Iida), pp. 113-125. Lecture Notes in Computer Science 1558, Springer-Verlag, Berlin, Germany.

Commercial chess software developers must walk a fine line between effectively hindering reverse engineering by their competitors and displeasing their customers.

[7] Buro, M. (1995). Statistical Feature Combination for the Evaluation of Game Positions. Journal of Artificial Intelligence Research 3, pp. 373-382, Morgan Kaufmann, San Francisco, CA.

[8] Buro, M. (1999). From Simple Features to Sophisticated Evaluation Functions. Computers and Games (eds. H. J. van den Herik and H. Iida), pp. 126-145. Lecture Notes in Computer Science 1558, Springer-Verlag, Berlin, Germany.

[9] Cliff, N. (1996). Ordinal Methods for Behavioral Data Analysis. Lawrence Erlbaum Associates.

[10] Elidan, G., Ninio, M., Friedman, N., and Schuurmans, D. (2002). Data Perturbation for Escaping Local Maxima in Learning. Proceedings AAAI 2002 Edmonton, pp. 132-139.

[11] Fawcett, T. E. and P. E. Utgoff (1992). Automatic feature generation for problem solving systems. Proceedings of the Ninth International Conference on Machine Learning, pp. 144-153. Morgan Kaufman.

[12] Fürnkranz, J. (2001). Machine Learning in Games: A Survey. In Machines that Learn to Play Games (eds. J. Fürnkranz and M. Kubat), pp. 11-59. Nova Scientific Publishers. ftp://ftp.ai.univie.ac.at/papers/oefai-tr-2000-31.pdf

[13] Gomboc, D., Marsland, T. A., and Buro, M. (2003). Ordinal Correlation for Evaluation Function Tuning. Advances in Computer Games: Many Games, Many Challenges (eds. H. J. van den Herik et al.), pp. 1-18. Kluwer Academic Publishers.

[14] Hartmann, D. (1987a). How to Extract Relevant Knowledge from Grandmaster Games, Part 1: Grandmasters have Insights – the Problem is What to Incorporate into Practical Programs. ICCA Journal, Vol. 10, No. 1, pp. 14-36.

[15] Hartmann, D. (1987b). How to Extract Relevant Knowledge from Grandmaster Games, Part 2: The Notion of Mobility, and the Work of De Groot and Slater. ICCA Journal, Vol. 10, No. 2, pp. 78-90.

[16] Hartmann, D. (1989). Notions of Evaluation Functions tested against Grandmaster Games. In Advances in Computer Chess 5 (ed. D.F. Beal), pp. 91-141. Elsevier Science Publishers, Amsterdam, The Netherlands.

[17] Hsu, F.-h., Anantharaman, T. S., Campbell, M. S., and Nowatzyk, A. (1990). Deep Thought. In Computers, Chess, and Cognition (eds. T. A. Marsland and J. Schaeffer), pp. 55-78. Springer-Verlag.

[18] Hyatt, R.M. (1996). Crafty – Chess Program. ftp://ftp.cis.uab.edu/pub/hyatt/v19/crafty-19.1.tar.gz.

[19] Kaneko, T., Yamagucki, K., and Kawai, S. (2003). Automated Identification of Patterns in Evaluation Functions. Advances in Computer Games: Many Games, Many Challenges (eds. H. J. van den Herik et al.), pp. 279-298. Kluwer Academic Publishers.

[20] Kendall, G. and Whitwell, G. (2001). An Evolutionary Approach for the Tuning of a Chess Evaluation Function. Proceedings of the 2001 IEEE Congress on Evolutionary Computation. http://www.cs.nott.ac.uk/~gxk/papers/cec2001chess.pdf.

[21] Levinson, R. and Snyder, R. (1991). Adaptive Pattern-Oriented Chess. AAAI, pp. 601-606.

[22] Marsland, T. A. (1983). Relative Efficiency of Alpha-Beta Implementations. IJCAI 1983, pp. 763-766.

[23] Marsland, T. A. (1985). Evaluation-Function Factors. ICCA Journal, Vol. 8, No. 2, pp. 47-57.

[24] Nunn, J. (1999). http://www.computerschach.de/test/nunn2.htm.

[25] Nowatzyk, A. (2000). http://www.tim-mann.org/deepthought.html.

[26] Pinchak, C., Lu, P., and Goldenberg, M. (2002). Practical Heterogeneous Placeholder Scheduling in Overlay Metacomputers: Early Experiences. 8th Workshop on Job Scheduling Strategies for Parallel Processing, Edinburgh, Scotland, U.K., pp. 85-105, also to appear in LNCS 2537 (2003), pp. 205-228. http://www.cs.ualberta.ca/~paullu/Trellis/Papers/placeholders.jsspp.2002.ps.gz.

[27] Plaat, A., Schaeffer, J., Pijls, W., and Bruin, A. de (1996). Best-First Fixed-Depth Game-Tree Search in Practice. Artificial Intelligence, Vol. 87, Nos. 1-2, pp. 255-293.

[28] Press, W., Teukolsky, S., Vetterling, W., and Flannery, B. (1992). Numerical Recipies in C: The Art of Scientific Computing, Second Edition, pp. 644-645. Cambridge University Press.

[29] Sahovski Informator (1966). Chess Informant: http://www.sahovski.com/.

[30] Samuel, A. L. (1959). Some Studies in Machine Learning Using the Game of Checkers. IBM Journal of Research and Development, No. 3, pp. 211-229.

[31] Samuel, A. L. (1967). Some Studies in Machine Learning Using the Game of Checkers. II – Recent Progress. IBM Journal of Research and Development, Vol. 2, No. 6, pp. 601-617.

[32] Sarle, W. S. (1997). Measurement theory: Frequently asked questions, version 3. ftp://ftp.sas.com/pub/neural/measurement.html. Revision of publication in Disseminations of the International Statistical Applications Institute, Vol. 1, Ed. 4, 1995, pp. 61-66.

[33] Schaeffer, J., Hlynka, M., and Jussila, V. (2001). Temporal Difference Learning Applied to a High-Performance Game-Playing Program. Proceedings IJCAI 2001, pp. 529-534.

[34] Shannon, C. E. (1950). Programming a Computer for Playing Chess. Philosophical Magazine, Vol. 41, pp. 256-275.

[35] Stevens, S. S. (1946). On the theory of scales of measurement. Science, 103, 677-680.

[36] Stevens, S. S. (1951). Mathematics, measurement, and psychophysics. In S. S. Stevens (ed.), Handbook of experimental psychology, pp 1-49). New York: Wiley.

[37] Stevens, S. S. (1959). Measurement. In C. W. Churchman, ed., Measurement: Definitions and Theories, pp. 18-36. New York: Wiley. Reprinted in G. M. Maranell, ed., (1974) Scaling: A Sourcebook for Behavioral Scientists, pp. 22-41. Chicago: Aldine.

[38] Sutton, R. S. (1988). Learning to Predict by the Methods of Temporal Differences. Machine Learning, Vol. 3, pp. 9-44.

[39] Tesauro, G. (1989). Connectionist Learning of Expert Preferences by Comparison Training. Advances in Neural Information Processing Systems 1 (ed. D. Touretzky), pp. 99-106. Morgan Kauffman.

[40] Tesauro, G. (1995). Temporal Difference Learning and TD-Gammon. Communications of the ACM, Vol. 38, No. 3, pp. 55-68. http://www.research.ibm.com/massive/tdl.html.

[41] Tesauro, G. (2001). Comparison Training of Chess Evaluation Functions. In Machines that Learn to Play Games (eds. J. Fürnkranz and M. Kubat), pp. 117-130. Nova Scientific Publishers.

[42] Thompson, K. (1982). Computer Chess Strength. Advances in Computer Chess 3, (ed. M.R.B. Clarke), pp. 55-56. Pergamon Press, Oxford, UK.

[43] Thompson, K. (1986). Retrograde Analysis of Certain Endgames. ICCA Journal, Vol. 9, No. 3, pp. 131-139.

[44] Tournavitis, K. (2002). Mouse(μ): A Self-Teaching Algorithm that Achieved Master-Strength at Othello. In Computers and Games: Third International Conference (CG 2002) (eds. J. Schaeffer et al.), pp. 11-28.

[45] Tukey, J. W. (1962). The Future of Data Analysis. In The Collected Works of John W. Tukey, Vol. 3 (1986) (ed. L. V. Jones), pp. 391-484. Wadsworth, Belmont, CA.

[46] Utgoff, P. E. and Clouse, J. A. (1991). Two Kinds of Training Information for Evaluation Function Learning. AAAI, pp. 596-600.

[47] Utgoff, P. E. (1996). ELF: An evaluation function learner that constructs its own features. Technical Report 96-65, Department of Computing Science, University of Massachusetts, Amherst, MA.

[48] Utgoff, P. E. and D. Precup (1998). Constructive function approximation. In Feature Extraction, Construction, and Selection: a Data-Mining Perspective (eds. Motoda and Liu), pp. 219-235. Kluwer Academic Publishers.

[49] Utgoff, P. E. and D. J. Stracuzzi (1999). Approximation via value unification. In Proceeedings of the Sixteenth International Conference on Machine Learning (ICML), pp. 425-432. Morgan Kaufmann.

[50] Utgoff, P. E. (2001). Feature Construction for Game Playing. In Machines that Learn to Play Games (eds. J. Fürnkranz and M. Kubat), pp. 131-152. Nova Scientific Publishers.

[51] Utgoff, P.E. and D. J. Stracuzzi (2002). Many-layered learning. In Neural Computation, Vol. 14, pp. 2497-2539.

[52] van der Meulen, M. (1989). Weight Assessment in Evaluation Functions. In Advances in Computer Chess 5 (ed. D.F. Beal), pp. 81-90. Elsevier Science Publishers, Amsterdam, The Netherlands.

[53] Velleman, P. and Wilkinson, L. (1993). Nominal, Ordinal, Interval, and Ratio Typologies are Misleading. http://www.spss.com/research/wilkinson/Publications/Stevens.pdf. A previous version appeared in The American Statistician, Vol. 47, No. 1, pp. 65-72.