

Experiments in Parameter Learning Using Temporal Differences

Jonathan Baxter
Department of Systems Engineering
Australian National University
Canberra 0200, Australia

Andrew Tridgell
Department of Computer Science
Australian National University
Canberra 0200, Australia

Lex Weaver
Department of Computer Science
Australian National University
Canberra 0200, Australia

{Jon.Baxter, Andrew.Tridgell, Lex.Weaver}@anu.edu.au

17th July 1998

Abstract

In this paper we discuss the problem of automatically learning evaluation function parameters in a chess program. In particular, we describe some experiments in which our chess program *KnightCap* learnt the parameters of its evaluation function using a combination of *Temporal Difference* learning and on-line play on FICS and ICC. KnightCap is freely available on the web from <http://wwwsyseng.anu.edu.au/lsg>. The main success we report is that KnightCap went from a (blitz) rating of 1650 to a rating of 2150 in just 3 days and 308 games. We discuss the details of our learning algorithm, details of KnightCap, and some of the principal reasons for KnightCap's rapid improvement.

1 Introduction

Temporal Difference learning, first introduced by Samuel [6] and later extended and formalized by Sutton [9] in his TD(λ) algorithm, is an elegant technique for

approximating the expected long term future cost (or *cost-to-go*) of a stochastic dynamical system as a function of the current state. In computer game-playing it is used to train the evaluation function to approximate the probability of a win. Perhaps the most remarkable success of TD(λ) to date is Tesauro's TD-Gammon, a neural network backgammon player that was trained from scratch using TD(λ) and simulated self-play. TD-Gammon is competitive with the best human backgammon players [10].

In this paper we describe our own experiments in which the parameters of a chess evaluation function were learnt using a variant of TD(λ) we call TDLeaf(λ) (TDLeaf(λ) is essentially just TD(λ) applied to minimax search). In particular, we incorporated TDLeaf(λ) into our chess program *KnightCap* and trained its evaluation function by on-line play on FICS (<http://www.freechess.org>) and ICC (<http://www.chessclub.com>). The main success story we report is that starting from an evaluation function in which all parameters were set to zero except the values of the pieces, KnightCap went from a 1650-rated player to a 2150-rated blitz player in just three days and 308 games. KnightCap is freely available and may be downloaded from <http://wwwsyseng.anu.edu.au/lsg>.

For reference purposes, note that on a Pentium Pro 200 MHz machine (the machine used for the experiments reported here), KnightCap searches from 8500 nodes per second to around 20,000 nodes per second depending on the position. It searches 2-3 ply shallower than Bob Hyatt's Crafty¹ given the same hardware and thinking time. In "3 0 Blitz" it will typically get to depth 5 in the middle game and depth 9-10 in the ending. It presently maintains a rating of 2400–2600² on ICC with its learnt parameters and 9702 book positions learnt from on-line games on ICC³. Crafty's rating on ICC is around 2700–2900.

The remainder of the paper is organized as follows. In section 2 we describe the TDLeaf(λ) algorithm as it applies to chess. KnightCap's architecture and how we incorporated the learning is described in section 3. Experimental results for internet-play with KnightCap are given in section 4. KnightCap's book learning algorithm is described in section 5. Section 6 contains some discussion and concluding remarks.

1.1 Related work

The earliest report of applying TD(λ) to chess of which we are aware is [3]. Gherity played his SAL program against an early version of GnuChess restricted to

¹Crafty is the strongest public domain chess program.

²There appears to be a systematic rating difference between FICS and ICC of about 200–250 points.

³The book learning is described in more detail in section 5.

search for only one second. He used straight $TD(\lambda)$ on the root nodes of the search. In 4200 games SAL drew 8 games and lost all the others, which was probably the main reason for its lack of success: if nearly all games are lost the $TD(\lambda)$ algorithm will tend to drive the evaluation of *every* position towards a loss, regardless of its true score. KnightCap did not suffer in this regard because it was learning by playing on the internet chess servers where people (and computers) tend to prefer playing opponents of similar strength, hence KnightCap’s win/loss ratio stayed near 1.

In [11] some further experiments on learning chess evaluation functions using Temporal Difference learning and Explanation-Based learning were reported. Thrun’s program NeuroChess again learnt using $TD(\lambda)$ applied to root nodes but in contrast to SAL it learnt by self-play, both from the opening position and from positions arrived at in Grandmaster play. The purpose of starting from Grandmaster positions was to ensure at least a minimal amount of relevant exploration. NeuroChess’ performance improved from a $< 1\%$ score against GnuChess to an 11% score in 2400 games. Both NeuroChess and GnuChess were restricted to 2-ply search with no quiescence extensions, and they were both using the same features (NeuroChess used a non-linear function of the features while GnuChess used a linear function). NeuroChess’ final performance may have been much better if it had been seeded with the “correct” material values, as was the case with KnightCap, and if it had learnt by playing a suitably weakened GnuChess, rather than by self-play. In our experiments with KnightCap we have found self-play to be a poor way of learning.

Beal and Smith [1], using $TDLeaf(\lambda)$, reported very positive results for learning piece values using temporal differences and *self-play* (in contrast to our use of on-line play). The fact that their evaluation function only contained material terms probably contributed to the success of self-play.

A more comprehensive review of automatic learning in all areas of computer chess programming may be found in [2].

2 The $TDLeaf(\lambda)$ algorithm

In this section we discuss the $TDLeaf(\lambda)$ algorithm and how it applies to a computer chess program with a tunable evaluation function. Note that $TDLeaf(\lambda)$ is essentially just Sutton’s $TD(\lambda)$ [9] adapted for use in minimax search.

Let x_1, x_2, \dots, x_N denote the the sequence of positions with the computer to move during the course of a game. In each position x_i , the computer performs a minimax search (or any of its faster variants like α - β) to some depth d , using a heuristic evaluation function $J(x, w)$ to assign values to the leaf nodes x . The

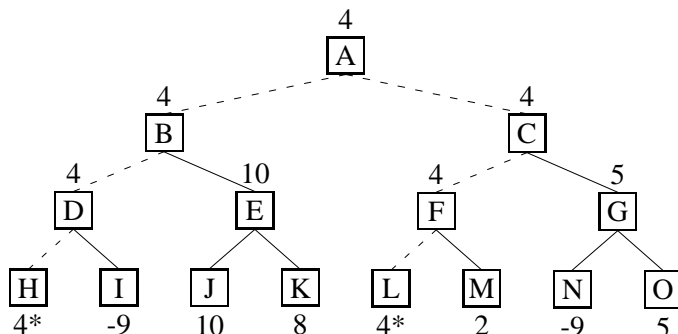


Figure 1: A minimax search tree showing the principal leaves (H and L).

parameter vector w consists of all the tunable parameters in the evaluation function (values of the pieces, isolated pawns, doubled pawns, king safety, etc). At the end of the search the computer makes its move and records the *leaf node of the principal variation* (also known as the *principal leaf*, see Figure 1), which we will denote by x_i^l . Note that the backed-up value assigned to the root node x_i after the search is just the heuristic value $J(x_i^l, w)$ of the principal leaf x_i^l . If the principal leaf is not unique, any one of the equivalent leaves may be recorded.

At the end of the game, the computer receives a *reward* $r(x_N)$ equal to -1 for a loss, 0 for a draw and $+1$ for a win. To convert the raw values of the principal leaves $J(x_i^l, w)$ into predictions of the final reward $r(x_i^l, w)$, they are passed through the hyperbolic tangent function (see Figure 2),

$$r(x_i^l, w) := \tanh\left(\beta J(x_i^l, w)\right) \quad (1)$$

where the constant β is chosen to ensure a not too steep evaluation function (in KnightCap $\beta = 0.255$ so that $r(x, w) = 0.25$ for a position x in which white's superiority is equivalent to one pawn). Note that any increasing differentiable function mapping large negative values of $J(x, w)$ to -1 , large positive values to $+1$, and 0 to 0 will do in place of the hyperbolic tangent.

Now, for each move $i = 1, \dots, N - 2$, define the *temporal difference* d_i by

$$d_i := r(x_{i+1}^l, w) - r(x_i^l, w) \quad (2)$$

For $i = N - 1$ define

$$d_{N-1} := r(x_N) - r(x_{N-1}^l, w). \quad (3)$$

Note that d_i measures the difference between the reward predicted by the computer at move $i + 1$, and the reward predicted at move i (d_{N-1} is the difference between the outcome of the game and the prediction at the penultimate move).

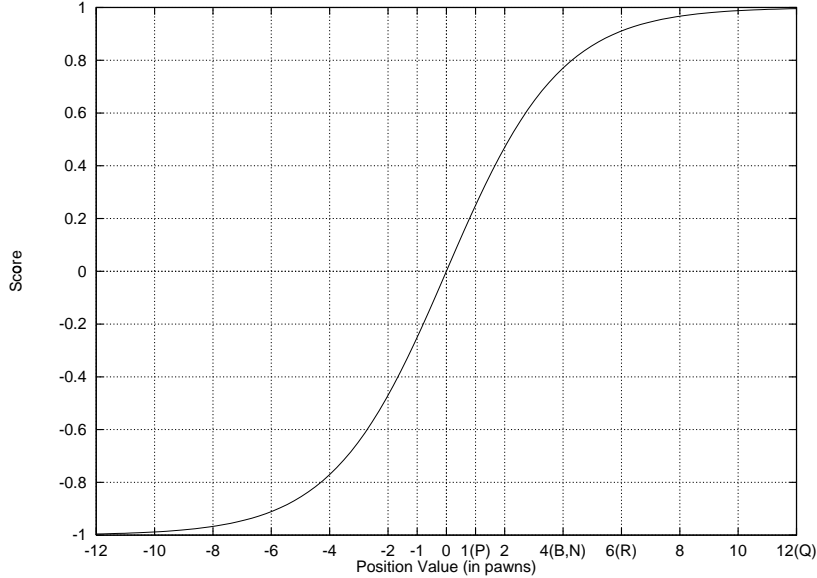


Figure 2: Predicted outcome of the game as a function of the underlying raw position value in pawns. Outcome is 1 for a win, 0 for a draw and -1 for a loss.

Barring blunders by the opponent, a “good” evaluation function should not “change its mind” too much from one move to the next and so should have small temporal differences d_i . To avoid KnightCap learning to predict weaker opponent’s blunders, we set all temporal differences d_i to zero if KnightCap did not predict the opponent’s move and the opponent’s rating was less than KnightCap’s. If the opponent was rated higher we left the temporal difference unchanged⁴.

The idea behind the TDLeaf(λ) algorithm is to adjust the parameters w so as to reduce the size of the temporal differences d_i . More precisely, after calculation of the temporal differences d_i , the TDLeaf(λ) algorithm updates the parameter vector w according to the formula (which is simply Sutton’s [9] TD(λ) formula rearranged and applied to the minimax domain)

$$w := w + \alpha \sum_{i=1}^{N-1} \nabla r(x_i^l, w) \left[\sum_{j=i}^{N-1} \lambda^{j-i} d_j \right] \quad (4)$$

⁴A more sophisticated method would be to reduce the size of temporal differences resulting from opponents’ blunders on a sliding scale as a function of the rating difference between KnightCap and its opponent. We did not investigate this possibility.

where $\nabla r(x, w)$ is the vector of partial derivatives of $r(x, w)$ with respect to its parameters w . The positive parameter α controls the learning rate and would typically be “annealed” towards zero during the course of a long series of games. The parameter $\lambda \in [0, 1]$ controls the extent to which temporal differences propagate backwards in time. To see this, compare equation (4) for $\lambda = 0$:

$$\begin{aligned} w &:= w + \alpha \sum_{i=1}^{N-1} \nabla r(x_i^l, w) d_i \\ &= w + \alpha \sum_{i=1}^{N-1} \nabla r(x_i^l, w) \left[r(x_{i+1}^l, w) - r(x_i^l, w) \right] \end{aligned} \quad (5)$$

and $\lambda = 1$:

$$w := w + \alpha \sum_{i=1}^{N-1} \nabla r(x_i^l, w) \left[r(x_N) - r(x_i^l, w) \right]. \quad (6)$$

Consider each term contributing to the sums in equations (5) and (6). For $\lambda = 0$ the parameter vector is being adjusted in such a way as to move $r(x, w)$ —the predicted reward at move i —closer to $r(x_{i+1}, w)$ —the predicted reward at move $i + 1$. In contrast, for $\lambda = 1$ the parameter vector is adjusted to move the predicted reward at move i closer to the final reward $r(x_N)$. Values of λ between zero and one interpolate between these two behaviors. Another way of looking at the role of λ is to note that for small values of λ evaluations over the next few moves are treated as reliable for the purpose of updating the parameters, whereas for large values of λ only evaluations near the end of the game are considered reliable. This observation provides a heuristic for choosing λ : if the evaluation function is already quite reliable, choose a small value of λ (0.3 to 0.7), if not choose λ close to 1 (say 0.95 or even 1). In our experiments with KnightCap we used $\lambda = 0.7$.

One easy way to compute the vector of partial derivatives $\nabla r(x_i^l, w) = \left(\frac{\partial r(x_i^l, w)}{\partial w_1}, \dots, \frac{\partial r(x_i^l, w)}{\partial w_k} \right)$ is numerically:

$$\frac{\partial r(x_i^l, w)}{\partial w_j} \approx \frac{r(x_i^l, w_j + \delta) - r(x_i^l, w_j)}{\delta},$$

for small enough δ . This requires only two calls to the evaluation function for each parameter, one with w_j at its original value and one with w_j incremented by δ , and has the advantage that complex new features may be added to the evaluation function without having to also compute their derivatives. Of course, it also requires that the parameters are stored in an easily adjustable form (in KnightCap they are elements of a single array).

Note that it is not necessary to adjust the parameters after every game. The updates can also be accumulated over several games in order to reduce the effects of noise. For reference, the TDLeaf(λ) algorithm is summarized in figure 3.

3 KnightCap

In this section we describe the details of KnightCap including implementation issues associated with applying TDLeaf(λ).

KnightCap is a reasonably sophisticated computer chess program for Unix systems. It has all the standard algorithmic features that modern chess programs tend to have as well as a number of features that are much less common. This section is meant to give the reader an overview of the type of algorithms that have been chosen for KnightCap. Space limitations prevent a full explanation of all of the described features, an interested reader should be able find explanations in the widely available computer chess literature (see for example [4]) or by examining the source code: <http://wwwsyseng.anu.edu.au/lsg>.

3.1 Board representation

This is where KnightCap differs most from other chess programs. The principal board representation used in KnightCap is the *topieces* array. This is an array of 32 bit words with one word for each square on the board. Each bit in a word represents one of the 32 pieces in the starting chess position (8 pieces + 8 pawns for each side). Bit i on square j is set if piece i is attacking square j .

The *topieces* array has proved to be a very powerful representation and allows the easy description of many evaluation features which are more difficult or too costly with other representations. The array is updated dynamically after each move in such a way that for the vast majority of moves only a small proportion of the *topieces* array need be directly examined and updated.

A simple example of how the *topieces* array is used in KnightCap is determining whether the king is in check. Whereas an `in_check()` function is often quite expensive in chess programs, in KnightCap it involves just one logical AND operation in the *topieces* array. In a similar fashion the evaluation function can find common features such as connected rooks using just one or two instructions.

The *topieces* array is also used to drive the move generator and obviates the need for a standard move generation function.

- Let $J(x, w)$ be an heuristic evaluation function parameterized by k parameters $w = (w_1, \dots, w_k) \in \mathbb{R}^k$.
- Let x_1, \dots, x_N be the N positions that occurred during the course of a game (with the computer to move), and for $i = 1, \dots, N$, let x_i^l be the leaf node of the principal variation (the *principal leaf*) of the computer's search starting from x_i . If there is more than one principal variation, choose an arbitrary leaf node from the available candidates.
- Let $r(x_N)$ be the outcome of the game (-1 for a loss, 0 for a draw, +1 for a win).
- For each principal leaf x_i^l , define

$$r(x_i^l, w) := \tanh\left(0.255 * J(x_i^l, w)\right)$$

where we are assuming $J(x, w)$ is measured in units of 1 pawn (scale $J(x, w)$ accordingly).

- For $i = 1, \dots, N - 1$, compute the temporal differences:

$$d_i := r(x_{i+1}^l, w) - r(x_i^l, w) \quad (7)$$

and the vector of partial derivatives $\nabla r(x_i^l, w) = \left(\frac{\partial r(x_i^l, w)}{\partial w_1}, \dots, \frac{\partial r(x_i^l, w)}{\partial w_k}\right)$.

- Set $d_i = 0$ if the move leading to x_{i+1}^l was not predicted by the computer (option: only do this if the opponent's rating is less than the computer).
- Update each parameter w_j according to the formula:

$$w_j := w_j + \alpha \sum_{i=1}^{N-1} \frac{\partial r(x_i^l, w)}{\partial w_j} \left[\sum_{m=i}^{N-1} \lambda^{m-i} d_m \right]. \quad (8)$$

λ should be around 0.5 to 0.7 if the evaluation function is already quite reliable, but close to 1 (> 0.95) if it is not. α should be chosen so that updates are of the order of 1/100 of a pawn.

Figure 3: The TDLeaf(λ) algorithm

3.2 Search algorithm

The basis of the search algorithm used in KnightCap is MTD(f) [5]. MTD(f) is a logical extension of the minimal-window alpha-beta search that formalizes the placement of the minimal search window to produce what is in effect a bisection search over the evaluation space.

The variation of MTD(f) that KnightCap uses includes some convergence acceleration heuristics that prevent the very slow convergence that can sometimes plague MTD(f) implementations. These heuristics are similar in concept to the momentum terms commonly used in neural network training.

The MTD(f) search algorithm is applied within a standard iterative deepening framework. The search begins with the depth obtained from the transposition table for the initial search position and continues until a time limit is reached in the search. Search ordering at the root node ensures that partial ply search results obtained when the timer expires can be used quite safely.

3.3 Null moves

KnightCap uses a recursive null move forward pruning technique. Whereas most null move using chess programs use a fixed R value (the number of additional plys to prune when trying a null move) KnightCap instead uses a variable R value in an asymmetric fashion. The initial R value is 3 and the algorithm then tests the result of the null move search. If it is the computers side of the search and the null move indicates that the position is “good” for the computer then the R value is decreased to 2 and the null move is retried.

The effect of this null move system is that most of the speed of a $R = 3$ system is obtained, while making no more null move defensive errors than an $R = 2$ system. It is essentially a pessimistic system.

3.4 Search extensions

KnightCap uses a large number of search extensions to ensure that critical lines are searched to sufficient depth. Extensions are indicated through a combination of factors including check, null-move mate threats, pawn moves to the last two ranks and recapture extensions. In addition KnightCap uses a single ply razoring system with a 0.9 pawn razoring threshold.

3.5 Asymmetries

There are quite a number of asymmetric search and evaluation terms in KnightCap, with a leaning towards pessimistic (i.e. careful) play. Apart from the asymmetric

null move and search extensions systems mentioned above, KnightCap also uses an asymmetric system to decide what moves to try in the quiesce search and several asymmetric evaluation terms in the evaluation function (such as king safety and trapped piece factors).

When combined with the TDLeaf(λ) algorithm KnightCap is able to learn appropriate values for the asymmetric evaluation terms.

3.6 Transposition Tables

KnightCap uses a standard two-deep transposition table with a 128 bit transposition table entry. Each entry holds separate depth and evaluation information for the lower and upper bound.

The ETTC (enhanced transposition table cutoff) technique is used both for move ordering and to reduce the tree size. The transposition table is also used to feed the book learning system and to initialize the depth for iterative deepening.

3.7 Move ordering

The move ordering system in KnightCap uses a combination of the commonly used history [7], killer, refutation and transposition table ordering techniques. With a relatively expensive evaluation function KnightCap can afford to spend a considerable amount of CPU time on move ordering heuristics in order to reduce the tree size.

3.8 Parallel search

KnightCap has been written to take advantage of parallel distributed memory multi-computers, using a parallelism strategy that is derived naturally from the MTD(f) search algorithm. Some details on the methodology used and parallelism results obtained are available in [12]. The results given in this paper were obtained using a single CPU machine.

3.9 Evaluation function

The heart of any chess program is its evaluation function. KnightCap uses quite a slow evaluation function that evaluates a number of computationally expensive features. The evaluation function also has four distinct stages: Opening, Middle, Ending and Mating, each with its own set of parameters (but the same features). We have listed the names of all KnightCap's features in table 1. Note that some of the features have more than one parameter associated with them, for example there are 64 parameters associated with rook position, one for each square.

These features all begin with “I”. To summarize just a few of the more obscure features: `IOPENING_KING_ADVANCE` is a bonus for the rank of the king in the opening, it has 8 parameters, one for each rank. `IMID_KING_ADVANCE` is the same but applies in the middle game (the fact that we have separate features for the opening and middle games is a hangover from KnightCap’s early days when it didn’t have separate parameters for each stage). `IKING_PROXIMITY` is the number of moves between our king and the opponents king. It is very useful for forcing mates in the ending. Again there is one parameter for each of the 8 possible values. `IPOS_BASE` is the base score for controlling each of the squares. `IPOS_KINGSIDE` and `IPOS_QUEENSIDE` are modifications added to `IPOS_BASE` according as KnightCap is castled on the king or queen sides respectively. The `MOBILITY` scores are the number of moves available to a piece, thresholding at 10. There is a separate score for each rank the piece is on, hence the total number of parameters of 80. The `SMOBILITY` scores are the same, but now the square the piece is moving to has to be safe (i.e controlled by KnightCap). `THREAT` and `OPONENTS_THREAT` are computed by doing a minimax search on the position in which only captures are considered and each piece can move only once. Its not clear this helps the evaluation much, but it certainly improves move ordering (the best capture is given a high weight in the ordering). `IOVERLOADED_PENALTY` is a penalty that is applied to each piece for the number of otherwise hung pieces it is defending. There is a separate penalty for each number, thresholding at 15 (this could be done better: we should have a base score times by the number of pieces, and have KnightCap learn the base score and a perturbation on the base score for each number). `IQ_KING_ATTACK_OPPONENT` and `INOQ_KING_ATTACK_OPPONENT` are bonuses for the number of pieces KnightCap has attacking the squares around the enemy king, both with and without queens on the board. `IQ_KING_ATTACK_COMPUTER` and `INOQ_KING_ATTACK_COMPUTER` are the same thing for the opponent attacking KnightCap’s king. Note that this asymmetry allows KnightCap the freedom to learn to be cautious by assigning greater weight to opponent pieces attacking its own king that it does to its own pieces attacking the opponent’s king. It can of course also use this to be aggressive. For more information on the features, see `eval.c` in KnightCap’s source code.

The most computationally expensive part of the evaluation function is the “board control”. This function evaluates a control function for each square on the board to try to determine who controls the square. Control of a square is essentially defined by determining whether a player can use the square as a flight square for a piece, or if a player controls the square with a pawn.

Despite the fact that the board control function is evaluated incrementally, with the control of squares only being updated when a move affects the square, the func-

Feature	#	Feature	#
BISHOP_PAIR	1	CASTLE_BONUS	1
KNIGHT_OUTPOST	1	BISHOP_OUTPOST	1
SUPPORTED_KNIGHT_OUTPOST	1	SUPPORTED_BISHOP_OUTPOST	1
CONNECTED_ROOKS	1	SEVENTH_RANK_ROOKS	1
OPPOSITE_BISHOPS	1	EARLY_QUEEN_MOVEMENT	1
IOPENING_KING_ADVANCE	8	IMID_KING_ADVANCE	8
IKING_PROXIMITY	8	ITRAPPED_STEP	8
BLOCKED_KNIGHT	1	USELESS_PIECE	1
DRAW_VALUE	1	NEAR_DRAW_VALUE	1
NO_MATERIAL	1	MATING_POSITION	1
IBISHOP_XRAY	5	IENDING_KPOS	8
IROOK_POS	64	IKNIGHT_POS	64
IPOS_BASE	64	IPOS_KINGSIDE	64
IPOS_QUEENSIDE	64	IKNIGHT_MOBILITY	80
IBISHOP_MOBILITY	80	IROOK_MOBILITY	80
IQUEEN_MOBILITY	80	IKING_MOBILITY	80
IKNIGHT_SMOBILITY	80	IBISHOP_SMOBILITY	80
IROOK_SMOBILITY	80	IQUEEN_SMOBILITY	80
IKING_SMOBILITY	80	IPIECE_VALUES	6
THREAT	1	OPONENTS_THREAT	1
IOVERLOADED_PENALTY	15	IQ_KING_ATTACK_COMPUTER	8
IQ_KING_ATTACK_OPPONENT	8	INOQ_KING_ATTACK_COMPUTER	8
INOQ_KING_ATTACK_OPPONENT	8	QUEEN_FILE_SAFETY	1
NOQUEEN_FILE_SAFETY	1	IPIECE_TRADE_BONUS	32
IATTACK_VALUE	16	IPAWN_TRADE_BONUS	32
UNSUPPORTED_PAWN	1	ADJACENT_PAWN	1
IPASSED_PAWN_CONTROL	21	UNSTOPPABLE_PAWN	1
DOUBLED_PAWN	1	WEAK_PAWN	1
ODD_BISHOPS_PAWN_POS	1	BLOCKED_PASSED_PAWN	1
KING_PASSED_PAWN_SUPPORT	1	PASSED_PAWN_ROOK_ATTACK	1
PASSED_PAWN_ROOK_SUPPORT	1	BLOCKED_DPAWN	1
BLOCKED_EPAWN	1	IPAWN_ADVANCE	7
IPAWN_ADVANCE1	7	IPAWN_ADVANCE2	7
KING_PASSED_PAWN_DEFENCE	1	IPAWN_POS	64
IPAWN_DEFENCE	12	ISOLATED_PAWN	1
MEGA_WEAK_PAWN	1	IWEAK_PAWN_ATTACK_VALUE	8

Table 1: KnightCap’s features and the number of parameters corresponding to each. Most of the features are self-explanatory, see the text for a description of some of the more obscure ones. Note that KnightCap’s large number of parameters is obtained by summing all the numbers in this table and then multiplying by the number of stages (four).

tion typically takes around 30% of the total CPU time of the program. This high cost is considered worthwhile because of the flow-on effects that this calculation has on other aspects of the evaluation and search. These flow-on effects include the ability of KnightCap to evaluate reasonably accurately the presence of hung, trapped and immobile pieces which is normally a severe weakness in computer play. We have also noted that the more accurate evaluation function tends to reduce the search tree size thus making up for the decreased node count.

3.10 Modification for TDLeaf(λ)

The modifications made to KnightCap for TDLeaf(λ) affected a number of the program's subsystems. The largest modifications involved the parameterization of the evaluation function so that all evaluation parameters became part of a single long weight vector. All tunable evaluation knowledge could then be described in terms of the values in this vector.

The next major modification was the addition of the full board position in all data structures from which an evaluation value could be obtained. This involved the substitution of a structure for the usual scalar evaluation type, with the evaluation function filling in the evaluated position and other board state information during each evaluation call. Similar additions were made to the transposition table entries so that the result of a search would always have available to it the position associated with the leaf node in the principal variation. This significantly enlarges the transposition table and means that to operate effectively with the MTD(f) search algorithm (itself a memory-hungry α - β variant), KnightCap really needs at least 30Mb of hash table when learning.

The only other significant modification that was required was an increase in the bit resolution of the evaluation type so that a numerical partial derivative of the evaluation function with respect to the evaluation coefficient vector could be obtained with reasonable accuracy.

4 Experiments with KnightCap

In our main experiment we took KnightCap's evaluation function and set all but the material parameters to zero. The material parameters were initialized to the standard "computer" values: 1 for a pawn, 4 for a knight, 4 for a bishop, 6 for a rook and 12 for a queen. With these parameter settings KnightCap (under the pseudonym "WimpKnight") was started on the Free Internet Chess server (FICS, `fics.onenet.net`) on 8/21/97. Both humans and computers play on FICS. We played KnightCap for 25 games without modifying its evaluation function so as

to get a reasonable idea of its rating. After 25 games it had a blitz (fast time control) rating of 1650, which put it at about B-grade human performance. We then turned on the TDLeaf(λ) learning algorithm, with $\lambda = 0.7$. The value of λ was chosen fairly arbitrarily, while α was set high enough to ensure rapid modification of the parameters (each update altered the parameters by at most 1/100 of a pawn).

By 8/24/97 KnightCap's rating had climbed to 2150, an increase of 500 points in three days and after only 308 games⁵. At this point KnightCap's performance began to plateau, primarily because it does not have an opening book and so will repeatedly play into weak lines⁶. KnightCap also does not search particularly deeply and so often gets tactically outgunned by deeper programs. It seems very difficult to add enough features to KnightCap's evaluation function to enable it to learn to predict significant tactics even as shallow as 2-ply.

There appear to be a number of reasons for the remarkable rate at which KnightCap improved.

1. As all the non-material parameters were initially zero, even small changes in these parameters could cause very large changes in the relative ordering of materially equal positions. Hence even after a few games KnightCap was playing a substantially better game of chess.
2. It seems to be important that KnightCap started out life with intelligent material parameters. This put it close in parameter space to many far superior parameter settings. We tried a similar experiment with the material parameters also initially set to zero but after 1000 games KnightCap was still rated down around 1300 (it had improved by 300 points but progress was very slow). This result is not inconsistent with those reported in [1]. There it took about 2000 games to learn the "correct" material values, and perhaps with another 1000 games KnightCap may have reached a rating of 1600, close to its performance with the material values set. However, we were using a small value of λ (0.7) compared to Beal and Smith's 0.95 and this may well have been a significant difference (for $\lambda = 0.95$ we are relying more on the outcome of the game for feedback, and less upon the evaluations of subsequent positions).
3. Most players on FICS prefer to play opponents of similar strength, and so KnightCap's opponents improved as it did. This may also have had the effect of *guiding* KnightCap along a path in parameter space that led to a strong set of parameters.

⁵A rating of 2150 on FICS corresponds to a rating of about 2350 on ICC.

⁶We have since implemented several different opening learning strategies in addition to the evaluation learning strategies. Our efforts to date are summarized in section 5.

4. KnightCap was not learning by self-play. In particular, by playing different opponents KnightCap is forced into positions that it evaluates highly but subsequently discovers are losing. These are precisely the positions that KnightCap needs to see to learn rapidly, but are unlikely to be seen when playing against itself.

4.1 Changes as KnightCap learned

In this first experiment we did not record the games KnightCap played, nor the evolution of the parameters or the evolution of KnightCap's rating, so we conducted a second experiment in which all these factors were recorded. One modification we made was to update the parameters after every fourth game to see whether this improved the performance. In addition, we had made major modifications to KnightCap's evaluation function between the two experiments: many more features had been added, stages had been introduced, and some small changes to the search algorithm had been made. By now there were nearly 6000 independent parameters in the evaluation function, whereas in the original experiment there were just under 500. KnightCap's rating over 1070 games is plotted in figure 4. In this plot,

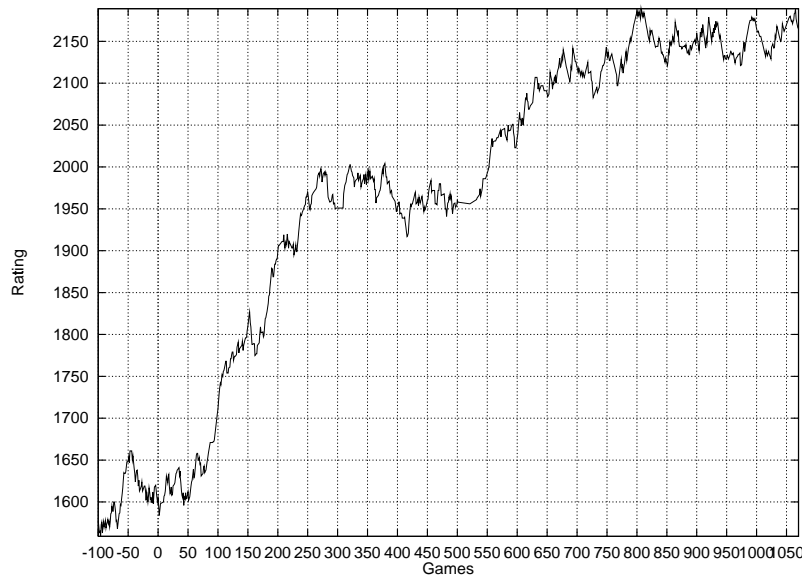


Figure 4: KnightCap's rating as a function of games played (second experiment).

KnightCap was playing with just material parameters until learning was turned on at game 0. The rating rose sharply and then began to plateau after about 300 games

at just under 2000. The sharp decline in KnightCap's rating from game 375 to 420 was caused by two "rogue" telnet processes using 2/3 of the processor during one night of play. At game 500, KnightCap's memory was increased to 40Mb (it had been running with only 8Mb until this point). One can clearly see the benefit of the increased memory in this plot (in the first experiment KnightCap was running with 24Mb). Again KnightCap's performance levelled out at around 2150, but this time after considerably more games which may well have been caused by the increased size of the evaluation function and also the decision to update the parameters after 4 games rather than every game.

Plots of various parameters as a function of the number of games played are shown in Figure 7. Each plot contains three graphs corresponding to the three different stages of the evaluation function: opening, middle and ending⁷. The ending is when there are less than or equal to six pieces (queens count for 2 pieces) and at least one pawn on the board. "Opening" is used when no more than a single piece has been exchanged and at least one side has the right to castle. "Middle" is not opening and not ending (and not mating). Note that KnightCap's parameters are independent for the stages, and also that the stage is only altered at the root of the search tree, never at an internal node or a leaf. This ensures all positions in a search are compared using the same parameters in the evaluation function (note also that the TDLeaf(λ) algorithm acts to keep evaluations between different stages reasonably consistent).

We have shown the final values (after 1070 games) for controlling the squares on the board for the different stages in Figure 8. In KnightCap's evaluation function, a square is controlled if a piece can fly to that square and not be immediately captured or kicked away by a pawn. In the figure, the lighter the square, the greater the bonus for controlling that square. Observe that in the opening the squares most desirable to control are the central squares on the kingside, the squares in front of a king-side castled king, and the king-rook file. The origin of the great value in controlling the king-rook file can be traced to the use of a standard "computer busting" strategy by several human players on the internet chess servers. The strategy involves sacrificing a knight on g4 (or g5 if the computer is black) and opening the h-file in the process. The human then quickly mates down the h-file. An example game of this type is shown in figure 5.

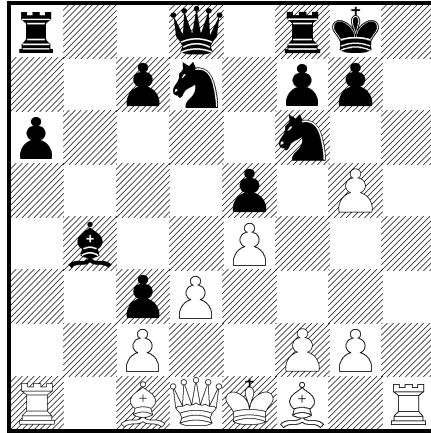
As one last example, in Figure 9 we have shown the final value of a rook located on each square of the board. The extreme undesirability of having rooks on their original squares in the opening and middle games is another way for KnightCap to

⁷KnightCap actually has a fourth and final stage "mating" which kicks in when all the pieces are off, but this stage only uses a few of the parameters (opponent's king mobility and proximity of our king to the opponent's king).

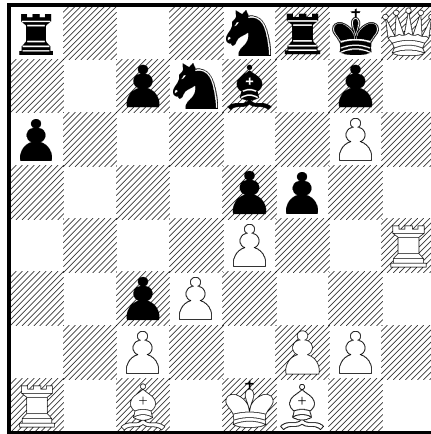
KNIGHTCAP MERCILOUS

Internet Chess Server, August 1997

1. d3 d5 2. ♘c3 e5 3. h3 d4 4. ♘b5 ♙d7 5. a4 a6 6. ♘a3 ♙xa4 7. ♘c4 ♘d7 8. ♙xa4 b5 9. ♚a1 bxc4 10. ♘f3 ♙b4+ 11. ♙d2 c3 12. bxc3 dxc3 13. ♙c1 ♘gf6 14. e4 O-O 15. ♘g5 h6 16. h4! hxg5 17. hxg5



KnightCap now has to return the Knight or be mated. These days KnightCap will play 17. ... ♚e8 and survive, however early versions of KnightCap would continue 17. ... ♘e8 and then 18. ♚h5 f5 19. g6 ♚h4 20. ♚xh4 ♙e7 21. ♚h8



1 : 0

[Mercilous]

Figure 5: An ICC game showing one often repeated attack that caused KnightCap to learn large values for controlling the h file, which then led to improved play.

KNIGHTCAP CRAFTY

5 minute game + 5 second increment per move

1. e4 e5 2. ♖f3 ♗c6 3. ♘b5 a6 4. ♙a4 ♗f6 5. O-O ♙e7 6. ♚e1 b5 7. ♘b3 (+0.56, d11)

KnightCap is now out of book although the depth of its last search indicates that the principal variation ends in a brain position.

7. .. O-O 8. c3 d6 9. h3 ♘b7 10. d3 ♗b8

Here KnightCap had expected 10. .. Na5 which it evaluated at -0.2 pawns for white (but only at depth 5). 10. .. Na5 was the other option in Crafty's book.

11. a4 (+1.12, d7) ♗bd7 12. axb5 axb5 13. ♚xa8 ♙xa8 14. ♗a3 c6 15. ♚e2 (+0.95, d5) ♗e8 (-0.14, d8)

Crafty is now out of book, probably because the large book was truncated at 30 ply.

16. ♗h4 d5 17. ♗f5 b4 18. ♗c2 bxc3 19. bxc3 dxe4 20. dxe4 ♗c5 21. ♙a3 ♘b7 22. ♙c4 (+1.84, d5) ♙c8 (-0.41, d8) 23. ♗xe7 (+1.87, d6) ♗xe7 (-0.58, d10) 24. ♗d2 (+2.12, d7) ♗fd7 (-1.26, d9)

Now Crafty sees a problem.

25. ♗e3 ♙b7 26. ♚d2 (+2.72, d7) ♚a8 (-1.39, d9) 27. ♚xd7 ♗xd7 28. ♙xc5 ♗d1 29. ♗h2 (+2.30, d8) ♗xc2 (-1.94, d8) 30. ♗f3 ♗h8 31. ♗xf7 ♗xe4 32. ♗xb7 ♚d8 33. ♗e7 ♚b8 34. ♙d4 ♗f4 35. g3 ♗f8 36. ♗xe5 ♚c8 37. ♙d3 ♗g8 38. ♙f5 ♚a8 39. ♗d6 ♚e8 40. ♗xc6 ♗f7 41. ♗h6 ♗g8 42. ♗h4 ♚f8 43. ♙e4 ♚e8 44. c4 ♚b8 45. ♗h1 ♚e8 46. c5 ♚c8 47. c6 ♚e8 **1 : 0 [KnightCap]**

Figure 6: One of KnightCap's better games against Crafty. Both were running on Pentium Pro 200 PC's and using 50M hash tables. Crafty was using 8M of pawn hash and its large book with the most recent learning data (as at 10-6-98). KnightCap was using evaluation function parameters tuned by TDLeaf(λ) from a handcrafted starting point, and a brain with 9702 positions. All evaluations are in units of 1 pawn.

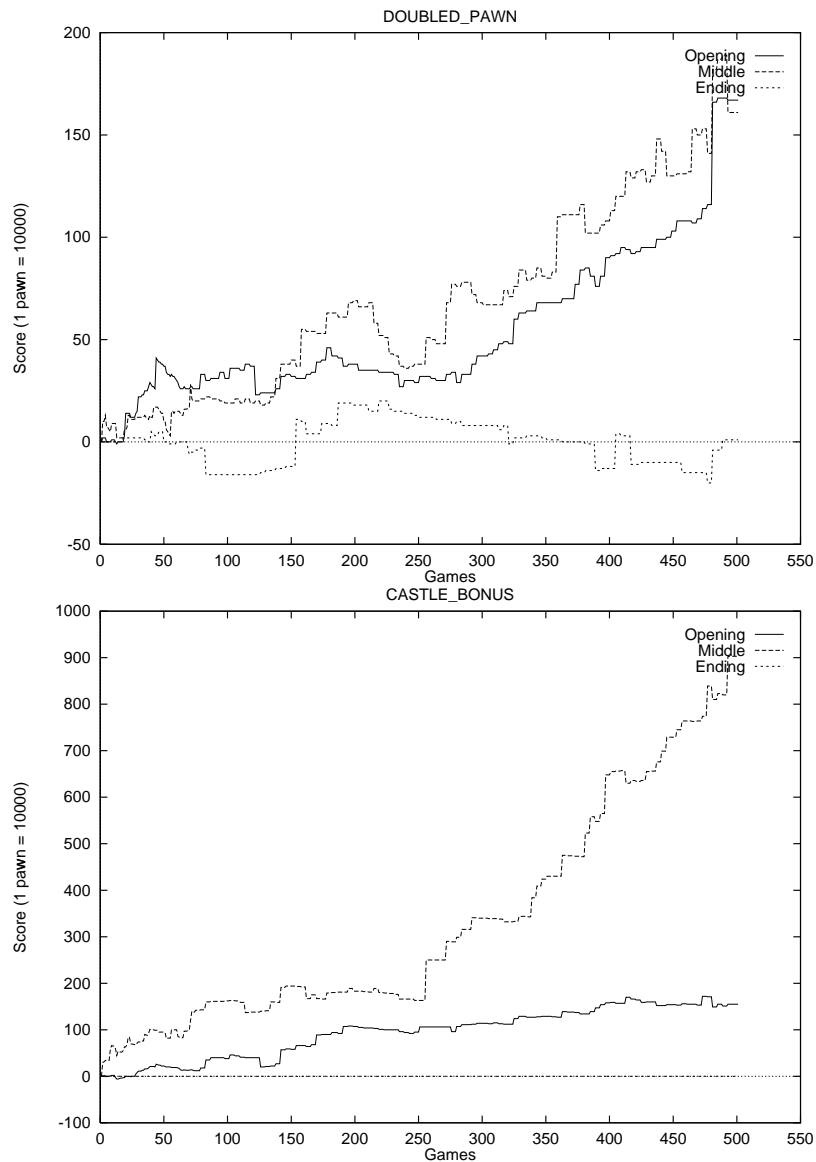


Figure 7: Evolution of two parameters as a function of the number of games played. “Doubled Pawn” is subtracted from the evaluation while “Castle Bonus” is added. Note that each parameter appears three times: once for each of the three stages in the evaluation function. There was no bonus for having castled in the endgame.

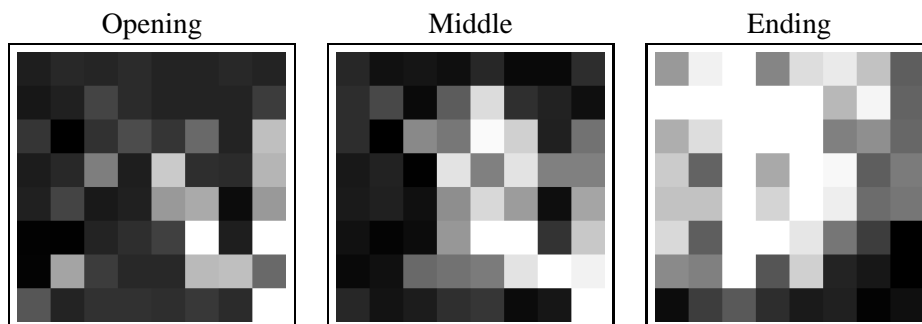


Figure 8: The value of controlling each square in the opening, middle, and ending (after 1070 games). The lighter the square, the greater the score for controlling it. Note the heavy emphasis on controlling the King-Rook file in the opening and middle game, and the central squares in the middle game. In the ending KnightCap has learnt to control advanced squares, presumably to assist in pawn promotion (although it appears not much promotion has occurred on the g and h files).

encourage castling, in addition to the CASTLE_BONUS parameter (Figure 7).

4.2 Comparison to hand-crafted parameters

The experiments reported so far demonstrate that a weak set of parameters for an evaluation function can be greatly improved by applying the TDLeaf(λ) algorithm and training with on-line play. But do the final parameters compare favourably to the performance achievable with hand-crafted parameters? We compared the performance of KnightCap with its learnt parameters to KnightCap’s performance with a set of hand-crafted parameters, again by playing the two versions on ICC. The hand-crafted parameters were close in performance to the learnt parameters (perhaps 50–100 rating points worse, although these figures are very noisy). We also tested the result of allowing KnightCap to learn starting from the hand-coded parameters, and in this case it seems that KnightCap performs better than when starting from just material values (peak performance was 2632 compared to 2575). We are conducting more tests to verify these results. However, it should not be too surprising that learning from a good quality set of hand-crafted parameters is better than just learning from material parameters. In particular, some of the hand-crafted parameters have very high values (the value of an “unstoppable pawn”, for example) which can take a very long time to learn under normal playing conditions, particularly if they are rarely active in the principal leaves. It is not yet clear whether given a sufficient number of games this dependence on the initial conditions can be made to vanish.



Figure 9: The value of placing a rook on each square (after 1070 games). The lighter the square, the greater the value of placing the rook on that square. The undesirability of the Rook home squares in the early stages of the game are another way for KnightCap to encourage castling.

5 Opening Learning

KnightCap’s lack of an opening book hampered its performance once its rating had improved to the level where it started playing sufficiently good opponents. The evaluation function is not sophisticated enough to prevent KnightCap from repeatedly playing opening blunders. Rather than supply KnightCap with an opening book of Grandmaster games, we tried implementing various opening book learning methods based on the “permanent brain” idea in Crafty. Crafty’s permanent brain is a set of losing positions and their evaluations that are inserted into the hash table at the start of every search. The search will then avoid playing again into those bad lines.

In our case we first tried inserting *all* positions that occurred during games and their backed up values into the brain (similar to [8]). A consistency check was also performed to locate later entries that invalidated positions 1 or 2 ply earlier, and if so the earlier positions were researched (with the later positions and their values inserted into the hash table). Although this method did improve performance somewhat, there were two difficulties:

- Some entries inserted into the brain had negative backed-up values but were in fact perfectly playable positions. In such cases KnightCap would avoid the positions in future and never discover that they are actually acceptable.
- It is nearly impossible to keep the brain consistent without spending large amounts of time researching the positions in the brain between games.

To circumvent these problems we implemented an alternative scheme. Instead of inserting all positions in a game in the brain, KnightCap only inserts the *losing*

position and the two subsequent positions. KnightCap inserts no positions if it won the game or drew against a stronger opponent. The losing position is defined to be last position in the game for which the evaluation was above zero, and the evaluation of the previous two moves were also above zero. By requiring at least three consecutive moves to be above zero we damp out some of the noise in the evaluation function. The three new positions are then researched in reverse order to at least depth 7 or depth one more than the depth searched in the game, whichever is greater (this takes around 30 seconds for a middle-game position on a Pentium Pro 200MHz PC (in blitz play)). The reason for storing the two positions after the losing position is to guarantee that the new search on the losing position is aware of the low score of the later positions.

One modification to this procedure is made in the case that the losing position occurs prior to a brain position that was actually encountered in the game. In that case we regard the final brain position as the losing move, insert its two successor moves and research all 3 in reverse order.

Every time a position is inserted in the brain, a 2-ply consistency check is performed: every brain position is examined to see if its recommended move leads to another brain position. If so, and if the second brain position contradicts the score of the first brain position both positions are marked to be researched. If the second position is not in the brain but there is a move in that position that leads to a brain position with a lower score than the first brain position then again the second and first brain positions are marked for research. All entries requiring research after the consistency check are searched in reverse order. A new game is not played until the brain is completely “clean” (no inconsistencies are found).

With this opening learning strategy⁸ KnightCap (playing under the pseudonym “KnightC” on ICC) has learnt many of the standard opening lines (KnightCap now has more than 10000 positions in its brain). Not being experts ourselves, we judge this by the fact that when playing versions of Crafty it is nearly always in Crafty’s book for five full moves and has been observed to be in book for 17 moves on one occasion. One nice thing about the opening learning is that KnightCap “understands” why certain opening moves are weak because it has played the weak lines itself in determining this fact. Such information is not available in Grandmaster games: the “correct moves” are always there but the computer does not necessarily know what to do if its opponent plays an incorrect move. Sometimes, however, KnightCap will lose a large number of games before eventually ruling out a particular weak line of play.

Note that we have not run book learning and evaluation function learning si-

⁸This is not strictly an *opening* learning strategy as brain positions can come from any phase of the game.

multaneously. The difficulty with applying both simultaneously is that as learning proceeds older book entries need to be periodically researched because they were originally searched with an obsolete evaluation function. This requires even more time to be devoted to the management of the brain. An example of KnightCap's play with a large brain learnt through on-line play on ICS is shown in Figure 6.

6 Conclusion and Future Work

We have introduced TDLeaf(λ), a variant of TD(λ) for tuning the parameters of an evaluation function to be used in minimax search. The only extra requirement of the algorithm is that the leaf node of each principal variation (the *principal leaf*) is stored throughout a game.

TDLeaf(λ) was incorporated into our chess program "KnightCap", which then learnt from a B-grade player to Master level in 308 games and 3 days of on-line play on the FICS Internet Chess Server. In this experiment KnightCap started out with widely used material parameter settings, all other parameters were set to 0. The final performance of the learnt parameters was comparable to the performance of a set of hand-crafted parameters. Also worthy of note is that when learning was seeded with the hand-crafted parameters, KnightCap's rating improved a further 50 to 100 points (this time playing on ICC), although the variance in these estimates is quite high. Hence the take home message from our experiments with KnightCap would be: incorporate as much knowledge as possible in the initial parameter settings, and then let learning proceed from there. A version of KnightCap "KnightC" having learnt in this way is presently running on ICC on PPro 200 hardware and has a blitz rating of 2400-2500 (the top rating is about 3000). This version is also using our opening book learning algorithm, a variation on Crafty's "permanent brain" idea.

It appears that learning via on-line rather than self-play was important for KnightCap's success. In particular, the knowledge of the opponent is conveyed in the positions played during the game: if the opponent opens up files next to KnightCap's king and then checkmates KnightCap, it tells KnightCap that having open files next to its king is a bad idea. Such information is unlikely to be discovered through self-play.

One weakness of the results presented here is that we have only tested the TDLeaf(λ) algorithm on one evaluation function. It would be good to see our results repeated for other evaluation functions in different programs, and a more systematic study of the influence of the parameter λ and the learning rate α on the learning performance.

Another area for further work is the automatic induction of features, since it is

the lack of tactical ability (and hence reliable features for predicting tactics) that has prevented KnightCap from improving even further. We did observe that as KnightCap's evaluation function improved it was able to search deeper in the same length of time. An even more radical avenue for further work is to learn to search selectively. The potential gains from reliable selective search are likely to be far greater than improvements in evaluation function technology.

References

- [1] D. F. Beal and M. C. Smith. Learning Piece values Using Temporal Differences. *Journal of The International Computer Chess Association*, September 1997.
- [2] J. Fürnkranz. Machine Learning in Computer Chess: The Next Generation. *International Computer Chess Association Journal*, 19:147–161, 1996.
- [3] M. Gherrity. *A Game-learning Machine*. PhD thesis, Department of Computer Science, University of California, San Diego, 1993.
- [4] T. A. Marsland and J. Schaeffer. *Computers, Chess and Cognition*. Springer Verlag, 1990.
- [5] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-First Fixed-Depth Minmax Algorithms. *Artificial Intelligence*, 87:255–293, 1996.
- [6] A. L. Samuel. Some Studies in Machine Learning Using the Game of Checkers. *IBM Journal of Research and Development*, 3:210–229, 1959.
- [7] J. Schaeffer. The History Heuristic and the Performance of Alpha-Beta Enhancements. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11:1203–1212, 1989.
- [8] T. Scherzer, L. Scherzer, and D. Tjaden. Learning in BEBE. In T. A. Marsland and Jonathan Schaeffer, editors, *Computers, Chess, and Cognition*, chapter 12, pages 197–216. Springer Verlag, 1990.
- [9] R. Sutton. Learning to Predict by the Method of Temporal Differences. *Machine Learning*, 3:9–44, 1988.
- [10] G. Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Computation*, 6:215–219, 1994.

- [11] S. Thrun. Learning to Play the Game of Chess. In G. Tesauro, D. Touretzky, and T. Leen, editors, *Advances in Neural Information Processing Systems 7*, San Francisco, 1995. Morgan Kaufmann.
- [12] A. Tridgell. KnightCap—A parallel chess program on the AP1000+. In *Proceedings of the Seventh Fujitsu Parallel Computing Workshop*, Canberra, Australia, 1997. ftp://samba.anu.edu.au/tridge/knightcap_pcw97.ps.gz. source code: <http://wwwsysneg.anu.edu.au/lsg>.