Automatic Generation of an Evaluation Function for Chess Endgames

Matthias Lüscher

Supervisors:
Thomas Lincke and Christoph Wirth

ETH Zürich
February 2000

Additions and corrections: March 2000
English translation: June 2004

Abstract. One not yet satisfactory solved problem of the evaluation function construction for chess is the efficient selection of features and the assignment of weights. Michael Buro presented in his paper "From Simple Features to Sophisticated Evaluation Functions" a practical framework for the semi-automatic construction of evaluation functions for games. According to his approach, only some simple domain specific features are hand coded. Complex rules become automatically derived from the simple features and are combined in a linear evaluation model. Each rule has a weight that will be fitted automatically according to a large set of classified game positions. This approach was very successful in the domain of Othello. The aim of the work presented here is therefore to investigate, whether this evaluation function generator can also be used for the domain of computer chess. The following text contains some helpful hints for an efficient implementation of such an evaluation function generator for chess and also the necessary mathematical formulas needed for the weight fitting algorithm.

Table of Contents

1 Introduction

Due to the high computing power of up to date processors chess programs are able to think ahead four and more moves even with short thinking time. Considering that, they are tactically far superior to even very strong human chess players. Nevertheless, the best human chess players are still able to occasionally win against the strongest chess programs. The reason for this is that a chess program has only a relatively modest positional knowledge compared to a world class chess player.

One of the most challenging tasks when creating a chess program is to convert the chess knowledge, which can be learned from many books, into computer code. Currently most computer chess programmers do this by explicitly coding many rules and afterwards weighting this rules. To successfully achieve this task, the programmer needs good positional chess knowledge and he shouldn't flinch from weeks of tedious optimizations.

The following work presents an evaluation method which automates the generation of rules as well as their weighting. The potential of the method is field-tested with chess endgames but the method is not limited to endgames.

It is important to mention that this work is based on a quite similar approach that has already been the key to one of the best Othello programs [1].

2 Functionality of an Evaluation Function

An evaluation function has to assign a value to a given chess position. This value should be a measure of the winning chance of the player that has to move next. It is sufficient if an evaluation function is only of positional nature. The most efficient way to get the tactical things right, is to use some kind of an α-β tree search. Combining the tree search with the evaluation function you get a move generator: Start the tree search and every time the search reaches a leaf of the tree, call the evaluation function and finally backpropagate its value to the root of the tree.
The following few chapters illustrate the structure of the evaluation function used here.

2.1 Atomic Features

A chess position is well-defined through pieces, their locations and some additional information like possible en passant and castling moves and finally the player that has to move next. For many reasons these informations are not adequate input parameters for an evaluation function.
As a first step, we describe a given chess position with a finite number of meaningful facts, also called „atomic features“ [1]. These atomic features serve as input parameters for the evaluation function. Human beings show a good intuition in selecting a number of atomic features that are relevant to a given problem. Although each atomic feature should be very simple, their sum and combinations should allow an appropriate description of a given chess position.
To simplify the following discussion, we assume that an atomic feature can only apply or not apply to a given position.

2.2 Configurations

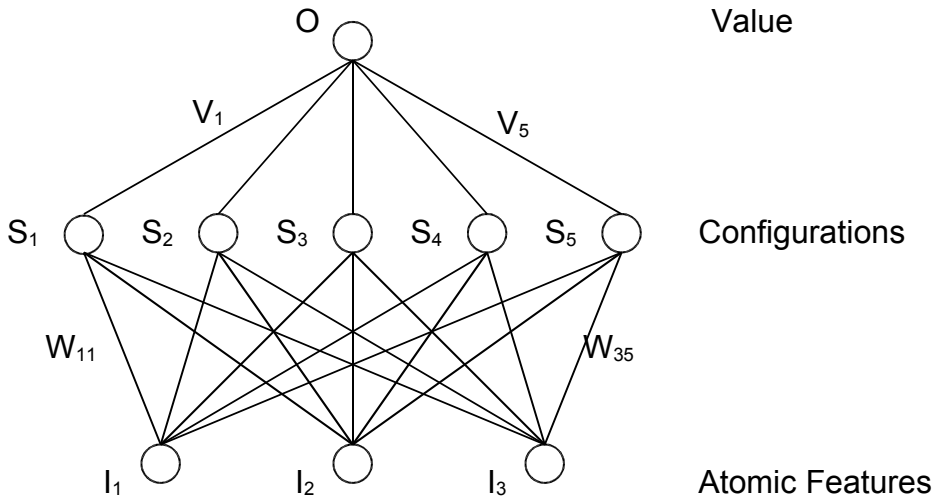
A configuration is a combination of several atomic features. Given a certain position, a configuration is called „active“ if all their atomic features apply. Through these configurations that combine simple atomic features the evaluation function is now able to describe complex coherences.
Let's assume that „the king is in the middle of the board“ is such an atomic feature. It becomes clear quite quickly, that this atomic feature is not sufficient to draw a conclusion: If a player has moved his king to the middle of the board right at the beginning of the game, this should be considered as suicide. On the other hand it might be essential to move the king to the middle of the board if you want to win a pawn endgame. Now we add a second atomic feature: „there are few pieces on the board“. Combining these two features to the configuration {„the king is in the middle of the board“, „there are few pieces on the board“} we are able to make a meaningful statement: If this configuration is active it has a positive influence over the game of the respective player.
Each configuration is associated with a value that represents its impact.
While the atomic features are hand coded, the creation of configurations and their evaluation is left to the computer.
Even with a few atomic features thousands of configurations are possible.

2.3 Evaluation

When evaluating a certain position we collect all atomic features that apply and pass them to the evaluation function. From this set of features the evaluation function derives the active configurations. The return value of the evaluation function is the sum of the values of the active configurations.
Since the evaluation function is called very often, an efficient locating of the active configurations is crucial. An optimized implementation is described in the following chapter.

2.4 Interpretation as a Multilayer Perceptron

Our evaluation function can easily be interpreted as a feedforward network with one hidden layer – also known as multilayer perceptron.
The atomic features I_1-I_n are the inputs. Assume I_i to evaluate to one if the atomic feature i applies, otherwise I_i is zero. The weights W_{ij} are either one or zero depending on whether the atomic feature i is part of the configuration S_j. S_j becomes one in case $\sum W_{ij}I_i$ is equal to the number of atomic features that the configuration S_j contains, otherwise zero. O is the return value of the evaluation function and is calculated as $O=\sum S_j V_j$.



To compare our evaluation function to a multilayer perceptron might look somehow enforced. The weights W_{ij} of real multilayer perceptrons are usually calculated with an error backpropagation algorithm and they take non discrete values. [2] suggests that despite our discretisation of the weights W_{ij} the possibilities of a real multilayer perceptron are well approximated.
This comparison helps to learn from experiences made with multilayer perceptrons.

3 Description of the Implementation

The following sub chapters provide an in-depth descriptions of the implementation of our evaluation function. We have to divide the evaluation process into two consecutive procedures. First we have to configure the evaluation function using a number of weighted chess positions. Within this step the configurations are generated and assigned to a value. After this step has been completed, the evaluation function can be used to evaluate an arbitrary chess position.

As previously mentioned, an efficient implementation is crucial because the evaluation function needs to be called very often during the configuration procedure and even more intensely during a brute force search. This evaluation function was realized with a relatively tight time frame and therefore a lot of room for improvements might still remain. Nevertheless the following text should provide some interesting approaches.

3.1 Atomic Features

Let's assume that we have n different atomic features. For this reason it is at least theoretically possible to generate 2^n configurations. Even if we try to keep n small we quickly run into such a big number of configurations that would make any efficient processing impossible. Unfortunately it turned out to be difficult to separate meaningful configurations from completely useless ones in advance.
To reduce the number of configurations we have chosen the following approach (cp. [1]): All n atomic features are divided into groups with $k < n$ atomic features. A certain configuration can only contain atomic features that belong to the same group. In order to not curtail the evaluation function too much we have to group atomic features that logically belong together.

In this evaluation function each piece on the board is associated with a group of atomic features. With this approach we can model the trait that is typical for each piece type.

Each atomic feature is encoded as a single bit. If an atomic feature applies for a given position, the bit is set to true and otherwise to false. The number of atomic features per piece has been limited to 32. For a given position each piece on the board therefore owns a bit vector with a length of four bytes.

The following table shows the encoding of the bit vector of a king:

bit	meaning
0	color of the piece is at a piece advantage
1	color of the piece is at a piece disadvantage
	piece is attacked by:
2,3	pawn
4	knight
5	bishop
6	rook
7	queen
8	...
	piece is covered by:
9,10	pawn
11	knight
12	bishop
13	rook
14	queen
15	...
16	it's the opponents turn
17-20	distance to the corner
21-26	distance to the opponents king
27	castling has been performed
28-31	location

3.2 Configurations

As mentioned in the previous sub chapter, configurations are separately combined and evaluated from atomic features of each piece. Each piece type – pawn, knight, bishop, rook, queen and king – owns a set of configurations. The evaluation function makes no difference between black and white pieces. Some care was taken to code the atomic features in a way that they act symmetrically with regard to color and piece types. Especially the pawn needed some special treatment because its moves are heavily dependent on its color.
With 32 atomic features per piece type we theoretically still manage to combine 2^{32} distinct configurations. Because we are unable to manage 2^{32} configurations we introduce two additional criteria to reduce the number of possible configurations (cp. [1]):

- We limit the maximum number of atomic features that belong to a configuration.
- We select a number of positions that are relevant to a certain area of interest. In order to be included into the set of valid configurations, a possible configurations has to be active in at least a given percentage of the above selected positions.

A configuration can also be encoded using a bit vector. This procedure allows a very efficient locating of active configurations. Given a certain position, each piece can provide the evaluation function with a bit vector that has marked all applying atomic features with true. On the other hand the evaluation function stores a list of all the possible configurations for each piece type. With the following simple comparison we can check whether the configuration B is active for piece A:

```
if ( (B.Bitvector & A.Bitvector) == B.Bitvector )
{
      configuration B is active for piece A
}
else
{
      configuration B is not active for piece A
}
```

The operator & stands for a bitwise comparison of the corresponding elements of the two bit vectors.
Because the number of possible configurations might still be very big we introduce an other optimization: The list of possible configurations not only contains the value of each configuration but also a jump address. If a certain configuration turns out to be inactive, a big number of the following configurations can be over jumped: If the configuration C turns out to be inactive, all the configurations that contain all the atomic features of configuration C plus additional atomic features will certainly be inactive as well.

3.3 Evaluation

Most chess programs do the tree search with negascout instead of minimax [3]. For our evaluation function this means that all evaluations have to be done from the point of view of the color that has to move next.
Given a certain position, let's assume that white has to move next: First we figure out all the active configurations for the white color and we add up all the values of these active configurations to the sum A. Next we do the same for the black color and call the resulting sum B. The value of the given position is then returned by the evaluation function and is calculated as A minus B.
Here an other optimization can be taken into account: A piece usually has thousands of possible configurations and the evaluation of the active configurations as well as the accumulation of their values is computationally expensive. The result of such an evaluation for a certain piece only depends upon the bit vector of the piece which is derived from the atomic features that apply for the given position. From now on we determine the active configurations for a piece and add up their values to the sum C. The sum C is not only used to calculate the value of the current position but it is also stored to the memory in a cache like manner: The address is derived form the bit vector and we store the value C plus the bit vector, which serves as a unique key.
When evaluating the sum of the active configurations for a given piece for any other position we first query the cache like storage to eventually retrieve an already calcu-

lated value instead of performing the whole calculation again. After some tuning it turned out that about 99% of the queries were successful and a vast speed up to the evaluation function was the result.

3.4 Determination of the Configuration Values

As previously mentioned, the meaning of a configuration is represented as a value. The introduction promised an automatic calculation of these configuration values. A prerequisite for such an automatic calculation is a set of somehow rated chess positions which should then be approximated by the evaluation function as good as possible. Using only precalculated endgames, an exact rating for each position is available: The value of the position is derived from the number of half moves that are needed to either promote a pawn or to achieve a checkmate.

For the calculation we introduce the following notation:

i	Index number of the chess positions
k	Index number of the possible configurations
$h_{i,k}$	Counter, how often configuration k is active on position i. Configurations that appear on pieces of the active color are counted positive and for the non active color negative.
r_i	Target value of position i
w_k	Weight or value of configuration k
$\underline{w}$	Weight of the configurations using vector notation

The value of position i is now calculated as

$$e_i(\underline{w}) = \sum_k w_k h_{i,k} \ .$$

The target value of the same position is r_i and we define a quadratic measure for the difference between the calculated and the target value. The sum of the quadratic errors over all i positions is

$$f(\underline{w}) = \sum_i (r_i - e_i(\underline{w}))^2 \ .$$

To create a good evaluation function we need to minimize $f(\underline{w})$. This turns out to be a quadratic optimization problem which is usually solved with a conjugate gradient method. The formulas required for the implementation of this method are given in the following text.

The method of conjugate gradients is composed of the following steps [7]:

a) For the first step ($m=1$) we can make no assumption on the configuration values and therefore they are initialized with zero:
$$\underline{w}^1 = \underline{0}$$
The first direction of descent is the negative gradient of the error function $f(\underline{w})$:
$$\underline{d}^1 = -\nabla f(\underline{w}^1)$$

b) A step with the descent direction $\underline{d}^m$ is performed. The configuration values after this step evaluate to
$$\underline{w}^{m+1} = \underline{w}^m + \alpha_m \underline{d}^m .$$

α_m is chosen in order that the error function $f(w)$ is minimized along the descent direction (for clarity's sake we omit the index m for α, $\underline{d}$ and $\underline{w}$):

$$\alpha = \frac{\sum_i (r_i - \underline{w}^T \underline{h}_i)\underline{d}^T \underline{h}_i}{\sum_i [\underline{d}^T \underline{h}_i]^2}$$

The new descent direction is:

$$\underline{d}^{m+1} = -\nabla f(\underline{w}^{m+1}) + \mu_m \underline{d}^m \quad \text{with} \quad \mu_m = \frac{\nabla f(\underline{w}^{m+1})^T \nabla f(\underline{w}^{m+1})}{\nabla f(\underline{w}^m)^T \nabla f(\underline{w}^m)}$$

c) Repeat step b) until a sufficient accuracy is achieved.

Remarks:

- The convergence of this method can be further improved by using a jacobi pre-conditioning
- The component wise representation of $\nabla f(\underline{w})$ is:

$$\frac{\partial f(\underline{w})}{\partial w_l} = \sum_i 2(r_i - \sum_k w_k h_{i,k})(-h_{i,l})$$

3.5 Stages of the Play

Our evaluation function has been tested with many stages of the chess play. It has come clear quite quickly that it is a bad idea to process all chess positions as a whole (cp. [1]). In fact there are many common facets for the pawn endgames KPK, KPPK, KPKP and KPPKP but a KBBK endgame is completely different from an opening. Taking this circumstance into account, we divide all possible chess position into groups. The configurations together with their values are then evaluated for each group separately. We could figure out many possible ways to do this subdivision. Possible criteria are endgame types, number of pieces on the board, king safety, pawn structures and so on.

When using our evaluation function in conjunction with a brute force search we have to keep in mind that selecting the group that a given position belongs to each time we call the evaluation function might be quite expensive. We therefore select the group when we start the tree search and quietly assume that the positions we encounter during the tree search are somehow related to the starting position. Because it is possible that we enter one or many other groups during our tree search we have to assure that the evaluation function still returns useful values. To achieve this we have made our groups somehow overlapping.

One other reason to choose the group at the root of the tree search is that it is not always possible to compare values returned from the evaluation function across group boarders. The returned values might be appropriate for the respective group but might not be comparable to the return value of another group.

4 Choice of the Training Set

Yet another challenging task is to prepare a good training set. The training set must contain a big number of positions that are relevant to the respective area of interest and we have to associate each position with a value. Because we need thousands of positions an automatic procedure to assign a value to each position is mandatory. A perfect solution is only possible for endgames that have been completely solved and

that have been stored to a database. From this database we can retrieve how many half moves are needed to either checkmate the opponent or to promote a pawn. If the number of half moves is odd, the active color will win the game. The following transformation makes this information more useful to an evaluation function:

```
if ( databaseValue % 2 == 1 )
      newValue = maxValue + OFFSET - databaseValue;
else
      newValue = -(maxValue + OFFSET - databaseValue);
```

maxValue is the biggest number of half moves that shows up in the database.

A more difficult task is to create a training set for openings or middle games. Satisfactory results have been achieved when examining games as a whole. Each individual position of a game receives a value that is derived from an averaged piece balance, the number of moves left to checkmate and the total number of moves of the game.
Although this simple approach yields good results one should keep in mind, that there are much more elaborate techniques to solve such problems. One example is Temporal Difference Learning [2], [10].

5 Interoperability with Conventional Algorithms

Because our evaluation function is just a replacement for a conventional evaluation function there is hardly any change needed to incorporate it in any chess program. In particular this evaluation function is compatible with modern tree search algorithms like α-β, nullmove, hash tables and so forth.
Some care must be taken if one tries to combine a hand optimized evaluation function with this automated one. One should first create the automated evaluation function and then apply the hand optimizations with respect to the generated evaluation function. In this work the automatically generated evaluation function has been combined with a piece balance.

6 Results

6.1 Endgames

We did some intensive testing with endgames to check out the effectiveness of our evaluation function. Endgames have two massive advantages: First of all it is easy to generate a training set and secondly for every position that can be retrieved from a database the best move is known.

The test layout was the following:

1. With the help of endgame databases we created a number of training sets for different endgame types. For simple endgames like KRK we were able to include all the possible positions into the training set. With more complex endgames like KPPKP we were forced to include only about 1% of all possible positions.
2. We rated each individual position as described in chapter 4.
3. We used the different training sets to derive the values of the configurations
4. The characteristics of the trained evaluation functions were merged into a table.
5. For the KBBK endgame we created a test set which shared no positions with the training set. We checked the characteristics of the trained evaluation function on the test set and compared it to the characteristics on the training set.

Here some comments on the terms used in the following statistics:

Configurations: Per endgame and piece type we indicate how many configurations have been generated.

Average error, average quadratic error: For each training set we calculated the average and average quadratic error between the value of the training set and the value of the evaluation function. The untrained evaluation function returns zero for any given position.

Win, draw, loss comparison: In this comparison we check the correctness of the evaluation function with regard to the outcome of the game.

Half move search: Given a specific position we perform a minimal tree search with the depth of exactly one half move to find the a priori best move. The piece balance has been taken into account when doing this test. The a priori best move is checked against the database to find out if it is really the best move. We thus receive a hit rate that indicates how often the half move search would find the best move. To get an idea of how good the hit rate is, we compared it to the hit rate we would achieve when doing just a legal random move.

6.1.1 KRK Endgame

To checkmate the opponents king in a king and rook against king endgame we have to push his king against the boarder of the board. To achieve this some characteristic move sequences are necessary. Any serious chess program will do a fine job when playing this endgame but we remarked that the program with our evaluation

function outperformed some strong freeware chess program when playing this endgame.

Statistics for the KRK endgame (training set)				
Number of configurations:		King:	798	
		Rook:	659	
Average error:		Before training:	15.6 half moves	
		After training:	2.13 half moves	
Average quadratic error:		Before training:	286 half moves^2	
		After training:	8.59 half moves^2	
		Evaluation function		
		Win	Draw	Loss
Database	Win	43.9%	0%	0%
	Draw	0%	5.48%	0.108%
	Loss	0%	0.112%	50.4%
Evaluation function hit rate:		99.8%		
Half move search:		Evaluation function hit rate:	68.7%	
		Hit rate when selecting random move:	27.5%	

6.1.2 KBBK Endgame

The king with two bishops against king endgame, which is slightly more difficult, was handled as well as the king and rook against king endgame.

Statistics for the KBBK endgame (training set, *test set*)							
Number of configurations:		King:	810				
		Bishop:	729				
Average error:		Before training:	16.4 half moves				
		After training:	2.47 half moves	*2.5 half moves*			
Average quadratic error:		Before training:	306 half moves^2				
		After training:	14 half moves^2	*14.8 half moves^2*			
		Evaluation function					
		Win		Draw		Loss	
Database	Win	41.9%	*41.8%*	0%	*0%*	0%	*0%*
	Draw	0.008%	*0.013%*	10.6%	*10.5%*	0.406%	*0.413%*
	Loss	0%	*0%*	0.205%	*0.23%*	46.9%	*47%*
Evaluation function hit rate:		99.4%	*99.3%*				
Half move search:		Evaluation function hit rate:				62.3%	*65.5%*
		Hit rate when selecting random move:				32.2%	*31.6%*

When comparing the effectiveness of the evaluation function on the training set to the effectiveness on the test set we can happily remark that the performance is almost the same on both sets. This means that our evaluation function is very good at extrapolating its knowledge to unlearned positions.

6.1.3 KBNK Endgame

The endgame where a king together with a bishop and a knight has to checkmate the opponents king is ranked as one of the most difficult endgames with four pieces.

To checkmate the opponent one must firstly push the opponents king to the boarder and then secondly try to drive it into the corner that can be reached with the bishop. It is impossible to checkmate the king in the corner that can't be reached by the bishop.

Statistics for the KBNK endgame (training set)				
Number of configurations:		King: 768 Bishop: 546 Knight: 528		
Average error:		Before training: 19.6 half moves After training: 4.90 half moves		
Average quadratic error:		Before training: 495 half moves^2 After training: 50.3 half moves^2		
		Evaluation function		
		Win	Draw	Loss
Database	Win	44.1%	0.0032%	0%
	Draw	0.214%	8.27%	1.8%
	Loss	0%	0.487%	45.1%
Evaluation function hit rate: 97.5%				
Half move search:		Evaluation function hit rate: 59% Hit rate when selecting random move: 26.3%		

After the training our evaluation function succeeds in pushing the opponents king to the boarder but it is unable – despite the good statistic results - to drive it afterwards to the correct corner.

6.1.4 KPPKP Endgame

After the training of this endgame the subjective impression when playing the endgame is very good. The important concepts of this endgame are handled properly. Also the extrapolation for more complex pawn endgames is encouraging.

Statistics for the KPPKP endgame (training set)				
Number of configurations:		King: 687 Pawn: 4099		
Average error:		Before training: 43.4 half moves After training: 18.2 half moves		
Average quadratic error:		Before training: 2091 half moves^2 After training: 537 half moves^2		
		Evaluation function		
		Win	Draw	Loss
Database	Win	50.1%	0.761%	0.975%
	Draw	3.3%	5.11%	6.3%
	Loss	0.926%	1.2%	31.3%
Evaluation function hit rate: 86.5%				
Half move search:		Evaluation function hit rate: 55.8% Hit rate when selecting random move : 39.8%		

6.2 Opening and Middle Game

After the successful introduction of the evaluation function to chess endgames we were curious whether we could also use it for openings and middle games. To test

this we swapped the evaluation function of Chessterfield (approx. 1900 ELO) for our new evaluation function. During the next days Chessterfield alternately played against strong freeware chess programs (> 2200 ELO) and learned from the games played. After approximately 500 games we remarked the following: The basic concepts of chess playing were handled properly and we guessed that it played about 250 ELO points stronger than before the learning process. With this improvement it played already better than with the old evaluation function.
Nevertheless we have to keep in mind, that there is still very much optimization potential. Much time has been used to fine tune the old evaluation function while the new one could still be much improved: run time optimizations, better atomic features and different training sets would help a lot.

7 Conclusion

The results presented here suggest that automatically generated evaluation functions like the one shown here would have good prospects in the domain of computer chess.
The technique first presented in [1] and extended here has two radical characteristics: On one hand it is almost as mighty as a neural network (here a multilayer perceptron) but on the other hand the calculation overhead is much smaller than the one of a common neural network [9]. The technique presented here makes it possible to combine brute force with an intelligent learning algorithm.
As far as I know – the evaluation functions of commercial chess programs are kept secret – almost all evaluation functions are hand coded and underwent long lasting hand optimizations. However they do not have to fear that they are outperformed by automatically generated evaluation functions in the near future. In the domain of automatically generated evaluation functions there is still a lot of development work waiting to be done and this might take quite some time.

8 Literature

[1] M. Buro; From Simple Features to Sophisticated Evaluation Functions; Lecture Notes in Computer Science LNCS 1558; Springer Verlag; 1998
[2] G. Tesauro; Temporal Difference Learning and TD-Gammon; Communications of the ACM; Vol. 38, No. 3, 1995
[3] D. Steinwender, F. A. Friedel; Schach am PC; Markt und Technik Verlag; 1995
[4] C. Wirth; Exhaustive and Heuristic Retrograde Analysis of the KPPKP Endgame; ICCA Journal; Vol. 22, No. 2; 1999
[5] J. Nunn; Taktische Schachendspiele; Falken Verlag; 1985
[6] P. Keres; Praktische Endspiele; Kurt Rattmann Verlag Hamburg; 1973
[7] I. N. Bronstein, K. A. Semendjajew, G. Musiol, H. Mühlig; Taschenbuch der Mathematik; Harri Deutsch Verlag; 4. Auflage, 1999
[8] M. Bain, S. H. Muggleton, A. Srinivasan; Generalising Closed World Specialisation: A Chess End Game Application; 1995
[9] S. Thrun; Learning to Play the Game of Chess; Advances in Neural Information Processing Systems (NIPS) 7; MIT Press; 1995
[10] Jonathan Baxter, Andrew Tridgell, Lex Weaver; KnightCap: A chess program that learns by combining TD(λ) with game-tree search

Appendix

A Software

Endgame Database of Christoph Wirth

Thanks to the endgame database programmed by Christoph Wirth we were able to generate training sets with all possible positions ranging from KK to KPPKP endgames. Each position value stored in this database is encoded as a single byte. This byte contains the number of half moves necessary to either checkmate the opponent or to promote a pawn. A sophisticated access routine makes it possible that the value of an arbitrary position can be retrieved from the database. [4] should be consulted for more information on this database.

The address of the author is:

Christoph Wirth
ETH Zürich, Institut für Theoretische Informatik
CH-8092 Zürich

wirthc@inf.ethz.ch

WinBoard, XBoard

WinBoard respectively XBoard is a graphical frontend for many chess engines like for example Crafty. WinBoard has a good reputation among freeware chess developers since it makes it possible to automatically test engines against other engines. In the meantime this concept has also been introduced into many commercial chess programs. Another nice feature of WinBoard is, that it is available for multiple platforms.

WinBoard has been programmed by Tim Mann and it is freeware:

http://www.research.digital.com/SRC/personal/mann/chess.html

Chessterfield

Chessterfield is a simple chess program written by Matthias Lüscher. There are two implementations: One is available with a graphical frontend for the Windows operating system and the other is a WinBoard compatible command line application.
The strength of the program is approximately 1900 ELO points and it has been written in C++.
Chessterfield served as a test platform for the new evaluation function. A lot of Chessterfield code has been reused to write the new evaluation function. The copyright holder is Matthias Lüscher but whole code of the command line edition is licensed under the GPL and can be downloaded from the following web page:

http://www.luescher-online.com/computerchess.html

B Used Standards

Portable Game Notation PGN

PGN is a format to digitally store chess data. It is optimized in a matter that it can be easily read by a computer program as well as by a human.
The standard is open and well documented and can be found here:

http://www.research.digital.com/SRC/personal/mann/Standard

Forsyth-Edwards Notation FEN

FEN is used to unambiguously describe single chess positions. The FEN specification is contained in the PGN specification.

C Hierarchy Chart of Chessterfield (Cutout)