# A simplex approach for the tuning of a chess evaluation function

Casper Willestofte Berg (s021692)

Hans Gregers Petersen (s021668)

January 23, 2006

Informatics og Mathematical Modelling

Technical University of Denmark

# Contents

# 1 Introduction

This report describes our study of the possibility of using the Nelder-Mead simplex algorithm to tune a chess evaluation function.

Chess has always been of interest of the academic community, and has been subject of analysis through centuries. Even before the rise of computers scientists have tried to create "chess brains" in form of chess machines, that would master the game of chess.

The current dominating trend in optimising chess brains is probably evolutionary theory. As we could not find any litterature describing the appliance of the Nelder-Mead simplex algorithm, we decided to study this approach for tuning.

We would like to thank Professor Henrik Spliid for kind supervision and the idea of studying the simplex approach. We would also like to thank Reader Robin Sharp for his consultation about the technical aspects of our computer grid.

January 2006

Casper Willestofte Berg and Hans Gregers Petersen

# 2 Computer chess

This section is an introduction to how computers play chess.

Ever since the arrival of the digital computer in the 1950's engineers and chess enthusiast have been making programs playing chess. In 1997 IBM's supercomputer "Deep Blue" defeated the former world champion Garry Kasparov, and today there are programs available for an ordinary pc that play world class chess. Still, constructing a strong chess playing program is a difficult task because chess is such a complicated game.

Chess is a game of perfect information, i.e. there is no randomness or information about the game that is hidden for either player. This means that the game of chess theoretically can be solved, so a computer could play a perfect game of chess. But this would require trying out all possible games of chess and recording the results. The total number of possible games is known as the *game-tree complexity* and was first estimated for chess by Claude Shannon to $10^{120}$. This is more than the number of atoms in the Universe, and thus making it practically impossible to solve. The big complexity stems from the big branching factor of 30-40 possible moves for each position on average.

## 2.1 Search algorithms

Computers play chess by making only a small part of the game tree by performing a search to a limited depth from the given position and applying an *evaluation function* at all the leaf nodes of the tree (some programs also evaluate internal nodes for pruning). By passing the best evaluation (depending on whose turn it is) upwards in the tree at each node, the computer can select the best move. This technique is known as *minimax*.

There have been some unsuccesful attempts to make a chess program that "thinks" like a human, i.e. looks at a very few number of moves and thereby rejects most moves in a position on a complicated heuristic basis. This is also known as forward pruning. The problem with this approach is that you easily miss a winning combination, and even if that only happens a small percentage of the time, it will probably prove fatal in the course of a whole game of chess. So instead of examining a relatively small number of moves extensively, chess programs opt to examine a huge number of moves and applying a relatively simple evaluation function.

All of todays strong computer programs use some variant of the so called alpha-beta search algorithm. The name comes from a technique called alpha-beta pruning which reduces the game tree substantially and thus makes deeper searches feasible. Alpha-beta pruning is different from forward pruning because it only cuts off parts of the game tree that can be guaranteed not to be relevant, so it does not change the search result in any way.

### 2.1.1 Quiescence search

Especially at small search depths it is crucial not to try to evaluate a position where the evaluation is likely to change drastically after the next move, e.g. in a position where the queen can be captured. Thus, most chess brains use a search extension technique called *quiescence search* which searches deeper than the fixed limit depth until the position is quiescent and thereby safe to evaluate.

## 2.2 Evaluation functions

The evaluation function is applied at the leaves of the game tree as mentioned earlier. It analyses the position and returns a numerical value that represents which player has the better position. A very simple evaluation function considers only the material balance giving each piece the values which can be found in any book on chess for beginners:

pawn = 1, knight = 3, bishop = 3, rook = 5, queen = 9.

Material is by far the most important factor for deciding who is ahead in a position. Some programs use slightly different values than the above mentioned, e.g. bishops are usually valued a little higher than knights. Stronger chess programs use a lot of other factors such as

- Pawn structure
- Positions of the individual pieces on the board
- Attacked and guarded pieces and squares
- King safety
- Mobility

Many of these are considered general rules of thumb and are some of the same used by human players. The problem is to determine the relative importance of these factors in order for the program to achieve the best results. The evaluation function, $E$, thus can be expressed as a sum of factors, $f$, each with a certain weight, $w_i$:

$$E = \sum w_i f_i \tag{1}$$

It is these weights we seek to optimise. It should be noted, that the optimum of many of these weights changes over the course of the game, e.g. a bad pawn structure is often fatal in the endgame phase as opposed to the opening and middlegame. Still, overall optima must exist, though they this fact can make them more difficult to find.

# 3 Other methods

In this section we will summarise some of the known methods of optimising (chess) evaluation functions.

## 3.1 The evolutionary approach

In the evolutionary approach you create several evaluation functions and let them "battle" each other in several rounds. The purpose of this is to let the (best) functions evolve – hopefully into a near perfect one. Variants of this approach is described in [Kendall].

One creates a pool of functions and after each round of battles a number of the poorest ones are replaced by randomly generated new ones. Thus at a certain evolutionary stage the evaluation functions in the pool will have converged to an

optimum (possibly local) and further rounds will not lead to a better function. At this point you have the best of all the randomly generated functions.

But the best of a pool of completely random functions is not certain to be the best of all functions. As a measure to avoid this problem, mutation or breeding are also often used. Mutation is simply changing the surviving functions a bit for every round eg. by taking the variance of the functions into consideration. In breeding one combines some of the the winning functions into new functions and adds some random functions - and thus functions containing the "best genes" will hopefully be the final survivors.

As the evaluation functions battle each other, one will have the strongest functions left after ending the evolutionary approach. It cannot be assured that the resulting functions are globally optimal, though. The evolutionary approach is often resolved to, when nothing else works, and it is bad for local optimisation problems ([Durand]). The evolutionary approach has actually been applied together with Nelder-Mead Simplex with success, the former for finding the global optimum area and the latter for the final optimum[Durand], though not on the chess domain to our knowledge.

### 3.2 Grandmaster fitting

It has also been tried to fit evaluation functions to games played by grandmasters to reach a function that would evaluate like a grandmaster – and thus hopefully reach the optimal evaluation.

The evaluation function of "Deep Thought" [Hsu] (which was later further developed into "Deep Blue") was made by hill climbing and least squares fitting to the winners of over 800 grandmaster games. It was revealed that this fit to the grandmasters was actually not the same as playing stronger.

In [Gomboc] it is described how a fit – using hill climbing – to all important master games from 1966 to 2002 also resulted in the acknowledgement of the fact that the strongest evaluation function was reached before a perfect fit to grandmaster games.

The prior mentioned hill climbing fit is simply to slowly follow the gradients to find the (possibly local) optimum. This method is very slow converging – if ever – because it only handles one parameter at a time. The Deep Thought team only used this method for a couple of parameters (out of over 120), which were almost impossible to fit with their other methods.

## 4 The Nelder-Mead Simplex algorithm

There are several algorithms dealing with maximisation of multidimensional functions. One of them is due to Nelder and Mead and has many names: Simplex, Nelder-Mead and "The Amoeba Algorithm". We find it suitable for our domain because this method is relatively robust and does not use derivatives, only function evaluations is required as opposed to many other optimisation schemes. Since we cannot calculate or even expect to measure any derivatives with any reasonable precision for our evaluation function because it's response is both stochastic and probably nonlinear, this choice seems natural.

The algorithm works by creating a *simplex*, which is a polytope of $n + 1$ vertices in $n$ dimensions, e.g. a triangle in the plane and a tetrahedron in three

dimensions and so forth. The simplex must of course be nondegenerate, i.e. have a nonzero volume and be a convex hull of the $n+1$ vertices. By substituting bad vertices with new ones directed away from the worst vertex and towards the best ones the simplex "moves like an amoeba" - hopefully towards the optimum. The algorithm is terminated when some predetermined tolerance condition is satisfied.

For an evaluation function $f$, the algorithm is defined by the following steps[1]:

1. **Ordering** The vertices $x_0 ... x_n$ are ordered by their respective function evaluations and relabelled such that $f(x_0) \geq f(x_1) \geq ... \geq f(x_n)$.

2. **Reflection** Compute the centroid of the $n$ best points, $\bar{x} = \sum_{i=0}^{n-1} \frac{x_i}{n}$, and the reflection point
$$x_r = \bar{x} + \alpha(\bar{x} - x_n) \tag{2}$$
Evaluate $f(x_r)$. If $f(x_0) \geq f(x_r) \geq f(x_n)$ we accept the reflection and start over.

3. **Expansion** If $f(x_r) > f(x_0)$ we have found a new maximum and thus we expand in that direction. We compute the expansion point $x_e$:
$$x_e = x_r + \beta(x_r - \bar{x}) \tag{3}$$
If $f(x_e) > f(x_r)$ accept $x_e$ otherwise accept $x_r$.

4. **Contraction** If $f(x_r) < f(x_{n-1})$, i.e. the reflected point is the new worst point, then perform a contraction between $\bar{x}$ and $x_n$:
$$x_c = \bar{x} + \zeta(\bar{x} - x_n) \tag{4}$$
If $f(x_c) \geq f(x_n)$ accept $x_c$.

5. **Shrink Simplex** Finally the simplex is shrunk around the best point $x_0$ by evaluating $f$ at the $n$ new vertices computed by:
$$x_i = x_0 + \eta(x_i - x_0), \qquad i = 1...n. \tag{5}$$

The standard values for the coefficients are $\alpha = 1, \beta = 1, \zeta = 0.5, \eta = 0.5$.

## 4.1 Convergence

This algorithm is widely used because it is easy to set up and gives good results in many cases. However, Mckinnon showed in 1998 that the algorithm "can fail to converge or converge to non-solutions on certain classes of problems"[Byatt]. Some limited convergence results exist for a restricted class of problems in one or two dimensions but unfortunately nothing in general. Even if such results were available, we cannot classify a chess evaluation function since we know very little about it's behaviour.

Recently, a new variant of the Nelder-Mead algorithm has been developed which is provably convergent, appears to work well in practice, maintains the nice features of the algorithm, and avoids some of its problems ([Byatt] p. 36). We have however chosen to refrain from using this new improved variant, since we found an advanced Nelder-Mead implementation in Java by Dr. Michael Thomas Flanagan ([Flanagan]), which saved us a lot of time, and we felt that we could trust it to be correctly implemented.

[1] http://www.research.ibm.com/infoecon/paps/html/amec99_bundle/node8.html

# 5 Design of experiments

This section describes how we designed the experiments we have performed.

## 5.1 Setup of the simplex

As described earlier this study considers the Nelder-Mead simplex.

### 5.1.1 Parameters to optimise

To avoid a very slow converging simplex we pruned the parameters of the evaluation function. Instead of optimising around thirty parameters we chose twelve which we considered as being possible to optimise and at the same time covering different parts of a game. We will not go into details about the selected parameters other than mentioning, that it was positional parameters such as king safety, pawn structure and mobility. We did not optimise on the values assigned to the different pieces, though this is a standard area for investigation too.

If more time is available one can choose to optimise all parameters in one simplex. Then it should be likely to find synergie effects between parameters etc. When doing so, one could start with the parameter values we have found optimal in this study and thus probably reduce the amount of calculations to be used on them.

### 5.1.2 Initial simplex

As the initial simplex we took the original twelve parameter values and added/subtracted a stepsize of around 30-40 percent of the value. This simplex was then used as the starting point, and after a while (usually around 200-400 iterations) we determined that the simplex had converged. In one experiment, we took the parameter values of one of the vertices this converged – and rather small – simplex and made this the centroid of a new initial simplex, where we used a step size of around 10-15 percent of the parameter values.

### 5.1.3 Games pr iteration

The Nelder-Mead algorithm needs a response for every vertex, and said response should be (near) continuous – or at least less discrete than the outcome of a single game of chess (win, draw or loss). For these reasons we choose to add the score of several chess games in each vertex. This should also make the certainty of improvements in the parameters greater.

**50 games** A game of chess has three possible outcomes for the players: win, draw and loss. Assuming that two players are given the same conditions (time to think, equal start positions and so on), and that both players play with unchanged strength, the chances of win/draw/loss are the same for each game $(p_1, p_2, 1-p_1-p_2)$. In that case, the number of wins $(X_1)$, draws $(X_2)$, and losses $(X_3)$ for a player in a match of $n$ games will be multinomial $(n, p_1, p_2, 1-p_1-p_2)$ distributed. Assigning 1 point for a win, 1/2 for a draw, and 0 for a loss, the score in such a match will be $X_1 + X_2/2$ and the win ratio $\frac{X_1+X_2/2}{n}$. We can

now find the variance of the win ratio: The variance of the individual scores $X_i$ are each binomial distributed, and thus each of them has the variance

$$\mathrm{Var}(X_i) = np_i(1-p_i) \tag{6}$$

and covariance

$$\mathrm{Cov}(X_i, X_j) = -np_ip_j \tag{7}$$

Using these equations the variance the of the win ratio becomes

$$\mathrm{Var}\left(\frac{X_1 + X_2/2}{n}\right) = \tag{8}$$

$$\mathrm{Var}(X_1/n) + \mathrm{Var}\left(X_2/2n\right) + 2\,\mathrm{Cov}(X_1/n, X_2/2n) = \tag{9}$$

$$\frac{\mathrm{Var}(X_1)}{n^2} + \frac{\mathrm{Var}(X_2)}{4n^2} + \frac{\mathrm{Cov}(X_1, X_2)}{n^2} = \tag{10}$$

$$\frac{1}{n}\left(p_1(1-p_1) + \frac{1}{4}p_2(1-p_2) - p_1p_2\right) \tag{11}$$

From this we see that the maximum variance occurs when $p_1 = 0.5$ and $p_0 = 0$ which is also what one would expect. It is of course hard to measure any increase in the win ratio the standard deviation is much larger in comparison. On the other hand the standard deviation decreases only by a factor $1/\sqrt{n}$ so one must weight that against the increase in computation time.

As it is seen in figure 1, the standard deviation is rather large compared to the increase in the win ratio we can expect to gain by the optimisation. Thus we initially decided to use both the test suite by Dr. John Nunn and that by Jeroen Noomen arriving at a total of 50 start positions (see section 5.3). This yields a standard deviation of 0.07 in the worst case, which is still rather large, but probably enough to detect convergence, in which case the number of games can be increased and a smaller start simplex can be used for faster convergence.

**Increasing the amount** Though 50 games of chess seemed as a good idea, we encountered some problems using a low number of games (as can be read in section 7). First we increased the number of games to 400, but for other reasons (see section 5.3.2) this was rejected. As a consequence we decided to increase the population of possible games to 1,000. We decided to increase the number of games to be played in each simplex vertex as well. As we consider a fairly large amount of games, it is reasonable to assume a normal distribution of the errors. The measure we are really interested in is the proportion of points out of the total amount possible. Hence we chose to use the formula for sample size determination for a proportion:

$$n = p\,(1-p)\left[\frac{z_{\alpha/2}}{E}\right]^2 \tag{12}$$

If one considers the worst case, it is seen that it occurs when $p\,(1-p) = 1/4$. We decide that the maximal error should at most be $1/20 = 0.05$ at 95% conficence, and this gives us the sample size needed:

$$n = \frac{1}{4}\left[\frac{z_{2.5}}{E}\right]^2 = \frac{1}{4}\left[\frac{1.96}{0.05}\right]^2 = 384.16 \tag{13}$$

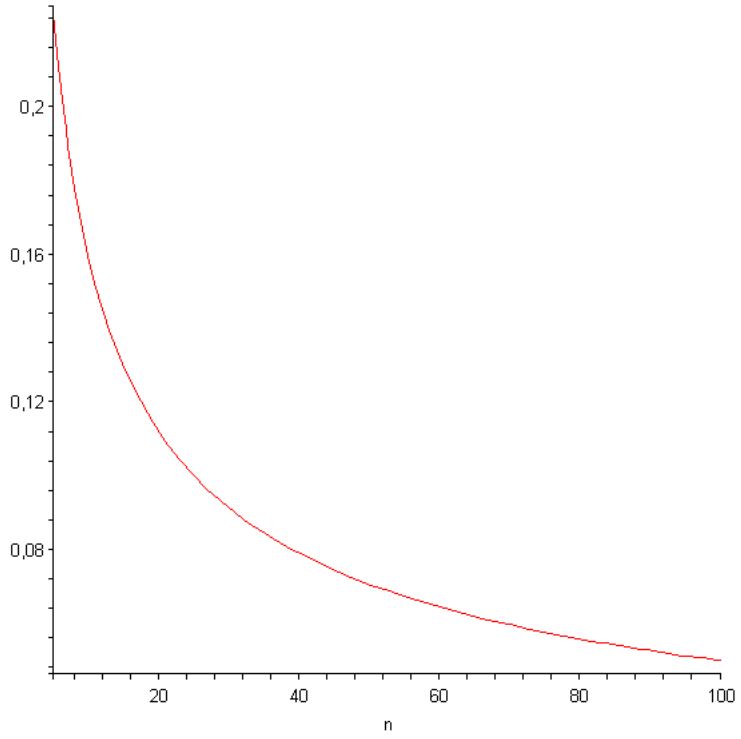


Figure 1: Worst case variance of win ratio against the number of games played.

So it is seen that we need a sample size of at least 385 games per vertex. We decided to randomly draw these games out of a pool of 1,000 games in each iteration.

## 5.2 The brains

In our study we use "three" brains:

- Alf with ordinary parameters
- Alf with test (simplex decided) parameters
- OliThink version 4.1.2 (Java version)

Alf is written by coauthor Casper W. Berg of this report, and the ordinary parameters used are his estimates handtunet by trial and error. OliThink is written by Dr. Oliver Brausch, and was the fastest and most reliable chess engine we could find written in Java.

In each simplex iteration we let Alf (with the test parameters) play an equal amount of games against the ordinary Alf and OliThink (the brains are randomly assigned). We did this to try to avoid fitting Alf to a single brain.

Though a fit to a specific brain could be useful, ie. in a tournament in which you know your opponent, we chose to mix the brains to reach a more general optimum.

The Nelder-Mead method could of course try to do a trade-off in which it "decides" that loosing to one type of opponent is okay as long as it wins more against the other. In fact we saw something that looked like this kind of effect in an early test run of our system. One should do checks, as described in section 5.4, for this kind of effect after a possible optimum is found.

### 5.2.1 Search depth

To assure a quick run of the games we fixed the search depth of all participating brains to four plies (four half moves). We made sure quiescence search was enabled in all brains such that they are allowed to search deeper than four plies if necessary. Our choice of fixing the search depth is to give both brains the same amount of tactical insight, thereby leaving more up to the positional evaluation. Four plies is a compromise between speed and tactical insight. As mentioned earlier if one adds a ply to the search depth, the computation time will increase by the branching factor (which will be above 10 in bad cases).

## 5.3 Test suites

In computer chess communities it is continuously discussed if a chess brain should be allowed to use opening books, learning capabilities, and other specialities which are really not the "brain"itself – the evaluation function. This is of course an important issue for evaluating the strength of a computer chess brain as a whole, as it is sometimes claimed that the very strong programs are only strong because they have extensive opening books and quickly learn their opponents' style.

In our experiment we chose not to allow the use of opening books – As we wish to optimise the evaluation function of Alf. Since almost all brains need an opening book to get a fair start on a game – most brains play really awful openings if they are based on the evaluation function – we chose to use the test suites which we describe in the following.

### 5.3.1 Nunn and Noomen

For the first rounds – where we calculated 50 games in each vertex – we used two test suites: one invented by Dr John Nunn and another by Jeroen Noomen[2]. These test suites are made for this purpose and are designed to cover different types of positions that occurs in chess. The test positions are also designed so that each side has about equal chances from the position i.e. a certain position and side does not infer a significantly larger probability of winning. Furthermore the chosen suites are acknowledged by the computer chess communities for testing evaluation functions.

[2] www.computerschach.de

### 5.3.2 A 400 games test suite

We then tried creating a test suite of 200 positions. The 50 of them were the above mentioned Nunn and Noomen suites, the last 150 positions were taken from the opening book: "Nunn's Chess Openings". The 150 positions were carefully selected from the book such that they were all regarded as positions with equal chances for both sides.

In this test suite we tried to mirror the positions, such that the brains played all positions as both black and white. However it turned out that many of the results were only a result of which brain got "the right" side, ie. the brain getting white in position X was certain to win said position. Therefore we rejected this type of mirroring and created the test suite described below.

### 5.3.3 Our own 1000 games test suite

Our suite consisting of 1000 test positions was created from a database of top grandmaster games (all players with rating above 2600). We used the hashtable function from Alf to store the position at ply nr. 19 in every game and skipped any game where the same position had occured before. This gave us 1000 unique start positions taken from real grandmaster games, that we could consider roughly equal due to the strength of the players in the database.

## 5.4 Test of improvement

To test the results of the parameter change reached by the Nelder-Mead algorithm, we decided to let the (possibly) improved version play 1,000 games against OliThink and 1,000 against the original Alf. This kind of test should reveal if the new version is actually improved and if so, if it has improved against both brains or only against one of them – ie. if the Nelder-Mead has made a kind of trade-off between the two brains.

The maximal error on the meassured proportion, $x/n$, at 95% confidence would then be given by:

$$E = \left| \frac{X}{n} - p \right| \leq z_{\alpha/2} \sqrt{\frac{p\,(1-p)}{n}} \tag{14}$$

The worst case would be if $p = 1/2 \Leftrightarrow p\,(1-p) = 1/4$, which gives us the following maximal error using a sample size of 1,000:

$$E = z_{\alpha/2} \sqrt{\frac{p\,(1-p)}{n}} = 1.96 \sqrt{\frac{1/4}{1000}} = 0.03 \tag{15}$$

So a 95% confidence interval on the measured proportion, $p'$, is – in the worst case – simply $p' \pm 0.03$. This should be enough to find any medium or large – and even some small – improvements in the evaluation function.

# 6 The grid

This section describes our technical solution to the problem of computing a large number of chess games as quickly as possible.

## 6.1 Grid vs. HPC

As the number of games to be played exceeded by far what could be computed on an ordinary PC, we had to consider alternate solutions. We considered using the high performance computing systems (HPC) at DTU[3]. As it would only be possible to gain access to a limited part of the HPC in a limited period of time, we regarded this solution to be infeasible.

Instead we focused on using grid technology: As our "tasks" – a number of games – can easily be broken up into small portions – a single game – this solution seemed to fit our problem very well. With a grid solution we would also be able to use the spare capacity of our own computers, and computers located around the campus. A grid solution is also more scalable and clients can be added and removed as needed.

## 6.2 The server

We decided to let the clients contact the server to fetch a task – a certain position from which two competing brains should play. We first considered using Suns JavaSpaces[4], but after consulting Reader Robin Sharp we decided to do a much simpler approach to transfer tasks: Java Remote Method Invocation (RMI).

The server offers two remotely invocable functions to the clients: one which gives a task to the client represented by a task object and one which accepts a result represented by a result object. This very general way of wrapping task and result ensures that we can add and remove parameters etc. without the need to change our interface. The interface can even be used for a completely different (grid) job. The chosen way of representing a problem is wellknown in other grid projects such as the "SETI@home" and "Folding@Home" projects. If the result of a task that has been given out is not returned within five minutes, the task is regarded as being timed out and is given to another client. The server chooses which side the engine with test parameters should play, which opponent it should play against, what situation they should start at etc.

We use Dr. Michael T. Flanagans Java implementation of the Nelder-Mead maximisation algorithm [Flanagan] which supports maximising any function fulfilling a defined function interface. Our server implements said interface and Flanagans Java package is thus able to optimise our "function." When the Nelder-Mead algorithm calls our function interface we create the tasks and put these in a taskpool from which the tasks are given to the clients. When all tasks have been "calculated" we sum the points and return this as our functions response. This results in a new call with adjusted parameters. The server is split into two threads: one for Flanagans libraries and our function interface and another for the communication-part.

## 6.3 The clients

The client uses Java RMI to contact the server. A remote method is called on the server to give the client a task. The parameters etc. are extracted from the task object and the specified situation is set up between two brains (also chosen via the task object). The client we used supported Alf (either with original

[3] http://www.hpc.dtu.dk

[4] http://java.sun.com/developer/products/jini/

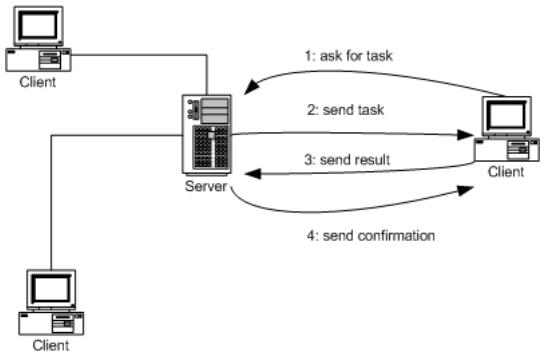


Figure 2: Communication between the server and a client in the grid

parameters or with the specified parameters) and the OliThink brain by Dr. Oliver Brausch[5].

When the situation has been set up the two chosen brains play the rest of the game against each other. The result is either zero – if Alf with the new parameters lost, $\frac{1}{2}$ – if it was a draw or one if it won. The result is wrapped in a result object and a method on the server is called to return the result.

After a task is completed the client repeats the process of asking for another task (unless the user has used the option of requesting the client to stop after completing the current task). If no tasks are available an exponential backoff is implemented to prevent the clients from continuously requesting tasks and thereby disturbing the network.

We have taken care of the situation in which an exception occurs in one of the brains participating in the game (in this situation we judged the game as a draw so the influence was minimal). After this situation all brains are reinitialised to avoid any effects of the exception on the following games.

## 6.4 Performance

During the period we conducted the study some clients were added to our grid – and some were removed. Many PCs around the DTU-campus are not fully utilized. We borrowed CPU-time from these PCs by running our grid client on them when they were not used. Some PCs ran our client all the time but at a low priority[6] so that any spare CPU-time was consumed by our client.

Using this method we managed to get a vast number of tasks calculated. We estimate the total active uptime of the grid to about a full week. During that time the grid processed over 150,000 tasks. As a task takes roughly a minut on a 2GHz Pentium4 one can estimate the time such a computer would use to process the tasks:

$$t_{p4} = \frac{150000}{60 \cdot 24 \cdot 7} \approx 15\text{weeks} \approx 3\text{months} \tag{16}$$

[5]http://home.arcor.de/dreamlike/

[6]In Unix-systems by setting nice-level to 19, in windows by setting process priority to "lowest"

This speedup certainly makes it worth spending a few days of work creating the grid.

# 7 Results

Figure 3 shows the results of each iteration by the simplex algorithm on the Nunn/Noomen test suite. We can observe a clear convergence but also a great amount of noise. We can conclude it is noise rather than actual differences in playing strength from the fact, that there is still great variation in the results in the late iterations, where each parameter is almost constant. Figure 5, 6, 7, and 8 (see appendix) shows examples of some of the parameter values for each iteration. The graphs of all the other parameters are similar and also almost constant in late iterations. The simplex has simply shrunk around the maximum result, which can be affirmed by the fact, that the parameters in the last iterations are almost identical to the parameters at the maximum peak. This makes a restart of the algorithm important, in order to "forget" that maximum point, because it might be a "lucky punch" due to the stochastic nature of the experiments. Another possibility to get around this problem, is to change the algorithm so that the measurement at the maximum point is replaced by a new one with slightly changed parameters, or simply to change the games used for that particular experiment.

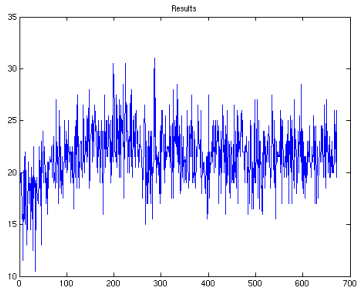


Figure 3: The results of each iteration/vertex in a 50 games run

Table 2 and 3 (in appendix) show the changes in parameters for the first experiment and the restart with a smaller initial simplex. All parameters except for one have a smaller change after the restart compared to the change from the originals, indicating that the restart was closer to the optimum area, but due to the amount of noise, it is hard to say if there are more local optima, or the deviations are caused by uncertainty or a mixture.

In figure 4 we see the results of each iteration on the 385-game matches from our 1000 positions suite. There are weak signs of convergence, but it is not as obvious as for the Nunn/Noomen suite. The sudden drops followed by increases can be accredited the restarts of the simplex algorithm when a possible optimum is reached. All the parameters were nearly constant after iteration number 100, which indicates that an optimum area has been found, but the noise is also significant as in the first experiments. Unfortunately we did not have time to

restart the algorithm in this experiment, though it of course is important in order to rule out the possibility of a "lucky punch" as mentioned earlier.

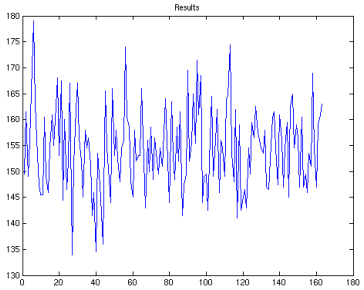


Figure 4: The results of each iteration/vertex in a 385 games run

Table 4 (in appendix) shows that the optimised parameters are actually very close to the original parameters. This indicates that many of the original parameters might actually be close to their optimum values.

Table 1 shows the results of the 1000-game matches between the old Alf, the optimised version of Alf in 50-game matches after the restart, the optimised version of Alf after the 385-game matches, and OliThink. It actually looks like if the optimised(50) version is not as strong as the original, but since the win ratios are within the error margin of 6% we cannot conclude that there is any significant difference. The win ratios of the optimised(385) version is not significantly different from the orignal Alf's, but higher as opposed to the ones of optimised(50) nonetheless.

| Match | Win ratio |
|---|---|
| Original Alf - Original Alf | 49.65 % |
| Orginial Alf - OliThink | 30.10 % |
| Optimised Alf(50) - Original Alf | 47.05 % |
| Optimised Alf(50) - OliThink | 28.35 % |
| Optimised Alf(385) - Original Alf | 50.35 % |
| Optimised Alf(385) - OliThink | 29.70 % |

Table 1: Results in 1000 game matches with Alf before and after optimisation

# 8 Future studies

In this section we will discuss some of the things we found could be studied further.

## 8.1 The quality of Nunn's and Noomen's testsuites

As described earlier, it seemed to us that our tests using the test suites of Nunn and Noomen were able to fit to the used positions. This is not a bad thing

from the simplex point of view, as it shows us that we are able to change the parameters such that the brain plays better in the tested positions. What could be a problem, is that after fitting to these test suites – that are recognised as covering the important parts of chess and giving each side an equal chance to win – we actually found that the brain had not improved, and maybe even changed for the worse in other games.

This gave rise to a discussion of the test suites. It could seem as if they do not cover all parts of chess - as one would then expect the brain optimised for the test suites to play a better game of chess, than one not optimised for them. If the positions do not cover a significantly large part of all typical positions in a game of chess, or if the chances of winning are actually not equal, then many of the current tests of strength of computer chess brains does not really cover the entire game of chess. We have actually shown that we could make Alf score higher in these specific test suites – which would increase it's "strength" when tested – whilst no improvement was found in other positions. Another reason could be that many of the games played diverge from each other at a relatively late stage from the start position. This would make it possible to enforce/dismiss the same crucial moves at a repeating position, which might not be with the globally optimal settings.

Of these reasons we think it could be interesting to study if the test suites actually cover that large a part of the common chess positions as believed in the chess community. It is at least our belief that larger suites – like our own 1000-game suite – would be useful for testing chess engines.

## 8.2 Mirroring vs one side play

As mentioned in section 5.3.2 we tested for improvement by playing each test position from both side of the board with both engines we were testing. This was done to eliminate any advantage effects from the test positions. We found that many supposedly equal positions was won by the same side at a higher rate than could be expected by randomness. This indicates that the positions were in fact *not* equal and those mirrored games therefore led the win ratio towards 50%. This made a higher number of experiments nescesarry to detect significant changes in the win ratio. Another aspect of this is, that the number of different types of positions is decreased when mirroring. Whether or not mirroring is a good strategy is thus a difficult question, that might be interesting to investigate further.

## 8.3 Reducing randomness by increasing search depth

The playing strength of a chess engine is highly dependant on the search depth. At low search depths at least, many experiments have shown a steep nearly linear increase in playing strength as the search depth increases[7]. This means, that the outcome of games played between two engines using low search depths are likely to be determined by tactical errors rather than the positional weights we are trying to optimise. Of course the risk of making tactical errors could be correlated with the type of positions "preferred" by the engine, but even so, a decrease in the number of tactical errors would almost certainly increase the

[7] http://supertech.lcs.mit.edu/ heinz/dt/node49.html

significance of the evaluation function weights. Therefore it might me worth increasing the search depth at the expense of computation time, in order to reduce the noise caused by tactical errors.

## 8.4 Retest of maximum results

As mentioned in section 7, the maximum result is never "forgotten" by the algorithm, and thus it could be a "lucky punch" and indeed not an optimal set of parameters. Future tests need to rule out that possibility by restarting the algorithm an appropriate number of times, or, retesting with slightly changed parameters around the the maximum, or, retesting with same parameters but on a different set of positions.

# 9 Conclusion

In this report we have analysed and tested a - to our knowledge - new method to optimise a chess evaluation function, namely the Nelder-Mead Simplex algorithm. We have constructed software to perform the calculations by grid computing, and thus enabling us to do calculations in one week that would have taken three months on a single computer.

In the first experiments, we used the widely recognised Nunn and Noomen suites of test positions, that are designed to test computer engines. The results showed a clear improvement, but a 1,000 game test revealed that the parameters were not significantly better than than the original ones, and maybe even worse. Our conclusion was, that a fit to the Nunn and Noomen suites is not the same as a fit to chess positions in general.

Later experiments, based on a random selection of 385 test positions from a population of 1,000, could not produce any significant improvements in playing strength. This can be partly due to the fact, that many parameters actually converged to near the original ones indicating that the original parameters are relatively close to the optimal ones. Another reason might be the high amount of noise observed in the experiments. We propose that the new experiments can be conducted with a higher search depth to eliminate some of the noise caused by tactical errors. We also propose additional restarts of the algorithm as to diminish the probability of a high score in a bad area by a "lucky punch", or, to change the algorithm so the maximum result is re-tested for every restart with slightly changed parameters or on another set of start positions.

Our most important conclusion however, is, that we find it very likely that using some of our proposed changes to our methods will make it possible to use the Nelder-Mead Simplex algorithm for tuning chess evaluation functions. As seen in our study we could not get a provable better brain by using the Simplex algorithm, but we have seen indications of the possibilities of improving evaluation functions: Even though the initial simplex' values were far away from the original – and near optimal – parameters, the algorithm managed to return to these values, with only a little difference on each. The algorithm managed to get the brain to win more in our 50 games run, and though probably it did so by forcing certain decisions for specific positions, it still got the brain to win more games in the test population – and even by only changing the parameters a little bit.

It is our hope, that the experiences we gathered in this study, will form a basis for further studies of the use of the Nelder-Mead Simplex algorithm for the tuning of a chess evaluation function.

# References


[Byatt] Byatt, David; Coope, Ian and Price, Chris: *40 Years of the Nelder-Mead Algorithm*. University of Canterbury, 2003.

[Durand] Durand, Nicolas and Alliot, Jean-Marc: *A Combined Nelder-Mead Simplex and Genetic Algorithm*. Laboratoire d'Optimisation Globale, 1999.

[Flanagan] Flanagan, Michael Thomas: *Michael Thomas Flanagan's Java Library*, http://www.ee.ucl.ac.uk/m̃flanaga/java/

[Gomboc] Gomboc, Marsland, and Buro: *Evaluation Function Tuning via Ordinal Correlation*. Presented at the 10th International Conference on Advances in Computer Games (ACG-10), Graz, Styria, Austria, Nov. 24-27, 2003. Published in Advances in Computer Games: Many Games, Many Challenges: Proceedings of the ICGA/IFIP SG16 10th International Conference on Advances in Computer Games (ACG-10), pp. 1-18. ISBN 1-4020-7709-2.

[Hsu] Hsu, F.H.; Anantharaman, T.; Campbell, M. and Nowatzyk, A.: *A Grandmaster Chess Machine*, Scientific American, Vol. 263, No. 4, October, 1990, pp. 44 - 50.

[Johnson] Johnson, Richard A.: *Probability and Statistics for Engineers*, sixth edition, Prentice Hall International, 2000.

[Kendall] Kendall, G. and Whitwell, G.: *An Evolutionary Approach for the Tuning of a Chess Evaluation Function using Population Dynamics*. In proceedings of Congress on Evolutionary Computation 2001 (CEC'01), COEX Center, Seoul, Korea, May 27-29, 2001, pp 995-1002 (an IEEE conference), ISBN : 0-7803-6657-3.

[Pitman] Pitman, Jim: *Probability*, Springer-Verlag, 1993.

# A Tables and figures

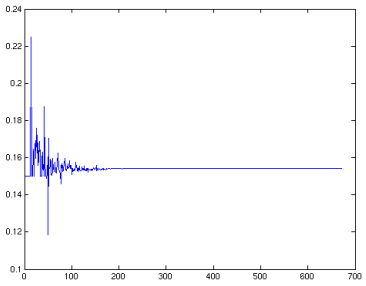


Figure 5: The changes in the parameter "bishopsc" in a 50 games run

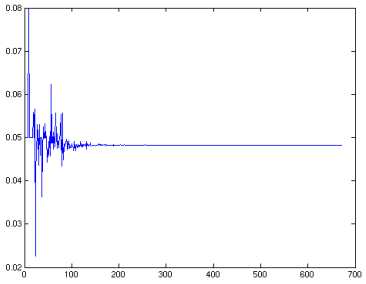


Figure 6: The changes in the parameter "pawnpush" in a 50 games run

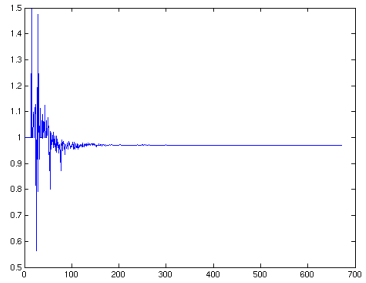


Figure 7: The changes in the parameter "kingsc" in a 50 games run

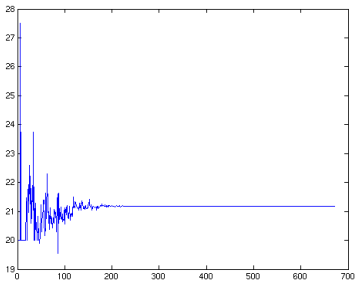


Figure 8: The changes in the parameter "threatdiff" in a 50 games run

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Original | 20.0 | 32.0 | 0.3 | 0.05 | 1.0 | 0.1 | 0.4 | 0.15 | 0.15 | 1.0 | 0.3 | 0.3 |
| 50 games run | 21.18 | 31.60 | 0.3076 | 0.04832 | 1.024 | 0.09600 | 0.3740 | 0.1544 | 0.1541 | 0.9719 | 0.34449 | 0.3854 |
| Change | 5.9% | -1.3% | 2.5% | -3.4% | 2.4% | -4.0% | -6.5% | 2.9% | 2.8% | -2.8% | 14.8% | 28.5% |

Table 2: Parameters – original and after a simplex using 50 games per vertex

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 50 games run | 21.18 | 31.60 | 0.3076 | 0.04832 | 1.024 | 0.09600 | 0.3740 | 0.1544 | 0.1541 | 0.9719 | 0.34449 | 0.3854 |
| Restarted | 21.66 | 31.72 | 0.3124 | 0.04717 | 1.015 | 0.09866 | 0.3832 | 0.1538 | 0.1570 | 0.9925 | 0.4809 | 0.3868 |
| Change | 2.2% | 0.38% | 1.6% | -2.4% | -0.95% | 2.8% | 2.5% | -0.35% | 1.9% | 2.1% | 39.6% | 0.36% |

Table 3: Parameters – simplex using 50 games per vertex and restarted

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Original | 20.0 | 32.0 | 0.3 | 0.05 | 1.0 | 0.1 | 0.4 | 0.15 | 0.15 | 1.0 | 0.3 | 0.3 |
| 385 games run | 21.17 | 32.00 | 0.3023 | 0.05019 | 1.123 | 0.1007 | 0.4052 | 0.1503 | 0.1510 | 0.9895 | 0.3021 | 0.2922 |
| Change | 5.9% | 0% | 0.8% | 0.4% | 12.3% | 0.7% | 1.3% | 0.2% | 0.6% | -1.1% | 0.7% | -2.6% |

Table 4: Parameters – original and after a simplex using 385 games per vertex