# Optimizing Selective Search in Chess

**Omid David-Tabibi** MAIL@OMIDDAVID.COM
Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel

**Moshe Koppel** KOPPEL@CS.BIU.AC.IL
Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel

**Nathan S. Netanyahu** NATHAN@CS.BIU.AC.IL
Department of Computer Science, Bar-Ilan University, Ramat-Gan 52900, Israel, and Center for Automation Research, University of Maryland, College Park, MD 20742

## Abstract

In this paper we introduce a novel method for automatically tuning the search parameters of a chess program using genetic algorithms. Our results show that a large set of parameter values can be learned automatically, such that the resulting performance is comparable with that of manually tuned parameters of top tournament-playing chess programs.

## 1. Introduction

Until the mid-1970s most chess programs attempted to perform search by mimicking the way humans think, i.e., by generating "plausible" moves. By using extensive chess knowledge, these programs selected at each node a few moves which they considered plausible, thereby pruning large parts of the search tree. However, as soon as brute-force search programs like TECH (Gillogly, 1972) and CHESS 4.X (Slate and Atkin, 1983) managed to reach depths of 5 plies and more, plausible move generating programs frequently lost to these brute-force searchers due to their significant tactical weaknesses. Brute-force searchers rapidly dominated the computer chess field.

The introduction of null-move pruning (Beal, 1989; Donninger, 1993) in the early 1990s marked the end of an era, as far as the domination of brute-force programs in computer chess is concerned. Unlike other forward-pruning methods which had great tactical weaknesses, null-move pruning enabled programs to search more deeply with minor tactical risks. Forward-pruning programs frequently outsearched brute-force searchers, and started their own reign which has continued ever since; they have won all World Computer Chess Championships since 1992. DEEP BLUE (Hammilton and Garber, 1997; Hsu, 1999) was probably the last brute-force searcher.

Nowadays, top tournament-playing programs use a range of methods for adding selectivity to their search. The most popular methods include null-move pruning, futility pruning (Heinz, 1998), multi-cut pruning (Björnsson and Marsland, 1998; Björnsson and Marsland, 2001), and selective extensions (Anantharaman, 1991; Beal and Smith, 1995). For each of these methods, a wide range of parameter values can be set. For example, different reduction values can be used for null-move pruning, various thresholds can be used for futility pruning, etc.

For each chess program, the parameter values for various selective search methods are manually tuned through years of experiments and manual optimizations. In this paper we introduce a novel method for automatically tuning the search parameters of a chess program using genetic algorithms (GA).

In the following section, we review briefly the main methods that have been used for selective search. For each of these methods, we enumerate the parameters that need to be optimized. Section 3 provides a review of past attempts at automatic learning of various parameters in chess. In Section 4 we present our automatic method of optimizing the parameters in question, which is based on the use of genetic algorithms, and in Section 5 we provide experimental results. Section 6 contains concluding remarks.

## 2. Selective Search in Chess

In this section we review several popular methods for selective search. All these methods work within the al-

Appearing in *Proceedings of the ICML Workshop on Machine Learning and Games*. Copyright 2010 by the author(s)/owner(s).

phabeta/PVS framework and introduce selectivity in various forms. A simple alphabeta search requires the search tree to be developed to a fixed depth in each iteration. Forward pruning methods, such as null-move pruning, futility pruning, and multi-cut pruning, enable the program to prune some parts of the tree at an earlier stage, and devote the time gained to other, more promising parts of the search tree.

Selective extensions, on the other hand, extend certain parts of the tree to be searched deeper, due to tactical considerations associated with a position in question. The following subsections briefly cover each of these pruning and extension methods, and specify which parameters should be tuned for each method.

## 2.1. Null-Move Pruning

Null-move pruning (Beal, 1989; David-Tabibi and Netanyahu, 2008b; Donninger, 1993) is based on the assumption that "doing nothing" in every chess position (i.e., doing a null-move) is not the best choice even if it were a legal option. In other words, the best move in any position has to be better than the null-move. This assumption enables the program to establish a lower bound $\alpha$ on the position by conducting a *null-move search*. The idea is to make a null-move, i.e., merely swap the side whose turn it is to move. (Note that this cannot be done in positions where the side to move is in check, since the resulting position would be illegal. Also, two null-moves in a row are forbidden, since they result in nothing.) A regular search is then conducted with reduced depth $R$. The returned value of this search can be treated as a lower bound on the position's strength, since the value of the best (legal) move has to be better than that obtained from the null-move search. In a negamax framework, if the returned value is greater than or equal to the current upper bound (i.e., $value \geq \beta$), it results in a cutoff (fail-high). Otherwise, if the value is greater than the current lower bound (i.e., $\alpha < value \leq \beta$), we define a narrower search window, as the returned value becomes the new lower bound. If the value is smaller than the current lower bound, it does not contribute to the search in any way. The main benefit of the null-move concept is the pruning obtained due to the cutoffs, which take place whenever the returned value of the null-move search is greater than the current upper bound. Thus, the best way to apply null-move pruning is by conducting a minimal-window null-move search around the current upper bound $\beta$, since such a search will require a reduced search effort to determine if a cutoff takes place.

Donninger (1993) was the first to suggest an adaptive rather than a fixed value for $R$. Experiments conducted by Heinz in his article on adaptive null-move pruning (1999) showed that, indeed, an adaptive rather than a fixed value could be selected for the reduction factor. By using $R = 3$ in upper parts of the search tree and $R = 2$ in its lower parts (close to the leaves) pruning can be achieved at a smaller cost (as null-move searches will be shallower in comparison to using a fixed reduction value of $R = 2$) while maintaining the overall tactical strength. An in-depth review of null-move pruning and our *extended null-move reductions* improvement can be found in (David-Tabibi and Netanyahu, 2008b).

Over the years many variations of null-move pruning have been suggested, but the set of key parameters to be determined has remained the same. These parameters are: (1) the reduction value $R$, (2) the Boolean adaptivity variable, and (3) the adaptivity depth for which the decremented value of $R$ is applied.

## 2.2. Futility Pruning

Futility pruning and extended futility pruning (Heinz, 1998) suggest pruning nodes near a leaf where the sum of the current static evaluation value and some threshold (e.g., the value of a knight) is smaller than $\alpha$. In these positions, assuming that the value gained in the remaining moves until reaching the leaf is not greater than the threshold, it is safe to assume that the position is "weak enough", i.e., that it is worth pruning (as its score will not be greater than $\alpha$). Naturally, the larger the threshold, the safer it is to apply futility pruning, although fewer nodes will be pruned.

The main parameters to be set for futility pruning are: (1) the futility depth and (2) the futility thresholds for various depths (usually up to a depth of 3 plies).

## 2.3. Multi-Cut Pruning

Björnsson and Marsland's multi-cut pruning (1998; 2001) suggests searching the moves at a given position to a shallower depth first, such that if several of them result in a cutoff, the current node is pruned without conducting a full depth search. The idea is that if there are several moves that produce a cutoff at a shallower depth, there is a high likelihood that at least one of them will produce a cutoff if searched to a full depth. In order to apply multi-cut pruning only to potentially promising nodes, it is applied only to cut-nodes (i.e., nodes at which a cutoff has occurred previously, according to a hash table indication).

The primary parameters that should be set in multi-cut pruning are: (1) the depth reduction value, (2) the depth for which multi-cut is applied, (3) the number of moves to search, and (4) the number of cutoffs to require.

### 2.4. Selective Extensions

Selective extensions (Anantharaman, 1991; Beal and Smith, 1995) are used for extending potentially critical moves to be searched deeper. The following is a list of major extensions used in most programs:

**Check extension:** Extend the move if it checks the opponent's king.
**One-reply extension:** Extend the move if it is the only legal move.
**Recapture extension:** Extend the move if it is a recapture of a piece captured by the opponent (such moves are usually forced).
**Passed pawn extension:** Extend the move if it involves moving a passed pawn (usually to 7th rank).
**Mate threat extension:** Extend the move if the null-move search returns a mate score (the idea is that if doing a null-move results in being checkmated, a potential danger lies at the horizon, so we extend the search to find the threat).

For each of the above extensions, fractional extensions have been widely employed. These are implemented usually by defining one ply to be a number greater than one (e.g., 1 ply = 4 units), such that several fractional extensions along a line (i.e., a series of moves from the root to a leaf) cause a full ply extension. For example, if a certain extension is defined as half a ply, two such extensions must occur along a line in order to result in an actual full ply extension. For each extension type, a value is defined (e.g., assuming that 1 ply = 4 units, an extension has a value between 0 to 4).

From this brief overview of selective search, there are a number of parameters for each method which have to be set and tuned. Currently, top tournament-playing programs use manually tuned values which take years of trial and improvement to fine tune. In the next section we review the limited success of past attempts at automatic learning of the values of these search parameters, and in Section 4 we present our GA-based method for doing so.

## 3. Automatic Tuning of Search Parameters

The selective search methods covered in the previous section are employed by most of the current top tournament-playing chess programs. They use manually tuned parameter values that were arrived at after years of experiments and manual optimizations.

Past attempts at automatic optimization of search parameters have resulted in limited success. Moriarty and Miikkulainen (1994) used neural networks for tuning the search parameters of an Othello program, but as they mention in their paper, their method is not easily applicable to more complex games such as chess. Temporal difference learning has been successfully applied in backgammon and checkers (Schaeffer, Hlynka, and Jussila, 2001; Tesauro, 1992). Although the latter has also been applied to chess (Baxter, Tridgell, and Weaver, 2000), the results show that after three days of learning, the playing strength of the program was only 2150 Elo, which is a very low rating for a chess program. Block *et al.* (2008) reported that using reinforcement learning, their chess program achieves a playing strength of only 2016 Elo. Veness *et al.*'s (2009) work on bootstrapping from game tree search improved upon previous work, but their resulting chess program reached a performance of between 2154 to 2338 Elo, which is still considered a very low rating for a chess program. Kocsis and Szepesvári's (2006) work on universal parameter optimization in games based on SPSA does not provide any implementation for chess.

Björnsson and Marsland (2002) presented a method for automatically tuning search extensions in chess. Given a set of test positions (for which the correct move is predetermined) and a set of parameters to be optimized (in their case, four extension parameters), they tune the values of the parameters using gradient-descent optimization. Their program processes all the positions and records, for each position, the number of nodes visited before the solution is found. The goal is to minimize the total node count over all the positions. In each iteration of the optimization process, their method modifies each of the extension parameters by a small value, and records the total node count over all the positions. Thus, given $N$ parameters to optimize (e.g., $N = 4$), their method processes in each iteration all the positions $N$ times. The parameter values are updated after each iteration, so as to minimize the total node count. Björnsson and Marsland applied their method for tuning the parameter values of the four search extensions: check, passed pawn, recapture, and one-reply extensions. Their results showed that their method optimizes fractional ply values for the above parameters, as the total node count for solving the test set is decreased.

Despite the success of this gradient-descent method for tuning the parameter values of the above four search extensions, it is difficult to use it efficiently for optimizing a considerably larger set of parameters, which consists of all the selective search parameters mentioned in the previous section. This difficulty is due to the fact that unlike the optimization of search extensions for which the parameter values are mostly independent, other search methods (e.g., multi-cut pruning) are prone to a high interdependency between the parameter values, resulting in multiple local maxima in the search space, in which case it is more difficult to apply gradient-descent optimization.

In the next section we present our method for automatically tuning all the search parameters mentioned in the previous section by using genetic algorithms.

## 4. Genetic Algorithms for Tuning of Search Parameters

In David-Tabibi *et al.* (2008a; 2009; 2010) we showed that genetic algorithms (GA) can be used to efficiently evolve the parameter values of a chess program's evaluation function. Here we present a GA-based method for optimizing a program's search parameters. We first describe how the search parameters are represented as a chromosome, and then discuss the details of the fitness function.

The parameters of the selective search methods which were covered in Section 2 can be represented as a binary chromosome, where the number of allocated bits for each parameter is based on a reasonable value range of the parameter. Table 1 presents the chromosome and the range of values for each parameter (see Section 2 for a description of each parameter). Note that for search extensions fractional ply is applied, where 1 ply = 4 units (e.g., an extension value of 2 is equivalent to half a ply, etc.).

| Parameter | Value range | Bits |
|---|---|---|
| Null-move use | 0–1 | 1 |
| Null-move reduction | 0–7 | 3 |
| Null-move use adaptivity | 0–1 | 1 |
| Null-move adaptivity depth | 0–7 | 3 |
| Futility depth | 0–3 | 2 |
| Futility threshold depth-1 | 0–1023 | 10 |
| Futility threshold depth-2 | 0–1023 | 10 |
| Futility threshold depth-3 | 0–1023 | 10 |
| Multi-cut use | 0–1 | 1 |
| Multi-cut reduction | 0–7 | 3 |
| Multi-cut depth | 0–7 | 3 |
| Multi-cut move num | 0–31 | 5 |
| Multi-cut cut num | 0–7 | 3 |
| Check extension | 0–4 | 3 |
| One-reply extension | 0–4 | 3 |
| Recapture extension | 0–4 | 3 |
| Passed pawn extension | 0–4 | 3 |
| Mate threat extension | 0–4 | 3 |
| Total chromosome length | | 70 |

*Table 1.* Chromosome representation of 18 search parameters (length: 70 bits).

For the GA's fitness function we use a similar optimization goal to the one used by Björnsson and Marsland (2002), namely the total node count. A set of 879 tactical test positions from the Encyclopedia of Chess Middlegames (ECM) is used for training purposes. Each of these test positions has a predetermined "correct move", which the program has to find. In each generation, each organism searches all the 879 test positions and receives a fitness score based on its performance. As noted, instead of using the number of solved positions as a fitness score, we take the number of nodes the organism visits before finding the correct move. We record this parameter for each position and compute the total node count for each organism over the 879 positions. Since the search cannot continue endlessly for each position, a maximum limit of 500,000 nodes per position is imposed. If the organism does not find the correct move when reaching this maximum node count for the position, the search is stopped and the node count for the position is set to 500,000. Naturally, the higher the maximum limit, the larger the number of solved positions. However, more time will be spent on each position and subsequently, the whole evolution process will take more time.

The fitness of the organism will be inversely proportionate to its total node count for all the positions. Using this fitness value rather than the number of solved positions has the benefit of deriving more fitness information per position. Rather than obtaining a 1-bit information for solving the position, a numeric value is obtained which also measures how quickly the position is solved. Thus, the organism is not only "encouraged" to solve more positions, it is rewarded for finding quicker solutions for the already solved test positions.

Other than the special fitness function described above, we use a standard GA implementation with Gray coded chromosomes, fitness-proportional selection, uniform crossover, and elitism (the best organism is copied to the next generation). All the organisms are initialized with random values. The following parameters are used for the GA: population size = 10, crossover rate = 0.75, mutation rate = 0.05, number of generations = 50.

The next section contains the experimental results using the GA-based method for optimization of the search parameters.

## 5. Experimental Results

We used the Falcon chess engine in our experiments. Falcon is a grandmaster-level chess program which has successfully participated in three World Computer Chess Championships. Falcon uses NegaScout/PVS search, with null-move pruning, internal iterative deepening, dynamic move ordering (history + killer heuristic), multi-cut pruning, selective extensions (consisting of check, one-reply, mate-threat, recapture, and passed pawn extensions), transposition table, and futility pruning near leaf nodes.

Each organism is a copy of FALCON (i.e., has the same evaluation function, etc.), except that its search parameters, encoded as a 70-bit chromosome (see Table 1), are randomly initialized rather than manually tuned.

The results of the evolution show that the total node count for the population average drops from 239 million nodes to 206 million nodes, and the node count for the best organism drops from 226 million nodes to 199 million nodes. The number of solved positions increases from 488 in the first generation to 547 in the 50th generation. For comparison, the total node count for the 879 positions due to Björnsson and Marsland's optimization was 229 million nodes, and the number of solved positions was 508 (Björnsson and Marsland, 2002).

To measure the performance of the best evolved organism (we call this organism EVOL*), we compared it against the chess program CRAFTY (Hyatt, Gower, and Nelson, 1990). CRAFTY has successfully participated in numerous World Computer Chess Championships (WCCC), and is a direct descendent of CRAY BLITZ, the WCCC winner of 1983 and 1986. It is frequently used in the literature as a standard reference.

First, we let EVOL*, CRAFTY, and the original manually tuned FALCON process the ECM test suite with 5 seconds per position. Table 2 provides the results. As can be seen, EVOL* solves significantly more problems than CRAFTY and a few more than FALCON.

| EVOL* | FALCON | CRAFTY |
|---|---|---|
| 652 | 645 | 593 |

*Table 2.* Number of ECM positions solved by each program (time: 5 seconds per position).

The superior performance of EVOL* on the ECM test set is not surprising, as it was evolved on this training set. Therefore, in order to obtain an unbiased performance comparison, we conducted a series of 300 matches between EVOL* and CRAFTY, and between EVOL* and FALCON. In order to measure the rating gain due to evolution, we also conducted 1,000 matches between EVOL* and 10 randomly initialized organisms (RANDORG). Table 3 provides the results. The table also contains the results of 300 matches between FALCON and CRAFTY as a baseline.

The results of the matches show that the evolved parameters of EVOL* perform on par with those of FALCON, which have been manually tuned and refined for the past eight years. Note that the performance of FALCON is by no means a theoretical upper bound for the performance of EVOL*, and the fact that the automatically evolved program matches the manually tuned one over many years of world championship level performance, is by itself a clear demonstration of the capabilities achieved due to the automatic evolution of search parameters.

| Match | Result | W% | RD |
|---|---|---|---|
| FALCON - CRAFTY | 173.5 - 126.5 | 57.8% | +55 |
| EVOL* - CRAFTY | 178.5 - 121.5 | 59.5% | +67 |
| EVOL* - FALCON | 152.5 - 147.5 | 51.1% | +6 |
| EVOL* - RANDORG | 714.0 - 286.0 | 71.4% | +159 |

*Table 3.* FALCON vs. CRAFTY, and EVOL* vs. CRAFTY, FALCON, and randomly initialized organisms (W% is the winning percentage, and RD is the Elo rating difference).

The results further show that EVOL* outperforms CRAFTY, not only in terms of solving more tactical test positions, but more importantly in its overall strength. These results establish that even though the search parameters are evolved from scratch (with randomly initialized chromosomes), the resulting organism outperforms a grandmaster-level chess program.

## 6. Conclusions

In this paper we presented a novel method for automatically tuning the search parameters of a chess program. While past attempts yielded limited success in tuning a small number of search parameters, the method presented here succeeded in evolving a large number of parameters for several search methods, including complicated interdependent parameters of forward pruning search methods.

The search parameters of the FALCON chess engine, which we used for our experiments, have been manually tuned over the past eight years. The fact that GA manages to evolve the search parameters automatically, such that the resulting performance is on par with the highly refined parameters of FALCON is in itself remarkable.

Note that the evolved parameter sets are not necessarily the best parameter sets for every chess program. Undoubtedly, running the evolutionary process mentioned in this paper on each chess program will yield a different set of results which are optimized for the specific chess program. This is due to the fact that the performance of the search component of the program depends on other components as well, most importantly the evaluation function. For example, in a previous paper on *extended null-move pruning* (David-Tabibi and Netanyahu, 2008b), we discovered that while the common reduction value for null-move pruning is $R = 2$ or $R = 3$, a more aggressive reduction value of adaptive $R = 3 \sim 4$ performs better for FALCON. It is interesting to note that our GA-based method managed to independently find that these aggressive reduction values work better for FALCON.

## References

T.S. Anantharaman (1991). Extension heuristics. *ICCA Journal*, 14(2):47–65.

J. Baxter, A. Tridgell, and L. Weaver (2000). Learning to play chess using temporal-differences. *Machine Learning*, 40(3):243–263.

D.F. Beal (1989). Experiments with the null move. *Advances in Computer Chess 5*, ed. D.F. Beal, pages 65–79. Elsevier Science, Amsterdam.

D.F. Beal and M.C. Smith (1995). Quantification of search extension benefits. *ICCA Journal*, 18(4):205–218.

Y. Björnsson and T.A. Marsland (1998). Multi-cut pruning in alpha-beta search. In *Proceedings of the First International Conference on Computers and Games*, pages 15–24, Tsukuba, Japan, November 1998.

Y. Björnsson and T.A. Marsland (2001). Multi-cut alpha-beta-pruning in game-tree search. *Theoretical Computer Science*, 252(1-2):177–196.

Y. Björnsson and T.A. Marsland (2002). Learning control of search extensions. In *Proceedings of the 6th Joint Conference on Information Sciences*, pages 446–449, Durham, NC, March 2002.

M. Block, M. Bader, E. Tapia, M. Ramirez, K. Gunnarsson, E. Cuevas, D. Zaldivar, R. Rojas (2008). Using reinforcement learning in chess engines. *Research in Computing Science*, No. 35, pages 31–40.

O. David-Tabibi, M. Koppel, and N.S. Netanyahu (2008). Genetic algorithms for mentor-assisted evaluation function optimization. In *Proceedings of the 2008 Genetic and Evolutionary Computation Conference*, pages 1469–1476. Atlanta, GA, July 2008.

O. David-Tabibi and N.S. Netanyahu (2008). Extended null-move reductions. In *Proceedings of the 6th International Conference on Computers and Games*, eds. H.J. van den Herik, X. Xu, Z. Ma, and M.H.M. Winands, pages 205–216. Springer (LNCS 5131), Beijing, China, October 2008.

O. David-Tabibi, H.J. van den Herik, M. Koppel, and N.S. Netanyahu (2009). Simulating human grandmasters: Evolution and coevolution of evaluation functions. In *Proceedings of the 2009 Genetic and Evolutionary Computation Conference*, pages 1483–1489. Montreal, Canada, July 2009.

O. David-Tabibi, M. Koppel, and N.S. Netanyahu (2010). Expert-Driven Genetic Algorithms for Simulating Evaluation Functions. *Genetic Programming and Evolvable Machines*, accepted for publication (online version available at www.springerlink.com/content/3346t8432n718821).

C. Donninger (1993). Null move and deep search: Selective search heuristics for obtuse chess programs. *ICCA Journal*, 16(3):137–143.

J.J. Gillogly (1972). The technology chess program. *Artificial Intelligence*, 3(1-3):145–163.

S. Hammilton and L. Garber (1997). DEEP BLUE's hardware-software synergy. *IEEE Computer*, 30(10):29–35.

E.A. Heinz (1998). Extended futility pruning. *ICCA Journal*, 21(2):75–83.

E.A. Heinz (1999). Adaptive null-move pruning. *ICCA Journal*, 22(3):123–132.

F.-h. Hsu (1999). IBM's DEEP BLUE chess grandmaster chips. *IEEE Micro*, 19(2):70–80.

R.M. Hyatt, A.E. Gower, and H.L. Nelson (1990). CRAY BLITZ. *Computers, Chess, and Cognition*, eds. T.A. Marsland and J. Schaeffer, pages 227–237. Springer-Verlag, New York.

L. Kocsis and C. Szepesvári (2009). Universal parameter optimisation in games based on SPSA. *Machine Learning*, 63(3):249–286.

D.E. Moriarty and R. Miikkulainen (1994). Evolving neural networks to focus minimax search. In *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 1371–1377. Seattle, WA, July 1994.

J. Schaeffer, M. Hlynka, and V. Jussila (2001). Temporal difference learning applied to a high-performance game-playing program. In *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 529–534. Seattle, WA, August 2001.

D.J. Slate and L.R. Atkin (1983). CHESS 4.5 - The Northwestern University chess program. *Chess Skill in Man and Machine*, ed. P.W. Frey, pages 82–118. Springer-Verlag, New York.

G. Tesauro (1992). Practical issues in temporal difference learning. *Machine Learning*, 8(3-4):257–277.

J. Veness, D. Silver, W. Uther, and A. Blair (2009). Bootstrapping from game tree search. *Advances in Neural Information Processing Systems 22*, eds. Y. Bengio, D. Schuurmans, J. Lafferty, C.K.I. Williams, and A. Culotta.