INVESTIGATION OF MULTI-CUT PRUNING IN GAME-TREE SEARCH

Research report

Spring 2011

Hrafn Eiríksson

B.Sc. in Computer Science

Leiðbeinandi: Yngvi Björnsson
Prófdómari: Magnús Már Halldórsson

T-619-LOKA
School of Computer Science

Abstract

Computers have long surpassed humans when it comes to playing chess, but this did not happen overnight. Modern chess programs are the product of decades of research which has predominantly dealt with the topic of game tree search. Consequently, there exist many search enhancement techniques for improving the effectiveness and decision quality of such programs.

In this report, we give a brief historical overview of how search paradigms in computer chess have shifted over the years. We also present a discussion on game tree search and popular search enhancement techniques. In particular, we investigate one such technique, *multi-cut pruning*, and introduce enhancements to it. The enhanced technique is implemented in our own chess program and quantitative experiments are carried out which demonstrate a statistically significant improvement over the original technique. Moreover, the enhanced technique is examined in the context of other popular search enhancements like *null-move pruning* and we find that it complements them much better than its predecessor.

Útdráttur

Það er langt síðan að tölvur tóku fram úr manneskjunni hvað varðar getu í skák, en þetta gerðist þó ekki á einni nóttu. Nútímaskákforrit eru afurð áratugalangra rannsókna sem varða fyrst og fremst leit í leitartrjám. Ein af afleiðingum þessara rannsókna er fjöldi leitaraðferða sem hjálpa skákforritum sem og öðrum forritum sem nýta sér leit í leitartrjám að taka betri ákvarðanir á styttri tíma.

Í þessari skýrslu lítum við örstutt á það hvernig helstu einkenni leitar í tölvuskák hafa þróast síðustu áratugi. Við fjöllum einnig almennt um leit í leitartrjám og tökum fyrir nokkrar vinsælar leitaraðferðir og kynnum þær. Sérstaklega tökum við þó fyrir eina leitaraðferð, *multi-cut pruning*, og kynnum til sögunnar endurbætta útgáfu af henni. Endurbætta útgáfan var útfærð í okkar eigin skákforriti og tilraunir framkvæmdar. Niðurstöður tilraunanna sýna með tölfræðilega marktækum hætti að hún ber höfuð og herðar yfir upprunalegu útgáfuna. Jafnframt skoðum við endurbættu útgáfuna í samhengi við aðrar vinsælar leitaraðferðir eins og *null-move pruning* en svo virðist sem hún falli þar betur inn en forveri sinn.

Contents

1 Introduction

The birth of the computer chess programs Tech [10] and Chess 4.x [1] in the mid-1970s marked an end of an era in the computer chess field. Until then, most chess programs had been trying to emulate the way human chess players think by relying on extensive chess knowledge to determine what subset of the positions in the search tree needed further investigation. This resulted in large parts of the search tree being pruned away which again, led to serious tactical shortcomings. When fast brute-forcers like Tech and Chess 4.x emerged, their dominance over the selective programs quickly marked a shift in the computer chess field.

Typical brute-forcers of the following era employed little selectivity in their search, or none at all. It was not until the beginning of the 1990s—with the introduction of *null-move pruning* [3,9,11]—that selective search in chess programs experienced a renaissance. Null-move pruning enabled chess programs to search deeper with minor tactical risk and quickly this kind of *speculative pruning* searchers began their reign which has continued ever since.

Since the introduction of the *null-move pruning* many selective search techniques have been developed and a modern state of the art game-playing program typically employs several such techniques concurrently. With new search enhancements being introduced every few years, and with continuous development in the field, it is only natural that some search techniques get left behind.

With this project, we plan to reinvestigate the effectiveness of one such technique, *multi-cut* [4, 5, 24]. This method has been employed in at least two world-class chess programs [4], Shredder and Hydra, although the last published results of the effectiveness of the method are almost a decade old. This report introduces our enhanced multi-cut algorithm and demonstrates empirically its improved efficiency in comparison to the original multi-cut algorithm.

We begin by reviewing some necessary background material concerning game tree search in Section 2. Section 3 reviews the original multi-cut algorithm and in Section 4 we present the enhancements we made to the algorithm. Section 5 presents the results of our experiments and Section 6 contains concluding remarks and future work.

2 Background

Chess programs use game tree search to reason about their decision making. Game trees are essentially *directed acyclic graphs* or DAGs (despite being called game *trees*) whose nodes are positions in a game and whose edges are legal moves from a given position to another. Unfortunately, the game trees for many games experience an exponential growth as their nodes are expanded. In chess, for example, there is an average of 35–38 legal moves per position, meaning that the average branching factor for a chess game tree is between 35 and 38. This means that finding the best possible move for an arbitrary chess position can be time consuming. But as mentioned in the introduction, much research has been established in the field of game tree search and computers have long surpassed humans when it comes to playing chess. However, in other games such as *Go* and *shogi*, humans are still far superior than the best computer players so there is still much room for additional research in this field.

2.1 Minimax

Finding the best move for a given chess position obviously involves exploring the game tree, but due to the vastness of it the computer is rarely able to search until it reaches a terminal node—that is, a node with a known outcome. Instead, it searches to a certain depth and uses a *static evaluation function* to estimate the desirability attractiveness of the leaf nodes. These values are then propagated up the tree resulting in a *game value*—the outcome when both players play perfectly. At least in theory, the best move for a given position can be found this way, but one has to be aware that the evaluation function only gives an estimation of the "goodness" of a node, and can thus be in error. The first search algorithms that dealt with this kind of two-person zero-sum perfect-information games, recursively expanded all possible continuations from the initial node to the leaf nodes and used the *minimax* rule to propagate the leaf node evaluations back to the initial node. With the minimax rule, *Max*, the player to move tries to maximize its gains by choosing the move that maximizes its value, while its opponent *Min* tries to minimize *Max*'s gains by choosing the moves with the minimal value.

2.2 The $\alpha\beta$ Algorithm

The minimax algorithm exhaustively explores *all* possible moves of the game tree. This is impractical in most cases, since in fact only a subset of the game tree needs to be searched to reach the game value. This subtree is often referred to as the *critical*- or the *minimal tree.* The fact that only a subset of the game tree has to be searched serves as a basis for a much more effective and a fundamental algorithm in game tree search, namely the $\alpha\beta$ algorithm [19]. The $\alpha\beta$ algorithm does this by establishing lower and upper bounds (named alpha and beta, respectively) on the range of possible values that subtrees being searched can have. When values fall outside of these bounds, the game tree can effectively be pruned without it affecting the search result; that is the game value. Ideally, we want the $\alpha\beta$ algorithm to explore the "best" moves first, since the number of nodes that it will expand greatly depends on the *move ordering.* A good move ordering ensures that a tight bound is established early, allowing for more of the tree to be pruned. In worst case, the $\alpha\beta$ algorithm will traverse the full game tree as the minimax algorithm does, but in the best case it will only traverse the minimal tree. According to Knuth and Moore [15] the number of nodes that the $\alpha\beta$ algorithm traverses in the best case is approximately the square root of the number of nodes that minimax needs to traverse.

2.3 Node Types in the Minimal Tree

Knuth and Moore realized that the nodes in the minimal tree could be classified into three categories which they simply called type-one, type-two and type-three. We will however use the more descriptive terminology introduced by Marsland and Popwich [18], who identified them as PV nodes, CUT nodes and ALL nodes, respectively:

- PV nodes are nodes that have all of their moves investigated. At least one of the moves returns a value above the lower bound (alpha), but no moves return a value greater than the upper bound (beta). Their value thus ends up being between the lower- and upper bound of the $\alpha\beta$ algorithm, that is $alpha < value < beta$.
- CUT nodes are nodes where a so-called beta-cutoff occurs (often just referred to as a cutoff). This happens when the value backed up by the search of one of its moves is greater or equal to the upper bound (beta), that is $value \geq beta$. Intuitively, it means that the search found something that was "too good", meaning that the opponent has some

way, already found by the search, of avoiding this position. When this happens no further moves need to be searched and a value of beta is returned. This also means that a minimum of one move needs to be investigated when dealing with a CUT node. These beta-cutoffs are the primary source of pruning that can be done within the $\alpha\beta$ framework and a good move ordering (where good moves are ordered first) is vital for these beta-cutoffs to occur early.

- ALL nodes are nodes containing no moves returning a value exceeding alpha, thus for all moves $value \leq alpha$. This means that every move at an ALL node needs to be investigated and if none of them return a value exceeding alpha, a value of alpha is returned.

Chess programs often try to predict the type of a node before it is actually searched. A node is then referred to as an *expected* PV node, *expected* CUT node or an *expected* ALL node. But predictions are not always correct: If none of the moves at an expected CUT node causes a beta-cutoff, the move subsequently becomes an ALL node. If one of the moves at an expected ALL node causes a beta-cutoff, the move becomes a CUT node. And when all expected CUT nodes on a path from the root to a leaf node have become ALL nodes, a new principal variation has emerged—all the nodes on the same path have in fact become PV nodes.

2.4 Algorithmic Enhancements

The $\alpha\beta$ algorithm can be improved and over the years a number of enhancements have been proposed with this intent. These enhancements can be categorized into the two following groups:

(a) Enhancements that deal with improving the search-efficiency of the $\alpha\beta$ algorithm *without* affecting the result of the search. These enhancements will often make use of a good move ordering.

(b) Enhancements that search the tree more selectively, where some lines of play are terminated prematurely (*speculative pruning*) whereas others are explored more deeply (*search extensions*). These types of enhancements may alter the result of the search.

In this report we will only cover the enhancements directly related to this project. For a more detailed discussion consider looking at [4, 16].

2.4.1 Principal Variation Search (PVS)

In 1982, Tony Marsland and Murray Campbell introduced *principal variation search* [17] or PVS. The idea behind PVS is that when the $\alpha\beta$ algorithm finds a PV move (a move leading to a PV node), the move ordering will be good enough that a better move will not be found. A "better move" in this context can both be a better PV move or a move that leads to a beta-cutoff (causing the node to become a CUT node). PVS is essentially an extension to the $\alpha\beta$ algorithm that assumes that once it has found a PV move, the rest of the moves only need to be proven inferior to that move. To prove this, the moves assumed to be inferior are searched with a *null*- or *zero window* with bounds $(alpha, alpha + 1)$ instead of the normal $(alpha, beta)$ bounds, causing more beta-cutoffs. If however the assumption is wrong and some move happens to return a value greater or equal to $alpha + 1$ the search has to be re-done with the normal bounds. When such double-evaluations occur, time and search effort is wasted. With good move ordering, however, this should be infrequent enough not to outweigh the gains from the increased number of beta-cutoffs.

The PVS algorithm is often split up into two routines: (a) PVS - the main driver which searches the PV nodes and (b) NWS (null window-search) - for the null-window searches of the CUT and ALL nodes. Another formulation and probably a more popular one as of today only uses one PVS routine to avoid unnecessary code repetition. The formulations are otherwise identical, that is they expand the same tree and return the same results.

2.4.2 Transposition Tables

In some board games it is possible that different sequences of moves lead to the same position on the board. When some sequence of moves results in a position that may also be reached by another sequence of moves it is called a *transposition*. An example of this is demonstrated in Figure 1, where two simple sequences of moves are shown to result in the same position.

Chess programs encounter the same positions repeatedly when searching. Without a so-called *transposition table* these positions would need to be searched down to a given depth each time they were encountered. Instead, chess programs employ a large hash table—a transposition table—that stores information about the outcome of previous searches as indexed by the position on the board. When an identical position is encountered later on, the outcome of the search can be retrieved from the transposition table without

1 e2-e4 e7-e5 2 ♘g1-f3
or
1 ♘g1-f3 e7-e5 2 e2-e4

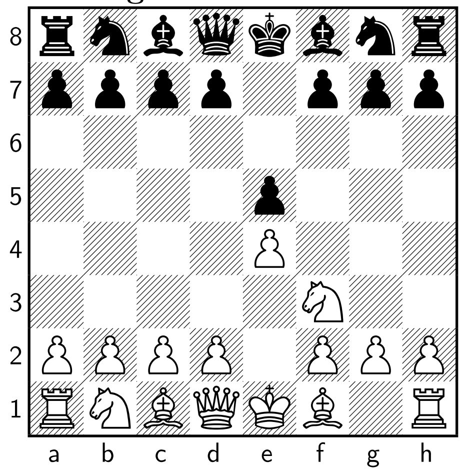

Figure 1: A *transposition* which can for example be reached by the two sequences of moves above.

any further search.[1] To be able to index a table like this by a position on the board, a *hash function* has to be employed that converts a board position into a almost unique, scalar signature. An example of a efficient, widely used hash function was introduced by Zobrist in 1970 [25].

Due to the upper bound of the number of reachable chess positions being approximately 10^{46} [7] it is obvious that transposition tables can only hold a small fraction of them. Corollary, transposition tables have to assume that *hash collisions* will occur. These hash collisions are dealt with using some kind of a *replacement schema*. There exist many implementations of replacement schemas for use with transposition tables but those will not be covered here (for those interested see Breuker [6]).

2.4.3 Quiescence Search

When the search reaches the leaf nodes (with *depth* = 0) the position is rarely evaluated immediately. Instead a more limited *quiescence search* [2,14] is employed. Quiescence search plays out all tactical moves until a "quiet position" is achieved (a position where no winning/tactical moves can be made). At that point it does a evaluation of the position. It is easy to

[1]Assuming that the depth of the previous search was deep enough.

imagine a position where this is mandatory, for example in chess where white has just captured a black piece and instead of giving black the opportunity to recapture, a evaluation of the position is returned immediately. This evaluation will be in huge error. This is the so-called *horizon effect* which is caused by the fixed search depth nature of the search algorithm.

2.4.4 Null-Move Pruning

Every chess player knows that in most cases making a move will be better than doing nothing.[2] This is of course with the exception of checked positions and *zugzwang*[3] positions; a position where every move will lead to a worse position, or in many cases a lost one. In the chess programming world, this observation is called the *null-move observation* and provides the foundation for many selective search techniques in computer chess, the most widely used one being *null-move pruning* [3, 9, 11].

Null-move pruning, also called null-move heuristic, is probably the most widely used selective search enhancement in computer chess and can be described as follows: If the side to move can give up the right to move (pass), letting the opponent play two moves in a row, and still have a good position, then clearly the side to move has a very strong position. In programming terms it reduces the search space by forfeiting a turn (making a null-move) and then performing an $\alpha\beta$ search on the resulting position to a shallower depth than it otherwise would have. This depth reduction is usually referred to as R. If the shallow search yields a beta-cutoff, it assumes that a deeper search would also have yielded a beta-cutoff. However, if it fails to yield a beta-cutoff, the program must redo the search and now to the full depth. Even though null-move pruning is applied recursively and can thus be applied often along the same search path, the technique only exhibits minor technical weaknesses while reducing the search space substantially.

Like most selective enhancements null-move pruning cannot, or should not, be used under some conditions. The most obvious conditions are the ones that we have already mentioned; when the side to move is in check or is experiencing a zugzwang position. But then again it is not trivial to detect whether a given position is a zugzwang position or not.

Many extensions to the null-move pruning technique have been proposed in later years. In 1999 Ernst A. Heinz proposed varying the depth reduction

[2]It must be noted though, that “doing nothing” is an illegal action in chess.

[3]German for “forced to move”.

R with depth. This is known as *adaptive null-move pruning* [13] and is employed by most top chess programs today. Another widely used extension proposed by David and Netanyahu in 2002 is called *verified null-move pruning* [8] and manages to detect most zugzwang positions.

2.4.5 Other Selective Enhancements

Since the popularization of null-move pruning in the 1990s many selective enhancements have been introduced. One of the more recent ones is called *late move reductions* [22] and although the idea behind it has been around for a long time, it only recently became popular within the chess programming community. It is based on the simple observation that in a program with a good move ordering, a beta-cutoff will usually occur either at the first node or not at all. Late move reductions make use of this observation by searching the first few moves with full depth, but the remaining ones with a reduced depth. If one of the reduced moves surprises by returning a good value, the move is re-searched with full depth. As with null-move pruning, some conditions need to be set on what moves to reduce; programs will for example typically not reduce tactical moves like captures and promotions.

Another important selective search enhancement that most modern chess programs will utilize is *futility pruning* [12]. Futility pruning comes in many flavours, but in its pure form it is applied at so-called *frontier nodes*; one ply above the leaf nodes. It reduces the search space by doing a static evaluation of the frontier node and adding to it a *futility margin*, a constant or variable denoting the *largest conceivable positional gain*. If this score is still not good enough futility pruning is triggered and the search will not advance to the next level since it is assumed that nothing good will be found there.

3 Multi-Cut

3.1 Multi-Cut Idea

Multi-cut [4, 5, 24] is a domain-independent, speculative pruning algorithm introduced by Yngvi Björnsson in 1998. It is based on the observation that in practice, it is common that if the first move does not cause a cutoff at a expected CUT node, one of the alternative moves will. Therefore, expected CUT nodes, where many moves have a good potential of causing a beta-cutoff, are less likely to become ALL nodes, and consequently such lines are unlikely to become part of a new principal variation. Central to the multi-cut algorithm is the definition of a *mc*-prune which goes as follows.

> *When searching node N to depth $d + 1$ using an $\alpha\beta$-like search, if at least c of the first e children of N return a value greater or equal to beta when searched to depth $d - r$, an* mc*-prune is said to occur and the local search returns.*

Figure 2 illustrates the basic idea. At node N, before we dive into a normal $\alpha\beta$ search which expands the subtree of m_1 to a full depth of d, we may decide that we want to apply multi-cut first. In that case, the subtrees for the first e moves $(m_1, m_2, \ldots, m_e)$ is expanded to a reduced depth of $(d-r)$. Each time one of these shallower searches return a value greater or equal to beta—a beta-cutoff—we increment a counter by one. If and when this counter ever reaches c we say that an *mc*-prune has occurred and the search returns. That is if c of the first e moves, which are expanded to a reduced depth of $(d - r)$ return a beta-cutoff, a *mc*-prune occurs and the search returns. The moves $(m_2, \ldots, m_e)$ represent the extra search overhead that is introduced by the algorithm. This overhead would not be incurred by the normal $\alpha\beta$ algorithm. The dotted lines however represent the savings that are possible if the *mc*-prune is successful. Because of the exponential nature of the game tree these savings can be substantial. If the *mc*-prune condition is never met we are left with the overhead but no savings and in that case we retreat to a normal $\alpha\beta$ search. It should be clear that by searching m_1 to a reduced depth, we risk overlooking some tactic that could be found had we searched m_1 to a full depth, but we are willing to take this risk because we expect at least one of the c moves that return a value greater or equal to beta when searched to a reduced depth to have caused a genuine beta-cutoff when searched to a full depth.

3.2 Multi-Cut Implementation

Algorithm 1 lists the pseudo-code for a null-window search (NWS) routine using the original multi-cut algorithm. As described in Section 2.4.1, the NWS routine is a part of the PVS algorithm and since the multi-cut algorithm is never applied at PV nodes, we only need to list the updated NWS routine here and not the PVS routine. For clarity we have omitted various details in the pseudo-code concerning transposition tables, time management, draw detection, quiescence search, move ordering schemes, and other search enhancements that are irrelevant to our discussion. A description of the pseudo-code follows. We, however describe it only in details sufficient

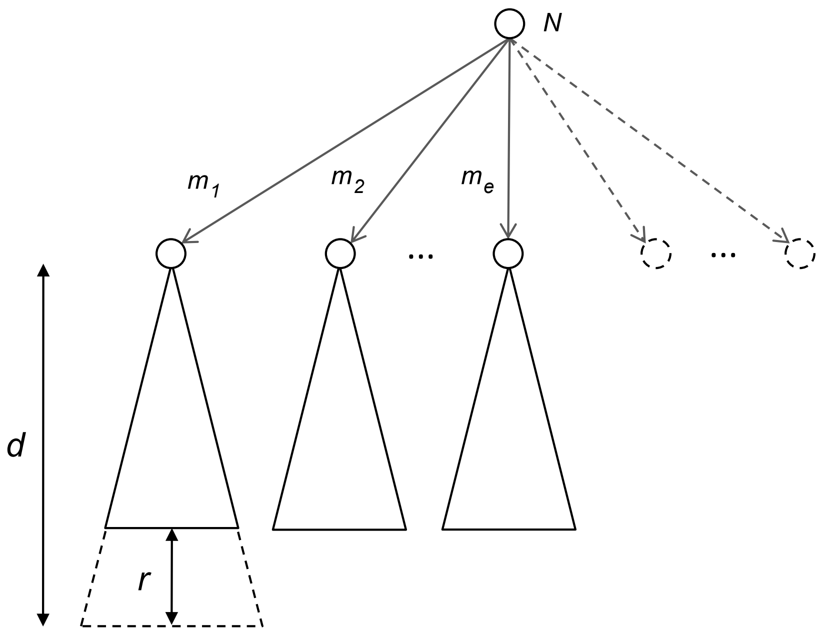

Figure 2: Multi-cut applied at node N.

to highlight the changes that we made during this project—which will be presented in Section 4.

The original multi-cut algorithm begins (lines 1–3) by checking whether it has reached the search horizon, that is whether it has reached a leaf node. If so, it does a evaluation of the current position which then propagates up the tree (in practice quiescence search would be called). Line 6 determines whether to apply multi-cut or not. The original algorithm applies multi-cut at expected CUT nodes, meaning that it is applied at every other layer in the null-window search. This is achieved with the negation of the `cut` variable in the recursive calls. Furthermore, multi-cut is not applied at levels of the tree close to the horizon, thus reducing the time overhead involved in the method. Lines 7–18 perform the actual multi-cut pruning and lines 20–30 do a normal $\alpha\beta$ search if the multi-cut pruning is not applied, or fails to cause a *mc*-prune.

Evidently, the configuration of the three multi-cut parameters E, C and R greatly influences both the efficiency and the error rate of the search.

- **Number of cutoffs needed (C):** This parameter specifies the number of cutoffs needed to cause a *mc*-prune. Setting this parameter too low will result in too many erroneous *mc*-prunes that will make the method too risky. On the other hand, setting this too high will result in a safe method that makes few mistakes but expands too much of the search tree. Ideally this parameter would lead to reasonably safe prunings without exploring too much of the tree.

- **Number of moves to extend (E):** The E parameter specifies how many moves to investigate at most when looking for a *mc*-prune. Again, setting this too high will result in too much of the tree being expanded. Given a program with a good move ordering, we should be able to keep this fairly low.
- **The depth reduction (R):** Reducing the depth of the search too much involves a great risk of overlooking some tactic. But yet again we have to keep the reduction high enough so that we will gain some node savings.

As one can see from these descriptions, it takes a good deal of adjustments and fine-tuning to find the appropriate values for these parameters. In his thesis, Yngvi presents the results of his experiments with various compositions of the three parameters:

> *"The depth reduction was fixed at 2, but the C and E parameters were allowed to vary from 2–6 and 2–12, respectively. We also experimented with different depth reduction factors, but we found that a value of $R = 1$ offers only limited node savings, while values of $R > 2$ were too error prone."*

and,

> *"[..], setting $C = 3$ and E somewhere in the high range of 8–12 looks the most promising. These settings give a substantial node savings (about 20%), while still solving over 99% of the problems that the standard version does."*

These were the values that we used as a baseline when we began enhancing the original algorithm. But it must be kept in mind that a certain configuration that works well for one program will not necessarily work well for another program.

Algorithm 1 Standard multi-cut — $mcNWS(pos, depth, alpha, beta, cut)$

```
Require: $E$ is the number of moves to expand when checking for a $mc$-prune
Require: $C$ is the number of beta-cutoffs needed to cause a $mc$-prune.
Require: $R$ is the depth reduction for multi-cut searches.
 1: if $depth \leq 0$ or $isTerminal(pos)$ then
 2:     return $evaluate(pos)$
 3: end if
 4: $best \leftarrow -\infty$
 5: $moves \leftarrow generateMoves(pos)$
 6: if $depth \geq R$ and $cut$ then
 7:     $c \leftarrow 0$
 8:     for all $m_i$ in $moves$ such that $1 \leq i \leq E$ do
 9:         $make(pos, m_i)$
10:         $v \leftarrow -mcNWS(pos, depth - 1 - R, -beta, -alpha, \neg cut)$
11:         $retract(pos, m_i)$
12:         if $v \geq beta$ then
13:             $c = c + 1$
14:             if $c = C$ then
15:                 return beta
16:             end if
17:         end if
18:     end for
19: end if
20: for all $m$ in $moves$ do
21:     $make(pos, m)$
22:     $v \leftarrow -mcNWS(pos, depth - 1, -beta, -alpha, \neg cut)$
23:     $retract(pos, m)$
24:     if $v > best$ then
25:         $best \leftarrow v$
26:         if $best \geq beta$ then
27:             return $beta$
28:         end if
29:     end if
30: end for
31: return best
```

4 Enhancements

The goal of this project was to reinvestigate the effectiveness of multi-cut as a search enhancement. Since multi-cut is a domain-independent pruning enhancement, we decided to focus our attention on the domain of chess. The motivation behind all of this is driven by the fact that none of the open source chess programs today seem to be utilizing multi-cut, despite it being a fairly known enhancement in the computer chess community. Joona Kiiski, one of the authors of the strongest open-source chess program STOCKFISH had, for example, this to say about his attempts to integrate multi-cut into his program: “Tried this with Stockfish. Couldn’t make it work. I don’t know why. (I also like the idea)” [23]. The need for a reinvestigation of the technique thus obviously exists.

As multi-cut seems to be a very effective method in domains other than chess, one of the first things that we had to figure out was *why* it is not as effective in chess. Our conclusion is that there exists a rather simple explanation for this. Namely, that multi-cut “collides” with null-move pruning. That is, if one method decides to prune a certain branch of the tree the other will in most cases choose to do the same. Null-move pruning (see Section 2.4.4) is the most widely used search enhancement in chess programs since it is incredibly effective while still making very few erroneous decisions. Multi-cut and null-move share at least two major similarities. First, they both prune the tree based on beta-cutoffs. Second, they both use reduced depth searches to compute an estimate for the value of the node and prune the trees if these estimates are above beta. But the two methods still differ in at least one major way, and this is probably the reason that null-move pruning is superior to multi-cut in the domain of chess: While multi-cut does not make any assumptions about its domain, null-move pruning makes a rather big one; that making any move will always be better than doing nothing. Due to this we came to the conclusion that we wanted our enhanced multi-cut algorithm to somehow complement null-move pruning and our empirical analysis in Section 5 reflect this decision. The next three Sections introduce the main enhancements that we made.

4.1 Applicability

The original multi-cut algorithm expects that it will only be successful at expected CUT nodes. Since expected CUT nodes appear on approximately every other layer of the game tree this happens quite frequently. Winands et al. [24] also experimented with applying multi-cut at expected ALL nodes, slightly modifying the PVS algorithm with mixed results.

One of our enhancements addresses where multi-cut is applied in the game tree. Instead of applying it at expected CUT or ALL nodes we apply it at nodes which are *known to have caused a cutoff when they were searched to a lesser depth in a previous search.* We can retrieve this information from the transposition table. Before each search, chess programs begin by doing a transposition table lookup of the current position and depending on the results of this lookup some action is taken. In Figure 3 we see a typical decision diagram for the results of these lookups. The dashed lines represent our enhancement to the diagram.

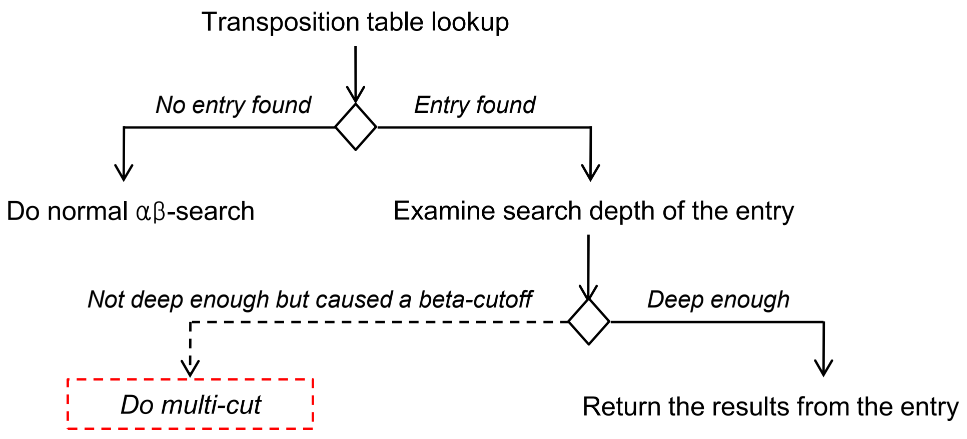
Transposition table lookup
No entry found
Entry found
Do normal αβ-search
Examine search depth of the entry
Not deep enough but caused a beta-cutoff
Deep enough
Do multi-cut
Return the results from the entry

Figure 3: A decision diagram showing how chess programs typically deal with the results of transposition table lookups. The dashed lines represent our changes.

What this essentially means is that we will only perform multi-cut when a shallower, previous search of the same position has resulted in a beta-cutoff. The rationale behind this is that if a shallow search resulted in a beta-cutoff it is often a good indicator that a deeper search will also result in a beta-cutoff. Since transpositions are very common in chess this will happen frequently. It will however not happen as frequently as in the previous scheme; that is at expected CUT nodes. The expectation is thus that we will be applying multi-cut at fewer nodes but with a better probability of success.

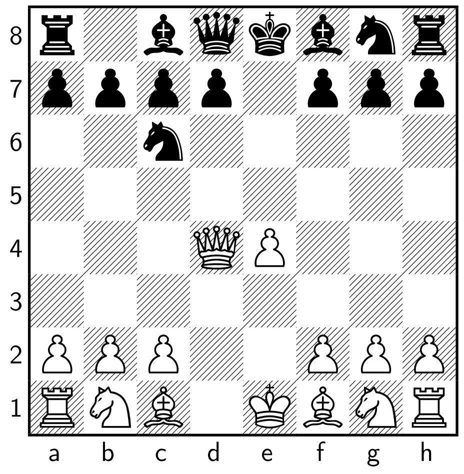

Figure 4: A position where the white queen is threatened.

4.2 Piece Independency

As explained in Section 3, a *mc*-prune will occur in multi-cut search if C of the first E moves yield a beta-cutoff when searched to a reduced depth. A potential problem with this approach is that it might very well be that the same piece is causing all of these cutoffs. For an example of this, consider the position in Figure 4. In this position the white queen is threatened by the knight on square c6. This would mean that if we were to search this position with the original multi-cut algorithm it would quickly result in a *mc*-prune since the queen only has to evade being captured and there are many ways to achieve this evasion. The original multi-cut algorithm would thus come to the conclusion that this particular position was very good for white, when it really is not. While this would not really be a problem in the position in Figure 4, it could easily become a problem in a more complex position. To make the algorithm even more safe we propose an enhancement that states that the beta-cutoffs that occur *need to be independent* of each other. One way to guarantee this independency is to require that the beta-cutoffs must be caused by different pieces. Thus when we are searching the moves with the multi-cut algorithm and come across a piece which has already caused a beta-cutoff we simply skip it and continue with the rest of the moves. This should in theory make the algorithm even more safe but will of course reduce the number of *mc*-prunes that will occur.

4.3 Move Reordering

The third enhancement that we propose in this report is what we call *move reordering.* This enhancement was also used by Winands et al. in [24]. This is a fairly simple and intuitive enhancement that should—at least theoretically—boost the effectiveness of the multi-cut algorithm. Move reordering tries to make use of multi-cut applications that fail to cause a *mc*-prune. As we saw in Algorithm 1, the original multi-cut algorithm will retreat to normal $\alpha\beta$ search if it fails to cause a *mc*-prune. Now let us assume that we configure multi-cut so that 3 beta-cutoffs are needed to cause a *mc*-prune ($C = 3$). Since three beta-cutoffs are needed, positions that cause 2 or less will thus be a waste of time and resources. This enhancement tries to make some use of the moves that cause beta-cutoffs in the multi-cut search but do not lead to a *mc*-prune. This is done by "remembering" the moves that cause beta-cutoffs in the multi-cut search, so when the search routine retreats to a normal $\alpha\beta$ search it will *begin by searching the moves that caused beta-cutoffs during the reduced multi-cut search.* Again, the rationale behind this is that reduced depth search for a certain position will most often give a good estimate on a deeper search of the same position. Moves that will cause a cutoff at reduced depth should therefore have a higher probability to cause a cutoff when searched with full depth and should thus be searched as quickly as possible.

4.4 Enhanced Multi-Cut Implementation

Algorithm 2 lists the pseudo-code for our enhanced version of the multi-cut algorithm which includes all three aforementioned enhancements. As with the pseudo-code for the original multi-cut algorithm in Algorithm 1, we skip many details. One of the things that we do not specify is how our enhancements should be implemented when it comes to data structures. The main reason for this is the fact that chess programs vary greatly in terms of their architecture and internal data structures. It should be up to the authors to decide how to implement each search enhancement. The pseudo-code thus only shows the absolute necessary functionality associated with our enhancements.

Lines 3–5 represent the "Applicability"-enhancement described in Section 4.1. Instead of doing multi-cut at an expected CUT node, we do it when the transposition table recognizes that a cutoff occurred in a shallower, previous search of the same position. Again we omit transposition table code that

Algorithm 2 Enhanced multi-cut — $emcNWS(pos, depth, alpha, beta)$

```
Require: $E$ is the number of moves to expand when checking for a $mc$-prune
Require: $C$ is the number of beta-cutoffs needed to cause a $mc$-prune.
Require: $R$ is the depth reduction for multi-cut searches.
 1: if $depth \leq 0$ or $isTerminal(pos)$ then
 2:     return $evaluate(pos)$
 3: end if
 4: $entry \leftarrow transLookup(pos)$ // Addition.
 5: $cut \leftarrow entry.depth < depth$ and $entry.isCutoff$ // Addition.
 6: $best \leftarrow -\infty$
 7: $moves \leftarrow generateMoves(pos)$
 8: if $depth \geq R$ and $cut$ then
 9:     $c \leftarrow 0$
10:     for all $m_i$ in $moves$ such that $1 \leq i \leq E$ do
11:         // Added "if" block:
12:         if $getPiece(m_i)$ has not caused a cutoff before then
13:             $make(pos, m_i)$
14:             $v \leftarrow -emcNWS(pos, depth - 1 - R, -beta, -alpha)$
15:             $retract(pos, m_i)$
16:             if $v \geq beta$ then
17:                 $markAsCutoff(m_i)$ // Addition.
18:                 $c = c + 1$
19:                 if $c = C$ then
20:                     return beta
21:                 end if
22:             end if
23:         end if
24:     end for
25: end if
26: $moveCutoffsInFront(moves)$ // Addition.
27: for all $m$ in $moves$ do
28:     $make(pos, m)$
29:     $v \leftarrow -emcNWS(pos, depth - 1, -beta, -alpha)$
30:     $retract(pos, m)$
31:     if $v > best$ then
32:         $best \leftarrow v$
33:         if $best \geq beta$ then
34:             return $beta$
35:         end if
36:     end if
37: end for
38: return best
```

does not directly concern our algorithm. In line 12 we introduce a new "if" block which enforces that only pieces that have not caused a cutoff in the past are examined as Section 4.2 explains. Line 17 and 26 represent our last enhancement, the move reordering (see 4.3). Line 17 marks the move that is causing the cutoff as a *cutoff-move*. This means that line 26 somehow has to order the moves that are marked as cutoff-moves appropriately in the list of moves to be searched. This will in most cases simply put these moves at the front.

5 Empirical Analysis

To evaluate the feasibility of our enhanced multi-cut algorithm extensive experiments had to be carried out. These experiments were conducted on our own chess program, ZIGGY. ZIGGY is an open-source[4], highly configurable Java based chess program with state of the art enhancements like null move pruning, late move reductions, multi-cut, PVS, static exchange evaluation [20, 21] and a two-level transposition table. It is built to serve as a test-bed for search enhancement development and analysis.

5.1 Experimental methods in chess programs

Before we introduce the results of the empirical analysis we want to discuss briefly the experimental methods that can be used when analyzing the effectiveness of enhancements and changes to chess programs, as well as the drawbacks of these methods. Research papers in computer chess often use a mixture of these testing methods and our experiments are no exception thereof.

Test-suites using fixed time search

A popular method of testing chess programs is based on *test-suites*. Test-suites are essentially a series of chess positions that pose some tactical challenge and have a known "best move". They have been used as a human chess exercising material for a long time and many of them were originally designed with humans in mind. The benefits of using test-suites as a testing method

[4]For those interested, the source code is publically available at `https://github.com/krummi/ChessEngine/`.

in chess programs are first and foremost that it does not take a long time to run them and that they provide the developer with some approximation on how much impact a change has introduced to his chess program. One should however be careful to tune programs based on test-suite results since they do not necessarily correlate with practical playing strength in matches against other opponents, which is the main problem with using test-suites as a testing mechanism.

When chess programs use test-suites to test their strength they are always given some restriction on for how long/how deep they can search each position. A fixed time restriction will for example only allow the programs to search each position in the test-suite for a fixed amount of time, usually in the range of 1 second (for the easier test-suites) to over 60 seconds (for the harder ones). The results of these tests will often just be a single number: how many positions the program solved given the limited time-frame, which can then be used for comparison purposes. The strength of chess programs can for example be tested in this way before and after some change has been made and the number of positions that it solves can then be compared. The problem with this type of time restriction testing is however the fact that the results are most often just a single number. Further, "time" is a fuzzy phenomena to computers and due to that, running the same test-suite twice with the same fixed time-frame can theoretically give two different results.

Test-suites using fixed depth search

As opposed to the fixed time restriction we can also decide to use a fixed depth restriction. This will simply search each of the positions in the test-suite to a certain depth no matter how much time it takes. This means that some positions will be searched very quickly while others may take much more time to be searched. The merit of using fixed depth testing is that it does not only allow for a comparison of the number of positions that are solved, but it also provides a comparison on the size of the trees that are searched. This can be done because most chess programs keep statistics about how many nodes they have searched in each search. For example, if we were to test two different versions of a chess program (a) with a new search enhancement and (b) without it, we could use fixed depth testing to estimate how many nodes (a) searches as compared to (b). If (a) searches 15% fewer nodes while solving the same number of positions as (b) one can assume that the new search enhancement has paid off. But this hypothesis can only be tested in a *self-play match* between (a) and (b), or by making them compete against the same opponent and to compare the results of these matches.

Matches

Most commonly though, chess program testing involves matching distinct versions of a program against each other in a series of matches and by analyzing the results. The reason is simply that this is by far the most reliable way of comparing chess programs. A single match will make the programs think about many realistic positions that are evenly distributed between the start/mid- and end-game with realistic time controls. The problem with this approach is however that many matches need to be played out to get a result that is statistically significant and can thus be used to state whether some enhancement has made a difference or not. To be able to play this many matches a considerable amount of time and computing power needs to be at hand, especially time if realistic time controls are used.

5.2 Test-suite results

Test-suites served an important role during the course of this project. Fixed depth test-suite testing allowed us to objectively compare new revisions of the multi-cut algorithm to older ones, providing us both with data on the number of positions solved and on the size of the tree expanded. Our main test suite during the experiments was the "Win At Chess" (WAC) test-suite which contains 300 positions. This test-suite seemed ideal since our program solved more than 80% of the positions in a matter of minutes when searched to a fixed depth of 8. While tests that utilize test-suites are usually done in a very systematic fashion with a test-suite containing more than 1000 positions that was not an option in our case. This was due to the fact that the process that we used during this project was very iterative: First we came up with some enhancement and carried out tests that we found fitting. We then made the results of these tests guide us and based on them we most often performed additional tests with some changes or parameter alterations. If we found a particular configuration interesting during the test-suite testing we made it a *candidate* for match testing.

Table 1 lists some of the main candidates that we used in the self-play matches. It must be noted though that we only show those that we find necessary. Their multi-cut configuration is shown, how they did against the test-suite and the number of nodes they explored. The original multi-cut parameters are shown in the first three columns (R, C and E). The fourth column ("Apply at") tells where multi-cut is applied (see 4.1): at expected CUT nodes ("Cut") or when a previous, shallower search indicates a beta-cutoff in the transposition table ("trans"). The fifth column ("Ind.") tells whether independency checking (see 4.2) is on or off and the sixth whether reordering (see 4.3) is done or not.

Table 1: A list of the main candidates, their configuration and test-suite performance. z-none neither employs null-move nor multi-cut pruning and is the baseline configuration. z-nm only employs null-move pruning. Others employ both null-move and multi-cut pruning.

Multi-cut configuration									
R	C	E	Apply at	Ind.	Reorder	Nodes	%	Solved	ID
-	-	-	-	-	-	1,991,573,761	100.0%	252/300	z-none
-	-	-	-	-	-	583,740,779	29.3%	246/300	z-nm
2	3	10	Cut	Off	Off	548,886,666	27.6%	238/300	z-thesis
2	2	10	Trans	Off	Off	467,711,698	23.5%	236/300	z-1
2	2	10	Trans	On	Off	480,274,921	24.1%	240/300	z-2
2	2	20	Trans	On	Off	455,480,763	22.9%	237/300	z-ind.
2	2	20	Trans	On	On	430,812,352	21.6%	237/300	z-trans
2	2	16	Cut	On	On	412,961,310	20.7%	236/300	z-cut
2	3	16	Trans	On	On	597,443,206	30.0%	245/300	z-3
3	3	16	Trans	On	On	505,147,481	25.4%	240/300	z-main

Originally our experiments were focused on trying to decrease the number of cutoffs needed for a mc-prune (C) from 3 to 2 and thus increase the number of mc-prunes. We were hoping to achieve this by introducing the independency check enhancement (see for example z-ind. and z-trans in Table 1). This worked well and outperformed the original multi-cut configuration but it never got to the point of outperforming the version of our program that only employed null-move pruning. That is, no multi-cut configuration of this type in blend with null-move pruning seemed to outperform a version that only employed null-move pruning (z-nm in Table 1)—which in turn meant that multi-cut was actually doing more harm than good. This told us that using a configuration with $C = 2$ along with the independency check was probably still too aggressive and consequently too error-prone. Instead we decided to try something completely different: to increase C to 3 while keeping the independency check on. We knew that this would result in a safer method that expanded too much of the tree (z-3 in Table 1). But since we were now making very safe *mc*-prunes, we realized that we could risk increasing the depth reduction factor to $R = 3$. The test-suite results seemed fine and we decided to match this version (z-main in Table 1) against the only-null-move configuration. Surprisingly this did very well and not only outperformed the original multi-cut configuration but the only-null-move configuration as well, as will be demonstrated in the next section.

5.3 Results of self-play matches

Self-play matches were used to assess two things. First, to estimate the improvement for each of the proposed enhancements. Second, to estimate the overall improvement of our enhanced multi-cut algorithm. Self-play matches consisted of around 200 matches where both programs had 5 minutes of thinking time, making each match at most 10 minutes. To prevent the programs from playing the same game over and over, fifty well-known opening positions were used as a starting point for each match. The programs played each opening once from the white side and once as black. Finally results were gathered and ELO ratings were computed with 95% confidence intervals based on a student t-test.

Enhancement improvements

To measure the improvement that each of the proposed enhancements introduced we simply ran two configurations of the program: one with the enhancement turned on and one with it turned off. This sounds logical but a few problems exist with this approach: First, you need to settle for some basic configuration. We used the basic configuration of $R = 2$ and $C = 2$ since it was our best configuration at the time of these experiments. Secondly, some of the enhancements rely on other enhancements—for example we would never use the independency check enhancement without using the applicability enhancement—and since they are not strictly independent their concurrent use can lead to skewed results. What follows is the results of these experiments.

As can be seen from these results the applicability enhancement seems to introduce the most improvement. The independency check introduces surprisingly little gains, but again this was an example of a enhancement that was hard to measure. The move reordering does not seem to provide us with any improvement. But on the other hand, a configuration of $C = 2$ means

Table 2: The results of matching versions of the program with and without the *applicability enhancement* against each other.

Applicability enhancement		
z-trans (with) vs. z-cut (without)		
Score	*Winning %*	*ELO difference*
109.5 - 86.5	55.9%	41 ± 34

Table 3: The result of matching versions of the program with and without the *independency check* enhancement against each other.

Independency check enhancement		
z-ind. (with) vs. z-1 (without)		
Score	*Winning %*	*ELO difference*
102.5 - 95.5	51.8%	13 ± 34

Table 4: The result of matching versions of the program with and without the *reordering enhancement* against each other.

Reordering enhancement		
z-trans (with) vs. z-ind (without)		
Score	*Winning %*	*ELO difference*
96.5 - 99.5	49.2%	-5 ± 34

that the move reordering enhancement never moves more than a single move to the front of the move-queue and given a good move ordering these moves will probably already be at the front of the queue.

Overall improvements

Ultimately, we wanted our enhanced multi-cut algorithm to (a) outperform the original multi-cut algorithm and (b) outperform the version of the program which only used null-move pruning and no multi-cut. The results of these matches follow.

As can be seen our main configuration outperformed both (a) and (b). As expected, the results against (a) show a definite improvement. Although not as convincing, the results against (b) are encouraging. At least there seems to be something there that may be worth examining more closely in a high-performance chess program like STOCKFISH.

Table 5: Results of our main configuration against (a).

z-main (enhanced) vs. z-thesis (original)		
Score	*Winning %*	*ELO difference*
111 - 86	56.3%	45 ± 36

Table 6: Results of our main configuration against (b).

z-main (enhanced) vs. z-nm (only null-move)		
Score	*Winning %*	*ELO difference*
105 - 90	53.8%	26 ± 36

6 Conclusion and Future Work

In this report we have given a brief historical overview of how search paradigms in chess have shifted over the years and how brute-force searchers have long lost their dominance to searchers that employ selective search techniques. We have also covered some of the basic components that make up chess programs. Furthermore we have reviewed the selective search technique multi-cut and introduced enhancements to it. Our enhanced multi-cut algorithm has been implemented and experimented with in our own chess program. The results of these experiments demonstrate a statistically significant improvement over the original multi-cut algorithm. The enhanced algorithm also seems to complement null-move pruning better than the original algorithm did, giving rise to optimism. But while our chess program served well as a first test-bed for the enhanced multi-cut algorithm, its real strength improvement cannot be assessed without it being implemented and experimented with in a high-performance, state of the art chess program.

References

[1] L. Atkin and D. Slate. Chess 4.5-the northwestern university chess program. In *Computer chess compendium*, pages 80–103. Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[2] D. F. Beal. Mating Sequences in the Quiescence Search. *ICCA Journal*, 7(3):133–137, 1984.

[3] D. F. Beal. Experiments with the null move. In *Advances in Computer Chess 5*, pages 65–79. Elsevier, 1989.

[4] Y. Björnsson. *Selective Depth-First Game-Tree Search*. PhD thesis, University of Alberta, 2002.

[5] Y. Björnsson and T. A. Marsland. Multi-cut pruning in alpha-beta search. In *Proceedings of the First International Conference on Computers and Games*, CG 1998, pages 15–24, London, UK, 1999. Springer-Verlag.

[6] D. Breuker. *Memory versus Search in Games*. PhD thesis, Universiteit Maastricht, The Netherlands, 1998.

[7] S. Chinchalkar. An Upper Bound for the Number of Reachable Positions. *ICCA Journal*, 19(3):14–18, 1996.

[8] O. David and N. S. Netanyahu. Verified null-move pruning. *ICGA Journal*, 25(3):153–161, 2002.

[9] C. Donninger. Null Move and Deep Search: Selective-Search Heuristics for Obtuse Chess Programs. *ICCA Journal*, 16(3):137–143, 1993.

[10] J. J. Gillogly. The technology chess program. In *Computer chess compendium*, pages 67–79. Springer-Verlag New York, Inc., New York, NY, USA, 1988.

[11] G. Goetsch and M. S. Campbell. Experiments with the null-move heuristic. In *Computers, Chess, and Cognition*, pages 159–168. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[12] E. A. Heinz. Extended Futility Pruning. *ICCA Journal*, 21(2):75–83, 1998.

[13] E. A. Heinz. Adaptive null-move pruning. *ICCA Journal*, 18(2):123–132, 1999.

[14] H. Kaindl. Quiescence Search in Computer Chess. *SIGART Newsletter*, (80):124–131, 1982.

[15] D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[16] Y. J. Lim. *On Forward Pruning in Game-Tree Search*. PhD thesis, National University of Singapore, 2007.

[17] T. A. Marsland and M. Campbell. Parallel search of strongly ordered game trees. *ACM Comput. Surv.*, 14:533–551, December 1982.

[18] T. A. Marsland and F. Popowich. Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7:442–452, 1985.

[19] A. Newell, J. C. Shaw, and H. A. Simon. Chess-playing programs and the problem of complexity. *IBM J. Res. Dev.*, 2:320–335, October 1958.

[20] F. Reul. *New Architectures in Computer Chess*. PhD thesis, Maastricht University, 2009.

[21] F. Reul. Static exchange evaluation with $\alpha\beta$-approach. *ICGA journal*, 33(1), 2010.

[22] T. Romstad. An Introduction to Late Move Reductions. `http://www.glaurungchess.com/lmr.html`, 2007. [Online; accessed 7-May-2011].

[23] TalkChess.com. Multi-cut, SE and ETC. `http://www.talkchess.com/forum/viewtopic.php?t=35697`, 2006. [Online; accessed 8-May-2011].

[24] M. H. M. Winands, H. J. van den Herik, J. W. H. M. Uiterwijk, and E. C. D. van der Werf. Enhanced forward pruning. *Inf. Sci.*, 175:315–329, November 2005.

[25] A. Zobrist. A New Hashing Method with Application for Game Playing. Technical Report #88, Computer Science Department, The University of Wisconsin, Madison, WI, USA, 1970.