Enhanced Forward Pruning

AI Games Seminar

Qingyang Cao

Matrikel-Nr. 3529070

Heidelberg University

August 2019

****

Contents

Executive Summary

This report discussed several strategies that can be used in pruning in Chess Game Programming.

The first is to use the searched value from Null Move searching as a lower bound. From Null Move Pruning to Verified Null Move Pruning, a further search improved the pruning performance and decreased the risk of mistaken pruning.

Another strategy is to use Principal Variable Search. And a new forward pruning method, Multi-Cut can be applied here. At first this pruning was only executed at CUT nodes. After experimental study, it has been proved that applying it at ALL nodes combined with that at CUT nodes can give a comparable pruning performance: the tree sizes of the combination MC-C and MC-A are approximately the same as the combination Null move and MC-C.

Introduction

In chess program, the speed is a function of its move generation cost, the complexity of the position under study, and the brevity of its evaluation. More important, however, is the quality of the mechanisms used to discontinue (prune) search of unprofitable continuations.

Before the mid-1970s, most chess programs were using the 'Plausible Move Generating' method to search. It required expert knowledge to select a few moves which they considered plausible at each node and then pruned large parts of the search tree. [1] However, selecting those discards based on an extensive analysis, this forward pruning is known to be error prone and dangerous.

The tactical shortcomings for plausible-move generating programs are serious and then it moved to the age of "Brute-Force Search Programs": TECH (Gillogly, 1972) [2] and CHESS 4.X (Slate and Atkin, 1977) managed to reach depths of 5 plies and more. Further, with check extensions and recaptures, there came the most successful brute-force programs like BELLE (Condon and Thompson,1983a,b), DEEP THOUGHT (Hsu, Anantharaman, Campbell, and Nowatzyk, 1990), HITECH (Berliner and Ebeling, 1990; Berliner, 1987; Ebeling, 1986), and CRAY BLITZ (Hyatt, Gower, and Nelson, 1990), which for the first time managed to compete successfully against humans.

The next stage was the forward-pruning program which took the idea of variable-depth search: exploring promising moves more deeply (extending the search) and non-promising moves less deeply (pruning the search). With null-move pruning introduced to it (Beal, 1989; Goetsch and Campbell, 1990; Donninger, 1993) in the early 1990s, it enabled programs to search more deeply with minor tactical risks. Ever since then, forward-pruning programs has reigned the chess programming world. Today almost all top-tournament playing programs use forward-pruning methods, null-move pruning is one of the most popular strategies.

Afterwards, a new forward-pruning mechanism, called Multi-Cut came up and it has been adopted by some of the world's strongest commercial chess programs. At first, this method was only applied at CUT nodes; later it has been proved to be beneficial to also use forward-pruning methods at expected ALL nodes in the Principal Variation Search (PVS) framework.

1 Minimax Search

Chess game can be described as two-person zero-sum perfect information games. Nodes in a two-person game-tree represent positions and the branches correspond to moves. Under the assumption of best play by both sides the aim of the search is to find a path from the root to the highest valued leaf node that can be reached. [1]

The Minimax rule is used to propagate the value of those outcomes back to the initial state, where the root of the tree is the initial state and the leaf nodes are the terminal states. The Minimax rule, the Max, the player to move at the root, tries to optimize its gains by returning the maximum of its children values. The other player, Min, tries to minimize Max's gains by always choosing the minimum value. This framework is referred to as NegaMax. [2]

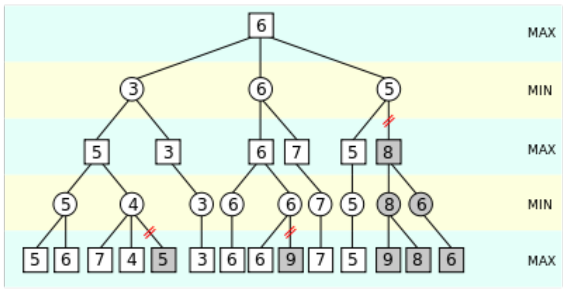


Figure 1: Minmax Search

However, visiting every node in a Minmax tree should be expensive. Fortunately the work can be reduced by excluding unnecessary nodes. It has been known since 1958 that pruning is possible in a minimax search. [3] : only a so-called Minimal Tree needs to be expanded. The minimax value depends only on the nodes in the minimal tree. The Minimal Tree contains three types of nodes: Every child of a PV-node or an ALL-node is a part of the Minimal Tree, but only one child of a CUT-node.

2 Alpha-Beta Pruning Search

2.1 Assumption of Alpha-Beta Algorithm

Using Alpha-Beta Pruning Search, a Minimal Subtree can be found which can deliver the minimax value.

Intuitively, if the opponent finds one continuation that makes the value of the current subtree inferior to the lower-bound already established at node n, then there is no need to search further and the current node becomes a CUR-node, which is also called a β-cutoff.

2.2 Procedure of Alpha-Beta Algorithm

Alpha-Beta algorithm is one of the most reliable brute-force pruning method in popular use. [4]. The α-β algorithm employs lower (α) and upper (β) bounds on the expected value of the tree, where α stores minimum score that the maximizing player is assured of at current node which was initially set to negative infinity and β stores the maximum score that the minimizing player is assured of and was initially set to positive infinity.

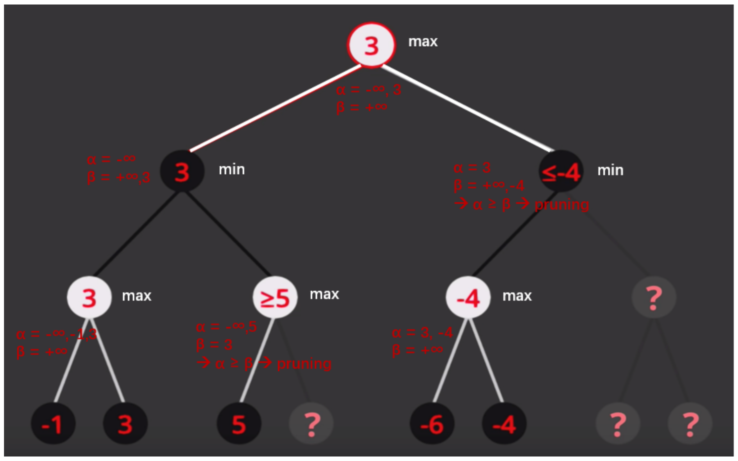


Figure 2: Alpha-Beta Search

The Alpha-Beta algorithm owes its efficiency to the employment of two bounds which form a window. In this case, the brench at a node can be pruned when $\alpha \geqslant \beta$, and it is more efficient when it is ordered with the most possible one first.

```
$S \leftarrow Successors(n)$
if $d \leq 0 \vee S \equiv \emptyset$ then
    return $f(n)$
$best \leftarrow -\infty$
for all $n_i \in S$ do
    $v \leftarrow -\alpha\beta(n_i, d-1, -\beta, -max(\alpha, best))$
    if $v > best$ then
        $best \leftarrow v$
        if $best \geq \beta$ then
            return $best$
return $best$
```

Figure 3: Beta-Cutoff Algorithm

2.3 Strength of Alpha-Beta Algorithm

It allows a large portion of the game-tree to be pruned, while still backing up the correct game-tree value.[5]

2.4 Weakness of Alpha-Beta Algorithm

Since the algorithm explores all continuations to some fixed depth, the number of nodes visited by the algorithm still increases exponentially with increasing search depth which obviously limits the scope of the search.

2.5 Improvement from Alpha-Beta Algorithm

To improve this, various heuristics allow variations in the distance to the search horizon, some move sequences can then be explored more deeply than others. In this way, "Interesting" continuations are expanded beyond the nominal depth, while others are terminated prematurely. This is refered as Forward

Pruning method which involves some risk of overlooking a good continuation. Here the time saved by pruning non-promising lines is better spent searching others more deeply, in an attempt to increase the overall decision quality.

3 Null Move Pruning & Verified Null Move Pruning

Null Move pruning can give a faster pruning in Minmax search.

3.1 Null Move Pruning

3.1.1 Assumptions of Null Move Pruning

The assumption in this method would be: the null move(doing nothing) is never a best choice. Then the outcome from a null move search can serve as an updating lower bound which is noted as α.

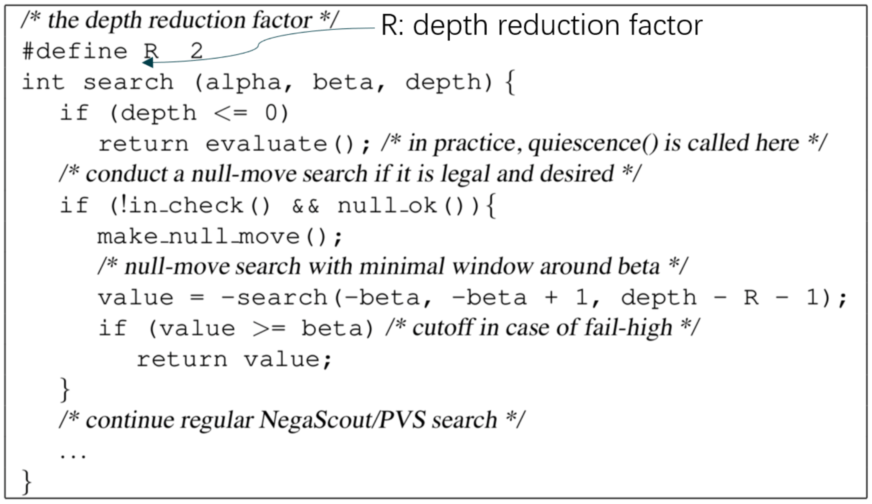
3.1.2 Procedure of Null Move Pruning

There are simple steps to achieve this standard Null Move Pruning:

1. Place a null move(assuming that a null move is legal): Swap the two sides of Min player and Max Player(this cannot be done in positions where that side is in check, since the resulting position would be illegal. Also, two null moves in a row are forbidden, since they result in nothing)

2. Conduct a regular search with a reduced depth there and save the searched value.

For step 2, suppose that R is the depth reduction factor indicating how many layers are about to be reduced in the algorithm. And after these two steps, a possible lower bound is returned as the searched value.


R: depth reduction factor

```
/* the depth reduction factor */
#define R  2
int search (alpha, beta, depth) {
   if (depth <= 0)
      return evaluate(); /* in practice, quiescence() is called here */
   /* conduct a null-move search if it is legal and desired */
   if (!in_check() && null_ok()){
      make_null_move();
      /* null-move search with minimal window around beta */
      value = -search(-beta, -beta + 1, depth - R - 1);
      if (value >= beta) /* cutoff in case of fail-high */
         return value;
   }
   /* continue regular NegaScout/PVS search */
   ...
}
```

Figure 4: Depth-reduced search

In this process, it would work dynamically:

1. if $\beta \leq$ searched value, then it indicates a "fail-high" here, which leads to a cutoff at this node;
2. if $\alpha <$ searched value $\leq \beta$, then update α value with the searched value and continue;
3. if searched value $< \alpha$, no cutoff nor updating is applied.

3.1.3 Strength of Null Move Pruning

Above together form a Minimal-window Null-Move search around current upper bound β. When it met the condition that $\beta \leq$ searched value, and the cutoff applied, then this method can save it from unnecessary further searching. And these cutoff decisions on dynamic criteria can give it greater tactical strength.[6]

3.1.4 Weakness and Flaws of Null Move Pruning

This standard Null Move Pruning method also got flaws. First, it will fail at Zugzwang positions when null move is actually the best choice. In another word, it does not work well when the assumption is violated. Although Zugzwang positions are rare in the middle game, they are not an exception in endgames, thus Null-Move pruning should be avoided in such endgame positions. And apparently the searched value is an upper bound in such cases.

Second, there could be Horizon effect (Berliner, 1974) when the reduced-depth search misses a tactical threat. When the delaying moves unnecessarily weaken the position or give up additional material to postpone the eventual loss, the insertion of delaying moves that cause any inevitable loss of material would occur beyond the program's horizon (maximum search depth), where the loss is then hidden. Generally, programs with a poor or inadequate quiescence search suffer more from the horizon effect. there is a possibility that it will make a detrimental move, but the effect is not visible because the computer does not search to the depth of the error (i.e., beyond its "horizon").

Further, in passed experiments, it has been proved that the performance is sensitive to the reduction factor R. Generally, R = 2 performs better and is mostly used (Feist, 1999), while R = 1 is too conservative and R = 3 is too aggressive (Heinz, 1999). Since using R = 3 is tempting, adaptive null-move pruning (Experiments by Heinz, 1999) used Adaptive R (Donninger,1993) (R = 3 in upper parts of the search tree and R = 2 in its lower parts) for improvement, which can save 10 to 30 percent of the search effort in comparison with a fixed R = 2, while maintaining overall tactical strength.

3.1.5 Improvement for Null Move Pruning

If no depth reduction was applied, then there would be no tactical threat due to horizon effect. The greater the depth reduction R, the greater the tactical risk due to the horizon effect. For improvement, extending the search algorithm with a quiescence search can ease the horizon effect. This gives the search algorithm ability to look beyond its horizon for a certain class of moves of major importance to the game state.

3.2 Verified Null Move Pruning

To cope with flaws in standard Null Move Pruning, here comes the Verified Null Move Pruning. The general idea is to apply a verification to check whether the returned value is indeed a lower bound on the position before pruning.

The horizon effect problem(tactical weaknesses) was solved by not to prune immediately based on the value returned from the shallow null-move search, but instead to continue the search with reduced depth (Goetsch and Campbell, 1990) when a fail-high was indicated in a Null-Move Search. And it can prevent mistaken pruning in Zugzwangs by verifying the searched value as a lower bound(Plenkner, 1995).

3.2.1 Procedure of Verified Null Move Pruning

Here is how the Verified Null Move Pruning method worked:

1. At each node, apply a standard Null Move Search with the reduction factor R set to 3;

2. Verification of cutoff: At each node that the searched value $\geq \beta$ (indicating a fail high), reduce the depth by one player and continue the search for that node's sub-tree using standard Null Move Pruning with reduction factor R = 3.

2.1) If nodes have another Null-Move search fail-high indication in one of its ancestors, then apply cutoffs there;
2.2) If the Null-Move search indicates a cutoff, but the search shows that the best value < β, then this position is a Zugzwang, then restore the original depth(add back one player depth) and research.

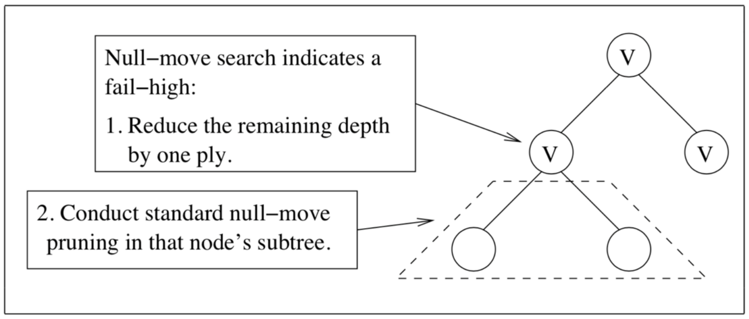
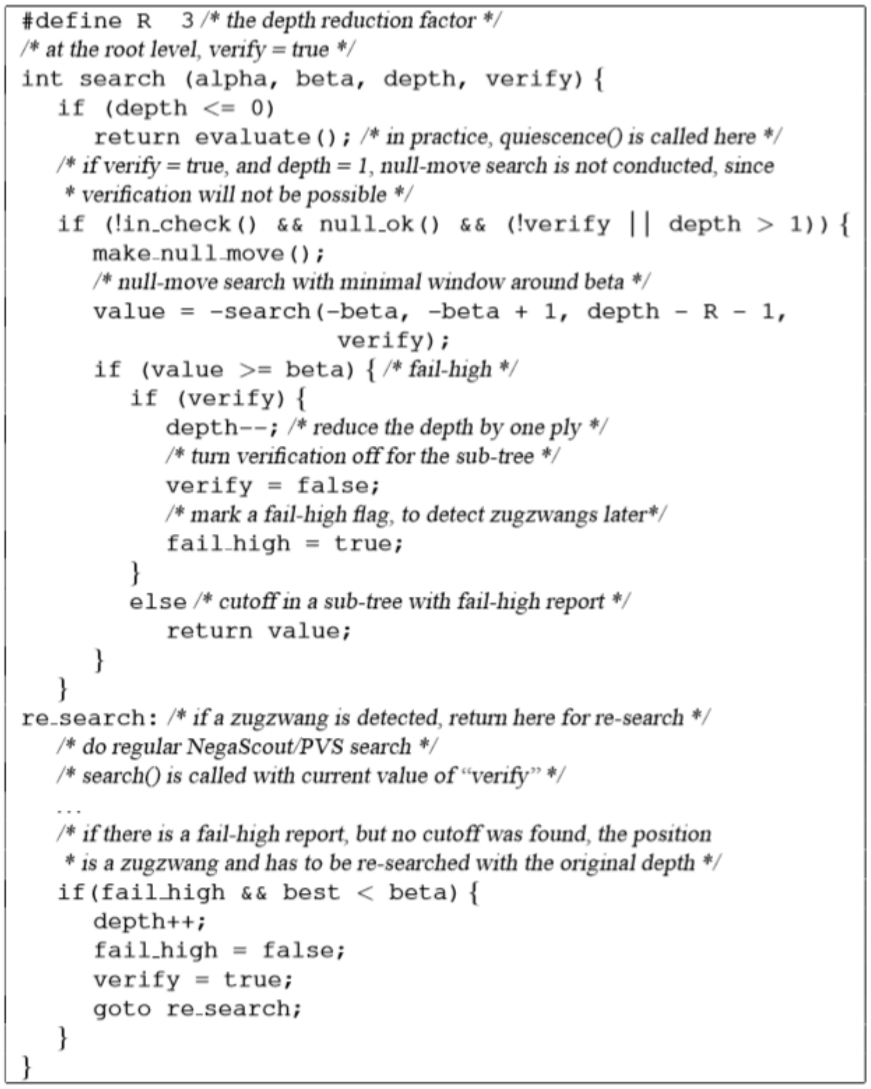

Null−move search indicates a fail−high:
1. Reduce the remaining depth by one ply.
2. Conduct standard null−move pruning in that node's subtree.
V
V
V

Figure 5: Verified Null Move Pruning

3.2.2 Strength of Verified Null Move Pruning

It would derive a smaller search tree which is close in size to that of standard Null-Move pruning with R = 3 with greater tactical strength than that of standard Null-Move pruning with R = 2, and it allows the use of R = 3 in all parts of the search tree. It manages to detect most Zugzwangs and in such cases conducts a re-search to obtain the correct result.

Further, it is easy to implement(only few codes need to be changed) and it is applicable to all standard Null Move Pruning program.



```
#define R   3 /* the depth reduction factor */
/* at the root level, verify = true */
int search (alpha, beta, depth, verify) {
   if (depth <= 0)
      return evaluate(); /* in practice, quiescence() is called here */
   /* if verify = true, and depth = 1, null-move search is not conducted, since
    * verification will not be possible */
   if (!in_check() && null_ok() && (!verify || depth > 1)) {
      make_null_move();
      /* null-move search with minimal window around beta */
      value = -search(-beta, -beta + 1, depth - R - 1,
                         verify);
      if (value >= beta) { /* fail-high */
         if (verify) {
            depth--; /* reduce the depth by one ply */
            /* turn verification off for the sub-tree */
            verify = false;
            /* mark a fail-high flag, to detect zugzwangs later*/
            fail_high = true;
         }
         else /* cutoff in a sub-tree with fail-high report */
            return value;
      }
   }
re_search: /* if a zugzwang is detected, return here for re-search */
   /* do regular NegaScout/PVS search */
   /* search() is called with current value of "verify" */
   ...
   /* if there is a fail-high report, but no cutoff was found, the position
    * is a zugzwang and has to be re-searched with the original depth */
   if(fail_high && best < beta) {
      depth++;
      fail_high = false;
      verify = true;
      goto re_search;
   }
}
```

Figure 6: Adding few lines to Null Move Searching codes

3.2.3 Experiments of Verified Null Move Pruning

The experiment was designed based on GENESIS eigine and NEGASCOUT/PVS (Campbell and Marsland, 1983; Reinefeld, 1983) search algorithm, with history heuristic (Schaeffer, 1983, 1989) and transposition table (Slate and Atkin, 1977; Nelson, 1985). And one-ply check extensions on leaf nodes was used to demonstrate the tactical strength differences.

138 test positions from Test Your Tactical Ability by Yakov Neishtadt Depths[7] were searched to depths of 9 and 10 plies, using standard R = 1, R = 2, R = 3, and verified R = 3.

The total node count for each method and the size of the tree in comparison with verified R = 3:

Depth	Std $R = 1$	Std $R = 2$	Std $R = 3$	Vrfd $R = 3$
9	1,652,668,804 (+267.46%)	603,549,661 (+34.19%)	267,208,422 (-40.58%)	449,744,588 -
10	11,040,766,367 (+661.64%)	1,892,829,685 (+30.57%)	862,153,828 (-40.52%)	1,449,589,289 -

Figure 7: Total node count of standard R = 1, 2, 3 and verified R = 3 at depths 9 and 10, for 138 Neishtadt test positions

The number of positions that each method solved correctly:

Depth	Std $R = 1$	Std $R = 2$	Std $R = 3$	Vrfd $R = 3$
9	64	62	53	60
10	71	66	65	71

Figure 8: Number of solved positions using standard R = 1, 2, 3 and verified R = 3 at depths 9 and 10, for 138 Neishtadt test positions

And for 869 positions from the Encyclopedia of Chess Middlegames (ECM) [8] with depth to 11 plies, the total node counts at depths 9, 10, and 11, using standard R = 2, R = 3, and verified R = 3 was shown as below:

Depth	Std $R = 2$	Std $R = 3$	Vrfd $R = 3$
9	5,374,275,763 (+10.84%)	2,483,951,601 (-48.76%)	4,848,596,820 -
10	16,952,333,579 (+17.40%)	7,920,812,800 (-45.14%)	14,439,185,304 -
11	105,488,197,524 (+106.51%)	24,644,668,194 (-51.75%)	51,080,338,048 -

Figure 9: Total node count of standard R = 1, 2, 3 and verified R = 3 at depths 9 and 10, for 138 Neishtadt test positions

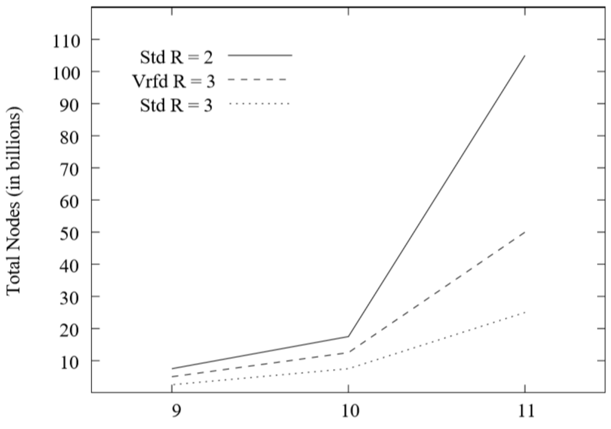


Figure 10: Three sizes of standard R=2, R=3, and verified R=3 at depths 9, 10, and 11, for 869 ECM test positions

These experiments have shown that Verified Null-Move pruning with a depth reduction of R = 3 constructed a search tree which is closer in size to that of the tree constructed by standard R = 3, and that the saving from the reduced search effort in comparison with standard R = 2 became greater as it searched more deeply.

4 Principal Variation Search

In PVS algorithm, it is guaranteed that forward pruning(like Multi-Cut and Null Move) is safe at ALL nodes. Especially, Multi-Cut at ALL nodes (MC-A) when combined with other forward-pruning mechanisms give a significant reduction of the number of nodes searched.

Principal Variation Search is built on the classification of 3 types of nodes: Principle Variation node, CUT node, and All node.

4.1 Three Node Types

Three types of nodes can be identified out of the $\alpha\beta$ Minmax tree. [9] And from former work, they were named as Principal Variation node, CUT node, and ALL node [10].

4.1.1 Principle Variation node

The fist source of Principle Variation node is the root of the tree. The second source is the best move found at a PV node, which is also called successor PV node. At the PV node, all its children have to be explored and the best move must be considered first. For the Returned score s and interval $[a, b]$, there should be $a < s < b$;

4.1.2 CUT node

CUT nodes, which are also called fail-high[1] nodes, includes all the other investigated children (other than the best move) at a PV node and successors of an All node. At Cut nodes, only the first child has to be explored in a perfect ordered tree. There exists $s \geq b$. 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

4.1.3 All node

All nodes are also called fail-low[2] nodes which are successors of a CUT node. All its children have to be explored (no move will cause a beta-cutoff).

4.1.4 Transformation

From expected nodes to truly classified nodes, there are rules for transformation:

[1]Fail high: The score returned is higher than the upper bound on the exact score of the node, which is usually noted as beta
[2]Fail low: The score returned is lower than the lower bound on the exact score of the node, which is usually noted as alpha

1. Expected CUT node would be transferred to ALL node if none of the moves causes a cutoff at this Expected CUT node;

2. Expected ALL node would be transferred to CUT node if one of the children turns out not to be a CUT node)

Therefore it can form a new principal variation: 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 path have become in fact PV nodes)

4.2 Principal Variation Search

The NWS routine is an integral part of the Principal Variation Search algorithm. As stated in an example algorithm in figure 11, the main driver, explores the expected PV-nodes, while the NWS part visits the expected CUT- and ALL-nodes, using the lower-bound established in PVS to reduce its search. The true type of a node is not known until after it has been searched. For each node, before its true type has been known after it had been searched, it is referred to refer an expected PV-, CUT- or ALL-node(depending on the current view of the structure of the game-tree).

```
function $PVS(n, d, \alpha, \beta)$
$S \leftarrow Successors(n)$
if $d \leq 0 \vee S \equiv \emptyset$ then
    return $f(n)$
$best \leftarrow -PVS(n_1 \in S, d-1, -\beta, -\alpha)$
for $n_i \in S \mid i > 1$ do
    if $best \geq \beta$ then
        return $best$
    $\alpha \leftarrow max(\alpha, best)$
    $v \leftarrow -NWS(n_i, d-1, -\alpha)$
    if $v > \alpha \wedge v < \beta$ then
        $v \leftarrow -PVS(n_i, d-1, -\beta, -v)$
    if $v > best$ then
        $best \leftarrow v$
return $best$
function $NWS(n, d, \beta)$
$S \leftarrow Successors(n)$
if $d \leq 0 \vee S \equiv \emptyset$ then
    return $f(n)$
$best \leftarrow -\infty$
for all $n_i \in S$ do
    $v \leftarrow -NWS(n_i, d-1, -\beta + \epsilon)$
    if $v > best$ then
        $best \leftarrow v$
        if $best \geq \beta$ then
            return $best$
return $best$
```

Figure 11: One example algorithm for PVS

The PVS follows following steps to go from expected nodes to truly-classified nodes:

1. The first node explored at the root (store the value there as the best value);
2. Search all its siblings using the NWS to ensure that they are inferior;
3. If one of its siblings returns a better value, then go to that node to establish the new principal variation.

This algorithm is in general more efficient than the original $\alpha\beta$. The algorithm considers the first node explored at the root (and at subsequent PV nodes) to be a PV node. The value of that node is therefore treated as best value, and all the siblings are searched using a closed $\alpha\beta$-window (i.e., $\beta = \alpha$ + 1) to prove that they are inferior. (called zero-width-window search or Null-Window Search (NWS)) . In the NWS, nodes are assumed not to be on the principal variation and therefore are expected to be alternately CUT and ALL nodes. If the NWS returns a score less than or equal to α, then that particular sibling has been proved inferior. if the NWS returns a better score then a re-search has to be done(the $\alpha\beta$-window is then opened and the child node under consideration is regarded as a PV node). In another word, forward-pruning methods are only used in the NWS part of the PVS framework.

When calling NWS recursively, the β bound is adjusted by an amount equal to ϵ, the smallest granularity of the value returned by the estimate function(if the value of function at node n returns integer values, ϵ would be set equal to 1.).

4.2.1 Assumptions of Principal Variation Search

The assumption in Principal Variation Search is that when all the alternate variations are searched with a zero window, all non-principal nodes will fail-low in any case. Should one of the moves not fail this way then it becomes the start of a new principal variation and the search is repeated for this move with a window which covers the correct range of possible values. [11]

4.2.2 Determine the expected type of a node

Expected node types are determined by tree topology, probing the transposition table, or comparing scores of a static evaluation; considering threats, or even a reduced search or quiescence search, with the bounds, may be considered by various (parallel) search algorithms and in decisions concerning selectivity.

To decide expected nodes:

1. The root of the game-tree is a pv-node.
2. At a PV-node, n, at least one child must have a minimax value v_{mm}, that child is a PV-node, but the remaining children are CUT-nodes.
3. At a CUT-node, a child node n with a minimax value $v_{mm}(n) < v_{mm}(n_{pv})$ is an ALL-node, where n_{pv} is the most immediate PV-node predecessor of n. At least one child must have such a value.
4. Every child node of an ALL-node is a CUT-node.

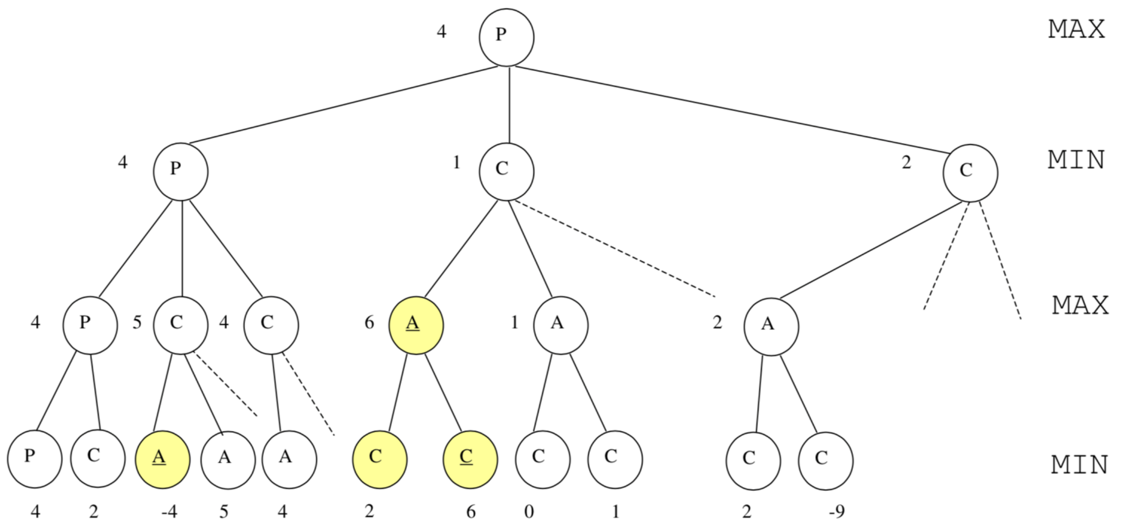


Figure 12: Pricipal Variation Search

4.2.3 Determine the true type of a node

Apply Null Window Searches for none PV-nodes to prove a move is worse or not than an already safe score from the principal variation.

To decide true nodes, None-Window search(closed $\alpha\beta$-window / NW: $\beta = \alpha + 1$ (J.P. Fishburn, 1981, 1984)) is used with score s being returned to test if they were correctly assigned as PV-nodes and none PV-nodes:

1. if $s \leq \alpha$, then this particular sibling has been proved inferior.
2. if $s > \alpha$, re-search ($\alpha\beta$-window is opened and should apply full window search at the child node then PV node.)

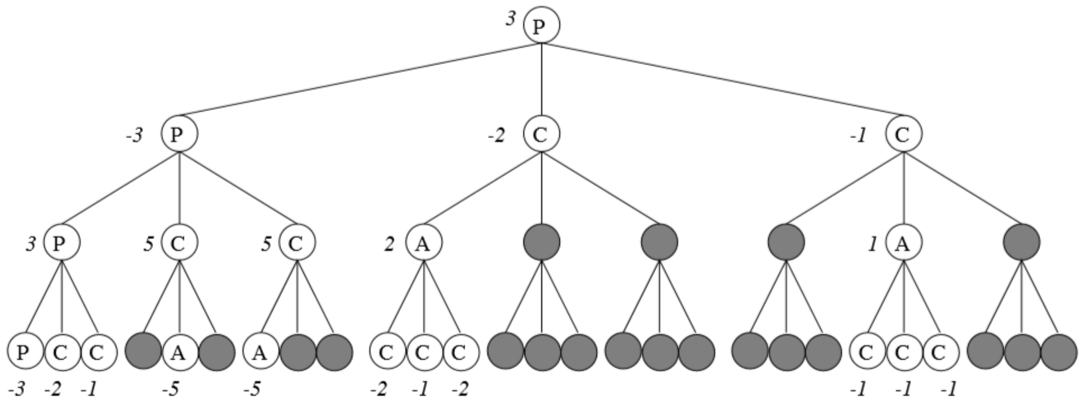


Figure 13: Possible Minimal Tree

Thus, torward-pruning is only applied for the NWS part. If erroneously pruned forward at some CUT node and the resulting score is backed up all the way to a PV node, it will obtain a fail low (the subtree seemingly has been proved inferior), forward pruning at a CUT node should also has further provisions is dangerous. If erroneously forward pruned at some ALL node and the resulting score is backed up all the way to a PV node, it will obtain a fail high and a re-search will be performed. The algorithm is then searching for a new principal variation and regards the subsequent nodes as PV nodes.

Here are outcomes of Forward pruning by mistake for different kinds of nodes:

1. at an expected PV node: too risky(a possible mistake causes an immediate change of the principal variation);

2. at an expected CUT node: fail low mistake(a possible new principal variation is overlooked in this position);

3. at an expected ALL node: fail high mistake.

Then in practice, no forward pruning is done at PV nodes. It is possible to find out whether there exists a new PV or not. In this case the result of the mistake will be that an extra amount of nodes has been searched. Forward pruning at an expected ALL node is not dangerous but can trigger unnecessary re-searches.

4.2.4 PVS Algorithm Modifications

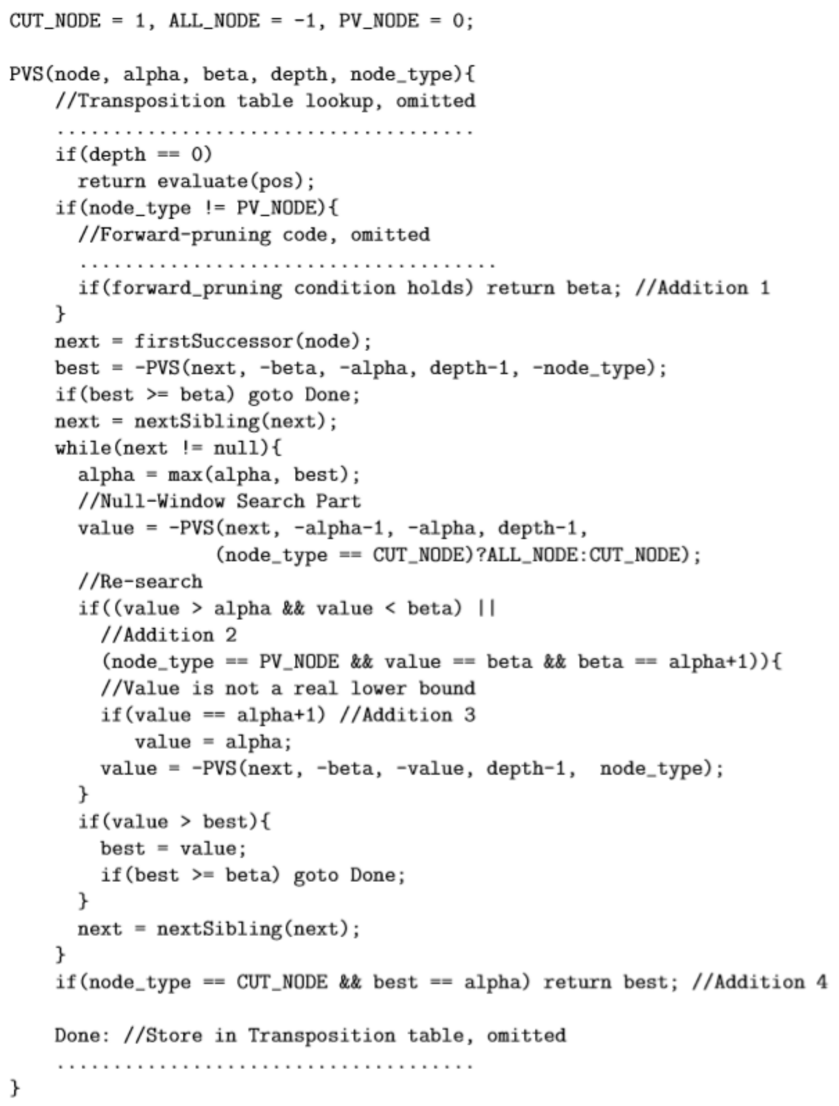
To ensure that a re-search is performed which is able to correct the value, some modification can be applied in PVS scheme:

1. To avoid that a backed-up value of a forward-pruned ALL node causes a beta-cutoff at the PV node lying above, the value of beta is returned in case of a cut-off (beta = alpha + 1 at an ALL node).

2. If the window of the PV node was already closed(with alpha, beta being updated) and the NWS should return a value of beta (alpha + 1), a re-search is still have to be done

3. If a re-search is done and the returned value of the NWS equals alpha + 1, we should do a re-search with alpha as lower bound.

4. CUT nodes where a fail-low has occurred with a value equal to Îś are not stored in the transposition table because their values are uncertain.



```
CUT_NODE = 1, ALL_NODE = -1, PV_NODE = 0;

PVS(node, alpha, beta, depth, node_type){
    //Transposition table lookup, omitted
    ....................................
    if(depth == 0)
      return evaluate(pos);
    if(node_type != PV_NODE){
      //Forward-pruning code, omitted
      ....................................
      if(forward_pruning condition holds) return beta; //Addition 1
    }
    next = firstSuccessor(node);
    best = -PVS(next, -beta, -alpha, depth-1, -node_type);
    if(best >= beta) goto Done;
    next = nextSibling(next);
    while(next != null){
      alpha = max(alpha, best);
      //Null-Window Search Part
      value = -PVS(next, -alpha-1, -alpha, depth-1,
                  (node_type == CUT_NODE)?ALL_NODE:CUT_NODE);
      //Re-search
      if((value > alpha && value < beta) ||
        //Addition 2
        (node_type == PV_NODE && value == beta && beta == alpha+1)){
        //Value is not a real lower bound
        if(value == alpha+1) //Addition 3
           value = alpha;
        value = -PVS(next, -beta, -value, depth-1,  node_type);
      }
      if(value > best){
        best = value;
        if(best >= beta) goto Done;
      }
      next = nextSibling(next);
    }
    if(node_type == CUT_NODE && best == alpha) return best; //Addition 4

    Done: //Store in Transposition table, omitted
    ....................................
}
```

Figure 14: Modified PVS Algorithm

5 Multi-cut

Multi-cut pruning is a new forward-pruning method which can make a pruning before examining an expected non-PV node to full depth. At first, it was only applied at CUT nodes. Afterwards, it was proved in experiments that it also worked well at ALL nodes.

5.1 Multi-Cut Prune

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

5.2 Multi-Cut Only at CUT Nodes

In new principal variation constructing, every expected CUT-node on the path from a leaf-node to the root must become an ALL-node. Meanwhile, if the first move does not cause a cutoff, 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 then such lines are unlikely to become part of a new principal variation.

Hereby the first M child nodes of an expected CUT node are searched to a depth reduced with a factor R before examining it to full depth:

1) If at least C child nodes return a value larger than or equal to β : cutoff;

2) Otherwise : re-exploring this node to a full depth d. [12]

In Multi-Cut $\alpha\beta$-search, an MC-prune is first only tested at expected CUT-nodes with levels of the search tree far from the horizon (to reduce the time overhead involved).

1. First, suppose that it starts with a node n, instead searching the subtree $_1$ of it to a full depth d(as it in the normal $\alpha\beta$-search), the first m successors of node n are expanded to a reduced depth of $d - r$;

2.1) If c of them return a value greater or equal to β, an MC-prune occurs and the search returns the β value;

2.2) Otherwise the search continues as usual exploring n_1 to a full depth d.

If the MC-prune is successful, there would be savings as it presented with the dotted area of the subtree below in Figure 15 as node n_1 represents.

By searching the subtree of n_1 to a shallower depth, there is some risk of overlooking a tactic that would result in n_1 becoming the new principal variation. And the risk is taken because there can be at least one of the c moves that returns a value greater or equal to β when searched to a reduced depth, and it would cause a genuine β-cutoff if searched to a full depth.

In practice, the pruning is disabled when the endgame is reached. since there are usually few viable move options there and the MC-searches are therefore not likely to be successful. Also, the positional understanding of chess programs in the endgame is generally poorer than in the earlier phases of the game.

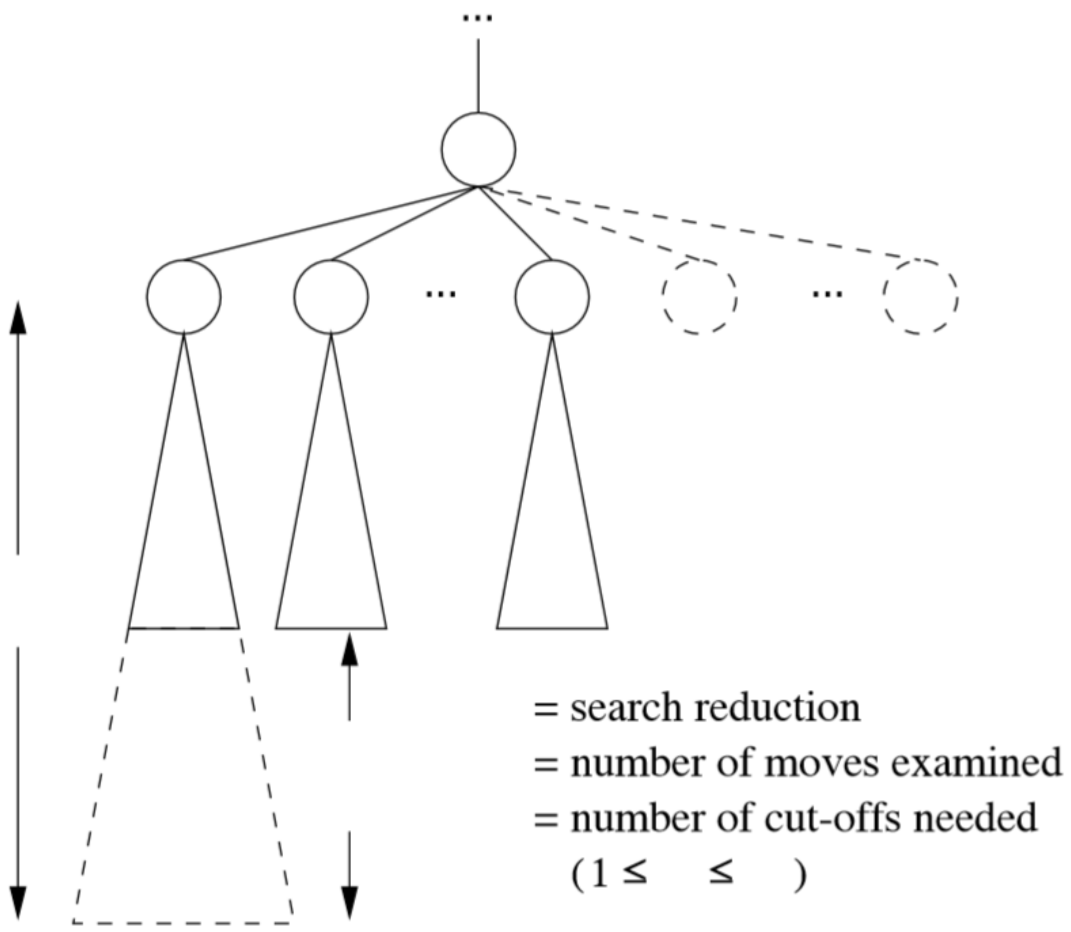


Figure 15: Multi-Cut

An example algorithm using Multi-cut can be illustrated as below Figure 16 where the parameter d is the remaining length of search for the position, and β is an upper-bound on the value we can achieve (where α is always equal to $\beta - \epsilon$).

The parameters M, R , and C are MC-prune specific and stand for: number of moves to look at (m), search reduction (r), number of cutoffs needed (c), respectively. Although they are shown here as constants, they could be determined dynamically and be allowed to vary during the search.

How they are set will affect both the efficiency and the error rate of the search:

1. Number of cutoffs (c): The larger the number C, the safer the method is. But at the same time it should be kept small enough to offer substantial node savings;

2. Number of moves (m): the higher M, the more likely it is that the pruning condition will be met, the more expensive for each unsuccessful MC-prune search, offsetting some of the node savings from the additional pruning. The better the scheme, the closer we can set $M$to C;

3. Depth reduction (r): influences the best settings for C and M; the larger R is, the larger C and M can be. too small R would cause the lack of flexibility in choosing values for C and M. Too aggressive R will make the search more error-prone.

```
Require:
    $M$ is the number of moves to look at when checking for $mc$-prune.
    $C$ is the number of cutoffs to cause an $mc$-prune.
    $R$ is the search depth reduction for $mc$-prune searches.
1: $S \leftarrow Successors(n)$
2: if $d \leq 0 \vee S \equiv \emptyset$ then
3:     return $f(n)$
4: if $d \geq R \wedge cut$ then
5:     $c \leftarrow 0$
6:     for $n_i \in S \mid i = 1, ..., M$ do
7:         $v \leftarrow -NWS(n_i, d - 1 - R, -\beta + \epsilon, \neg cut)$
8:         if $v \geq \beta$ then
9:             $c \leftarrow c + 1$
10:            if $c = C$ then
11:                return $\beta$
12: $best \leftarrow -\infty$
13: for all $n_i \in S$ do
14:     $v \leftarrow -NWS(n_i, d - 1, -\beta + \epsilon, \neg cut)$
15:     if $v > best$ then
16:         $best \leftarrow v$
17:         if $best \geq \beta$ then
18:             return $best$
19: return $best$
```

Figure 16: Multi-Cut Algorithm

The new parameter, CUT, is true if the node we are currently visiting is an expected CUT-node, but is otherwise false. In a Null-Window search we are dealing only with alternating layers of CUT and ALL-nodes.

To sum up, the routine follows several steps:

1. Starts by checking whether the horizon has been reached, and if so, evaluates the position and returns its value.

2. Otherwise,if using a fully enhanced search routine, useful information about the position in the transposition table would be the next to look for, followed by a Null-Move search. If the Null-Move does not cause a cutoff, before apply a standard Null-Window $\alpha\beta$ search (lines 12-18). Instead, a Multi-Cut search (lines 4-11) would be inserted here to see if the MC-prune condition applies.

The potentially useless subtrees can be detected by studying number of good move alternatives a player has at CUT-nodes. Multi-Cut $\alpha\beta$-pruning makes its pruning decisions based on the number of promising moves at CUT-nodes. Instead of finding any such refutation like it in the minmax sense, shallow searches was used to identify moves that "seem" good. If there are several such moves, Multi-Cut pruning assumes that a cutoff will occur and so prevents the current line of play from being expanded more deeply.

Because many children of Cut-nodes may qualify as belonging to a Minimal Tree and the one is selected arbitrarily, there may exist more than one Minimal Subtree in any game-tree.

5.3 Improvements in Multi-Cut: MC-A

First, when at a reduced depth a winning value is found, the search is stopped and the winning value is returned. Second, if the Multi-Cut does not succeed in causing a cutoff, the moves causing a beta-cutoff at the reduced depth are tried first in the normal search. The remaining question is whether Multi-Cut is also useful at ALL nodes. The inventors of the Multi-Cut algorithm anticipated that it would not be successful elsewhere [6] and therefore did not test it at expected ALL nodes. Henceforth, we call Multi-Cut at expected CUT nodes MC-C and at expected ALL nodes MC-A. In the past, some experiments were reported on testing whether MC-A is beneficial.

5.4 MC-A Experimental Results

With search engine MIA in the game Lines of Action (LOA), there have been experiments testing MC-A under different parameter settings. Also, different combinations of forward-pruning methods were investigated. Variable Null-Move bound was tested and compared to MC-A. And the increase in playing strength was tested.[12]

5.4.1 Game of Lines of Action (LOA)

Lines of Action (LOA) is a two-person zero-sum chess-like connection game with perfect information. It has an 8×8 board by two sides and 12 pieces at its disposal. It starts with Black and a move takes place as many squares as there are pieces of either colour anywhere along the line of movement. A player may jump over its own pieces, not the opponent's. One player can capture pieces by landing on them. The goal for both players is to be the first to create a configuration in which all own pieces are connected in one unit.

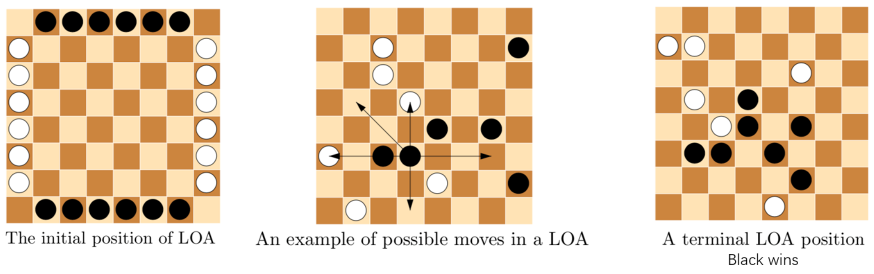

The initial position of LOA
An example of possible moves in a LOA
A terminal LOA position
Black wins

Figure 17: Game of Lines of Action (LOA)

5.4.2 Search engine MIA (Maastricht In Action)

The program has been written in MIA performs an $\alpha\beta$ depth-first iterative-deepening search in the PVS framework. [3]

To prune a subtree or to narrow the $\alpha\beta$ window, a two-deep transposition table was applied. At all interior nodes which are more than 2 ply away from the leaves, all the moves were generated and Enhanced Transposition Cutoffs (ETC) scheme was applied. Also, a null move is applied with a reduced depth R before any other move, at CUT nodes(adaptive null move is used to set R: If the remaining depth is more than 6, R is set to 3. When the number of pieces of the side to move is lower than 5 the remaining depth has to be more than 8 for setting R to 3. And for all other cases R is set to 2.) and ALL nodes(R = 3 is used).

[3]The program and the test sets can be found at the website: http://www.cs.unimaas.nl/m.winands/loa/

MC-A (C, M, R) $(2, 10, 2)$ is the most efficient while $C = 1$ was too aggressive and it would cause many re-searches, and $C = 4$ is too conservative that it caused not much pruning.

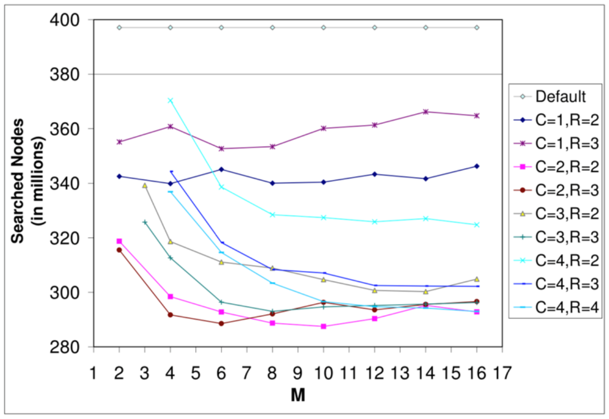


Figure 18: Three sizes for different C, M, and R

For move ordering, the move stored in the transposition table, if applicable, is always tried first.

With above optimal parameter setting, the results the performance of MC-A in combination with the already included pruning methods (i.e., MC-C and null move) can be seen as the following:

	No Forward pruning			Only Null move		
d	No MC-A	MC-A	%	No MC-A	MC-A	%
5	5,071,689	4,995,845	98.5	3,504,759	3,404,882	97.2
6	19,896,101	19,286,868	96.9	10,109,533	9,518,082	94.1
7	113,653,808	110,663,056	97.4	36,265,257	34,671,647	95.6
8	416,549,038	406,489,302	97.6	92,749,650	89,483,140	96.5
9	2,427,406,280	2,395,844,102	98.7	314,507,126	303,466,596	96.5
10	9,635,185,102	9,460,591,510	98.2	891,348,022	813,032,326	91.2
11	-	-	-	2,930,142,106	2,599,157,486	88.7
12	-	-	-	8,362,297,395	7,080,475,905	84.7
	Only MC-C			Null move and MC-C		
d	No MC-A	MC-A	%	No MC-A	MC-A	%
5	2,097,908	1,955,564	93.2	2,012,835	1,897,600	94.3
6	7,314,731	5,549,772	75.9	6,083,136	5,122,496	84.2
7	28,656,432	20,221,202	70.6	20,491,711	17,423,109	85.0
8	85,103,638	50,333,688	59.1	50,018,470	42,242,144	84.5
9	297,239,554	149,671,128	50.4	142,182,834	116,784,068	82.1
10	1,286,515,396	393,490,307	30.6	397,092,800	283,350,391	71.4
11	4,860,474,957	1,358,658,246	28.0	1,223,918,717	846,066,886	69.1
12	23,806,355,059	3,536,842,482	14.9	3,328,838,963	2,162,692,924	65.0
13	-	-	-	9,869,101,893	6,289,563,990	63.7
14	-	-	-	30,087,791,323	17,578,589,423	58.4

Figure 19: Value of MC-A added

Apparently, when no forward pruning is used at all, Multi-Cut is useless at expected ALL nodes. In the case of using only null move adding MC-A gives a further reduction increasing to 15 per cent at depth 12. If only MC-C was used as forward-pruning method, introducing MC-A to the search brings a further reduction increasing to 85 per cent at depth 12 (but null-move pruning was already performed at ALL nodes). When null move and MC-C are available in the engine (the original situation), MC-A reduces the search tree with another 42 per cent at depth 14.

Depth	Original Set (171 positions)	Validation Set (156 positions)
5	94.3	94.7
6	84.2	86.8
7	85.0	83.7
8	84.5	82.0
9	82.1	79.8
10	71.4	73.9
11	69.1	72.0
12	65.0	68.5
13	63.7	64.8
14	58.4	61.4

Figure 20: Relative Performance of MC-A in combination with null move and MC-C

In the first column of Figure 20 it can be seen that the relative performance of MC-A on the the original test set. In the second column the relative performance of MC-A is given for the validation set. There is not much difference between them and similar results are achieved.

5.4.3 Variable Null-Move Bound

A Null-Move cutoff can be forced if the returned null-move search value is larger than or equal to $\beta - t$, where t is the minimal value of a tempo depending on the evaluation function. It allows a larger part of the null-move searches to cause cut-offs

To compare variable null-move bound (VNMB) with MC-A, experiments about whether VNMB at expected ALL nodes gives a reduction of nodes were investigated (the search was not enhanced with MC-A.):

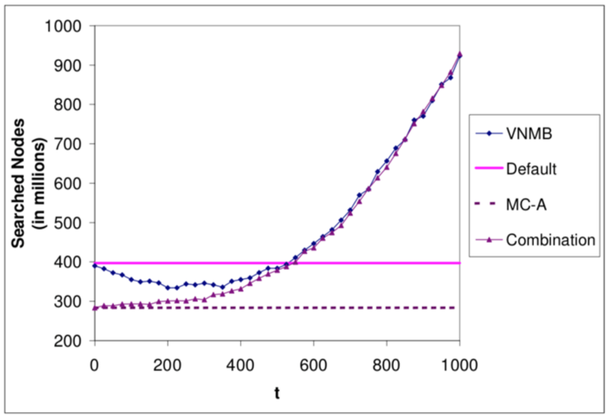


Figure 21: Variable null-move bound

The combination of MC-A and variable null-move was tested with different t:

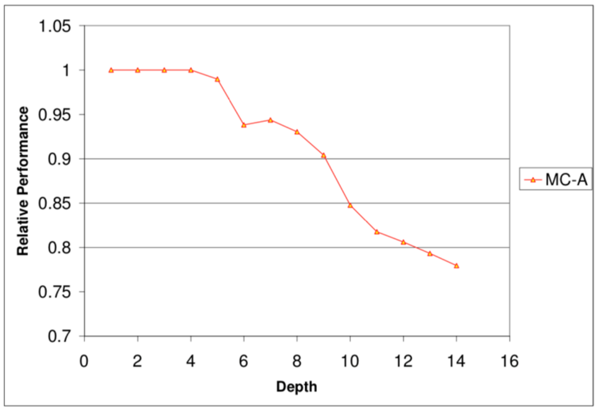


Figure 22: MC-A compared to variable null-move bound with optimal parameter t = 200

And there was no setting of t for which the combination was better than the MC-A enhanced search. When t became sufficiently large enough the combination behaved the same as the Variable Null-Move.

Experiments suggested that MC-A is not surpassed by aggressive Null-Move pruning at expected ALL nodes:

Two versions of MIA were matched against each other, one using MC-A and the other without MC-A. These versions used all the enhancements described in subsection 5.2. Both programs played 1000 games, switching sides halfway. The thinking time was limited to 60 seconds per move, simulating tournament conditions. The results:

	Score	Winning ratio
MC-A vs. Default	549-451	1.21

Figure 23: 1000-game match results

The modified version outplayed the original version with a winning ratio of 1.21 (i.e., scoring 21% more winning points than the opponent). This result revealed that MC-A improved the playing strength of MIA significantly. MC-A is a significant enhancement in a PVS search where forward pruning is used.

Multi-Cut at expected ALL nodes gives a safe reduction of approximately 40 percent of the number of nodes searched in combination with null move and the regular Multi-Cut, MC-C. Experiments suggested that parameters more aggressively chosen than MC-C lead to an additional improvement. It has been observed that MC-A still searches 22 percent less nodes than Variable Null Move Bound at expected ALL nodes. Moreover, MC-A was able to increase significantly the playing strength of the program MIA.

6 Conclusion

In Chess Programming, pruning can hugely save it from unnecessary searching. Different forward pruning strategies discussed above uses different assumptions for pruning, while verification process is also applied as an important part to decrease tactical risk. And with varying search depth, depth-first algorithms can be selective in practice. Although the search occasionally goes wrong, the time saved by pruning non-promising lines is generally better used to search other lines deeper and therefore, hopefully, to increase the overall decision quality.

Also, these algorithms become more careful when it comes to the end since it has been seen in Null-Move pruning and Multi-Cut pruning that forward-pruning scheme is more likely to be harmful at the ending.

Based on experimental outcome, two principal refinements has been found as Narrow Window Aspiration search and the use of memory tables. [4] Minimal Window search employs a full window only on the candidate principal variation. And a zero window searches normally cut off quite quickly. If this is not the case, then a profitable heuristic is to curtail the search and repeat immediately with the appropriate window. But in PVS the correct score for a candidate principal variation is not needed until a potential rival arises. For failing searches the bound returned may be better than the alpha limit. This means that the re-search of a new principal variation normally proceeds with a narrower window.

When Minimal Window search is used, it is usual to re-visit a position. In this case the performance can be improved by implementation of transposition and refutation memory tables. When a position is reached again, the corresponding table entry:

1. It may be possible to use the table score to narrow the (alpha, beta) window bounds;

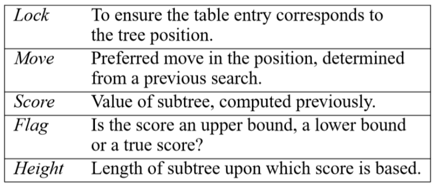
2. The best move that was found before can be tried immediately(It had probably caused a cut-off and may do so again: Here the table entry is being used as a move re-ordering mechanism);

3. Enable recognition of move transpositions(exact forward pruning) that have lead to a position (subtree) that has already been completely examined. In such a case there is no need to search again.

It was found that preservation and use of the refutations from a previous iteration was more important than aspiration searching, where a full window search with refutation table support is superior to a Narrow Window Aspiration search without a memory table.[6]

[4]1. Refutation table is a sequence of moves which is sufficient to cut off the search. The candidate principal variation is fully searched, but for the alternate variations the moves in the refutation table may again be sufficient to cut off the search, thus save the move generation that would normally occur at each node. The storage overhead is very small, although a small triangular table is also needed to identify the refutations. [13]

2. Transposition table holds more than refutations and the main subvariations: once a subtree has been searched, its transposition table entry will contain not only the length of the search tree and the value of the subtree, but also whether that value represents the true score or an upper/lower bound on the score. [14]



Lock	To ensure the table entry corresponds to the tree position.
Move	Preferred move in the position, determined from a previous search.
Score	Value of subtree, computed previously.
Flag	Is the score an upper bound, a lower bound or a true score?
Height	Length of subtree upon which score is based.

Figure 24: Typical Transposition Table Entry

References

[1] T. A. Marsland. A review of game-tree pruning, 2013.

[2] James J Gillogly. The technology chess program. *Artificial Intelligence*, 3:145 – 163, 1972.

[3] Allen Frederick Newell, J. C. Shaw, and Herbert Alexander Simon. Chess-playing programs and the problem of complexity. *IBM Journal of Research and Development*, 2:320–335, 1958.

[4] A L. Brudno. Bounds and valuations for abridging the search of estimates. *Problems of Cybernetics*, 10, 01 1963.

[5] Yngvi Björnsson and Tony Marsland. Multi-cut pruning in alpha-beta search. In H. Jaap van den Herik and Hiroyuki Iida, editors, *Computers and Games*, pages 15–24, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[6] T. A. Marsland. Relative efficiency of alpha-beta implementations. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'83, pages 763–766, San Francisco, CA, USA, 1983. Morgan Kaufmann Publishers Inc.

[7] Yakov Neishtadt. Test your tactical ability. *Batsford, ISBN 0-7134-4013-9*, pp:110–135, 1993.

[8] Chess Informator. *Encyclopedia of chess middlegame*. Times Mirror, Toronto, 1985.

[9] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6:293–326, 1975.

[10] Ieee transactions on pattern analysis and machine intelligence.

[11] John P. Fishburn. *Analysis of Speedup in Distributed Algorithms*. University Microfilms International (UMI), Ann Arbor, MI, USA, 1984.

[12] Mark H. M. Winands, H. Jaap van den Herik, Jos W. H. M. Uiterwijk, and Erik C. D. van der Werf. Enhanced forward pruning. *Inf. Sci.*, 175(4):315–329, November 2005.

[13] Selim G. Akl and Monroe M. Newborn. The principal continuation and the killer heuristic. In *Proceedings of the 1977 Annual Conference*, ACM '77, pages 466–473, New York, NY, USA, 1977. ACM.

[14] T. A. Marsland and M. Campbell. Parallel search of strongly ordered game trees. *ACM Comput. Surv.*, 14(4):533–551, December 1982.