: Accepted (March 1999) for publication in Theoretical Computer Science

Multi-Cut $\alpha\beta$-Pruning in Game-Tree Search

Yngvi Björnsson and Tony Marsland

University of Alberta, Department of Computing Science,
Edmonton AB, Canada T6G 2H1
{yngvi,tony}@cs.ualberta.ca

Abstract. The efficiency of the $\alpha\beta$-algorithm as a minimax search procedure can be attributed to its effective pruning at so-called cut-nodes; ideally only one move is examined there to establish the minimax value. This paper explores the benefits of investing additional search effort at cut-nodes by also expanding some of the remaining moves. Our results show a strong correlation between the number of promising move alternatives at cut-nodes and a new principal variation emerging. Furthermore, a new forward-pruning method is introduced that uses this additional information to ignore potentially futile subtrees. We also provide experimental results with the new pruning method in the domain of chess.

1 Introduction

The $\alpha\beta$-algorithm is the most popular method for searching game-trees in such adversary board games as chess, checkers and Othello. It is much more efficient than a plain brute-force minimax search because it allows a large portion of the game-tree to be pruned, while still backing up the correct game-tree value. However, the number of nodes visited by the algorithm still increases exponentially with increasing search depth. This obviously limits the scope of the search, since game-playing programs must meet external time-constraints: often having only a few minutes to make a decision. In general, the quality of play improves the further the program looks ahead[1].

Over the years, the $\alpha\beta$-algorithm has been enhanced in various ways and more efficient variants have been introduced. For example, although the basic algorithm explores all continuations to some fixed depth, in practice it is no longer used that way. Instead, various heuristics allow variations in the distance to the search horizon (often called the search depth or search tree height), so that some move sequences can be explored more deeply than others. "Interesting" continuations are expanded beyond the nominal depth, while others are terminated prematurely. The latter case is referred to as *forward-pruning*, and involves some risk of overlooking a good continuation. The rationale behind the

[1] Some artificial games have been constructed where the opposite is true; when backing up a minimax value the decision quality actually decreases with increasing search depth. This phenomenon has been studied thoroughly and is referred to as *pathology* in game-tree search [10]. However, such pathology is not seen in chess or the other games we are investigating.

approach is that the time saved by pruning non-promising lines is better spent searching others more deeply, in an attempt to increase the overall decision quality.

To effectively apply forward-pruning, good criteria are needed to determine which subtrees to ignore. Here we show that the number of good move alternatives a player has at cut-nodes can be used to identify potentially futile subtrees. Furthermore, we introduce a new forward-pruning method, called *multi-cut* $\alpha\beta$-*pruning*, that makes its pruning decisions based on the number of promising moves at cut-nodes. In the minimax sense it is enough to find a single refutation to an inferior line of play. However, instead of finding any such refutation, our method uses shallow searches to identify moves that "look" 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.

In the following section we give a brief introduction to game-tree searching, and introduce necessary terminology and definitions. We introduce the basic idea behind our new pruning scheme and provide a sound foundation for the work. The pruning scheme itself has been implemented and tested in an actual game-playing program. Experimental results follow; first the promise of the new pruning criterion is established, and second the method is tested in the domain of chess. Finally, before drawing our conclusions, we explain how some related works use complementary ideas.

2 Game-Tree Search

We are concerned here with two-person zero-sum perfect information games. The value of a such games is the outcome with perfect play by both sides, and can be found by recursively expanding all possible continuations from each game state, until states are reached with a known outcome. The minimax rule is then used to propagate the value of those outcomes back to the initial state. The state-space expanded in this way is a tree, often referred to as a *game-tree*, where the root of the tree is the initial state and the leaf nodes are the terminal states.

2.1 Minimax

Using the minimax rule, *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. However, for zero-sum games one player's gain is the other's loss. Therefore, by evaluating the terminal nodes from the perspective of the player to move and negating the values as they back up the tree, the value at each node in the tree can be treated as the merit for the player who's turn is to move. This framework is referred to as *NegaMax* [5], and has the advantage of being simpler and more uniform, since both sides now maximize their values. We use this framework as the basis for our subsequent discussion.

In theory, at least, the outcome of a game can be found as described above. However, the exponential growth of game-trees expanded this way is prohibitively time-expensive. Therefore, in practice, game-trees are only expanded to a limited depth d and the resulting "leaf nodes" are assessed. Their values are propagated back up the tree using the minimax rule, just as if they were true game-outcome values. The rule for backing up these values can be defined as follows:

Definition 1 ($v_{mm}(n,d)$)**.** *The minimax value, $v_{mm}(n,d)$, of a game-tree expanded to a fixed depth d is:*

$$v_{mm}(n,d) = \begin{cases} f(n) & \text{if } S_n \equiv \emptyset \text{ or } d = 0; \\ Max_i(-v_{mm}(n_i, d-1)) \quad n_i \in S_n & \text{otherwise.} \end{cases}$$

where $f(n)$ is a scalar function that returns an estimate of the true value of the game position corresponding to node n (relative to the side to move at that point), and S_n is the set of all children (successors) of node n.

Note, the true value of these leaf nodes is normally not known, since the function $f(n)$ can usually only estimate the outcome. Typically, the estimate is a number that measures the "goodness" of the state[2]. The exact meaning of the estimate is not that important; the purpose is to provide a ranking of the leaf nodes. The higher the value, the more likely the state is to lead to a win. Note, too, that as $d \to \infty$ the method reduces to a pure unbounded minimax search.

Algorithm 1 shows a function for calculating the minimax value of a depth-limited game-tree. The function, $MM(n,d)$, implements Definition 1.

Algorithm 1 $MM(n,d)$

```
1: S ← Successors(n)
2: if d ≤ 0 ∨ S ≡ ∅ then
3:    return f(n)
4: best ← −∞
5: for all n_i ∈ S do
6:    v ← −MM(n_i, d − 1)
7:    if v > best then
8:       best ← v
9: return best
```

2.2 The Minimal Tree and Alpha-Beta

It is not essential to investigate all branches of a game-tree to find its minimax value; only a so-called *minimal tree* needs to be expanded. The minimax value

[2] Sometimes terminal game positions are reachable within the search horizon, and the estimates are exact in such cases. In chess, for example, (stale)mate are terminal states with a known outcome.

depends only on the nodes in the minimal tree; no matter how the other nodes are assessed, the value at the root does not change.

The minimal tree contains three types of nodes: pv-, cut- and all-nodes[3]. More formally, we can determine the minimal tree as follows:

Definition 2 (Minimal tree). *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. Given any game-tree, we can derive a minimal tree by identifying its nodes as follows:*

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 $-v_{mm}(n)$ (when there are several such children pick one arbitrarily). 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; when there are several, pick one arbitrarily.*
4. *Every child node of an all-node is a cut-node.*

From the above definition it is clear that there may exist more than one minimal subtree in any game-tree, because many children of cut-nodes may qualify as belonging to a minimal tree. Figure 1 shows an example game-tree of uniform width and a fixed depth of 3. The non-shaded nodes represent one possible minimal tree. The letters P, C, and A represent pv-, cut- and all-nodes, respectively.

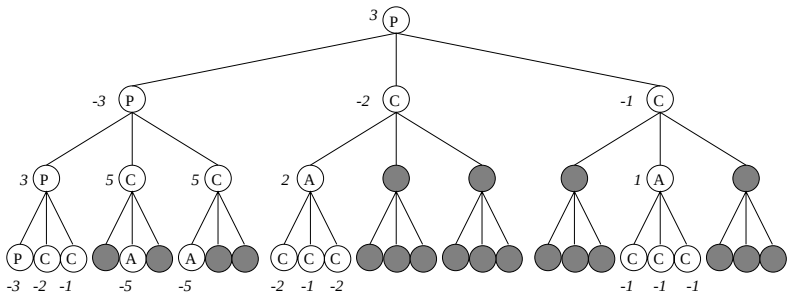

Fig. 1. Minimal tree

The $\alpha\beta$-algorithm, thoroughly analyzed by Knuth and Moore [5], is based on the observation that the minimax value can be found from the search of any minimal subtree. Once we have searched one child of some node n, the value

[3] Knuth and Moore [5] called these type 1, type 2 and type 3 nodes, but here the more descriptive terminology [8] of calling them pv-, cut- and all-nodes, respectively, is used.

returned from that search is a lower-bound for the actual minimax value of n. Moreover, this lower-bound is an upper-bound for our opponent, and can be used to prune the subtrees of the remaining child nodes of n (that is, identify those nodes that do not belong to the minimal tree). Intuitively, if the opponent finds one continuation that makes the value of the current subtree inferior to the lower-bound already established at n, there is no need to search further and the current node becomes a cut-node. Algorithm 2, below, shows a NegaMax formulation of the $\alpha\beta$-algorithm. It keeps track of the lower and upper bounds that a player can achieve via two parameters named α and β, respectively. The pruning condition is checked at lines 9-10. If a move returns a value greater or equal to β, the local search terminates at that particular node; this is often referred to as a β-cutoff (in the NegaMax formulation of the algorithm there is no distinction between α and β cutoffs). To ensure that the value of the tree will be found, the values of α and β are initialized to $-\infty$ and ∞, respectively.

Algorithm 2 $\alpha\beta(n, d, \alpha, \beta)$

```
 1: S ← Successors(n)
 2: if d ≤ 0 ∨ S ≡ ∅ then
 3:    return f(n)
 4: best ← −∞
 5: for all n_i ∈ S do
 6:    v ← −αβ(n_i, d − 1, −β, −max(α, best))
 7:    if v > best then
 8:       best ← v
 9:       if best ≥ β then
10:          return best
11: return best
```

2.3 Alpha-Beta Enhancements

The performance of the $\alpha\beta$-algorithm is sensitive to the order in which nodes in the tree are examined. In the worst case, it expands the same exhaustive tree as the minimax algorithm, while in the best case only a minimal tree is traversed. For the $\alpha\beta$-algorithm to achieve optimal performance the best move must be expanded first at pv-nodes, but at cut-nodes any move sufficiently good to cause a cutoff can be searched first[4]. Various heuristics are used to accomplish a good move-ordering (see for example [13, 14]) and, over the years more efficient

[4] Because of a non-uniform branching factor, local variability in the actual search depth (through search extension/reduction), and the possibility of reaching the same state via alternative paths (transpositions), the size of the many possible minimal trees varies considerably. For efficiency, we would like to search first not only a move that returns a value that is sufficient to cause a cutoff, but *also* one that leads to the smallest subtree.

variants of the $\alpha\beta$-algorithm have been developed that take full advantage of better move ordering. Algorithm 3, *Principal Variation Search* [6,7] is one such variant. PVS, 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. Therefore, during the search we refer to the node as an *expected* pv-, cut- or all-node, depending on our current view of the structure of the game-tree. The algorithm considers the first node explored at the root (and

Algorithm 3 $PVS(n, d, \alpha, \beta)$

```
 1: function PVS(n, d, α, β)
 2: S ← Successors(n)
 3: if d ≤ 0 ∨ S ≡ ∅ then
 4:    return f(n)
 5: best ← −PVS(n_1 ∈ S, d − 1, −β, −α)
 6: for n_i ∈ S |i > 1 do
 7:    if best ≥ β then
 8:       return best
 9:    α ← max(α, best)
10:    v ← −NWS(n_i, d − 1, −α)
11:    if v > α ∧ v < β then
12:       v ← −PVS(n_i, d − 1, −β, −v)
13:    if v > best then
14:       best ← v
15: return best
16: function NWS(n, d, β)
17: S ← Successors(n)
18: if d ≤ 0 ∨ S ≡ ∅ then
19:    return f(n)
20: best ← −∞
21: for all n_i ∈ S do
22:    v ← −NWS(n_i, d − 1, −β + ε)
23:    if v > best then
24:       best ← v
25:       if best ≥ β then
26:          return best
27: return best
```

at subsequent pv-nodes), to be a pv-node. The value of that node is therefore treated as the best value, and all the siblings are searched using the NWS routine to prove them inferior. Occasionally, one of the siblings returns a better value and in that case the algorithm researches that node to establish the new principal variation (lines 11–12). When calling NWS recursively (line 22) the β-bound is adjusted by an amount equal to ϵ, the smallest granularity of the value returned by the estimate function. For example, if $f(n)$ returns integer values,

ϵ would be set equal to 1. In Section 5 we show how our new forward-pruning method can be incorporated into the PVS algorithm.

2.4 Selective Search

All the algorithms described above traverse the game-tree in a depth-first manner. That is, they fully explore each branch of the tree before turning their attention to the next. They all return the same minimax value, the primary difference is the search efficiency: where the more enhanced algorithms search a smaller tree (always at least the minimal tree necessary for determining the minimax value is explored). There exists a different class of algorithms for searching game-trees. These algorithms traverse the trees in a best-first fashion, and commonly search more selectively than depth-first methods. They temporarily stop exploring branches to visit other more interesting subtrees, possibly later returning to the abandoned branches to search them more deeply. However, these best-first algorithms are generally not time- and space-efficient and so have not found a wide use in practice. For an overview of these alternative approaches see Junghanns [4].

Although, the term selective search has most often been associated with best-first search, the depth-first algorithms can also be selective in practice. The selectivity is introduced by varying the search horizon, some branches being searched beyond the nominal depth, while others are pruned prematurely. The former case is referred to as a search extension, and the second as forward-pruning. As such, the search can return a value quite unlike that from a fixed depth minimax search. In the case of forward-pruning, the full minimal tree is not explored, and good moves may be overlooked. However, the rationale is that 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. Our algorithm falls into this latter category so far as selectivity is concerned.

3 Error Propagation

A forward-pruning scheme should only curtail the search if it is unlikely that the pruned subtree contains a better continuation. But inevitably, any forward-pruning method will once in a while make a wrong decision. However, we can minimize the risk that such errors will affect our move choice at the root.

Figure 2 shows two different game-trees. The solid lines identify the parts of the tree that have already been visited, while the dotted lines show nodes that are still to be expanded. Assume that the search is currently situated at node n, and that the subtree n_1 has already been searched. Furthermore, assume that a part of that subtree has been pruned using some forward-pruning technique, and that the value returned is greater or equal to the β-bound used at node n (when n is a cut-node this is what we would expect). Therefore, a cutoff occurs and the value of the subtree n_1 will back up to the root. From the root's perspective

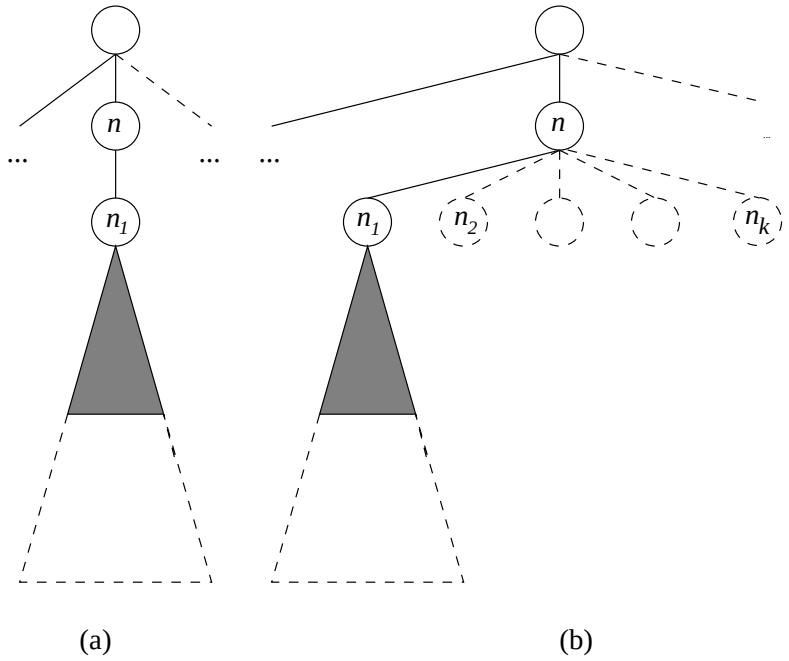

Fig. 2. Controlling error propagation

this branch is inferior to the current principal variation, and the search therefore continues to expand the other children of the root without changing the principal variation.

If the pruned subtree in Figure 2(a) does not contain a better line, search effort has been saved. The case of interest here is: *what if a better line is present?* In Figure 2(a), if a better line is overlooked, the value of n_1 is wrong and the error propagates through node n and may affect the root value. However, if alternatives to n_1 are present, as in Figure 2(b), it is possible that one of the alternative subtrees n_2, ..., n_k may return a value that causes a cutoff at n. Thus in Figure 2(b), an error in the subtree n_1 does not necessarily propagate to the root. This situation is common in practice: if the first move fails to cause a cutoff, one of the alternative moves may do so. As a consequence, even though the reduced search of n_1 is risky, the danger of affecting the move decision at the root is lower for the tree in Figure 2(b) than in Figure 2(a), because one of the other subtrees n_2 ..., n_k might preserve the cutoff even if the reduced search of n_1 does not. Thus, *even though the truncated search of n_1 is in error it will not necessarily affect the move decision at the root.* This illustrates that, when assessing risk, pruning methods should not only take into account the expected return value of a pruned node, but also assess the likelihood that an erroneous pruning decision will propagate up the tree. The idea underlying our pruning

method is partially based on this observation, and the method prunes only if it considers it unlikely that an erroneous pruning decision will affect outcomes closer to the root.

4 Multi-Cut Idea

In the traditional $\alpha\beta$-search, if a cutoff occurs there is no reason to examine that position further, and the search can return. For a new principal variation to emerge, every expected cut-node on the path from a leaf-node to the root must become an all-node. In practice, however, it is common that 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 β-cutoff, are less likely to become all-nodes, and consequently such lines are unlikely to become part of a new principal variation.* This observation forms the basis for the new forward-pruning scheme we introduce here, *multi-cut $\alpha\beta$-pruning.* Before explaining how it works, let us first define an *mc*-prune (multi-cut prune).

Definition 3 (*mc*-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.*

In multi-cut $\alpha\beta$-search, we test for an *mc*-prune only at expected cut-nodes (we would not anticipate it to be successful elsewhere). Figure 3 shows the basic idea. At node n, before searching the subtree n_1 to a full depth d, like a normal $\alpha\beta$-search does, the first m successors of n are expanded to a reduced depth of $d-r$. If c of them return a value greater or equal to β an *mc*-prune occurs and the search returns the β value, otherwise the search continues as usual exploring n_1 to a full depth d. The subtrees of depth $(d-r)$ below $n_2, ..., n_m$, represent extra search overhead introduced by *mc*-prune. This overhead would not be incurred by normal $\alpha\beta$-search. The dotted area of the subtree below node n_1 represents the savings that are possible if the *mc*-prune is successful. However, if the pruning condition is not satisfied, we are left with the overhead but no savings. Clearly, 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. We are willing to take that risk, because we expect at least one of the c moves that returns a value greater or equal to β when searched to a reduced depth, will cause a genuine β-cutoff if searched to a full depth.

5 Multi-Cut Implementation

Algorithm 4 is a pseudo-code version of null-window search (NWS) routine using multi-cut. The *NWS* routine is an integral part of the Principal Variation Search algorithm. Multi-cut could equally well be implemented in other enhanced $\alpha\beta$-variants like *NegaScout* [11]. For clarity, we have omitted details

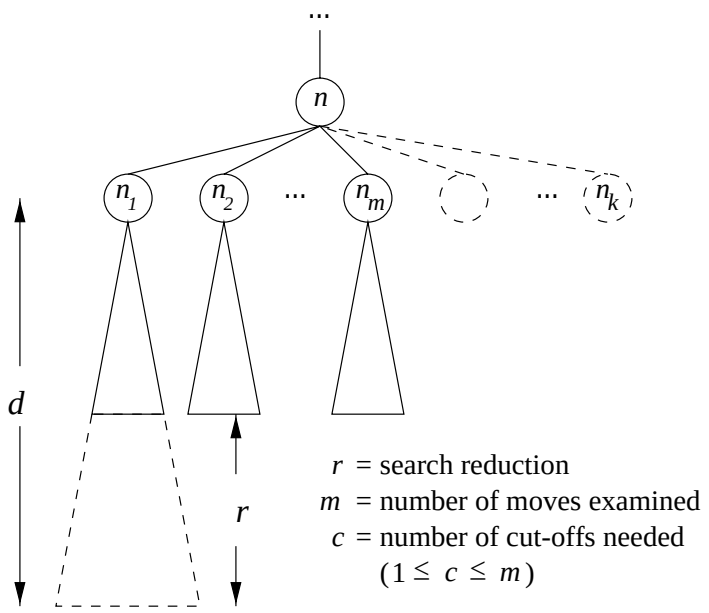

Fig. 3. Applying the *mc*-prune method at node n

about search extensions, transposition table lookups, quiescence searches, null-move searches, and history heuristic updates that are irrelevant to our discussion. For an overview of some of these techniques see for example [7, 3]. The parameter d is the remaining length of search for the position, and β is an upper-bound on the value we can achieve. There is no need to pass α as a parameter, because it is always equal to $\beta - \epsilon$. On the other hand, 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.

As is normal, the routine starts by checking whether the horizon has been reached, and if so evaluates the position and returns its value. Otherwise, if we are using a fully enhanced search routine, we would next look for useful information about the position in the transposition table, followed by a null-move search. If the null-move does not cause a cutoff, a standard null-window $\alpha\beta$ search would follow (lines 12–18). Instead, we insert here a multi-cut search (lines 4–11) to see if the *mc*-prune condition applies. 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.

We do not check for the *mc*-prune conditions at every node in the tree. First, we only test for them at expected cut-nodes. Second, they are not applied at the

Algorithm 4 $NWS(n, d, \beta, cut)$

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 ← Successors(n)
 2: if d ≤ 0 ∨ S ≡ ∅ then
 3:   return f(n)
 4: if d ≥ R ∧ cut then
 5:   c ← 0
 6:   for n_i ∈ S | i = 1, ..., M do
 7:     v ← −NWS(n_i, d − 1 − R, −β + ε, ¬cut)
 8:     if v ≥ β then
 9:       c ← c + 1
10:       if c = C then
11:         return β
12: best ← −∞
13: for all n_i ∈ S do
14:   v ← −NWS(n_i, d − 1, −β + ε, ¬cut)
15:   if v > best then
16:     best ← v
17:     if best ≥ β then
18:       return best
19: return best
```

levels of the search tree close to the horizon, thus reducing the time overhead involved in this method. Finally, there are some game-dependent restrictions that apply, but are not shown in the pseudo-code. In our experiments in the domain of chess (see later) 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. The programs rely more heavily on the search to guide them in the ending, and any forward-pruning scheme is therefore more likely to be harmful. Furthermore, the pruning is not done if the side to move is in check, or if search extensions have been applied for any of the three previous moves leading to the current position.

6 Multi-Cut Parameters

It is not clear how to select the most appropriate values for the parameters c, m, and r. How they are set will affect both the efficiency and the error rate of the search, each parameter influencing the search in its own way:

- *Number of cutoffs (c):*
  The more cutoffs that are required for an mc-prune to occur, the safer the method is. On the other hand, the higher the value is, the larger the tree

expanded. Not only does each check for *mc*-prune require more nodes to be searched, but also the less often *mc*-prunings occur. Therefore, c should be set large enough for the method to be safe, but still small enough to offer substantial node savings.

- *Number of moves (m):*
  The m parameter tells how many moves to investigate when checking for an *mc*-prune. The higher m, the more likely it is that the pruning condition will be met. However, each unsuccessful *mc*-prune search will be more expensive, offsetting some of the node savings from the additional pruning. The right balance between these two counter-acting effects will depend, among other things, on the quality of the move ordering scheme used. The better the scheme, the closer we can set m to c.

- *Depth reduction (r):*
  The depth reduction factor r will influence the best settings for c and m; the larger r is, the larger c and m can be. Obviously, if the goal is to improve search efficiency, the depth reduced multi-cut searches must explore, in total, fewer nodes than the full depth search they replace. Therefore, if r is very small there is not much flexibility in choosing values for c and m. On the other hand, too aggressive search depth reduction will make the search more error-prone.

From the above discussion we see how intertwined the parameters are, altering one will bias the selection of the others. It is impossible to analytically determine the most appropriate settings for the parameters, because not only do they depend on different characteristics of the search-space, but also on various properties of the game-playing program itself (e.g. the move-ordering scheme). We empirically determined a suitable setting of these parameters for our experiments.

7 Experimental Results

To test the idea in practice, multi-cut $\alpha\beta$-pruning was implemented in *TheTurk*[5]. Three different kinds of experiments were performed. First, we verified the feasibility of the idea by correlating the number of promising move alternatives at cut-nodes to an actual cutoff occurring. Secondly, we experimented with different multi-cut parameter settings to both give some insight into how they alter the search, and to find an appropriate setting for our program. Finally, a version of the program using multi-cut played several self-play matches against an unmodified version of the program.

[5] TheTurk is a chess program developed at University of Alberta by Yngvi Björnsson and Andreas Junghanns.

7.1 Criteria Selection

The multi-cut idea stands or falls with the hypothesis that nodes having many promising move alternatives are more likely to cause a β-cutoff than those with fewer. We will refer to any node where a β-cutoff is anticipated as an expected cut-node. Only after searching the node do we know if it actually causes a cutoff; if it does we call it a *True cut-node*, otherwise a *False cut-node*. What we seek is a scheme that accurately predicts which expected cut-nodes are False. We experimented with the following four different ways of anticipating cut nodes:

1. Number of legal moves (NM):
   The most straight-forward approach is simply to assume that every move has the same potential for causing a β-cutoff. Therefore, the more children an expected cut-node has, the more likely it is to be a True cut-code. Although this assumption is not realistic, it can serve as a baseline for comparison.
2. History heuristic ($HH > \Delta$):
   A more sensible approach is to distinguish between good and bad moves. For example, by using information from the *history-heuristic* table [13]. Moves with a positive history-heuristic value are known to be useful elsewhere in the search-tree. This method defines moves with a history-heuristic value greater than a constant Δ as potentially good. One advantage of this scheme is that no additional search is required.
3. Quiescence search ($QS() \geq \beta - \delta$):
   Here quiescence search is used to determine which children of a cut-node have a potential for causing a cutoff. If the quiescence search returns a value greater or equal to $\beta - \delta$ the child is considered promising. The constant δ, called the β-cutoff margin, can be either positive or negative. Although, this scheme may require additional search, it will hopefully give a better estimate than the aforementioned schemes.
4. Null-window search ($NWS(d - r) \geq \beta - \delta$):
   This scheme is much like the one above, except instead of using quiescence search to estimate the merit of the children, a null-window search to a closer horizon at distance $d - r$ is used.

To establish how well the number of promising moves, as judged by each of the above schemes, correlates to an expected cut-node being a True cut-node or not, we had the program gather statistics about cut-nodes. When the program visits an expected cut-node it calculates the number of promising move alternatives in the position according to each of the above schemes. Then, after searching the node to a full depth to determine if it really is a cut-node, information about the number of promising moves is logged to a file along with a flag indicating whether the node is a True cut-node.

The resulting data was classified into two groups, one with the True cut-nodes, and the other with the False cut-nodes. The program gathered statistics about 100,000 expected cut-nodes, and of these only 2.5% were classified incorrectly (i.e. were False cut-nodes). The average number of promising moves, as judged by each scheme, is presented in Table 1. The second column shows the

average for the True cut-node group and the third column the average for the False cut-node group. By comparing the averages and the standard deviations (also shown in the table) of the two groups we can determine the scheme that can best predict False cut-nodes. That is, we are looking for the scheme that has the greatest difference between the averages for the two groups, and the lowest standard deviation.

Table 1. Comparison of different schemes for identifying False cut-nodes

Method	True cut-nodes		False cut-nodes	
	$\overline{x}$	σ	$\overline{x}$	σ
NM	35.60	11.74	24.83	14.46
$HH > 0$	22.27	8.87	16.35	9.77
$HH > 100$	9.15	5.72	7.13	5.33
$QS() > \beta$	20.48	15.03	0.32	1.44
$QS() > \beta$-25	23.70	14.08	1.66	4.20
NWS(d-2) $> \beta$	20.62	14.88	0.17	0.55
NWS(d-2) $> \beta$-25	23.75	14.00	1.46	3.75

In Table 1, it is interesting to note that even a simplistic scheme like looking at the number of legal moves shows a difference in the averages. However, the difference is relatively small and the standard deviation is high. The history heuristic schemes have lower standard deviation, but unfortunately the averages are too similar. This renders them useless. The methods that rely on search, $QS()$ and $NWS()$, do much better, especially those where δ (the β-cutoff margin) is set to zero[6]. Not only are the averages for the two groups far apart, but the standard deviation is also very low. From the data in Table 1 the two schemes look almost equally effective. Therefore, to discriminate between them further, we filtered the data for the False cut-nodes looking only at non-zero data-points (that is, we only consider data-points where at least one promising move alternative is found by either scheme). The result using the filtered data is given in Table 2. Now we can see more clearly that the null-window (NWS) scheme is a better predictor of False cut-nodes. Not only does it show on average fewer false promises, but the standard deviation is also much lower. This means that it only infrequently shows False cut-nodes as having more than several promising move alternatives. Even in the worst case there never were more than 6 moves listed as promising, whereas for the $QS()$ scheme at least one position had 32 wrong indicators.

The above experiments clearly support the hypothesis that there is a way to discriminate between nodes that are likely to become true cut-nodes and those that are not. As a result, we selected the shallow null-window searches as the scheme for finding promising moves in multi-cut $\alpha\beta$-pruning.

[6] In *The Turk*, a δ value of 25 is equivalent to a quarter of a pawn.

Table 2. Comparison of selected schemes using filtered data

Method	False cut-nodes	
	$\overline{x}$	σ
$QS() > \beta$	2.31	3.20
$NWS(d-2) > \beta$	1.45	0.86

7.2 Multi-Cut Parameters

Next, after implementing the multi-cut algorithm in our chess program, we experimented with different instantiations of the multi-cut parameters both to give a better insight into how they alter the search behavior, and to find the most appropriate parameter setting for the program. The program was tested against a suite of over one thousand tactical chess problems [12]. For each run a different set of multi-cut parameters was used, and information was collected about both the total number of nodes explored, and the number of problems solved. The program was instructed to search to a nominal depth of 7-ply, and use normal search extensions and null-move search reductions. Basically, we are looking for the parameters that give the most node reduction, while still solving the same number of problems that the original program does.

Figure 4 shows the search effort under a range of parameter settings. The search effort is given as a percent of nodes searched by the standard version of the program. The depth reduction is fixed to 2, but the c and m parameters are

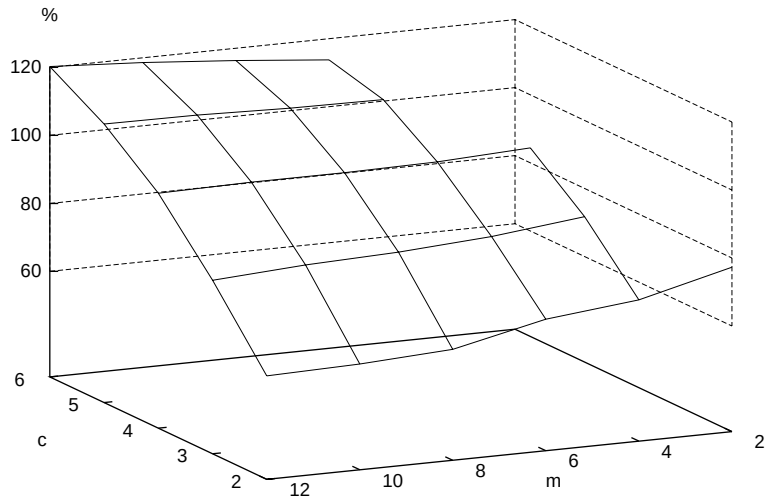

Fig. 4. Search efficiency when $r = 2$

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 limited node savings, while values of $r > 2$ were too error prone. The data from all the experiments is included in tabular form as an appendix. As expected, the fewest nodes are examined for small values of c. For example, the program with $c = 2$ and $m = 12$ searches over 40% fewer nodes than the original program. However, the node savings decrease rapidly as c increases, breaking approximately even at $c = 4$, and searching considerately more nodes for higher values. We also see how m influences the search, although these changes are more subtle. An interesting observation is that for low values of c the total number of nodes decreases as m increases, but the opposite is true for higher values of c. This can be explained by the counter-acting effects we discussed earlier. For low values of c, we observe more *mc*-prunings as m increases, and the extra cutoffs more than offset the additional search overhead of each *mc*-prune search. However, for larger values of c there are far fewer additional cutoffs, and the increased cost of each *mc*-prune search starts to show. From looking only at this graph, using a low value of c and a relatively high value for m, results in the best search efficiency. However, we still have to look at the other side of the coin, namely the error rates associated with the different parameter settings.

Figure 5 shows a similar graph, except here we are looking at the percentage of problems solved (as compared to the standard version of the program). Most notable is the steep increase in the percentage of problems solved as c is increased from 2 to 3. However, increasing c further only yields slow improvement. There is also a slight trend towards an improved accuracy as m is decreased, at least for the smaller values of c. This is understandable, by decreasing m the criterion for *mc*-prune is being set more conservatively.

From the above data, setting $c = 3$ and m 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.

7.3 Multi-Cut in Practice

Ultimately, we want to show that a game-playing program using the new pruning method can achieve increased playing strength. Although, the aforementioned experiments are useful in giving insight into the feasibility of the idea and the behavior of the search, they do not tell how beneficial the new method is in practice. For that actual chess matches are needed. Generally, when using a forward-pruning scheme playing games is the only way to show the proper balance between improved search efficiency and added risk of overlooking good continuations.

Two versions of the program were matched against each other, one with multi-cut pruning and the other without. Four matches, with 80 games each, were played using different time controls. To prevent the programs from playing the same game over and over, forty well-known opening positions were used as a starting point. The programs played each opening once from the white side

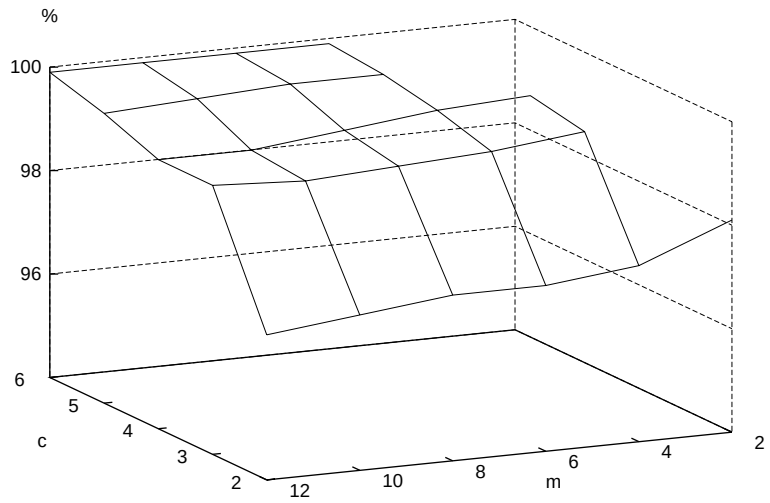

Fig. 5. Decision quality when $r = 2$

and once as black. Table 3 shows the match results. T represents the unmodified version of the program and $T_{mc(c,m,r)}$ the version with multi-cut implemented. We experimented with the case $m = 10$, $r = 2$, and $c = 3$ (i.e. 10 moves searched with a depth reduction of 2 ply and with 3 cutoffs required to achieve the *mc*-prune condition).

Table 3. Summary of 80-game match results

$T_{mc(3,10,2)}$ versus T		
Time control	Score	Winning %
40 moves in 5 minutes	46 - 34	57.5
40 moves in 15 minutes	42 - 38	52.5
40 moves in 25 minutes	43.5 - 36.5	54.4
40 moves in 60 minutes	43 - 37	53.8

The multi-cut version shows definite improvement over the unmodified version. In tournament play this winning percentage would result in about 35 points difference in the players' performance rating. Although the results are encouraging, it is still too early to state the exact strength difference between the two versions, based only on this single set of experiments: for that more games are needed.

One final insight: the programs gathered statistics about the behavior of the multi-cut pruning. The search spends about 25%-30% of its time (in terms of

nodes visited) in shallow multi-cut searches, and an *mc*-prune occurs in about 45%-50% of its attempts. Obviously, the tree expanded using multi-cut pruning differs significantly from the tree visited when it is not used.

8 Related Work

The idea of exploring additional moves at cut-nodes is not entirely new. There exist at least two other variants of the $\alpha\beta$-algorithm that explore more than one alternative at cut-nodes, although the resulting information is used quite differently in our work.

The *Singular Extensions* algorithm [2] extends "singular" moves more deeply than others. A move is defined as singular if its evaluation is higher than all its siblings by some specified margin, called the singular margin. Moves that fail-high, i.e. cause a cutoff, automatically become candidates for being singular (the algorithm also checks for singular moves at pv-nodes). To determine if a candidate move that fails-high really is singular, all its siblings are explored to a reduced depth. The move is declared singular only if the value of all the alternatives is significantly lower (as defined by the singular margin) than the value of the principal variation. Singular moves are "remembered" and extended one additional ply on subsequent iterations. This method improved the playing strength of Deep Thought (predecessor of Deep Blue) by about 30 USCF rating points [1]. One might think of multi-cut as the complement of singular-extensions: instead of extending lines where there is seemingly only one good move, it prunes lines where many promising (refutation) moves are available.

The *Alpha-Beta-Conspiracy* algorithm [9] is essentially an $\alpha\beta$-search that uses conspiracy depth, instead of classical ply depth, to decide when to stop searching a branch. The conspiracy depth is updated at each node in the tree, but instead of reducing the depth always by one ply, it can be reduced by a fraction of a ply, all depending on how many good alternative moves there are. The fewer alternatives, the smaller will be the conspiracy depth reduction. Quiescence searches are used to establish the number of good alternative moves. This algorithm encourages forced lines to be searched more deeply. Another distinct feature of the algorithm is that two separate conspiracy depth parameters are used, one for each player. At each level, only the conspiracy depth parameter for the player to move is updated. The search explores a branch until either both conspiracy depths parameters converge to zero, or alternatively, when the conspiracy depth for the player to move is zero and a static evaluation delivers a cutoff. However, empirical results using this algorithm were not favorable.

9 Conclusions

We have shown that there exists a strong correlation between the number of promising move alternatives available at an expected cut-node, and the node becoming a True cut-node. We investigated how this can affect error propagation when using a minimax-based search algorithm, and we introduced a new

forward-pruning method, multi-cut, that exploits this correlation. Furthermore, to show the feasibility of the idea, we implemented and experimented with the technique in an actual game-playing program. Our experimental results give rise to optimism. In match play, a version of our chess program using the new method, consistently outplayed an unmodified version of the program. This indicates that our search method, while expanding a tree that is radically different from the $\alpha\beta$-algorithm, has seemingly improved playing strength.

The multi-cut method is still in its infancy. There is still scope for improvement through further tuning and enhancement. For example, we have parameterized our method using variables instead of constants for c, m, and r, and propose that their values be adjusted dynamically as the game/search progresses. The multi-cut method as described and implemented here is not the only way of using the information about the number of promising move alternatives at cut-nodes, and by no means necessarily the best. Our experiments show that there is room for innovative domain-independent pruning methods, based on exploiting the structure of the minimal tree.

References

1. T. Anantharaman. *A Statistical Study of Selective Min-Max Search in Computer Chess*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 1990.
2. T. Anantharaman, M. S. Campbell, and F. Hsu. Singular extensions: Adding selectivity to brute-force searching. *Artificial Intelligence*, 43(1):99–109, 1990.
3. D. G.Beal. *Experiments with the Null Move*, pages 65–89. Elsevier Science Publishers B.V., 1989. D.F. Beal (Editor).
4. A. Junghanns. Are there practical alternatives to alpha-beta? *ICCA Journal*, 21(1):14–32, 1998.
5. D. E. Knuth and R. W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.
6. T. A. Marsland. Relative efficiency of alpha-beta implementations. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-83)*, pages 763–766, Karlsruhe, Germany, August 1983.
7. T. A. Marsland. Single-Agent and Game-Tree Search. In A. Kent and J. G. Williams, editors, *Encyclopedia of Computer Science and Technology*, volume 27, pages 317–336, New York, 1993. Marcel Dekker, Inc.
8. T. A. Marsland and F. Popowich. Parallel game-tree search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-7(4):442–452, July 1985.
9. D. A. McAllester and D. Yuret. Alpha-beta-conspiracy search, 1993. URL: http://www.research.att.com/~dmac/abc.ps.
10. D. S. Nau. Pathology on game trees: A summary of results. In *Proceedings of the ACM National Conference on Artificial Intelligence*, pages 102–104, 1980.
11. A. Reinefeld. An improvement to the Scout tree search algorithm. *ICCA Journal*, 6(4):4–14, 1983.
12. F. Reinfeld. *1001 Brilliant Ways to Checkmate*. Sterling Publishing Co., New York, N. J., 1955. Reprinted by Melvin Powers Wilshire Book Company.
13. J. Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(1):1203–1212, 1989.

14. D. J. Slate and L. R. Atkin. *Chess Skill in Man and Machine*, chapter 4. CHESS 4.5 – Northwestern University Chess Program, pages 82–118. Springer-Verlag, New York, NY, 1977.

A Appendix

The result of the experiment described in Section 7.2 is shown in Table 4 below. Both the number of nodes searched and problems solved are relative to the performance of the standard (unmodified) version of the program.

Table 4. $T_{mc(c,m,r)}$ searches

r	c	m	Nodes	Solved	r	c	m	Nodes	Solved	r	c	m	Nodes	Solved
1	2	2	92.05	98.10	2	2	2	77.28	98.10	3	2	2	79.21	96.80
1	2	4	93.33	97.60	2	2	4	70.48	97.40	3	2	4	71.60	95.80
1	2	6	93.02	97.20	2	2	6	67.61	97.20	3	2	6	67.71	95.80
1	2	8	91.71	97.20	2	2	8	61.56	97.20	3	2	8	63.17	95.50
1	2	10	92.10	96.80	2	2	10	60.04	97.00	3	2	10	60.57	95.20
1	2	12	93.39	96.80	2	2	12	59.38	96.80	3	2	12	57.13	95.10
1	3	4	134.17	99.20	2	3	4	87.46	99.50	3	3	4	86.07	97.70
1	3	6	144.14	99.20	2	3	6	84.41	99.30	3	3	6	82.92	97.50
1	3	8	150.31	98.90	2	3	8	82.60	99.20	3	3	8	79.30	97.50
1	3	10	153.00	98.70	2	3	10	81.66	99.10	3	3	10	75.86	97.10
1	3	12	157.34	98.50	2	3	12	79.95	99.20	3	3	12	72.21	97.00
1	4	4	175.38	99.40	2	4	4	100.14	99.70	3	4	4	98.33	98.60
1	4	6	194.19	99.40	2	4	6	98.86	99.60	3	4	6	94.20	97.90
1	4	8	210.41	99.30	2	4	8	98.50	99.40	3	4	8	89.96	97.90
1	4	10	222.67	99.10	2	4	10	98.51	99.20	3	4	10	87.39	97.70
1	4	12	234.33	99.00	2	4	12	98.04	99.20	3	4	12	84.89	97.60
1	5	6	227.73	99.50	2	5	6	109.63	99.80	3	5	6	97.23	98.50
1	5	8	252.26	99.60	2	5	8	109.93	99.80	3	5	8	94.95	98.10
1	5	10	276.16	99.50	2	5	10	110.67	99.70	3	5	10	92.02	97.90
1	5	12	286.82	99.40	2	5	12	110.88	99.60	3	5	12	90.24	97.80
1	6	6	239.81	99.70	2	6	6	113.77	99.90	3	6	6	100.97	99.20
1	6	8	269.33	99.70	2	6	8	116.40	99.90	3	6	8	99.42	98.30
1	6	10	312.24	99.70	2	6	10	118.61	99.90	3	6	10	100.24	98.30
1	6	12	335.51	99.70	2	6	12	120.23	99.90	3	6	12	95.66	98.00