Risk Management in Game-Tree Pruning

Y. Björnsson and T.A. Marsland

Department of Computing Science, University of Alberta, Edmonton, AB,
CANADA T6G 2H1

Abstract

In the half century since *minimax* was first suggested as a strategy for adversary game search, various search algorithms have been developed. The standard approach has been to use improvements to the Alpha-Beta (α-β) algorithm. Some of the more powerful improvements examine continuations beyond the nominal search depth if they are of special interest, while others terminate the search early. The latter case is referred to as forward pruning.

In this paper we discuss some important aspects of forward pruning, especially regarding risk-management, and propose ways of making risk-assessment. Finally, we introduce two new pruning methods based on some of the principles discussed here, and present experimental results from application of the methods in an established chess program.

1 Introduction to Decision Quality and Search

The standard approach to game-tree search is to use improvements to the Alpha-Beta (α-β) algorithm to explore all combination of moves to some fixed depth (continuation length or search horizon). In practice, however, the algorithms are not used that way, instead heuristics vary the distance to the search horizon, exploring some variations more deeply than others. In an indirect way, this resembles the human thinking process. Continuations that are thought to be of special interest are expanded beyond the nominal depth, while others are terminated prematurely. The latter case is referred to as *forward pruning*.

Humans are adept at simplifying the search process, by reasoning about the choices and then selecting a few prime candidates [4], while computers try to search all possible combinations of moves. Thus the number of nodes visited by Alpha-Beta increases exponentially with increasing search depth. This limits the scope of the search, especially because game-playing programs must meet external time-constraints: often having only a few minutes to reach a decision (choose of move). In practice, the quality of the move decision usually improves

the further the search looks ahead. The question now becomes, how to find a good move while making best use of the available time. One approach is to prune unpromising lines, so that the time saved can be used to search selected lines more deeply.

There exist several well established forward-pruning methods for game-tree search. For example, in chess the null-move heuristic is very effective (best described by Beal [2] and Goetsch & Campbell [5]), while in Othello Buro's ProbCut method is the most successful [3]. On the other hand, the theoretical issue surrounding forward pruning and how it affects decision quality has not been studied much. However, Smith and Nau [11] analyzed a model of forward pruning using (over)simplified game trees, and concluded that forward pruning works best when there is a high correlation among the minimax values of sibling nodes in the game tree. For specific games like chess, checkers and Othello, Junghanns *et al.* [6] have done some empirical studies to investigate how error in leaf-node evaluation affects the move decision at the root.

2   Forward Pruning

The real task when doing forward pruning is to identify move sequences that are worth considering more closely, and others that can be pruned off with minimal risk of overlooking a good continuation. Several factors should be considered for effective forward pruning:

- *Risk-assessment.*
  How safe is the forward-pruning method? We want to minimize the risks that this speculative pruning introduces into the search.
- *Applicability.*
  To maximize the possible gains from forward pruning we would like to apply the method frequently in the tree, especially where there is a potential for big savings.
- *Cost-effectiveness.*
  The investment of time and effort to decide whether to prune a node should be kept low. In any case, the savings achieved through pruning must exceed the additional effort introduced.
- *Domain-independency.*
  Ideally, we want a pruning method that can be applied in more than one specific search domain.

The above factors are by no means independent, improving one usually involves compromising another. For example, reducing the risks often means limiting the applicability, while improving cost-effectiveness can introduce other risks. Finally the more general (domain-independent) methods tend to be less

efficient. A useful forward-pruning heuristic must find the appropriate trade-off between the above factors, and this process may require careful tuning.

2.1 Risk-assessment

When using forward pruning there is always some danger of overlooking good moves. We would like to minimize the risk of doing so. When deciding whether to examine a node v, the basic question is: how likely is it that the sub-tree below v includes a continuation that, if searched, would yield a new principal variation (pv). For a new variation to emerge two things must occur; first the value returned to v must exceed the best value found so far, and second the value must propagate to the root of the tree. This in turn implies that the pruning method should be able to

- predict with reasonable accuracy the range of values for node v, and
- measure the likelihood that the anticipated value will back up to the root of the tree.

Existing forward-pruning methods address the first issue while often ignoring the second one.

2.1.1 Error Introduction

For most subtrees we are not so much interested in knowing the exact value of each particular node, but rather whether the value lies outside the bounds of the α-β window. This is because we know that continuations which result in values outside the window can never become a part of the principal variation. When using a null-window search the bound is the value of the current principal variation, so when comparing node values to the bound we are determining whether a better continuation is found. In that case we are simply interested in knowing if a value returned by searching a node further is at least as good as the β-bound, since then it causes a cutoff.

When predicting where the value of a node v lies relative to the α-β bounds, most pruning methods carry out a shallow search. They use the value returned to estimate the range in which the actual value of node v is likely to be found when the node is searched more deeply. For example, a 5-ply-deep search is used to predict the window for a 6-ply-deep search. The outcome of the shallow search decides whether to search node v further. If we are confident enough that further search will not yield an improvement, node v is not expanded. The exact criteria used to relate the value of the shallow search to the anticipated return value of the deeper search varies with the pruning technique. Some approaches rely on statistical methods to define confidence intervals, while

others simply use *ad hoc* heuristics. Error is introduced into the search when a wrong pruning decision is made.

Although, values returned by shallow searches are usually reasonable estimates of the values found by deeper searches, additional information can enhance the overall prediction capabilities of the pruning heuristics, thereby reducing the risk involved. For example, consider the tree in Figure 1. The shaded area
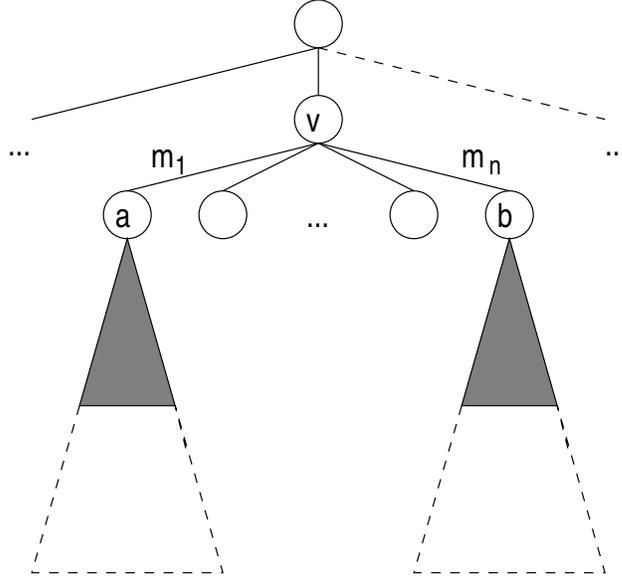
Fig. 1. Different risk-assessment.

marks the parts of the tree searched to decide whether to prune nodes a and b. Each pruning decision is made independently of the other, based only on the outcome of the local search. However, by looking at each node in isolation information is lost. For example, when looking at move m_n current pruning methods are interested in knowing if the move will lead to a value that causes a cutoff, that is, in estimating the probability

$$P(v(m_n) \geq \beta).$$

But having already searched moves $m_1,...,m_{n-1}$, and knowing that none caused a cutoff, provides a strong indicator that move m_n will also fail to do so, especially because the preliminary move-ordering scheme believes that move m_n is no better than the moves already considered. Instead one should compute the probability that move m_n causes a cutoff, given that moves $m_1,...,m_{n-1}$ have failed to do so, that is compute:

$$P(v(m_n) \geq \beta \mid v(m_1, ..., m_{n-1}) < \beta).$$

The values of the moves are not independent of each other, and so by assuming otherwise we are ignoring potentially useful information. Existing pruning methods and probability based best-first search algorithms totally ignore the

dependencies, or unrealistically assume the search values (or the error in the values) are independent of each other. *Instead, the fact that the values tend to be dependent should be used to make more informed pruning decisions.*

2.1.2 Error Propagation

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 correspond to nodes that have not been expanded. Assume that the search is currently situated at
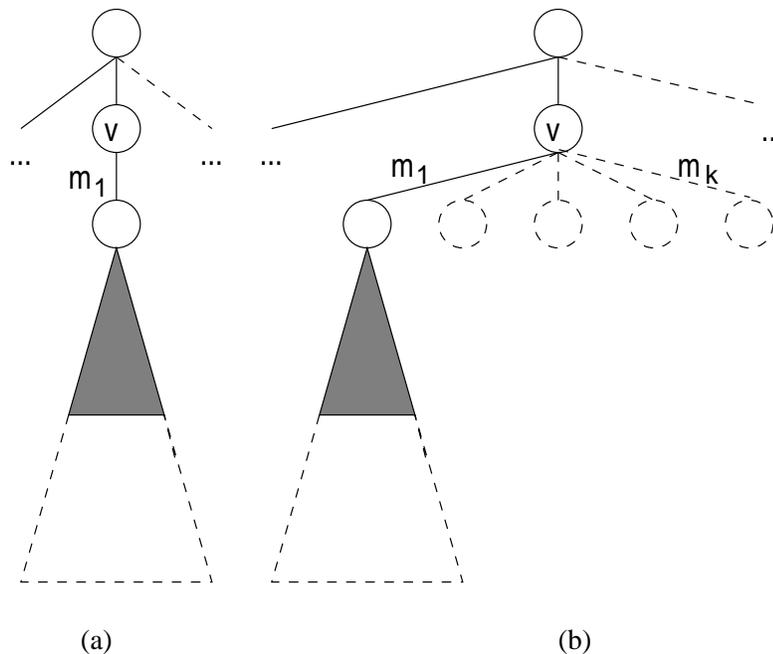
Fig. 2. Controlling error propagation.

node v and that the sub-tree resulting from playing move m_1 has already been searched. Furthermore, assume that a part of that sub-tree has been cut away using some forward-pruning technique, and that the value returned is greater or equal to the β-bound for node v (if node v is a cut-node this is what we would expect). Therefore, a β-cutoff occurs and the value returned by move m_1 will back up to the root. From the root's perspective this branch is inferior to the current principal variation and the search therefore continues to expand the other children of the root without switching 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 present but is overlooked, the value of m_1 is wrong and the error may propagate through node v to the root. However, if alternatives to m_1 are present, as in Figure 2(b), it is possible that one of the alternative moves in $[m_2, ..., m_k]$ may contain a line that enables it to deliver a beta-cutoff at v, acting as a substitute for m_1, and thus preserving the value

assigned at node v. Thus in Figure 2(b), an error in the subtree below m_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. This means that even though the pruning below m_1 was flawed, the risk of affecting the move decision at the root is less in Figure 2(b) than in Figure 2(a), because one of the other moves m_2 ..., m_k might preserve the cutoff if m_1 changes it's value. Thus, *even though an erroneous pruning is made 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.

2.2   Applicability

The most popular pruning heuristics used in two-person game-playing programs have one thing in common: they apply frequently, though not without restriction. The more frequently a pruning heuristic is applied in the search, especially at places where there is a high probability of big savings, the more potential it has for being effective. However, the applicability is restricted, since pruning can only be done where it is expected to be safe. Depending on the heuristics used, this can differ substantially. Some heuristics need additional pre-requirements for them to be applied. An example of one such pruning method is the special cutoff introduced by Reinefeld's NegaScout [8] algorithm. Although the cutoff is risk-free when search extensions are not used, the savings are very small (less than 2%). This is because the necessary pre-requirement for the cutoff is met infrequently. That is, a change in the principal variation is occurring two or fewer plies from the search horizon.

2.3   Cost-effectiveness

Although some pruning methods offer low risks and substantial savings in terms of nodes searched, the overhead needed to implement them is often prohibitive. The effort expended gathering and tracking in real-time the information required by the heuristics may outweigh the potential time-savings introduced by the pruning. An example of such a heuristic is the method of analogies. Although, the method offers almost risk-free pruning, the overhead of tracking how pieces influence each other originally proved too high for practical use in a competitive chess playing program [1]. However, changes in software and hardware technology may improve the efficiency of such methods. It might also be possible to approximate the original heuristic by another that is less costly to maintain, and yet achieve most of the savings. Therefore

the method of analogies is again a topic worthy of investigation.

2.4 Domain-dependency

Ideally we want domain-independent pruning. Those methods would not rely on such explicit knowledge as: is a king in check, or whether a corner square is occupied. If domain-specific knowledge, is used, it is incorporated in a domain independent way, for example express the knowledge in a language that can be interpreted in a general way by the search.

In the domain independent methods, the only information revealed to the search by the evaluation function is a numerical estimate of a problem state's quality. This clear separation of the search and the problem encourages more domain-independent pruning methods. On the other hand the methods are then denied access to potentially useful information about the problem domain, thereby restricting their pruning capabilities. However, there is a wealth of information to be gathered about the problem by simply looking at the shape of the expanded search tree. This knowledge is accessible without having to uncover any additional domain-specific knowledge. We have already mentioned a few cases of interest as part of our risk-assessment discussion.

In practice, it is extremely difficult for pruning methods to be domain independent. As said earlier, there is a trade-off between generality and effectiveness, and to achieve the full pruning capability we must exploit some special characteristics of the search space. Most existing forward-pruning methods are therefore domain specific. Even though methods like null-move [2] and Prob-Cut [3] do not use explicit knowledge about their domain, they make implicit assumptions that tie them down for use in one, or at best very few, two-person games. For example, the null-move heuristic is very effective in chess, but inappropriate in Othello. Conversely, ProbCut is the pruning heuristic of choice in Othello but has not yet been shown useful in chess or checkers.

3 New Forward-Pruning Methods

As mentioned before, pruning heuristics should be concerned with the question: *What is the likelihood of making an erroneous pruning decision, and if an erroneous decision is made how likely is it to affect the principal variation?* Existing forward-pruning methods generally do not consider the second part of this question. When assessing risk, pruning methods should not only speculate whether a subtree contains a good continuation, but also determine if there are alternatives to any potentially overlooked continuation that could

preserve the principal variation. To answer these questions the methods must consider each node in the context of its location in the game-tree, instead of looking at each node (and the subtree below it) in isolation.

In the following sections we introduce two new forward-pruning methods: *Multi-Cut*, and Variable Null-move Bound. Both our new methods indirectly consider the likelihood that the consequences of an erroneous pruning decision propagate to the root of the game tree.

4 Multi-Cut

In this section we present the idea behind the multi-cut method, then give implementation details, and experimental results.

4.1 Multiple cut-offs idea

In a traditional $\alpha\beta$-search, if a move returns a value greater or equal to β there is no reason to examine that position further, and the search can return. This is often referred to as a β-cutoff. Intuitively, this means that the player to move has found a way to refute the current line of play, so there is no need to find something better. By way of explanation, and to introduce our terminology, we are seeking the principal variation. This is the sequence of moves from the root node (current position in the game) to the best of the accessible nodes on the search horizon. We expect β-cutoffs to occur at so called cut-nodes (that is, nodes that are refuted). The root node of a game-tree is a pv-node (principal variation node), the first child of a pv-node is also a pv-node, while the other children are cut-nodes. Every child of a cut-node is an all-node (where every successor must be explored) and vice versa every child of an all-node is a cut-node. In a perfectly ordered tree only one child of a cut-node is expanded. If a new best move is found at a pv-node, the node it leads to is also a pv-node. At pv- and all-nodes every successor is examined. Most often it is the first child that causes the cutoff, but if it fails to do so the sibling nodes are expanded in turn, until one of them returns a value greater or equal to β (thus causing a cutoff), or all the children have been searched. If none of the moves causes a cutoff, the cut-node becomes an all-node.

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 con-*

sequently 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 a *mc*-prune (multi-cut prune).

Definition 1 (*mc*-prune) *When searching node v to depth $d + 1$ using $\alpha\beta$-search, and if at least c of the first m children of v return a value greater or equal to β when searched to depth $d - r$, a mc-prune is said to occur and the search can return.*

In multi-cut $\alpha\beta$-search, we attempt an *mc*-prune only at expected cut-nodes (we would not expect it to be successful elsewhere). Figure 3 shows the basic idea. At node v, but before searching v_1 to a full depth d, as normal $\alpha\beta$-search does, the first m successors of v are expanded to a reduced depth of $d - r$. If c of them return a value greater or equal to β a *mc*-prune occurs and the search returns the value of β, otherwise the search continues as usual exploring v_1 to a full depth d. The subtrees below $v_2, ..., v_m$ represent extra
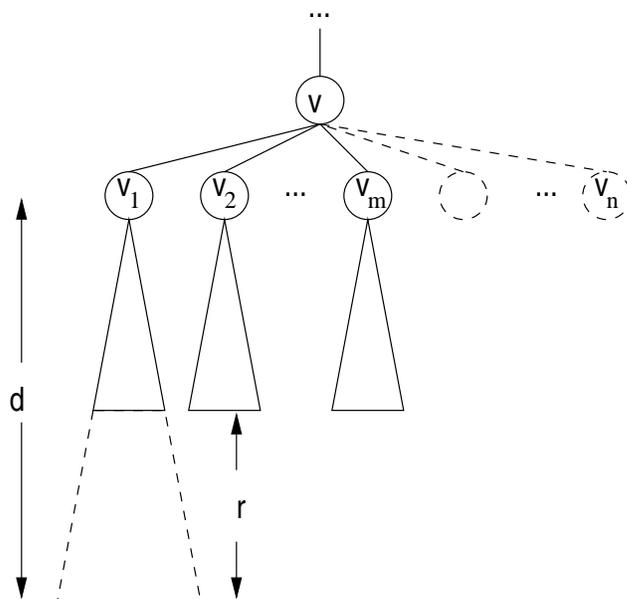
Fig. 3. Principles of multi-cut pruning.

search overhead introduced by *mc*-prune, since they would not be expanded by normal $\alpha\beta$-search. The dotted area of the subtree below node v_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. By searching the subtree of v_1 to a shallower depth, there is of course some risk of overlooking a tactic that would make v_1 become a 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 still cause a genuine β-cutoff if searched to a full depth.

Figure 4 is a C-code version of a null-window search (NWS) routine using multi-cut. For clarity we have omitted details about search extensions, transposition table lookups, null-move searches, and history heuristic updates that are not immediately relevant to our discussion. The *NWS* routine [7] could for example be called by an enhanced $\alpha\beta$-variant like *Principal Variation Search* or *NegaScout*. The parameter *depth* 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 β - 1. In our case, however, the new parameter *ntype* is needed to identify the nodes where *mc*-pruning applies.

As is normal, the routine starts by checking whether the horizon has been reached, and if so to use a quiescence search (QS) to return the value of the position. Otherwise, we look in the transposition table for useful information to guide the search. This is followed by a null-move search (most chess programs use this powerful technique) [2]. A normal $\alpha\beta$ search would then start exploring the possible moves to the requested depth. Instead we insert here a multi-cut search to see if the *mc*-prune condition applies. If a multi-cut occurs the local search terminates, but otherwise it continues as a normal $\alpha\beta$ search. The parameters *mc_M*, *mc_R*, and *mc_C* stand for m (number of moves to look at), r (search reduction), and c (number of cutoffs needed), respectively. Although they are shown here as constants, they could be determined more dynamically and allowed to vary during the search.

We do not check for the *mc*-prune condition at every node in the tree. First, we test for it only at expected cut-nodes. Second, it is not applied at the 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. In Figure 4 these restrictions are encapsulated in the function *TryMultiCut()*. In our experiments in the domain of chess (see Section 4.3) the pruning is disabled when the endgame is reached. Usually only a few viable move options exist there, and so the *mc*-search is 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. Therefore the programs rely heavily on the search to guide them, and so any forward-pruning scheme is more likely to be harmful. Finally, the pruning is not done if the side to move is in check, or if a search extension has been applied at any of the three previous moves leading to the current position, since it is preferable that these forced situations be assessed correctly.

```
// Multi-Cut parameters
#define mc_M    10  //# of moves to look at
#define mc_C     3  //# of cuts to cause a mc-prune
#define mc_R     2  //depth reduction
#define CUT      2
#define ALL      3
#define TYPE(t) (((t)==CUT) ? ALL : CUT)

VALUE NWS(int depth, VALUE beta, NODETYPE ntype) {
  VALUE score;
  MOVE  move;

  if ( depth <= 0 ) return QS(beta-1);
  // Transposition table lookup, and nullmove search omitted ...

  // Multi-Cut pruning
  if ( (ntype == CUT) && (depth > mc_R) && TryMultiCut() ) {
    int m = 0, c = 0;
    move = MoveFirst();
    while ( m < mc_M && move ) {
      MakeMove( move );
      score = -NWS(depth-mc_R-1, -beta+1, TYPE(ntype));
      RetractMove( move );
      if ( score >= beta ) {
        c++;
        if ( c == mc_C ) return beta;
      }
      m++;
      move = MoveNext();
    }
  }

  // Standard null-window (minimal window) search
  move = MoveFirst();
  while ( move ) {
    MakeMove( move );
    score = -NWS(depth-1, -beta+1, TYPE(ntype));
    RetractMove( move );
    if ( score >= beta ) break;
    move = MoveNext();
  }

  // Store node information, omitted ...
  // ... update trans table and history heuristic...
  return score;
}
```

Fig. 4. Multi-Cut null-window search.

Table 1. $TT_{mc(r,c,m)}$ searches

r	c	m	Nodes	Solved	r	c	m	Nodes	Solved	r	c	m	Nodes	Solved
1	2	4	93.33	97.60	2	2	4	70.48	97.40	3	2	4	71.60	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	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	8	150.31	98.90	**2**	**3**	**8**	**82.60**	**99.20**	3	3	8	79.30	97.50
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	8	210.41	99.30	2	4	8	98.50	99.40	3	4	8	89.96	97.90
1	4	12	234.33	99.00	2	4	12	98.04	99.20	3	4	12	84.89	97.60

4.3 Experimental Results

Ultimately, we want to show that game-playing programs using the new prun-
ing method can achieve increased playing strength. To test the idea in practice
multi-cut $\alpha\beta$-pruning was implemented in *The Turk*[1] . Next, 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 appropri-
ate parameter setting for the program. The program was tested against a suite
of over one thousand tactical chess problems [9]. 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. For each parameter
setting, Table 1 shows the number of nodes searched and problems solved.
However, instead of using absolute values, the values are given relative to the
performance of the standard version of the program. The high-lighted data
indicates that a setting of $r = 2$, $c = 3$, and m in the range 8-12 looks the
most promising. Using these settings the program searches 20% fewer nodes,
while still solving 99.2% of the problems handled by the original version. A
higher value for c, does not result in any improvement in search efficiency,
while a lower value significantly lowers the decision quality. Similarly there is
little scope for varying r. On the other hand, there is some freedom in choosing
the m parameter.

Finally, two versions of the program were matched against each other, one us-
ing multi-cut pruning and the other without. Several matches, with 80 games
each, were played using different time controls. To prevent the programs from

---

[1] *The Turk* is a chess program developed at University of Alberta by Yngvi
Björnsson and Andreas Junghanns.

Table 2. Summary of 80-game match results.

$TT_{mc(2,3,10)}$ versus TT		
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

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 and once as black. Table 2 shows the match results. TT stands for the unmodified version of the program and $TT_{mc(r,c,m)}$ for the version with multi-cut implemented. We experimented with the case $m = 10$, $r = 2$, and $c = 3$ (i.e. 10 moves searched using a depth reduction of 2 and requiring 3 β-cutoffs to achieve the mc-prune condition). This choice of parameter values is based on the test-suite of data presented above. 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. More games are needed for determining exactly the relative strength of the two versions, but based on these experiments we can state with a 95% confidence level that the multi-cut version is the stronger.

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 mc-prune occurs in about 45%-50% of its attempts. Obviously, the tree expanded using multi-cut pruning differs significantly from the tree expanded when it is not used.

5   Variable Null-move Bound

In this section we describe a new enhancement to the null-move heuristic, *variable null-move bound* search, that utilizes some of the aforementioned observations to search more efficiently.

5.1   Basis

Goetsch and Campbell [5] mention as a future research idea the possibility of permitting a null-move cutoff not only when a null-move search returns a value greater or equal to β, but also if the returned value is slightly less.

They propose that a cutoff be forced if $v \geq \beta - t$, $t \geq 0$, where v is the value returned by the null-move search and t is a small positive number that can be interpreted as the value of a tempo. This allows null-move cutoffs to be applied more frequently thereby reducing the tree-size even further, although at the cost of introducing additional errors. Furthermore, they state that the value of t must be lower than the actual value of a tempo to avoid inadvertent cutoffs, and that the value of t would be dependent on the evaluation function and could vary during the course of the game. Both these factors help reduce the risk of erroneous pruning.

The method we introduce here is based on the same idea. However, we use a different approach for approximating t. Instead of having t depend on the evaluation function, we let it vary according to how likely we think an erroneous pruning decision affects the principal variation. Indeed, in some parts of the tree we allow the value of t to exceed what would normally be considered an appropriate value for a tempo.

5.2   Implementation

First we need a metric to show how likely it is that a pruning error affects the principal variation. The more opportunities a player has to refute the opponent's play, the less likely it is that an oversight in assessing an individual node will affect the move decision at the root (see the discussion with Figure 2). This suggests that one metric is the "Number of Potentially Good Alternative Moves" ($NoPGAM$) that a player has on the path leading from the root to the current node in the search tree. However, there are a couple of difficulties. First, at cut-nodes only one, or at most few, moves are considered, leaving us with no information about the remainder. Second, since programs commonly employ a null-window search, for most nodes in the tree we have only bounds on the actual value of a node, making it difficult to compare the merits of any two moves. One approach would be to perform additional shallow searches to estimate the value of each move, but this would imply a considerable extra search overhead, possibly offsetting any gains. Fortunately, there are more cost-effective means of approximating the number of potentially good alternative moves. Most programs use some form of Schaeffer's history-heuristic [10] for move ordering. Whenever a move causes a cut-off it is rewarded by increasing its history heuristic value. We simply define a move to be a potentially good alternative if it has a positive history heuristic value. Although this is not the most accurate approximation it is a cost-effective one.

We implemented the variable null-move bound heuristic in *The Turk*. The program uses principal variation search, and null-moves are applied recursively with a search reduction of 2. Several restrictions control where and when

14

the null-move heuristic applies. For example, a null-move is not allowed on the principal variation or if the side to move is in check. Neither are two consecutive null-moves allowed. Figure 5 shows how the null-move heuristic is applied in the variable bound scheme. The variable $NoPGAM$ is the number of potentially good alternative moves that are found on the path from the root, but are still unexplored. A separate count is kept for each player and is updated incrementally as the tree is traversed.

```
if ( NULLMOVE_OK() ) {
    int bound, t = 0;
    if ( !InNullMoveSearch() ) {
        if ( NoPGAM > 15 ) {
            t = 20;
        }
        else if ( NoPGAM > 0 ) {
            t = 10;
        }
    }
    bound  = beta - t;
    Make ( Nullmove );
    score = -NWS ( depth+1, -bound+1, max_depth-2 );
    Retract ( Nullmove );
    if ( score >= bound ) {
        return ( beta );
    }
}
```

Fig. 5. Variable bound null-move cutoff decision.

In the current implementation, the number of potentially good alternative moves is recorded during a zero-window search, with the exception of the first move expanded at each node. The main reasons for this is that the first move is most often taken from the transposition table and expanded before any legal moves are generated. Because we count the number of good alternative moves for each level in the tree at the time of move generation, that information is not available to pass down for the first move. Since the $NoPGAM$ count is not updated for that level, this makes the program less aggressive in pruning along these paths [2].

---

[2] Since the first move expanded is often the most critical one, this compromise might actually be beneficial.

Table 3. Performance on tactical problems.

Method	Nodes	Solved
$TT_{variable}$	89.38	99.40
TT_{10}	84.44	98.90
TT_{20}	75.06	98.80

5.3 Experimental Results

We did experiments to assess the viability of the new heuristic. Three different variants of the chess program (TT_t) were created, each using a different value for the tempo, t. For two of the variants, TT_{10} and TT_{20}, t was set to a fixed constant, 10 and 20 respectively [3]. The third, $TT_{variable}$, varied the value of t using history data (see above). The relationship between $NoPGAM$ and t is as shown in Figure 5, and was chosen based on some trial and error tests. In a future implementation a more appropriate relationship will be empirically determined.

First, the programs were tested against the same suite of one thousand tactical chess problems as used in the multi-cut experiments. The result is shown in Table 3. As before, both the number of nodes searched and problems solved are relative to the performance of an unmodified program. We see that adjusting the cutoff margin offers substantial node savings, but at the cost of overlooking a few problem solutions. All the programs seem to do reasonably well, but notably the version that varies the cut-off margin misses the fewest solutions. This experiment is useful for showing that the pruning does not alter significantly the tactical ability of the program. This is particularly important for games like chess. On the other hand, the experiment does not show how adjusting the margin affects the positional play. This is best tested by playing actual games, as we do next.

The modified programs were matched against an unmodified version of the program (TT). Each match consisted of 100 games, with the time controls set to 40 moves in 5 minutes. To prevent the programs from playing the same game over and over, fifty well known opening positions were used as a starting point. The programs played each position twice, once from the white side and once as black. As is customary in actual tournament play, a player scores one point for a win, half a point for a draw, but nothing for losing. The match results are shown in Table 4. Of the three variants, $TT_{variable}$ performed the best, out-playing the original version with a winning ratio of 1.25 (i.e. scoring 25% more winning points than the opponent). The TT_{10} version also demonstrated increased playing strength, while the TT_{20} version lost its match convincingly.

---

[3] These numbers are related to a value of a pawn, which is set to 100.

This shows that for small margin adjustments, the benefits of increased search efficiency outweigh the drawbacks of the error introduced by more aggressive pruning. On the other hand, if the margin adjustment is too high the errors

Table 4. 100-game match results.

Match		Score			Winning ratio
$TT_{variable}$	vs. TT	55.5	-	44.5	1.25
TT_{10}	vs. TT	53.5	-	46.5	1.15
TT_{20}	vs. TT	41.0	-	59.0	0.69

become the dominating factor in determining the playing strength. Not only will the program make more tactical errors, but the positional play also suffers. This clearly shows in the match result where the TT_{20} version of the program performs rather poorly. However, by selectively adjusting the margin depending on how likely an error is to affect the principal variation, we achieve some of the search efficiency benefits from using a higher margin, but at only minimal risk of introducing additional errors. This follows from our match results where the $TT_{variable}$ version performs the best.

Although the preliminary results are encouraging, care must be taken in interpreting them. First, more than 100 games are required to reliably determine a difference in playing strength between any two programs, and second, games with actual tournament time controls must be played. The preliminary results indicate that this method has potential and is definitely worth refining further. For example, more study is needed to find the optimal values of the controlling parameters (r, c, m).

6   Conclusions

We discussed some important characteristics of forward pruning in game-tree search, emphasizing risk-management, and we propose ways to improve risk-assessment by considering

- move dependency information, and
- the likelihood that erroneous pruning decisions influence the principal variation.

We develop two new forward-pruning methods based on the above principles. Experiments with the new methods demonstrate consistent improvement over the currently best search techniques. This clearly indicates that pruning methods should consider the likelihood of an erroneous pruning decision propagating to the root of the game tree, something that has been neglected previously.

The new methods need additional practical experience before their full potential can be determined. There is still scope for improvement through tuning and further enhancements. For example, in our experiments we fixed the pruning parameters to constant values. Instead, the parameters could be adjusted dynamically, and their values allowed to vary as the game/search progresses. The new forward-pruning methods introduced here are by no means the only way to exploit the above principles for improved risk-assessment, there is room for other innovative methods.

References

[1] G. M. Adelson-Velskiy, V. L. Arlazarov, and M. V. Donskoy. Some methods of controlling the tree search in chess programs. *Artificial Intelligence*, 6(4):361–371, 1975.

[2] D. F. Beal. A generalized quiescence search algorithm. *Artificial Intelligence*, 43:85–98, 1990. See also: Experiments with the Null Move. In Advances in Computer Chess 5, 1989, 65-79.

[3] M. Buro. ProbCut: An effective selective extension of the alpha-beta algorithm. *ICCA Journal*, 18(2):71–76, 1995.

[4] A. de Groot. *Thought and Choice in Chess*. Mouton Publishers, The Hague, second edition edition, 1978.

[5] G. Goetsch and M.S. Campbell. Experimenting with the null move heuristic. In T. Marsland and J. Schaeffer, editors, *Computers, Chess and Cognition*, pages 158–168, 1990. See also: 1988 AAAI Spring Symposium Proceedings, 14-18.

[6] A. Junghanns, J. Schaeffer, M. Brockington, Y. Björnsson, and T. Marsland. Diminishing returns for additional search in chess. In *Advances in Computer Chess 8*, pages 53–67, June 1997.

[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] A. Reinefeld. An improvement to the Scout tree search algorithm. *ICCA Journal*, 6(4):4–14, 1983.

[9] F. Reinfeld. *1001 Brilliant Ways to Checkmate*. Sterling Publishing Co., New York, N. J., 1955. Reprinted by Melvin Powers Wilshire Book Company.

[10] 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.

[11] S. J. J. Smith and D. S. Nau. An analysis of forward pruning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1386–1391, 1994.