Searching with Uncertainty Cut-offs

Yngvi Bjornsson, Tony Marsland,
Jonathan Schaeffer and Andreas Junghanns
University of Alberta
Department of Computing Science
Edmonton, Alberta
CANADA T6G 2H1
Email: {yngvi,tony,jonathan,andreas}@cs.ualberta.ca

Abstract

A new domain-independent pruning method is described that guarantees returning a correct game value. Even though alpha-beta-based chess programs are already searching close to the minimal tree, there is still scope for improvement. Our idea hinges on the recognition that the game tree has two types of node, those where cut-offs occur and those that must be fully explored. In the latter case one of the moves is best and yields the subtree value, thus for the remaining alternatives one is simply proving their inferiority. This offers an opportunity for pruning, while introducing some potential for uncertainty in the search process. There are two cases of interest. One considers the immediate alternaitves to the Principal Variation itself, and here a safe uncertainty cut-off is presented, the second is a proposal for an unsafe generalization, one which offers substatnial search reduction but with the potential for control of the error probability. Experiments with the new pruning method show some potential for savings in the search.

1 Introduction

Alpha-beta is the fundamental algorithm used for searching game trees in chess and various other two-person games. It is much more efficient than a plain brute-force minimax search because it allows a large portion of the search tree to be pruned off, while still backing up the correct minimax value. Forward pruning is a different kind of pruning that does not guarantee that the correct game value is returned. Although forward pruning methods can achieve significant search reduction, they involve some risk since it is always possible that some important move is overlooked and therefore a wrong move decision is made.

In this paper, a new look at pruning is made. A common scenario in a search is that expectations change. Uncertainty in the search results in changes of the principal variation

(PV). When this happens, some branches get explored that, with hindsight, didn't have to. There is an opportunity here for savings. A new pruning technique, *uncertainty cut-offs*, is applied at carefully selected places in the search tree and bookmarks are kept where the pruning is done, so that one can tell if a backed up value is a true game value or an *uncertain* value. Even if forward pruning is used in the search, it does not necessarily affect the reliability of the game value at the root of the search tree. If the pruning is only done in sub-trees that turn out to be of no relevance for proving the game value, a guaranteed value can be backed up to the root. The book-marks tell us whether the pruning applied in the tree is affecting the reliability of the game value, thus giving us the opportunity to correct it by re-searching the uncertain moves. Hopefully, the gains of applying the pruning will out-weigh the extra search overhead of occasional additional re-searches of uncertain values.

In the next section we look at the minimal tree that must be searched to prove a value of a game tree. This is followed by two sections that describe the uncertainty cut-off pruning method, the idea and the implementation, respectively. Finally, some experimental results and our assessments are given.

2 Searching the minimal tree

To find a value of a game tree, at least the so-called *minimal tree* must be searched [KM75]. The nodes of the minimal tree can be categorized into three different types based on their properties as shown in figure 1. All moves have to be searched at PV- and ALL-nodes, but in perfectly ordered trees only one move need to be searched at CUT-nodes.

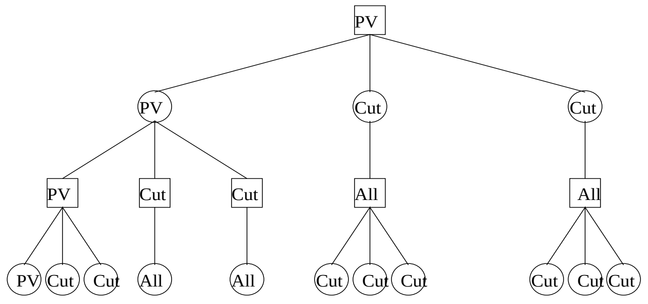

Figure 1: Minimal (perfectly ordered) search tree.

The performance of the alpha-beta algorithm is affected by the order in which nodes in the tree are searched. In the best case only the minimal tree is expanded. 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 cut-off can be searched first[1]. Because of the alpha-beta algorithm's sensitivity to the move ordering, it is important to expand good moves as

[1]Because of a non-uniform branching factor, search extensions and various possible transpositions, the sub-trees generated by different moves may vary considerably in size. In general, we would like to search first

early as possible. Various heuristics to achieve a good move ordering have been developed in the past, such as saving the previous best move for a position in the transposition table and the *history-heuristic* [Sch89]. By using these heuristics, empirical evidence shows that in over 90% of the cases where a cut-off occurs the cut-off is indeed caused by the first move [PSPdB96]. Enhanced alpha-beta algorithms like NegaScout [Rei83] and PVS [Mar83] that employ null window search, take advantage of the fact the moves are ordered such that good moves are more likely to be searched first. The algorithms have been shown, both theoretically [RM87] and empirically [Sch89] to be more efficient than plain alpha-beta.

3 Uncertain cut-offs - Idea

Current tree search algorithms equipped with various search enhancements are searching quite efficiently. But there is still some scope for improvement. Search overhead from imperfect move ordering can be introduced in two ways:

- at a CUT-node, the first move does not cause a cut-off, or
- at a PV-node, the first move is not the best.

Both these cases occur when there is uncertainty in the search - previous expectations are changing. In the first case additional moves must be searched until a move (if any) causes a cut-off. The sub-trees of the sibling nodes searched prior to the node that caused the cut-off have been searched unnecessarily. In figure 2 this search overhead is shown as the grayed sub-tree T_1. In the second case, assuming null window search is used, if a new best move is found it must be re-searched with a normal window. The search overhead here consists primarily of the initial null window search that failed high[2]. In figure 2 we assume that the third move searched at the root (i.e. c) failed high; the null window search that was performed (the grayed sub-tree of c in figure 2) is the search overhead and the sub-tree T_2 represents the necessary re-search. However, information stored in the transposition table during the null window search efficiently guides the re-search, saving some move generations and node expansions.

At CUT-nodes it is most important that the move which causes the cut-off be searched as soon as possible. To improve the prospect of choosing a move that will cause a cut-off, we make use of available move information (e.g. the transposition table entry and the history heuristic). However, *while searching the sub-tree of this move we might, based on other information, start to believe that this move will not cause a cut-off.* The question that we are then faced with is whether to continue searching this sub-branch, or stop searching it and start searching another candidate cut-off move nearer the root of the tree. Similarly at PV-nodes, when searching a child with a null window, where we believe the search will

not only a move that returns a value that is sufficient to cause a cut-off, *but* also one that has the smallest sub-tree.

[2]The search efficiency is also somewhat degraded because prior sibling nodes have been searched with an inappropriate window.

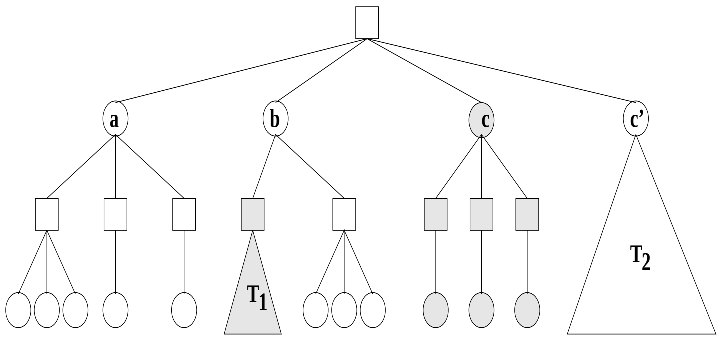

Figure 2: Search overhead.

fail high, we might choose to stop the search and start the re-search of the suspected new PV right away. This is the basic idea behind the pruning method introduced here. Instead of having only the two scenarios (either expanding all children of a node or having a child cause a cut-off) a third scenario is now also possible, where only some of a node's children are searched before we stop. This type of pruning shows some resemblance to the pruning done in null-window search. The artificial upper bound of the null-window enables early cut-offs at ALL-nodes, whereas the uncertain cut-offs cause early termination of CUT-nodes that start to behave like ALL-nodes.

To illustrate this idea in practice, let us look at the chess position in figure 3. Here it is

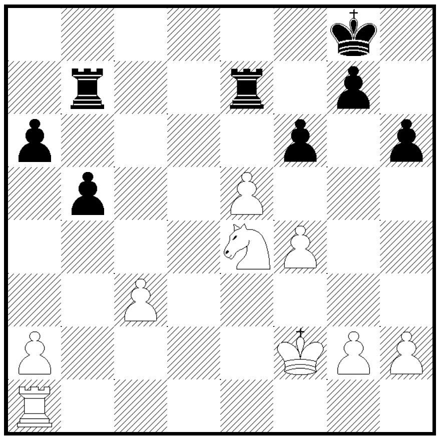

Figure 3: Example chess position.

White's turn to move. The pawn on $e5$ is being threatened but White has several possible continuations. Assume that White has already found a tentative principal variation and is now thinking of 1.$e6$ as an alternative move. Black's obvious reply 1....$R{\times}e6$ fails to 2.$Nc5$, putting a fork on both of Black's rooks. That was, however, outside the search horizon of the previous search iteration, so the search expands the move 1....$R{\times}e6$ first and White responds with 2.$Nc5$ (not necessarily the first move considered). In the resulting position Black is faced with the problem of saving the exchange. Black has over 25 legal moves but all of them fail to prevent White from winning the exchange. Instead of exhaustively searching all the possible legal moves, we can abandon the others after only a few have been examined, and start to look at alternative to 1....$R{\times}e6$. A better move is easily found (e.g. 1....$f5$) and, assuming the new move leads to a cut-off, then we save considerable search effort, but still return the same game value. The savings arise because the sequence $e6, R{\times}e6; Nc5,$ "$anything$" looks like a new PV. Rather than exploring all the alternatives for "anything" we assume that a new PV is indeed emerging and so retreat up the tree, using the current "uncertain value" as a bound, in the search of alternatives to $R{\times}e6$, where we quickly refute the candidate PV with $f5$.

4 Uncertainty cut-offs - Algorithm

Two fundamental questions must be addressed when implementing the above pruning method; how to guarantee that a correct game value is returned, and how to make decisions about when to apply the pruning method. These questions will now be addressed.

Let us assume that the pruning method described above is applied in the sub-tree T_1 in figure 2, such that the value backed up to the root of T_1 is not guaranteed to be the correct game value, i.e. the value is uncertain. The specially interesting case occurs when the value returned by T_1 does not cause a cut-off, but another child of b does. In that case the sub-tree T_1 is not a part of the minimal tree and any pruning made in there will not affect the game score in any way. The value returned by b will be the value returned by the move that caused the cut-off. Given that this value was not uncertain, then neither will the value returned by b. However, in cases where b fails low the value returned by b will be uncertain if *any* of its children's values are uncertain. If an uncertain value is backed up all the way to a PV-node, that node is re-searched. By keeping track of how uncertain values back up the tree, we know if the returned value is guaranteed to be the correct game value or not. Figure 4 shows the NegaScout algorithm with the uncertainty cut-offs code highlighted in bold-faced. Note also that we must be careful to mark uncertain nodes when inserting them into the transposition table, so that their re-use is restricted to suggesting the best move, and not to adjusting the search bounds or the search value.

The other fundamental question is where and when to apply the pruning heuristic. We cannot blindly apply the pruning everywhere in the tree, because this would result in frequent re-searching of uncertain nodes, resulting in the search overhead of the re-searches exceeding the gains of the pruning. Basically, we would like to prune only in sub-trees that are not likely to become a part of the minimal tree. What is needed is a good criteria for identifying these sub-trees. Typically what happens is that when we are searching on a path that is off the minimal tree, the characteristics of the search tree shape are different. Nodes that we expect to be CUT-nodes start to behave like ALL-nodes and vice verse. This can be seen in figure 2. For move c to fail high (and therefore is no longer a part of the minimal tree because of the re-search) it must be true that all children of c are searched and fail low.

```
int NS( int depth, int alpha, int beta )
{
  uncertain[depth] = FALSE;
  if ( depth == max_depth )
      return ( Evaluate() );

  if ( GetTT(&tt) ) {
      if ( tt.height+depth >= max_depth && !tt.uncertain ) {
         // Return score or update alpha/beta bounds
      }
      best_move = tt.best_move;
  }
  else NextMove(&best_move);

  MakeMove( best_move );
  best = -NS( depth+1, -beta, -alpha );
  UnmakeMove();
  uncertain[depth] = uncertain[depth+1];

  while ( (best < beta) && NextMove(&move) ) {
      if ( UncertaintyCutoff( depth, best ) ) {
         uncertain[depth] = TRUE;
         break;
      }
      MakeMove( move );
      alpha = MAX( alpha, best );
      score = -NS( depth+1, -(alpha+1), -alpha );
      if ( (score > alpha) && (score < beta) )
         score = -NS( depth+1, -beta, -alpha );
      if ( score > best ) {
         best = score;
         best_move = move;
      }
      UnmakeMove();
      if ( best < beta )
         uncertain[depth] = uncertain[depth] || uncertain[depth+1];
      else
         uncertain[depth] = uncertain[depth+1];
  }
  PutTT( best_move, best, max_depth-depth, uncertain[depth] );
  return ( best );
}
```

Figure 4: Uncertainty cut-off pruning.

Because of the move ordering, the moves that are most likely to cause a cut-off are searched first, but if none of the promising cut-off candidate moves causes a cut-off we have a good reason to believe that the rest of the moves will also fail to do so. Therefore, after searching only some of the possible moves at c we may decide not to search the rest, i.e. we make an uncertainty cut-off, and return right away. This will cause node c to be re-searched. The criteria used here to decide when to apply the pruning is as follows: *if during a null-window search, a node that is expected to be a CUT-node does not cause a cut-off after searching n-moves, then the rest of the moves are ignored.* The number of moves to look at in each position, n, can be determinated in various ways, for example a fixed percent of legal moves could be searched. But some more dynamic measure might be better.

```
int UncertaintyCutoff( int depth, int best )
{
  // This function assumes the existence of a global structure
  // SearchInfo[MAX_DEPTH] that stores various information about
  // the current search path.

  int move_limit = SearchInfo[depth].no_moves * CUT_RATIO;

  if ( ( depth > 0 )
      && ( SearchInfo[depth].move_no > move_limit )
      && ( SearchInfo[depth].type == CUT_NODE )
      && ( SearchInfo[depth-1].type == PV_NODE )
      && ( -best > SearchInfo[depth-1].alpha )
      && ( -best < SearchInfo[depth-1].beta ) )
  {
      // also check if a non-capture move
      return ( TRUE );
  }

  return ( FALSE );
}
```

Figure 5: Pruning decision.

5 Experimental Results

The new pruning method was implemented in *The Turk*[3], an experimental chess program. The program's search engine uses the NegaScout algorithm and also includes most search enhancements found in contemporary chess programs, such as search extensions, quiescence-search and a transpositions table. The move ordering scheme generates capture moves first (most valuable piece captures generated first), and the history heuristic is used to sort the remaining moves. The best move previously found in a position is stored in the transposition table and searched first where applicable.

The chess program, both with and without the new pruning technique, was tried on the Bratko-Kopec test. A normal 5-ply search is done, but with search extensions. Uncertainty cut-offs are only done at expected CUT-nodes. At such nodes a fixed percent of the legal moves are searched with the proviso that all capture moves are always searched. In the current experiment the pruning was further limited to CUT-nodes that are children of PV-nodes, and pruning is done if and only if the value returned would result in the node being re-searched[4]. The coding of the pruning constraints is shown in figure 5. The function assumes the existence of a global structure *SearchInfo* that stores information about the path being explored. The possible savings here correspond to minimizing the second type of search overhead described earlier, i.e. stopping a null-window search when we expect it to

[3] *The Turk* was developed at the University of Alberta by Yngvi Bjornsson and Andreas Junghanns.

[4] Note that because of how restricted these particular cut-off constraints are there is indeed no need to back up uncertainty information. This is because the pruning is only applied if it is guaranteed that the sub-branch will be re-searched. The value, even though being uncertain, will therefore always be corrected in the re-search before propagating further up the tree. This is however a special case, in general the uncertain information must be kept.

fail-high anyway.

	10%	30%	50%		10%	30%	50%
1	89	91	87	13	103	104	102
2	106	100	101	14	92	94	94
3	108	99	98	15	101	111	100
4	97	98	98	16	98	96	96
5	103	101	99	17	100	100	100
6	100	90	88	18	105	104	97
7	101	100	100	19	102	100	100
8	102	100	100	20	113	99	99
9	105	100	100	21	100	99	99
10	104	100	100	22	117	104	94
11	96	89	88	23	108	100	86
12	101	100	100	24	101	99	99
				Total %	104	100	97
				Avg. %	102	99	97

Table 1: Percentages of nodes searched.

Table 1 shows the result of the experiment by displaying the percentage of nodes searched with the program using the new pruning method compared to the same program without the pruning. The total savings are shown both as a percentage of nodes searched in total, as well as the average percent saving over all the positions. The average number might be more representative because the search effort differs widely from one position to another. As can been seen from the table, when we apply the pruning aggressively–cutting-off after searching only 10% of legal moves–the total number of nodes searched is bigger than nodes seen without the pruning. This is because we are making too many wrong pruning decisions. However, by searching a larger percentage of the nodes we get some savings. When searching 50% of the moves at CUT-nodes the total saving in number of nodes searched is about 3%, and for some of the positions the savings are considerable (between 10-15%). It is interesting to note that only in a few cases does the pruning cause more nodes to be searched, and then only marginally. The savings did not improve further when searching over 50% of the legal moves and tapers to zero as a larger move percentile is searched. Even though the data here are presented as savings in nodes searched, the run time overhead with the method is negligible so the search time results are in the same ratio.

The above experiment only applies to a limited case of the more general pruning mechanism described. Some preliminary experiments have been run where we allow pruning at all CUT-nodes. Even though there is more potential for savings, the pruning decisions are also more prone to error. The experiments indicate that a more sophisticated scheme is needed. The history heuristic does a good job in getting the few good moves at the head of the move-list, but for the most part it does not give very good ordering for the remaining moves. A criteria, possibly based on additional domain knowledge, is needed to classify/order these moves better, such that moves with little or no potential are considered last.

In "stable" positions where PV-changes only occur infrequently there is little benefit in using the uncertainty cut-off. However, in highly dynamic positions where the search is more uncertain, big savings are possible. These positions are often the most critical ones in the game, so savings there are especially beneficial.

6 Extending the Pruning

Another interesting application of this method, only briefly addressed here, is to accept uncertain values at the root and make move decision based on them. Given some certainty measurement of the returned game value, e.g. bound on the error or some probability of the value being correct, one might decide to make a move at the root even though it has not been proven to be absolutely the best one. The time saved by doing the pruning can be used to search deeper and therefore, hopefully, increasing the overall accuracy of the search.

In the above experiments the pruning was only applied in very restricted cases, i.e. in sub-branches that we truly expected not to be a part of the minimal tree. Obviously this limits the possible benefits of the pruning because existing algorithms are already searching quite close to the minimal tree. Another possibility is to apply the pruning more widely in the search tree, even though it means that an uncertain value may be backup up to the root. For example, at ALL-nodes we might decide to exclude from the search moves that we believe are of no relevance whatsoever. The move ordering makes it more likely that the best move is searched early on, so we could partially base the decision on how many moves have been searched in the position. In figure 6 we see a distribution of where in the move-list the best move is on average at ALL-nodes. The data is based on 5-ply searches on the Bratko-Kopec test using *The Turk*[5]. As can been seen, there is a high probability that the best move is early in the move-list. However, because of the risk involved, a more complex domain-dependent pruning mechanism is likely necessary to decide where and when to prune. Instead of maintaining a Boolean variable indicating if a game value is uncertain or not, a probability-based value could be maintained that is associated with the certainty of the returned value.

7 Conclusions

Preliminary experiments indicate that this pruning technique can improve search efficiency. Even though the savings are small, the experiments still show some potential for the method. Because move ordering in chess programs is already very good, this technique does not make much difference in most cases, but it can significantly decrease the search effort when our

[5]ALL-nodes with less than 10 legal possible moves were not included in the data.

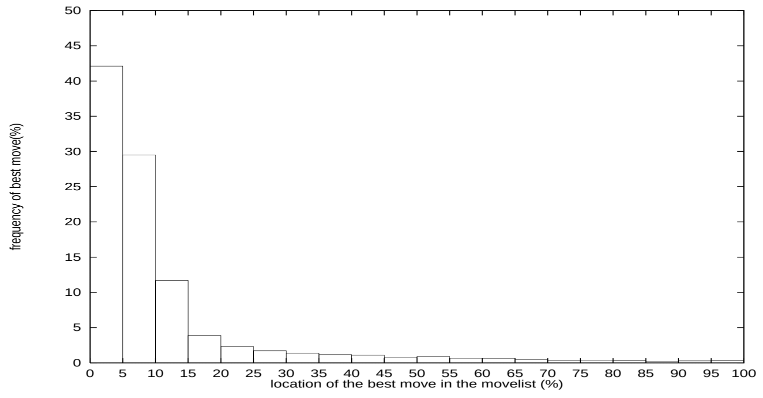

Figure 6: Distribution of the best move location at ALL-nodes.

move ordering heuristic fails. Also program run-time overhead of embedding this pruning method is negligible, so the savings come almost at no cost. In the experiments a very simplistic cut-off criteria was used. There is still room for improvement:

- Instead of cutting after searching fixed % of legal moves, this parameter could vary from place to place in the tree, based on both various positional features and information about the search tree.
- Use an improved move ordering scheme that sorts moves with little potential to the end of the list.
- Extend this pruning method to prune also at ALL-nodes, allowing for uncertain values to back up all the way up to the root. This allows for much greater node savings at the cost of risking the game value.

Further experiments are needed to reach a final verdict. The experiments here are only the first step in exploiting the potential of this pruning method. The improvements mentioned above will be tried, and the idea tested in other two-person games.

References

[KM75] Donald E. Knuth and Ronald W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[Mar83] 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.

[PSPdB96] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. Exploiting graph properties of a state space. In *AAAI*, 1996. To appear.

[Rei83] A. Reinefeld. An improvement of the Scout tree search algorithm. *ICCA Journal*, 6(4):4–14, 1983.

[RM87] Alexander Reinefeld and T.A. Marsland. A quantitative analysis of minimal window search. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-87)*, pages 951–953, 1987.

[Sch89] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(1):1203–1212, 1989.