PRE-SEARCHING

Markian Hlynka[1] *and Jonathan Schaeffer*[2]

Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2E8

ABSTRACT

This paper introduces the idea of *pre-searching* a position—searching it before $\alpha\beta$ determines that it needs to be searched. Consider the case where an iteration of $\alpha\beta$ search comes across a position p that needs to be searched to depth $d1$, and the transposition table reveals that p will likely need to be searched later in that iteration to depth $d2$ greater than $d1$. Pre-searching involves speculatively searching p to depth $d2$, even though it has not been demonstrated that search to this depth is needed. If the gamble is successful, then additional accuracy is introduced to the search (using a result with extra search depth $d2 - d1$). While any search extension scheme is not without some risk, empirical results indicate that the impact on the $\alpha\beta$ search tree size is small, but the additional accuracy that is introduced improves program performance.

1. INTRODUCTION

As part of a project to better understand the nature of game-tree search, the $\alpha\beta$ search algorithm was instrumented to generate numerous informative statistics that can be used to characterize two-player search trees. These characterizations will form part of the data provided to learning algorithms. These learners will in turn be designed to further enhance $\alpha\beta$ search through self-exploration, learning, and acquired knowledge.

A better understanding of the dynamics of a search tree can lead to the development of new search enhancements, as seen, for example, with the development of Enhanced Transposition Table cutoffs (Plaat *et al.*, 1996). One search characteristic that statistics were gathered on was the prevalence of cycles and transpositions in a tree. The experimental data suggested that the search trees were bigger than they had to be—some positions were being searched several times on the same iteration. This can occur for several reasons. Consider coming across position p twice in the same iteration. The first time p is encountered, it needs to be searched to depth $d1$; the second time to depth $d2$. The repeated position is detected using the transposition table and one of three situations arises:

- $d1 == d2$: The first search of p should be sufficient for the second search of p. However, the search windows used for the two searches may be different. Consequently, the value returned by the first search may not be sufficient to cause a transposition table cutoff for the second search.
- $d1 > d2$: The first occurrence of p will be searched deep enough that its value can be reused for the second search (modulo the search window bounds on the value from the transposition table).
- $d1 < d2$: Position p is searched to depth $d1$, only to discover later on in the iteration that p needs to be re-searched to the deeper depth $d2$.

This paper addresses the case where $d1 < d2$. We introduce the notion of searching a position to a depth greater than is required in anticipation that the deeper search depth will be needed later in the iteration. The idea is to reduce the frequency of duplicate search effort, while at the same time introducing greater heuristic accuracy into the search (replacing the depth $d1$ result with the more accurate depth $d2$ result).

[1]email: markian@cs.ualberta.ca
[2]email: jonathan@cs.ualberta.ca

The idea of *pre-searching* a position is captured in our pre-search enhancement to the $\alpha\beta$ algorithm. We show that pre-search might introduce a small overhead to the search, but can improve the number of problems solved. Pre-searching is evaluated in a chess and a checkers program. Although the performance improvements are modest, the major contribution of this paper is a new way of looking at $\alpha\beta$ search. In much the same way as other novel $\alpha\beta$ enhancements showed only modest initial promise (e.g. uncertainty cutoffs (Björnsson, 2002)), looking at the tree search from a different point of view can be the stepping stone for new creative ways of getting the maximum search value from given search resources.

2. PRE-SEARCHING

Consider an $\alpha\beta$ search, where the same position is reached in different parts of the search tree. In Figure 1, we see the search tree represented by the large triangle. By convention, search proceeds from top to bottom, left to right (depth-first search). When the search reaches the node marked A, it will explore the subtree underneath this node represented by the triangle with its apex at A. Suppose this subtree has a height of $d2$ and the search returns a value of $v2$. Position A and the search information $d2$ and $v2$ will be saved in the transposition table and the search continues. Later in the search, position B is encountered, and B is a transposition of A. The transposition table is consulted, and the repeated position detected. Since B's search depth $d1$ is $<= d2$, the search result of A can be used (depending on the value $v2$ and its relation to the search window of B). In this scenario, the deep search occurs before the shallow search.

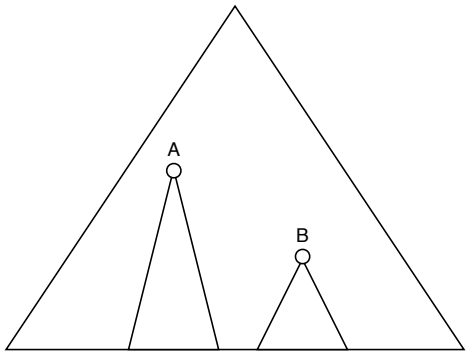

Figure 1: Deep search first

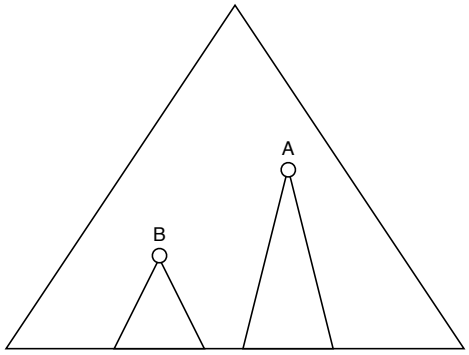

Figure 2: Shallow search first

Figure 1 is the ideal situation—the deep search precedes the shallow search. However, the situation in Figure 2 can just as easily occur. Now B is encountered first and is searched to depth $d1$. When A is reached, the transposition table discovers that the position needs to be re-searched since $d1 < d2$. In this scenario, the search of B was redundant[3].

[3]There may be a benefit to searching B, such as an improved search window. However, the search must be repeated to a greater depth which will give both a better search window *and* greater probability of a cut later on. It is in this light that we use the term redundant.

The pre-search enhancement recognizes when the situation shown in Figure 2 occurs. Pre-search will speculatively search B to depth $d2$ (not $d1$ as $\alpha\beta$ would normally do) in the expectation that this search result will be re-used when it comes time to search A.

We will refer to the node requiring the shallow search (B in Figure 2) as the shallow node, and the node requiring the deeper search (A in Figure 2) as the deep node.

There are several benefits to pre-searching shallow nodes:

1. Search efficiency might be improved by the elimination of redundant shallow-node searches. In Figure 2 pre-searching B to depth $d2$ means that the shallow search to depth $d1$ is not done.
2. Search accuracy will improve since shallow nodes are searched to the depth of deep nodes. In Figure 2, B's search value will be the result of a $d2 > d1$ search, improving the quality of the backed-up heuristic score for this subtree.

One important case where pre-searching can make a qualitative difference is by helping to alleviate some horizon effect (Berliner, 1974) problems. Analysis of a line of play may have the win pushed beyond the search horizon by the interjection of postponing moves by the opponent. In some cases, these postponing moves are merely a diversion, and the program's analysis eventually leads back to the main line of play. In this case, pre-search is of great assistance. Positions in the main line of play will have been analyzed without the effects of the postponing moves. In the line of play with the postponing moves, pre-search will search these positions to at least the same depth as in the main line of play. Hence, in some cases, the postponing moves have no/little effect.

Unfortunately, pre-searching is speculative and, hence, there are risks associated with doing additional search before it is actually proven to be needed. Pre-search anticipates that a deeper search is needed, and so it does the search now rather than later. But the use of the pre-search result may not happen; the $\alpha\beta$ search may end up cutting off the second search. In Figure 2, there is no guarantee that the $\alpha\beta$ search will reach A; a cut-off may occur, or the move ordering might change. In this case, B has been searched deeper than was necessary, without achieving the immediate savings expected by reusing it at A (savings may still occur, as the deeper result may be useful on the next iteration of an iterative-deepening search).

Nevertheless, the potential to enhance the search in this speculative manner is alluring, and with careful tuning the benefits may range from smaller search trees to more accurate results.

2.1 The Pre-search $\alpha\beta$ Enhancement

Figure 3 gives the pseudo-code for adding the pre-search enhancement to the $\alpha\beta$ algorithm. The assumption is that $\alpha\beta$ has been enhanced with both iterative deepening and transposition tables. For simplicity, fixed-depth searches are assumed, but the code can be easily extended to handle variable-depth searches.

The code illustrates how pre-search works. Transposition tables are used to identify when a pre-search is possible. The table is augmented to record the iteration (IDDepth) at which a position is encountered. This is used by pre-search to know whether a position has been seen in the current iteration. When a position is encountered in the search with:

1. a transposition table entry from the previous iteration, and
2. a search depth less than that of what is in the transposition table,

then a shallow node has been found and is a candidate for a pre-search.

Adding search extensions to $\alpha\beta$ is always a dangerous proposition. Unless the extensions are kept in check, the search effort could increase unreasonably. Hence, in our implementation, we use four conditions to limit when pre-searching is initiated. These conditions are detailed in Figure 4 (`presearchFilter()` routine):

- Recursive: Conceptually, there is nothing wrong with using pre-searches within pre-searches. In our experience, however, allowing recursive pre-searching frequently resulted in searches being extended to unreasonable search depths (increasing the cost of the speculative search).

```
// Assumptions:
// * search depth counts down to 0 (leaf node)
// * IDDepth is the iterative deepening search depth
// * searches are to a fixed depth
int AlphaBeta( state, depth, alpha, beta, presearch ) {
  if( TerminalNode( state ) || depth == 0 )
    return( Evaluate( state ) );

  save_alpha = alpha;
  save_beta = beta;
  tt_found = TTLook( state, &tt_depth, &tt_value,
                           &tt_bound, &tt_iteration );
  if( tt_found == TRUE && tt_depth >= depth ) {
    if( tt_bound == LOWER ) alpha = MAX( alpha, tt_score );
    if( tt_bound == UPPER )  beta = MIN( beta, tt_score );
    if( tt_bound == ACCURATE ) alpha = beta = tt_bound;
    if( alpha >= beta ) return( alpha );
  }

  // Check for pre-search conditions
  if( tt_found == TRUE &&         // node is in TT
      tt_iteration != IDDepth &&  // not updated this iteration
      // Check that entry has deeper search needs
      tt_depth + ( IDDepth - tt_iteration) >= depth ) {
    // Found a shallow node
    new_depth = tt_depth + ( IDDepth - tt_iteration );
    if( presearchFilter( newdepth, depth, tt_depth,
                         tt_iteration, presearch ) == YES ) {
       presearch = TRUE;
       depth = new_depth;
    }
  }

  score = -INFINITY;
  num_moves = Generate( state, moves[] );
  for( i = 1; i <= num_moves; i++ ) {
    result = -AlphaBeta( state.moves[i], depth-1,
                         -beta, -alpha, presearch );
    score = MAX( score, result );
    alpha = MAX( alpha, score );
    if( alpha >= beta )
      break;
  }
  // Normal TT update, but need to also save IDDepth
  TTUpdate( state, score, save_alpha, save_beta, depth, IDDepth );
  return( score );
}
```

Figure 3: Pre-search pseudo-code.

```
presearchFilter( new_depth, depth, tt_depth, tt_iteration, presearch ) {
  // Do not allow a pre-search in a pre-search
  if( presearch == YES ) return( NO );

  // Ensure that we are searching deeper
  if( new_depth <= depth ) return( NO );

  // Limit the amount of the search extension
  if( new_depth - depth > Risk ) return( NO );

  // Prefer deep nodes close to the root
  if( tt_iteration - tt_depth > NearRoot ) return( NO );

  // Prefer shallow nodes that are not close to the leaves
  if( depth < NearLeaf ) return( NO );

  // Pre-search
  return( YES );
}
```

Figure 4: Pre-search filter.

- `NearRoot`: Having the deep node close to the root of the search tree is desirable. The further the deep node is from the root, the greater the risk that it will be cut-off by $\alpha\beta$ (meaning that the shallow node's search was possibly extended unnecessarily).
- `NearLeaf`: Having the shallow node be a comfortable distance from the leaves of the search tree is desirable. The closer the shallow node is to the leaves, the more frequent pre-searching will apply and the increased risk that deep searches will be done for little benefit.
- `Risk`: The increase in search depth for a pre-search search can be limited. Consider having to do a 2-ply shallow search that gets extended to 10-ply to meet the needs of the deep search. This is a huge investment of search effort. If pre-searching fails, then a substantial amount of work has been invested in improving the accuracy of a subtree that has a low chance of impacting the search result.

The decision to use or not use these restrictions is application dependent.

3. EXPERIMENTAL RESULTS

The pre-search enhancement was added to THE TURK, a chess program developed at the University of Alberta by Yngvi Björnsson and Andreas Junghanns[4] (Björnsson, 2002). THE TURK has all the standard $\alpha\beta$ search enhancements. To better measure the impact of pre-searching, search extensions and reductions were disabled.

Two data sets were chosen for testing the effectiveness of pre-searching: The Encyclopedia of Chess Endgames Volume 3 (Informant, 2005a) (1,797 positions) and The Encyclopedia of Chess Midgames (Informant, 2005b) (879 positions). These databases will be referred to as the 'endgames' and 'midgames' test sets, respectively

Pre-experiment predictions were that pre-searching would be much more effective in the endgame than in the middle game. Endgame positions typically have more transpositions than middle games. As well, endgame searches often search considerably deeper than middlegame searches.

Experiments were performed on a 2.0 GHz dual processor AMD Athlon running Red Hat Linux release 9, kernel 2.4.20-13. Our first experiments had all positions searched to a fixed depth. This did not give us meaningful data, since there was a huge variation in the time taken to complete these searches (varying from a second to several hours). Instead we tried to approximate the searches done by tournament programs. All searches were performed to at least 4,000,000 nodes (unless a forced mate was found). When 4,000,000 nodes were reached, the current search iteration was allowed to complete. This methodology simplified the comparison of two implementations by eliminating partially completed iterations.

We experimented with numerous combinations of parameter settings. We report one set of experiments in detail,[5] as they give a good indication of the potential for pre-searching. The experiments reported in this paper used the following parameter settings: no recursive pre-searches, `NearRoot` = ∞, `NearLeaf` = 3, and `Risk` = 2.

[4] The program competed in the 1995 World Micro-Computer Chess Championship.

[5] A few examples of the effect of parameter changes are given at the end of this section.

Our first result was disappointing: the midgame tests showed no benefit to pre-searching. Search tree size with pre-searching enabled was comparable to that without pre-searching, but there was no difference in the number of problems solved by each program. This result is not too surprising since the middlegame positions are generally complex, limiting the depth of search achievable (to an average of 9) given the experiment's resource limitations.

The endgame experiments, however, were more interesting. Here the search depths are larger (13 on average) and the frequency of transpositions considerably greater. On the endgame test set, pre-search resulted in reducing the search 26.4% of the time (by an average of 8.2% nodes), while increasing it 73.6% of the time (by an average of 4.5%).

For some depths, pre-searching results in as many as 8 more problems being solved. In many cases, the extra heuristic search accuracy introduced by pre-searching results in the correct move being found an iteration (or more) before the non-pre-searching version. In a particularly spectacular example, pre-search found a correct answer at a search depth of 13, having expanded roughly 1.3 million nodes, while the non-pre-search version did not find the solution at that depth (but expanded 6.4 million nodes).

A difference of 8 out of 1,797 positions may not initially sound encouraging. However, this improved accuracy comes almost for free. Further, the results reported here are for a very restricted version of pre-searching. The benefits are more dramatic if, for example, one increases the risk threshold. Increasing the risk threshold to 3, for example, results in a peak of 12 more problems solved, leveling off at 8 at the maximum depth. However, parameter tuning is finicky. Increasing the risk to 4 results in presearch solving fewer problems. However, in such cases the additional search effort thus induced requires tuning of the `presearchFilter` parameters to keep the search extensions in line. This tuning is application specific, and we did not feel comfortable reporting a result that was tuned to a specific program.

Pre-searching has been implemented in the checkers program CHINOOK (Schaeffer, 1997). A test set of 535 positions were used, randomly chosen from a set of positions that had been solved using proof-number search (part of the effort to solve the game of checkers). Two experiments were performed, varying the `NearLeaf` parameter. Searches were done using $\alpha\beta$ to 21 ply with no search extensions. The first experiment resulted in a 1% reduction in search nodes examined, while the second experiment saved 4%. Obviously the node reductions are small, but the very fact that they are reductions is very encouraging. With a 21-ply alpha-beta search, only 123 of the positions could be solved. Of these, experiment 1 was able to solve 6 of them at an earlier iteration, while experiment 2 solved 1 earlier. Clearly, there are trade-offs to be had in the setting of the filter parameters, and this is application dependent.

An experiment was performed with search extensions enabled. The results have to be taken in context, since the search extensions strongly depend on factors that are difficult to control, including move ordering and the search window. The number of problems solved within the depth limit almost doubled to 232 (out of 535), clearly showing the value of the search extensions. On the test set, 13 positions were solved in an earlier iteration while one position was solved in a later iteration. In this experiment, presearch ended up examining 10% additional nodes.

4. CONCLUSIONS

A new method has been presented for speculatively searching a node in advance of when the search result is required. While performing such a pre-search is a gamble, results show that generally pre-search adds little overhead to a high-performance $\alpha\beta$ search algorithm. Future work should include a more systematic test of the pre-search parameters and how they are influenced by the $\alpha\beta$ enhancements (especially search extensions). With the proper tuning of the pre-search parameters, this new enhancement can outperform ordinary $\alpha\beta$ search, sometimes by a significant margin.

ACKNOWLEDGMENTS

This research was supported by grants from the Natural Sciences and Engineering Research Council of Canada (NSERC) and Alberta's Informatics Circle of Research Excellence (iCORE).

5. REFERENCES

Berliner, H. J. (1974). *Chess as Problem Solving*. Ph.D. thesis, Carnegie-Mellon University.

Björnsson, Y. (2002). *Selective Depth-First Game-Tree Search*. Ph.D. thesis, University of Alberta.

Informant, C. (2005a). *The Encyclopedia of Chess Endgames*, Vol. 3. Chess Informant.

Informant, C. (2005b). *The Encyclopedia of Chess Midgames*. Chess Informant.

Plaat, A., Schaeffer, J., Bruin, A. de, and Pijls, W. (1996). Exploiting Graph Properties of Game Trees. *American Association for Artificial Intelligence National Conference*, pp. 234–239.

Schaeffer, J. (1997). *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer-Verlag.