INFORMATION ACQUISITION IN MINIMAL WINDOW SEARCH

Alexander Reinefeld t
Jonathan Schaeffer
T.A. Marsland ft

Computing Science Department,
University of Alberta,
Edmonton,
Canada TOG 2111

ABSTRACT

The alpha-beta tree search algorithm can be improved through the use of minimal windows. Branches are searched with a minimal window [α,α+l] with the expectancy that this will show the sub-tree to be inferior. If not, then that sub-tree must be re-searched. In this paper, several methods are discussed to minimize the cost of the re-search. Two new algorithms, INS and PNS, are introduced and their performance on practical trees is shown to be comparable to SSS*, but with considerably smaller overhead.

1. Introduction

The use of minimal windows [l] provides an improvement to the alpha-beta tree-searching algorithm (AB) [2], Minimal window search is based on the assumption that all subtrees are inferior to the best subtree searched thus far, until proven otherwise. Having searched the lirst branch with a full window [α,0], all remaining branches are searched with a minimal window [α,α+1], where a represents the best minimax value found so far. If the value returned is indeed < a, then our assumption was correct and the subtree is inferior. Otherwise, this subtree is superior and usually must be re-searched with a wider window.

The re-search idea originally appeared in Pearl's Scout algorithm [3]. Subsequently, there have been two generalizations, Principal Variation Search [4] and NegaSeout [5]. Figure 1 shows the NegaSeout (NS) algorithm for searching a tree of width *w* and depth *d*. If a node *p* is terminal, *Evaluate(p)* returns its value. For interior nodes, *Gencrate(p)* determines the *w* branches from *p*. Those branches whose minimal window search produces a better minimax value of *v* usually must be re-searched. Only when $\alpha < v < 0$, and the remaining depth of search is greater than 2, is a re-search with a window [v,0] necessary.

t Current address: Universitaet Hamburg, Farhbereich Informatik, Schlueterstr.70, D-2000 Hamburg 13, West-Germany.
tt Research reported here was supported in part through Canadian NSERC grants A7902 and E5722.

This paper introduces two new algorithms. Those use information acquired from the original search of a. subtree to minimize the cost of a possible re-search. Informed NegaScout (INS) uses all available information to generate the smallest possible trees, but does so with increased storage overhead. Partially Informed NegaScout (PNS) is a compromise between NS and INS. The performance of NS, PNS, and INS is compared with AB and SSS* [6,7]. INS searches trees of size comparable to those traversed by SSS[4]', but does so with lower overheads.

2. Information Acquisition

When the initial search of a subtree is performed, the values returned by each of the descendants is maintained in a tree-like data structure. This information can be used to minimize the cost of any necessary re-search, otherwise it is discarded.

```
FUNCTION NS (p. POS; α, β, depth: INTEGER):
                                        INTEGER;
VAR i, v, a, b : INTEGER;
    succ : array [1..w] of POS;

BEGIN
  IF depth = 0 THEN        { Terminal node? }
    RETURN (Evaluate (p));
  succ := Generate (p);    { Generate succ. }
  a := -∞;
  b :=  β;

  FOR i := 1 TO w DO BEGIN
    v := -NS (succ[i], -b, -MAX(α, a), depth-1);

    IF v > a THEN          { Re-search needed? }
      IF i = 1 OR v ≤ α OR v ≥ β OR depth ≤ 2
      THEN                 { v is accurate enough }
        a := v
      ELSE                 { Re-search required }
        a := -NS (succ[i], -β, -v, depth-1);

    IF a ≥ β THEN RETURN (a);  { β cut-off }

    b := MAX (α, a) + 1; { Minimal window }
  END;
  RETURN (a);
END;
```

Figure 1. The NegaScout Algorithm (NS)

Information gathered from the initial search of the subtree is used on a re-search to allow two new types of cut-offs. Figure 2 illustrates the *ignore left* cut-off. In Figure 2a, the subtree has been searched with a minimal-window of [100,101]. The descendant *B* returned a value of 105 causing a normal beta cut-off. If at some future point it is necessary to re-search this subtree, descendant *A* need not be looked at again since it has already been shown to be inferior to *B,* Figure 2b.

Figure 3 illustrates the *prove best* cut-off. At these nodes, a beta cut-off has not occurred and all descendants have been examined. Each of the values returned is an *upper bound* on the subtree's true value, Figure 3a. If a re-search is necessary on this subtree, there are three things that can be done to minimize tree size. First of all, the branches can be re-ordered according to their values from the initial search. By sorting the branches in descending order of value, the branch with the highest upper bound (and therefore with the highest probability of being the root of the best subtree) is searched first. Figure 3b.

Secondly, since the initial value for each subtree represents an upper bound, the re-search can be done with a narrow window instead of a minimal-window. By doing this, no re-searches of re-searches can ever occur.

Finally, if the search of a subtree returns a true value that is greater than the upper bound of any of the other descendants, then those descendants can be discarded without any further work. For example, in Figure 3b, if move *B* is re-searched and returns a true value of 88, then moves *A,* and *C* need not be searched again, since their values can never exceed that of *B.*

It turns out that *ignore left* cut-offs are just a special case of *prove best* cut-offs. Branches proven inferior can be treated as having value - and the rest of the branches as having a +∞ value. Retrieving this information and performing a stable sort creates the *prove best* condition. The cut-offs are treated differently because in an actual implementation the *ignore left* cut-offs require less storage to maintain the necessary information, e.g. only the number i of the best descendant thus far need be saved. On a re-search, descendants 1 through i-l are ignored and the remainder searched. At *prove best* nodes, the values for all descendants must be saved.

3. Algorithms

NegaScout can be enhanced to use information from the initial search of a subtree to aid in any re-searches. Every time a node is visited, a record is kept of the results obtained from searching each descendant subtree. Either a beta cut-off occurs, and *ignore left* information is available for a re-search, or all descendants are examined, and

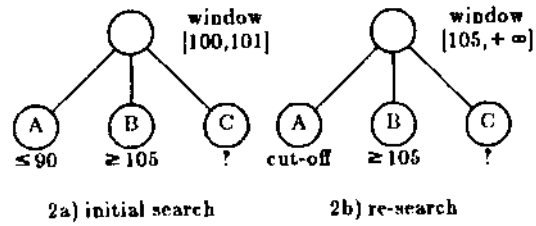

Figure 2. Ignore left cut-offs

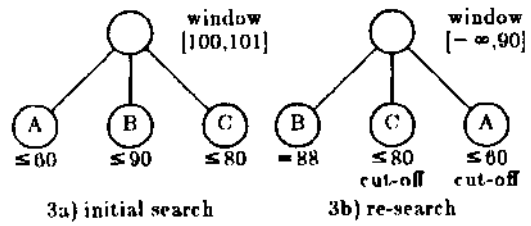

Figure 3. Prove best cut-offs

prove best information is available. In both cases, this information can be linked together to form a map of the subtree just searched. If a re-search is necessary, the map data can be used to achieve *ignore left* and *prove best* cut-offs that are not possible in NegaScout. Informed NegaScout (INS), see Appendix, does exactly this for all nodes in a tree.

The storage overhead in saving all this information is proportional to $\sum_{i}^{\lfloor \frac{d-1}{2} \rfloor} w^{*}$ entries, which is less than for SSS*. Nevertheless this may be too much, even if one reclaims storage whenever possible. As an alternative, Partially Informed NegaScout (PNS) has been implemented, providing a compromise between the complete information of INS and the zero information of NS. One can devise many different compromise algorithms, our version of PNS only retains information about *prove best* cut-offs near the root of each subtree and maintains the *principal variation,* the path to the terminal node that the initial search considered best. This algorithm tries to provide many of the benefits of INS without the storage overhead.

An important point to note is that the information used by INS is not a hash table or a transposition table [4). Whereas transposition tables are most useful in directed graphs, the methods described here are applicable to any tree structure and do not depend on the properties of the application.

4. Results

Figures 4 and 5 illustrate some results comparing AB, NS, PNS, INS, and SSS*. The number of leaves se are lied by each algorithm is normalized to the size of the minimal game tree [2]. Each data point represents an average over 20 runs. Random trees, where each descendent has an equal probability of being best, and strongly ordered trees [4], where the first descendent has a 60% probability of being the best, were searched by all algorithms. In Figure 5, data at depths 7 and 8 are not available for SSS* and INS, because of memory constraints.

The graphs provided, as well as results for other widths not reported here, show that the curves oscillate, with SSS* varying the least. The oscillation is normal and occurs because the formula for the minimal tree size depends on whether the tree is of even or odd depth. SSS* fluctuates least since it is a best first search algorithmt; the other algorithms are (partly) directional. For even search depths SSS* visits fewer nodes than INS, but for odd search depths INS is usually better, because INS postpones node expansion until it is proven that the principal variation lies in this part of the tree. Of course, minimal window techniques are favored in strongly ordered trees, since re-searches are less probable. Here even a modest amount of information is enough to allow PNS to outperform SSS* at odd depths.

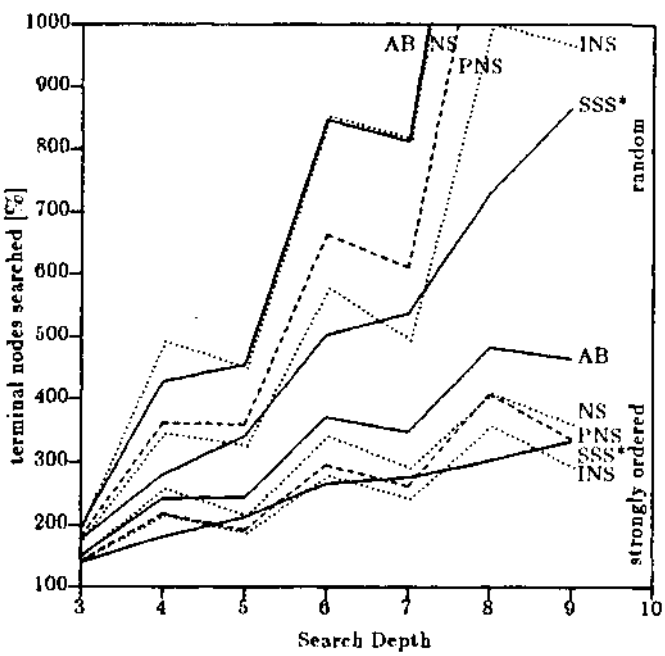

Figure 4. Comparison to minimal tree (w = 5)

Nodes visited is not the only consideration when comparing tree searching algorithms. SSS* and INS have significant overheads when compared to AB and NS. Obviously, any timing results are implementation dependent. Our experience is that a call to INS is, on the average, about twice as expensive as a call to AB, NS, or PNS, and that SSS* is 10 times slower than INS. Whether this overhead is significant or not depends on the application.

5. Conclusions

INS has been shown to be competitive with SSS* in terms of leaf nodes searched. However the data structure to support INS is more efficient. Not only is it slightly smaller, but it can be processed in one tenth the time required by SSS*. Perhaps more importantly, by storing the information acquired during the minimal window search in a hash table, rather than as a map of the re-search tree, the memory needs can be reduced to the space available. The cost for use of such reduced memory is increased search overhead, but the search time is bounded below by NS.

PNS represents a good compromise, yielding significant reductions in tree size with little time and space overhead. In our experience, PNS is the preferred algorithm for large trees, especially under conditions of well ordered interior nodes.

The results reported here show the relative properties of the algorithms. Experiments are continuing to obtain a better measure of the standard deviation. Current work includes empirical performance analyses of these algorithms in practical game playing programs.

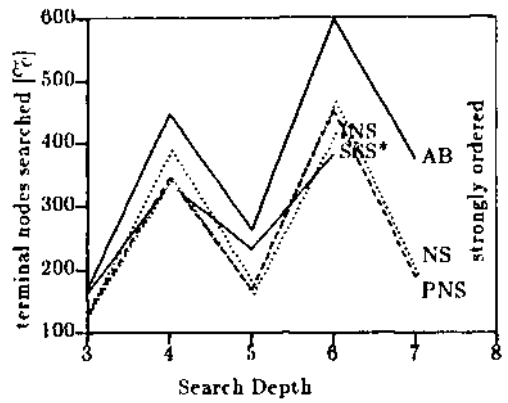

Figure 5. Comparison to minimal tree (w = 20)

† Sometimes the best first data misleads SSS* so that it expands more nodes than NS!

References

1. J.P. Fishburn, Analysis of speedup in distributed algorithms, University of Wisconsin, Tech. Rep. 431, Madison, May 1981.
2. D.E. Knuth and W. Moore, An analysis of alpha-beta pruning, *Artificial Intelligence 6*, (1975), 293-326.
3. J. Pearl, Asymptotic properties of minimax trees and game searching procedures, *Artificial Intelligence 14*, (1980), 113-138.
4. T.A. Marsland and M.S. Campbell, Parallel search of strongly ordered game trees, *ACM Computing Surveys 14,* 1 (11)82), 533-552.
5. A. Reinefeld, An improvement of the Seoul tree search algorithm, *ICC A Journal 6,* \ (1983), 1-H.
6. GO. Stockman, A minimax algorithm better than alpha-beta, *Artificial Intelligence 12,* 2 (1979), 179-196.
7. M. Campbell and T.A. Marsland, A comparison of minimax tree search algorithms, *Artificial Intelligence 20,* (1983), 347-367.

Appendix: Informed NegaScout (INS)

```
FUNCTION INS (p: POS; α, β, depth: INTEGER;
                          research: BOOLEAN): INTEGER;
VAR i, w, a, b: INTEGER; res: BOOLEAN;
    kind : (PROVE_BEST, IGNORE_LEFT);
    score: array [1..w] of INTEGER;
    succ : array [1..w] of POS;

BEGIN
  IF depth = 0 THEN RETURN (Evaluate(p)); { Terminal node?
  succ[] := Generate(p);                  { Generate successors }
  res := research;                        { Save research status }
  IF research THEN BEGIN
    kind := GetInfo (p, score[]);         { In research mode, }
    Sort (succ[], score[]);               { get scores and sort }
  END
  ELSE kind := PROVE_BEST;
  a := -∞;
  b := β;

  FOR i := 1 TO w DO BEGIN                { Start searching }
    v := -INS (succ[i], -b, -MAX(α, a), depth-1, res);

    IF v > a THEN                         { Re-search needed? }
      IF i = 1 OR v ≤ α OR v ≥ β OR depth ≤ 2 THEN
        a := v
      ELSE IF research AND kind = PROVE_BEST THEN
        a := v                            { Narrow window before }
      ELSE                                { Re-search required }
        a := -INS (succ[i], -β, -v, depth-1, YES);

    IF a ≥ β THEN BEGIN
      kind := IGNORE_LEFT;
      GOTO done; END;                     { β cut-off }

    IF res AND i < w THEN
      IF MAX (a, α) ≥ score[i+1] THEN BEGIN
        a := MAX (a, score[i+1]);
        GOTO done;                        { Prove-best cut-off }
      END ELSE
        b := score[i+1]                   { Set narrow window }
    ELSE
      b := MAX (α, a) + 1;                { Set minimal window }

    IF kind = IGNORE_LEFT THEN            { Only 1st subtree has }
      res := FALSE;                       { been seen before     }
  END;
done:
  IF NOT research THEN SaveInfo (p, kind, score[]);
  RETURN (a);
END;
```