Low Overhead Alternatives to SSS*

T.A. Marsland
Alexander Reinefeld
Jonathan Schaeffer

Computing Science Department
University of Alberta
Edmonton, T6G 2H1
Canada

ABSTRACT

Of the many minimax algorithms, SSS* is noteworthy because it usually searches the smallest game trees. Its success can be attributed to the accumulation and use of information acquired while traversing the tree. The main disadvantages of SSS* are its high storage needs and management costs. This paper describes a class of methods, based on the popular alpha-beta algorithm, that acquire and use information to guide a tree search. They retain a given search direction and yet are as good as SSS*, even while searching random trees. Further, although some of these new algorithms also require substantial storage, they are more flexible and can be programmed to use only the space available, at the cost of some degradation in performance.

Acknowledgement

Financial support from both the Natural Sciences and Engineering Research Council Grant A7902 and the Killam Exchange Scholarship Program, made it possible to complete the experimental work.

Revised edition for Artificial Intelligence. Generated from files dated 9 October 1986.

The present address of Alexander Reinefeld is: Fachbereich Informatik, Universitaet Hamburg, Schlueterstrasse 70, D-2000 Hamburg 13, West Germany.

Low Overhead Alternatives to SSS*

T.A. Marsland
Alexander Reinefeld
Jonathan Schaeffer

Computing Science Department
University of Alberta
Edmonton, T6G 2H1
Canada

1. Introduction

After more than a decade of use, the efficiency of the widely used **alpha-beta** ($\alpha\beta$) algorithm for searching game trees was questioned by Stockman's introduction of the **State Space Search** (SSS*) [1]. By saving information during the search, SSS* tries to expand subtrees in a *best-first* manner. Since the information maintenance entails significant overheads, SSS*'s application is restricted to the search of small trees. More recently, *minimal window* search techniques [2-4] have also been found superior to $\alpha\beta$ for applications [5] as well as for artificially constructed trees [6-8]. Minimal window search is more efficient than $\alpha\beta$ whenever the current subtree is inferior to the best subtree visited so far. If the current subtree is superior, it must be searched a second time to compute its correct value. Like $\alpha\beta$, minimal window search normally expands more nodes than SSS*. However, unlike $\alpha\beta$, minimal window search occasionally obtains the correct minimax value while traversing smaller trees than SSS*.

In this paper, new information acquisition methods to improve minimax search are presented. Minimal window search, modeled here by the **NegaScout** (NS) variant [7], is enhanced so that the initial search of a subtree gathers information for use if a second search is needed. The resultant algorithm will be referred to as **Informed NegaScout** (INS). It is compared to **Partially Informed NegaScout** (PNS) [9], a compromise algorithm that uses less storage.

A study of SSS*'s traversal of non-random trees shows that many nodes are stored, but

---

NegaScout is based on ideas from **Scout** [2] and **Palphabeta** (PAB) [3, 6]. It is equivalent to **Principal Variation Search** (PVS) [4, 5], but has some practical implementation advantages. On narrow trees all these algorithms have similar average performance [8].

subsequently are not expanded. This disadvantage is reduced by another algorithm, here called **DUAL***, which uses the *dual* of SSS* and hence incorporates some directional properties by doing a left to right search at the root. Thus, most subtrees are searched with a better bound than SSS* would use. Our experiments show that DUAL* often traverses smaller trees than SSS*, even in the random case. The performance of these algorithms is compared on both random and *strongly ordered* [4] trees of uniform width *w* and constant depth *d*. Experiments show that for odd-ply trees the new algorithms are comparable to SSS* in terms of nodes visited, and yet have significantly lower overheads.

2. Minimal Window Search

Minimal window search relies on being able to prove a subtree inferior, rather than on finding its true value. *Aspiration* or *narrow* window search [10] also employs this notion by seeking a value for a tree within tight limits. If these limits take on adjacent values, then one has a *zero-width* or minimal window. The minimax value, *v*, of a tree may be determined by invoking the NegaScout function as follows:

$$v = NS\ (p,\ \alpha,\ \beta,\ d);$$

where *p* represents the root position, (α,β) the *search window*, and *d* the remaining search depth. After the expansion of the first successor with an appropriate window (α,β), the others are traversed with the minimal window $(a,a+1)$, where *a* represents the best available *merit* (score) not less than α. Clearly, every minimal window search fails. If it fails low ($v \le a$), then the subtree is inferior and can be ignored. If the search fails high ($v > a$), it may be necessary to re-search the subtree with the opened window (v,β) to determine its exact value. No re-search is needed if $v \ge \beta$, since the cut-off value has already been achieved, nor if $d \le 2$ [7]. If the minimax value, *v*, of a tree were known, the most efficient aspiration $\alpha\beta$ search would use the narrow window $(v-1,v+1)$. All but one of the subtrees would then be refuted cheaply by using such a window. This raises the possibility of developing algorithms that scan the range of plausible values for the tree, successively moving a narrow window to eliminate subtrees until only one remains.

2.1. Refutation Wall

It is a well-known that the narrower the range of the search window, the smaller the tree that is traversed. What is not so well-known is how the tree size varies with the location of the window relative to the minimax value, *v*, of the tree. For a window of $(s,\ s+1)$, the node count partly depends on the difference $i = s - v$, which will be referred to as the *distance* to the minimal window. An experiment was conducted to explore this point, using twenty different random uniform trees of constant width *w* and depth *d*. Each tree had a known minimax value, *v*, and was searched fifty times using the following distinct windows

$$(v + i,\ v + i + 1),\ \text{ for } i = -25,\ -24,\ -23,\ ...,\ 23,\ 24$$

which covered all distances from -25 to +24. For each distance, the node count was averaged over the twenty trees. Fig. 1 shows two sample plots of average nodes visited (normalized to the largest value) versus distance. One graph is for a set of random uniform trees and the other for a set of strongly ordered trees.

As the distance of the window from the minimax value decreases, the node count increases slightly until a window of $(v-1,\ v)$ is reached. When the window moves to $(v,\ v+1)$ the node count rises abruptly! Fig. 1 shows that the better the tree order, the larger the increase. Thus it is easier to show that a tree has a value greater than the window bound, than to prove that it does not. In the latter case all immediate descendants of the root must be examined, while in the former case the search stops as soon as a value greater than *s* is found. The steep rise in the node count when the window reaches *v* will be referred to as the *refutation wall*. The step function shape of the wall is less pronounced in typical applications, but the consequences are just as important [11].

2.2. Ignore-Left and Prove-Best Cut-offs

Other than the current window, NegaScout does not retain information. If a re-search occurs, all nodes of the initial search are re-visited plus some additional ones. In an initial search, one piece of infor-

---

Redefined here so that the left-most descendant has a 60% chance of being best, otherwise the best is found with equal probability from the other $w-1$ siblings.

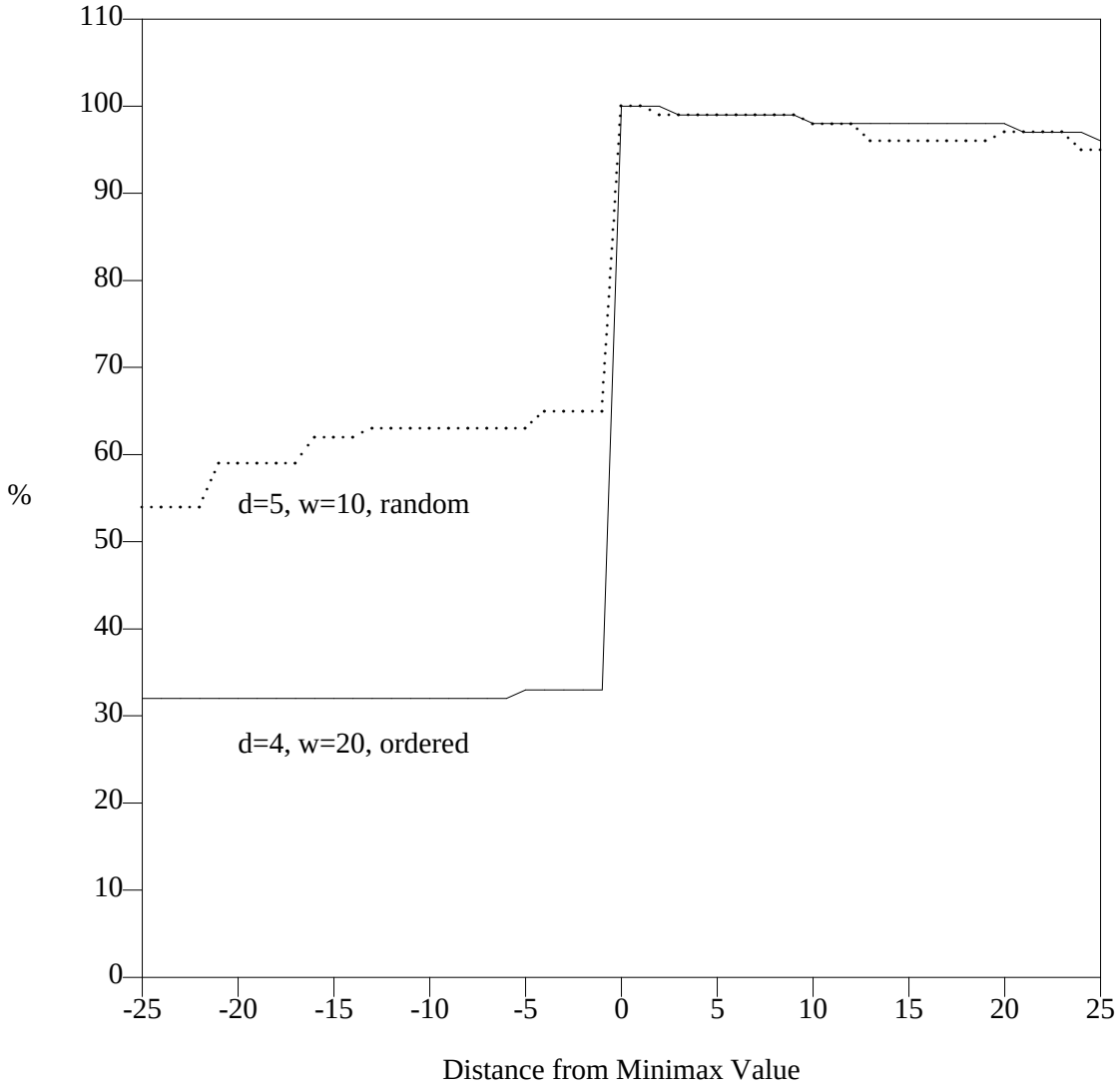
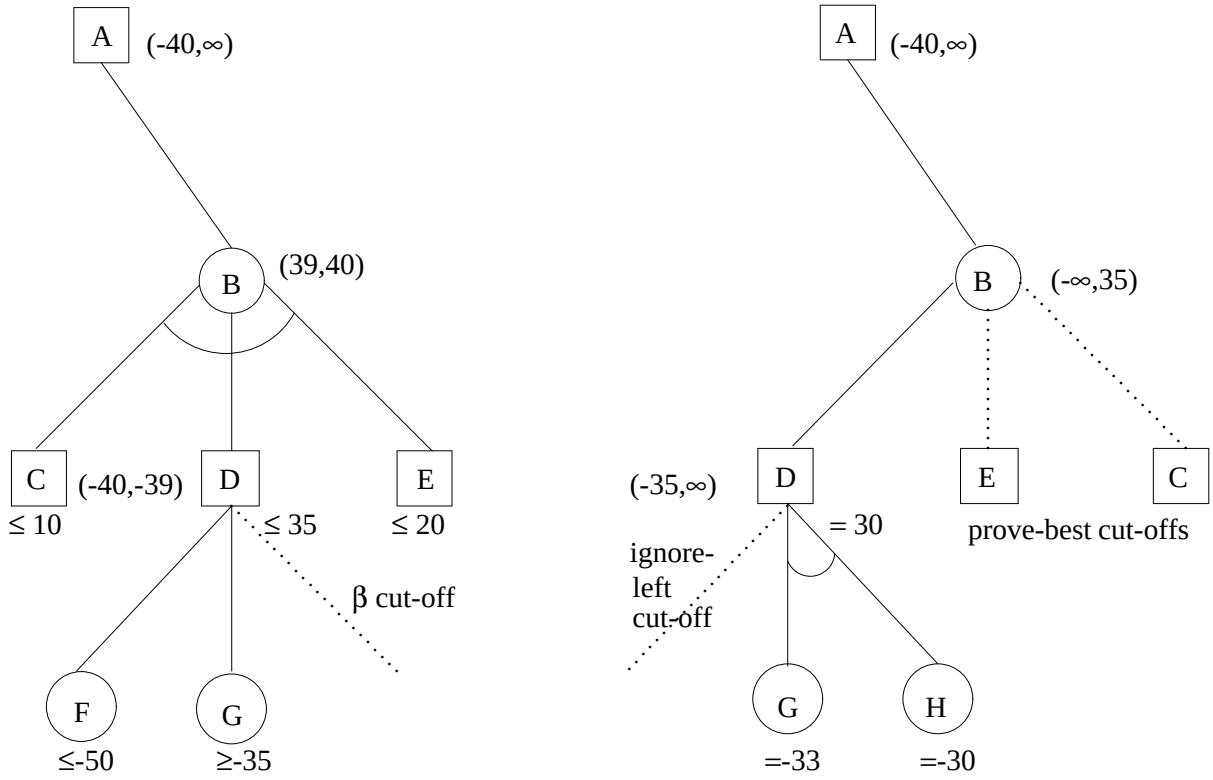


Fig. 1. Refutation Wall

Average % Node Count Relative to Largest Search

mation that is inexpensive to maintain is the *sub-variation* (path) to the leftmost terminal node that caused the failure. At even depths from the start of the re-search, all branches previously lying to the left of the sub-variation can be ignored. These branches have already been examined and shown to be inferior. Such an *ignore-left* cut-off is illustrated with the first successor of node D in Fig. 2(a). Node F cannot possibly return a better value, because it did not stop the initial search and therefore is ignored in Fig. 2(b).



(a) Initial Search **(b) Re-Search**

Fig. 2. Three cut-off types

Another piece of information that can be kept is the merit for each successor of nodes an odd depth away from the root of the re-search. At these nodes, such as node B in Fig. 2(a), a β cut-off has not occurred and the merits represent upper bounds on the exact values of the subtrees. This information can be used in three ways. First, the bounds can be used to re-order the successors, hence improving the probability that the best subtree will be seen sooner. For example, in Fig. 2(b), subtree D is re-expanded first and then E and C. Secondly, D will be proved superior to both E and C if a re-search of subtree D returns a value ≥ 20, since it is already known from the initial search that the value of E is ≤ 20 and of C is ≤ 10. Fig. 2(b) shows that this *prove-best* cut-off eliminates nodes E and C. If both D and E return values between 10 and 20, a prove-best cut-off discards node C. Finally, assuming a prove-best cut-off does not occur, the upper bounds can be used to narrow the window for a re-search. Node D can be searched with the narrow window (-35, ∞), perhaps returning the value $v_1 > -10$. Since v_1 is in the search window it is a

lower bound on the true merit, and so node E can be searched with the narrower window (-20, v_1). If this search returns the value v_2, then node C can use the window (-10, $max\{v_1, v_2\}$). Each time, the true value is guaranteed to lie inside the window and no further re-searches occur within a re-search.

2.3. Informed NegaScout (INS)

The basis for Informed NegaScout is the recursive saving of prove-best and ignore-left information at the nodes visited during the initial minimal window search, and using that information to guide the re-search. A description of the entire subtree generated by the initial search is saved in case a re-search is necessary. The *w* subtree values are retained for every odd subtree level where a prove-best cut-off might occur. Analogously, ignore-left information is saved for all intermediate (even) levels. This accounts, in a symmetrical way, for all possible ignore-left and prove-best cut-offs in all regions of the re-search tree.

Fig. 3. Informed NegaScout (INS).

An examination of the INS algorithm, presented in a C language pseudo code in Fig. 3, reveals that only β and prove-best cut-offs are recognized, illustrating that ignore-left cut-offs can also be treated as a special case of prove-best. One way of transforming an ignore-left cut-off into a prove-best one is to initialize to -∞ the merit of each inferior successor $p.1,..,\ p.i-1$, with the remaining nodes retaining a value of +∞. After the sorting operation, the successor list becomes $p.i,..,\ p.w,\ p.1,..,\ p.i-1$, and a simple prove-best cut-off trims the inferior nodes. In terms of storage requirements, however, a distinction should be made. In the initial search, at ignore-left nodes only the number *i* of the best successor need be saved, while at prove-best nodes the merits returned by all *w* successors are retained. These storage management issues have been hidden by the *SaveInfo* and *GetInfo* routines, which access and maintain the data structure used to gather information from the initial search.

Unlike SSS*, INS's ignore-left and prove-best cut-off information is such that it can be maintained in a hash table (similar to the transposition tables used in chess programs [4]). Thus INS's storage requirements can be tailored to the memory size of the system on which it is running. To obtain maximum cut-offs, all information must be retained and not lost through hash conflicts. For results presented here, a tree-like linked data structure was used instead of a hash table, ensuring that all information was retained

```
int INS ( p, α, β, depth, research )
position p; int depth, α, β, research;
{
   int i, v, a, b, kind, resflag;
   int merit[w], succ[w];                        /* Assume w successors */

   if ( depth == 0 )
      return ( Evaluate(p) );
   succ[]  = Generate (p);          /* returns move list with w>0 sons */
   resflag = research;                           /* save re-search flag */
   if ( research == TRUE ) {
      kind = GetInfo ( p, merit[] );       /* get merits of successors */
      Sort ( succ[], merit[] );                     /* and sort them */
   }
   else
      kind = PROVEBEST;

   a = -∞;
   b = β;                          /* use open window for first successor */
   for ( i=1; i≤w; i++ )
   {
      v = -INS ( succ[i], -b, -Max( a, α ), depth-1, resflag );
      if ( v > a )
         if ( i == 1 || v ≤ α || v ≥ β || depth ≤ 2 )
            a = v;
         else if ( research == TRUE && kind == PROVEBEST )
            a = v;                     /* searched with a narrow window */
         else                                          /* re-search */
            a = -INS ( succ[i], -β, -v, depth-1, TRUE );

      if ( a ≥ β ) {
         kind = IGNORELEFT;                                /* β cut-off */
         goto done;
      }
      if ( resflag == TRUE )                    /* omit case i == w too */
         if ( Max( a, α ) ≥ merit[i+1] ) {
            a = Max( a, merit[i+1] );            /* prove-best cut-off */
            kind = IGNORELEFT;
            goto done;
         }
         else
            b = merit[i+1];                            /* narrow window */
      else
         b = Max( a, α ) + 1;                         /* minimal window */

      if ( kind == IGNORELEFT )
         resflag = FALSE;     /* haven't seen right-most sons of node */
   }
done:
   if ( research == FALSE )           /* save info for later re-search */
      SaveInfo ( p, kind, merit[] );
   return ( a );
}
```

and maximum cut-offs were achieved. In addition, an integer array of size *w* is used at each level to hold the upper bounds of the prove-best cut-offs that are returned during the initial search. As a compromise, PNS only retains these upper bounds at the first level. Note also the essential differences between the memory scheme used by INS and the simple use of transposition tables in chess programs [4]. First the

INS method is general purpose and application independent, and second transposition tables are used only to guide the re-search down the principal variation. Although modifying transposition tables to provide ignore-left and prove-best cut-offs is a possibility, this would not be space efficient and is not at present being done.

3. The DUAL* Algorithm

SSS* is a powerful algorithm for searching random trees, even though it often does not use all the information it stores, especially in applications where trees are well-ordered. Simple examples can be constructed where the directional algorithm NS is better than SSS* [12]. Because directional searches have benefits, we present in Fig. 4 **DUAL***, a small variation on SSS* that does a left to right search at the root node of subtrees that are each fully expanded by the dual of SSS*. The dual of SSS* is formed from SSS* by exchanging the tests for MIN and MAX nodes, by doing a maximization instead of a minimization, and by changing the *insert* operation to maintain the OPEN list (stack) in increasing order. A similar routine was used by Kumar and Kanal in their proposed parallel algorithm for game tree search [13]. In their simulation *w* processes were used at the root node, each searching a game subtree with an algorithm called dual-SS*. Unfortunately they did not report on the effectiveness of their method for the one process case. Our results for DUAL* may be those missing data.

Although DUAL* would be the normal function to search a game tree rooted at a MIN node, we have taken here the unusual step of using it to search a MAX rooted tree. So, in contrast to SSS*, DUAL* does a strict left to right search at the root, ensuring that the right successors profit from bounds already established. This usage stems from our interest in forming new hybrid algorithms which capitalize on the even/oddness of the remaining search depth. Clearly, SSS* and DUAL* share the same node descriptor, which consists of a node identifier (*node*), the node status (*status* which is either *LIVE* or *SOLVED*) and the merit (*h*). Similarly, the OPEN list actions of *pop* (p, s, h) and *push* (p, s, h) in Fig. 4 are clear. Finally, the function *insert* (p, s, h) puts the node descriptor behind all nodes of equal merit and *purge* (p) deletes from OPEN all node descriptors which are successors of *p*.

Fig. 4. The DUAL* algorithm

```
int DUAL ( root, bound )                        /* bound is initially -∞ */
{
   push ( root, LIVE, bound );                          /* save root node */
   while ( true )
   {
      pop ( node, status, h );                /* restore node description */
      if ( status == LIVE )
      {                                                        /* Phase 1 */
         if ( node is a LEAFNODE )
            insert (node, SOLVED, Max (Evaluate(node), h));

         if ( node is a MAXNODE )              /* save first successor */
            push (node.1, LIVE, h);

         if ( node is a MINNODE )               /*  save all successors */
            for (j=w; j>0; j--)
               push (node.j, LIVE, h);
      }
      else                                      /* node status == SOLVED */
      {                                                        /* Phase 2 */
         if ( node == root )
            return( h );                                /* problem solved */

         if ( node is a MAXNODE )
         {
            purge ( parent(node) );      /* remove parent's successors */
            push ( parent(node), SOLVED, h);   /* save updated parent */
         }
         if ( node is a MINNODE )
            if ( node has an unexamined brother )
               push ( brother(node), LIVE, h );  /* save next sibling */
            else
               push ( parent(node), SOLVED, h );
      }
   }
}
```

4. Performance Comparison

There are several ways of comparing algorithms. For tree searching, the usual measure is bottom positions visited [14]. However, for practical purposes, this measure is often inadequate because it fails to take into account the time and space overheads. We detail those overheads in our report [12], and show that although SSS*, DUAL* and INS have decreasing space needs, their requirements are still exponential with depth. INS is the fastest and its simple node descriptor fits into less space. PNS and NS, on the other hand, are faster still and require negligible storage.

4.1. Terminal Node Visits

In practice, trees rarely have a random distribution of values at terminal nodes. More usually application dependent knowledge is available that allows the searching program to examine siblings of interior

nodes in order from most to least promising. The stronger the ordering, the smaller the trees that are built. Many papers on tree searching consider only the case of random trees. Although these give an adequate indication of the relative performance of the methods, they do not measure the true efficiency in typical applications. To assess tree-searching algorithms, it is necessary to consider their performance under differing conditions. These include varying the degree of ordering, as well as the tree width (w) and depth (d).

4.1.1. Generating Trees

Several approaches for generating trees have appeared in the literature [6, 8]. These methods usually suffer from inflexibility (for example, are restricted to the random tree case) or require excessive storage (the trees must be pre-computed). For our experiments a different method was developed [11], one which is both flexible with respect to ordering and uses only $O(wd)$ storage. It works on the principle of deriving the value of a subtree from information available at its parent node. Thus, the entire tree need not be generated to know its minimax value. Rather, a minimax value is chosen and used with the specified ordering criteria to build a tree from the root down, and by this means ensure that consistent values are assigned throughout the tree.

Initially, the user specifies w *weights* reflecting the probability that each of the w siblings at any node will be the root of the subtree having the minimax value. These weights determine the degree of ordering for the tree, and obviously must total 100. For a random tree each sibling has equal weight. Assigning a weight of 100 to the first sibling and zero to the remainder produces an optimal tree, since the left-most descendant is always best. In a more typical case, with exactly w=20 branches per node, the twenty weights (70,5,5,5,5,2/3,..,2/3) yield strongly ordered trees as they were originally defined [4].

Based on the pre-selected weights, at any node, r, a random number is used to select which of the w descendants will have the known high value, v. Initially this minimax value of the tree is computed randomly from the range -∞..∞. If descendant i is selected as best, the values of all siblings to the left of i are chosen randomly from the range -∞..v-1, and those to the right from -∞..v. These w values are stored on a stack in an array $V(k)$. Thus at any interior node, $r.k$, the value for that subtree is known from the stack as $V(k)$. The best successor j of $r.k$ is again randomly chosen according to the weights, and is

assigned the negative its parent's minimax value, i.e., $-V(k)$. This approach is applied recursively, generating a tree whose leaf values are everywhere consistent with the computed minimax value of the tree.

To be deterministic but varied, a unique *seed* is associated with each tree searched. This seed is multiplied by a function of the width and depth, so that *d*-ply trees are not treated as a proper subset of (*d*+1)-ply trees of the same uniform width. Hence, given the same *w*, *d*, *weights*, and *seed*, the same tree will always be generated. The merit of such simulations has been questioned because formulae exist to measure the performance of $\alpha\beta$ and SSS* on random trees [15]. The $\alpha\beta$ formula seems to show that simulations can sometimes be in error by as much as 30%. Such discrepancies illustrate the wide variation in random trees, so that even searches of a hundred independent trees may not yield precise average results. Nevertheless our technique has many compensations, including providing results where no formula exists (e.g., new variations of algorithms, or searches of non-random trees), and making possible a comparison of all algorithms on exactly the same basis.

4.1.2. Average Bottom Positions Seen

Fig. 5 presents some representative experimental data for the case of uniform trees of width five. The graph plots search depth versus terminal nodes visited (expressed as a percentage of the minimal game-tree size [16]). All data points were averaged over the same twenty sample trees, generated in the random fashion described earlier. Two types of characteristic trees are represented: random trees, and strongly ordered trees (as redefined in Section 2.1). Comparable data was also gathered for the cases w = 10, 15 and 20 and depths up to the computational limits of our resources.

Perhaps the most interesting result is the poor performance of SSS* on odd-ply trees. If the first successor proves to be best, SSS* usually expands more nodes than either DUAL* or INS. It seems that SSS*'s best-first search is often mislead into jumping from one node to another in subtrees that are later cut off. On the other hand, SSS* can be a great advantage, sometimes visiting only one quarter of the nodes traversed by any other algorithm. This erratic behavior leads to a standard deviation of about 30% in our SSS* results, which could not be reduced by averaging the data over more runs.

Another question is, why are the even-ply results so poor for directional searches like NS? Here the

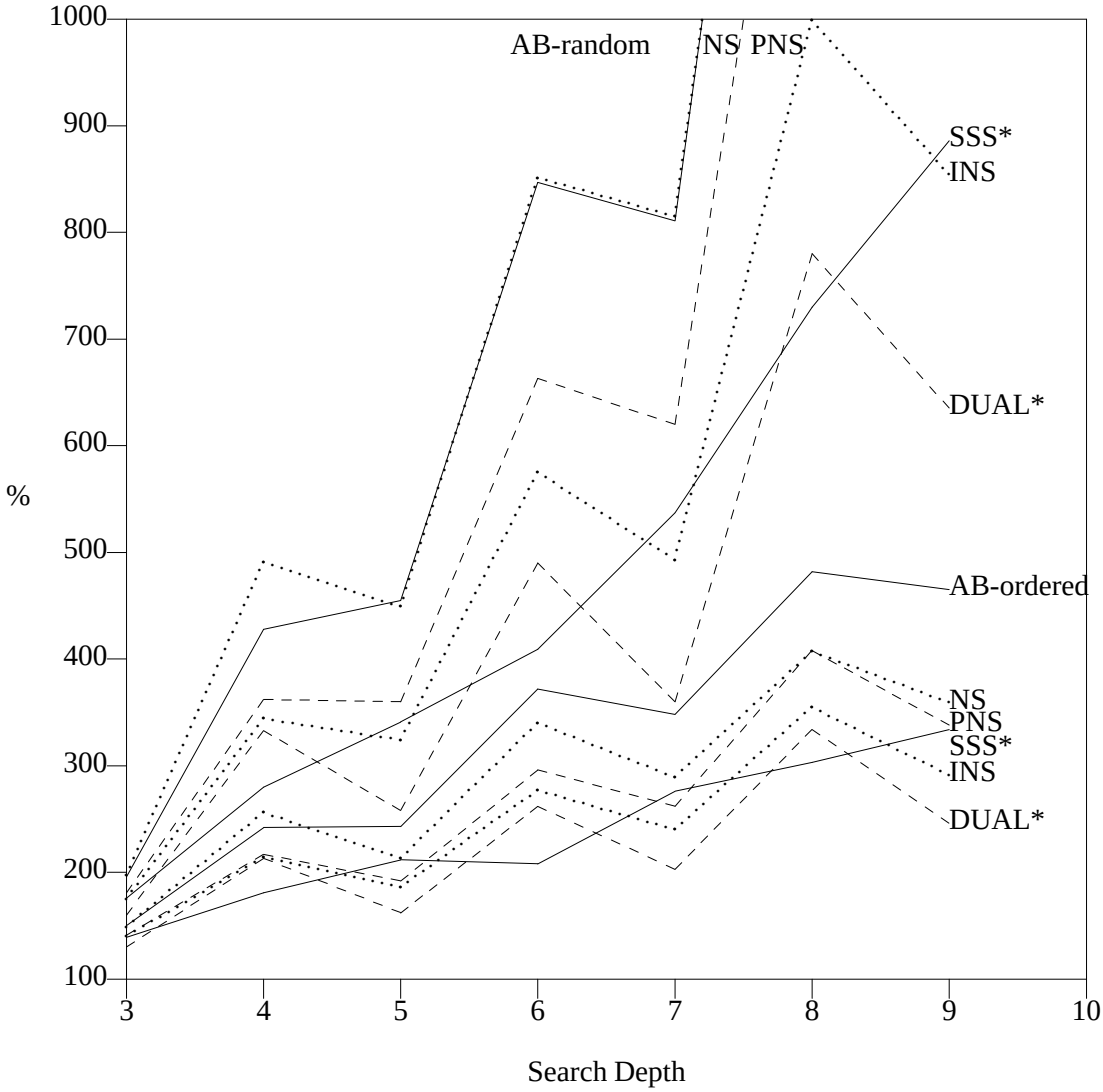


Fig. 5. Width = 5, Comparison to Minimal Tree

Average % Node Count Relative to a Minimal Tree

answer is more clear. It is easy to show that for perfectly ordered even-depth trees the principal variation holds about $\frac{w+1}{2w}$ of the total terminal nodes, while for similar odd-depth trees that factor is only $\frac{2}{w+1}$ [11]. It follows directly that the alternative variations of even depth each hold about $\frac{1}{2w}$ of the terminal nodes. Also, for all widths, a change in the principal variation costs more in even depth trees, because more than 50% of the terminal nodes are in the first subtree. Since changes in principal variation are pos-

sible with directional searches, NS, PNS, INS and DUAL* are inevitably less efficient than SSS* in searching even-depth trees.

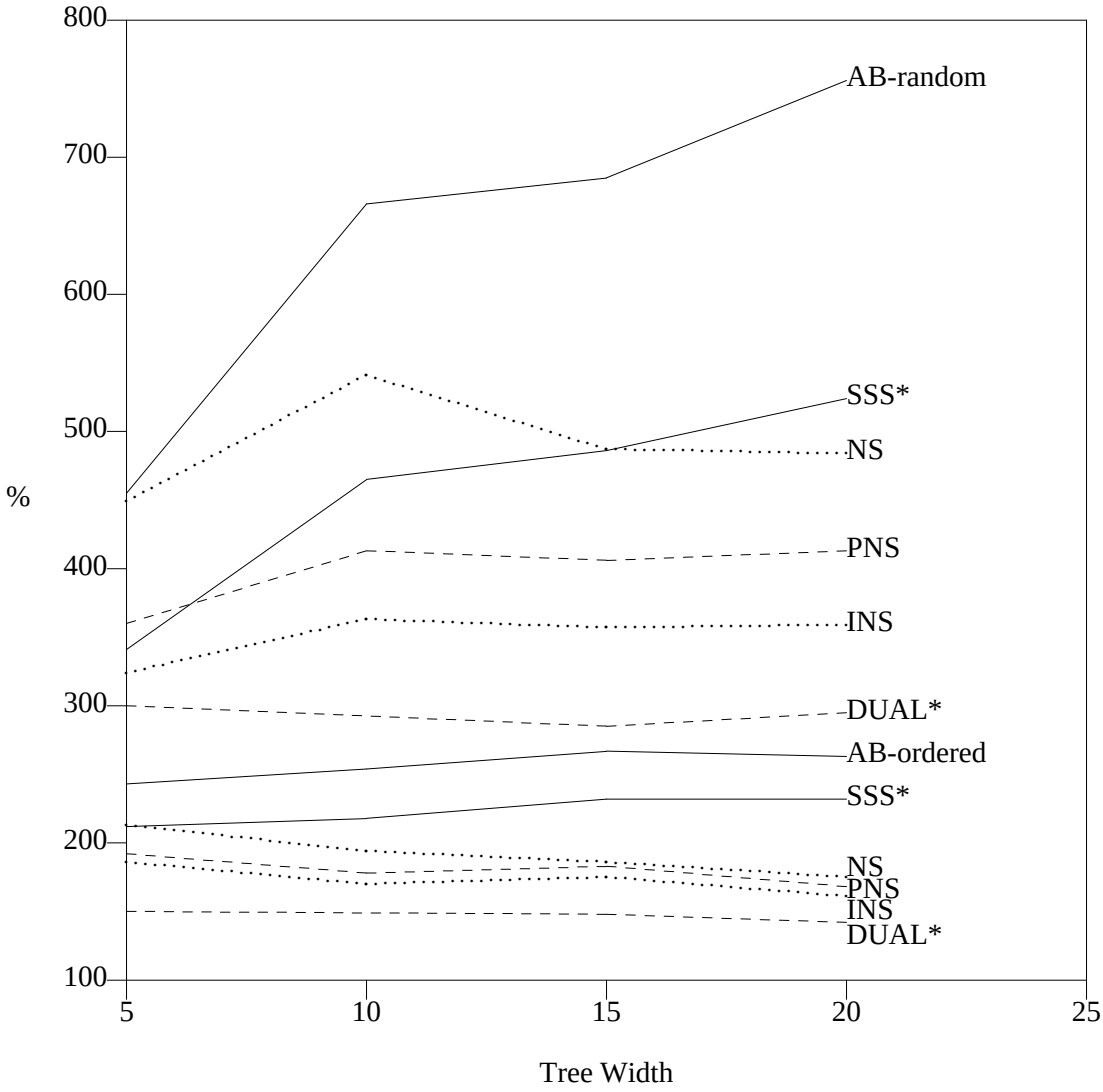


Fig. 6. Depth = 5, Comparison to Minimal Tree.

Average % Node Count Relative to Minimal Tree

Despite the common basis for SSS* and DUAL*, they exhibit different search characteristics. Figs. 5 and 6 show that the performance of DUAL* is especially good in odd-ply searches, but parallels more closely that of the minimal window search algorithms. Like NS, PNS and INS, DUAL* benefits from a good successor ordering at the root level of the tree. If the minimax value is found in the leftmost subtree,

DUAL* searches about the same nodes as the minimal window techniques. On the other hand, when the minimax value lies in a right subtree, DUAL* is able to search more efficiently even than INS, because re-searches do not arise. Fig. 6 shows how the relative performance of the methods changes for a typical odd-depth search, $d = 5$, as the tree width increases from $w = 5$ to $w = 20$. Unlike a recent paper [8], which concentrated on the search of extremely narrow trees of widths from 1 to 5 branches, we have considered the full range from narrow ($w \leq 10$) to bushy ($w \geq 20$) trees. Theoretical studies [15] and our own experience shows that the relative characteristics of the algorithms do not change as the width increases beyond 20 branches per node. Our results in Figs. 5 and 6 show that for odd-depth trees of width greater than 5, SSS* dominates only $\alpha\beta$. As the tree width increases, DUAL*, INS, PNS and eventually even NS are superior on a terminal node count basis. As well as requiring much more space, and an order of magnitude more time, SSS* also examines more terminal nodes than the simple algorithms like NS on odd-depth trees.

SSS*'s best performance occurs in trees of even depths, because SSS* depends on successor ordering at odd levels in the tree. Trees of both depth d (even) and $(d-1)$ have exactly the same levels where node ordering is important, and so even-ply trees are relatively more efficient for SSS*. Fig. 7 shows typical data for the even depth $d = 6$ with increasing widths from 5 to 20 branches. Here we observe that SSS* is superior to the other methods. Even in the best case of ordered bushy trees, DUAL* visits about 10% more nodes than SSS*, and NS about 25% more. Minimal window searches are particularly hurt each time a new subtree proves to be superior, but even $\alpha\beta$ and DUAL* are affected.

4.2. Execution Time

The creation of new search algorithms is motivated by the need for reduced search time. A more time consuming algorithm, no matter how well informed, is certainly less desirable than a faster one, all things being equal. Our preliminary results show that the workspace management overhead in INS is modest, making INS only slightly slower than NS and other $\alpha\beta$ implementations. On the other hand, the overhead in managing the OPEN list is significant, thus DUAL* and SSS* are 5 and 10 times slower, respectively, than NS [12]. Execution profiles of SSS* show that 90% of its time is spent in adding to

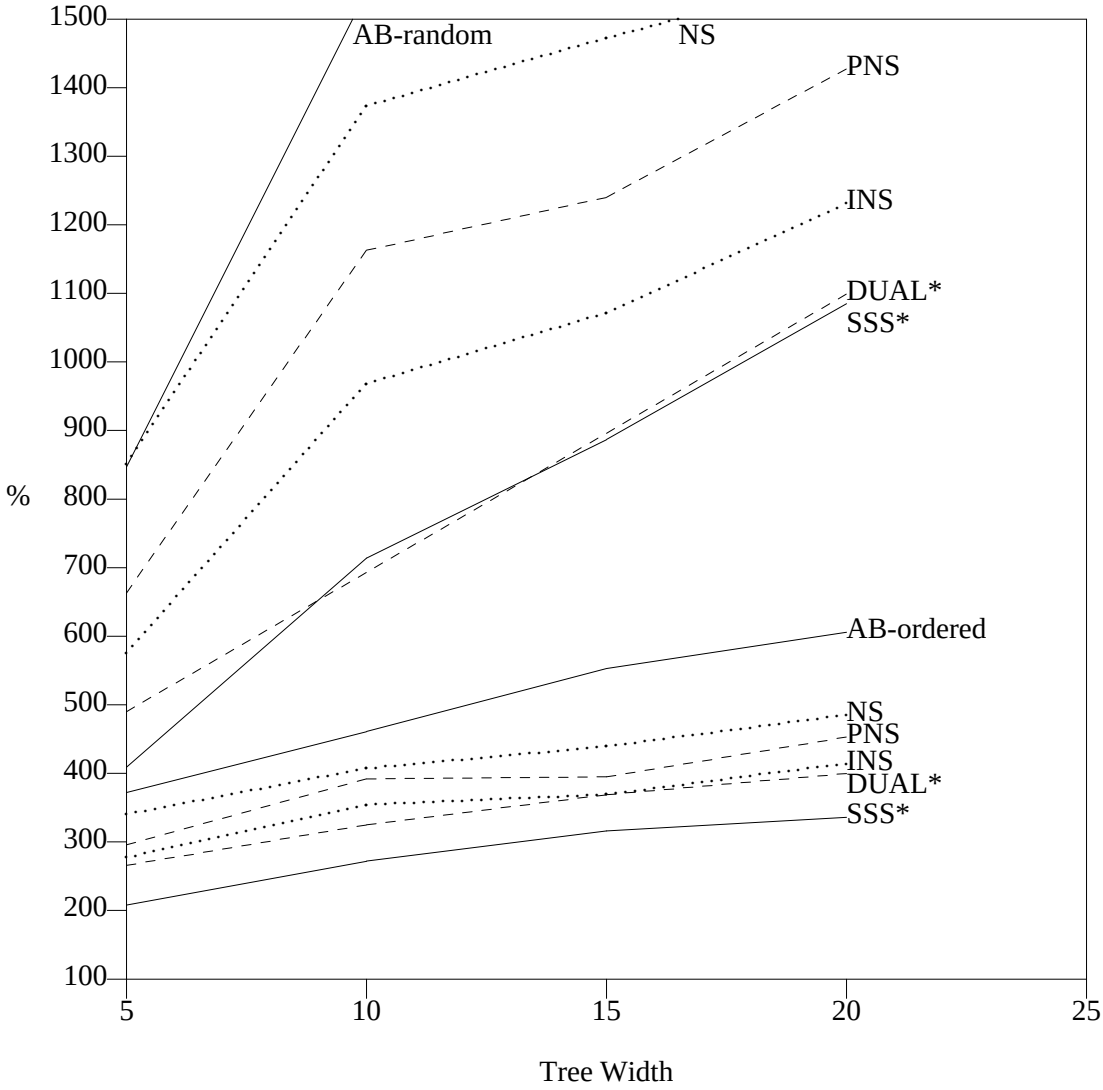


Fig. 7. Depth = 6, Comparison to Minimal Tree

Average % Node Count Relative to Minimal Tree

and deleting from the OPEN list. DUAL* is also degraded by these operations but to a lesser extent because it has a shorter OPEN list, since in effect only $d-1$ ply trees are searched.

5. Discussion

An important point, not brought out by our graphs, is that SSS* outperformed INS on even-ply searches because it was highly efficient on a few problems. In terms of how often each method was preferable, SSS* was only marginally better than INS. On odd-ply trees, in contrast, INS not only out-

performed SSS* in absolute terms, but did so in about 80% of the trees being searched. Two things work in favour of minimal window techniques. First an inferiority proof on a variation is never more than by $\alpha\beta$, and second the identification of a new principal variation is cheap, since it is equivalent to a search on the low side of the refutation wall. However, if a subtree is found superior, NS, PNS and INS must search it once more. DUAL*, in contrast, simply continues searching in a best-first fashion until the new minimax value is found. Consequently, DUAL* usually expands fewer nodes than INS, and both do better than SSS* for odd search depths. Perhaps the most important observation is SSS*'s inability to exploit its node information in an optimal way. The free-ranging best-first search jumps from one subtree to another, trying to prove them superior. Since there is only one superior root subtree, but $w - 1$ inferior ones, most of the node information is never used.

References

1. G.C. Stockman, A minimax algorithm better than alpha-beta?, *Artificial Intelligence 12(2)*, (1979), 179-196.
2. J. Pearl, Asymptotic properties of minimax trees and game searching procedures, *Artificial Intelligence 14(2)*, (1980), 113-138.
3. J.P. Fishburn, *Analysis of speedup in distributed algorithms*, UMI Research Press, Ann Arbor, Mich., 1984.
4. T.A. Marsland and M.S. Campbell, Parallel search of strongly ordered game trees, *ACM Computing Surveys 14(4)*, (Dec. 1982), 533-552.
5. T.A. Marsland, Relative efficiency of alpha-beta implementations, *8th IJCAI Conf. Procs.*, Karlsruhe, 1983, 763-766.
6. M.S. Campbell and T.A. Marsland, A comparison of minimax tree search algorithms, *Artificial Intelligence 20(4)*, (July 1983), 347-367.
7. A. Reinefeld, An improvement of the Scout tree search algorithm, *ICCA Journal 6(4)*, (1983), 4-14.
8. A. Musczycka and R. Shinghal, An Empirical Comparison of Pruning Strategies in Game Trees, *IEEE Trans. on Systems, Man and Cybernetics SMC-15(3)*, (1985), 389-399.
9. A. Reinefeld, J. Schaeffer and T.A. Marsland, Information acquisition in Minimal Window Search, *9th IJCAI Conf. Procs.*, Los Angeles, 1985, 1040-1043.
10. G.M. Baudet, The design and analysis of algorithms for asynchronous multiprocessors, Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, Apr. 1978.
11. J. Schaeffer, Experiments in Search and Knowledge, Ph.D. thesis, Univ. of Waterloo, Waterloo, Ont., Spring 1986.
12. A. Reinefeld, T.A. Marsland and J. Schaeffer, Is Best First Search Best?, Tech. Rept. TR85-16, Computing Science, Univ. of Alberta, Edmonton, Oct. 1985.
13. V. Kumar and L.N. Kanal, Parallel Branch-and-Bound formulations for AND/OR tree search, *IEEE Trans. on Pattern Anal. and Mach. Intell. PAMI-6(6)*, (1984), 768-778.
14. J. Slagle and J. Dixon, Experiments with some programs that search game trees, *J. ACM 2*, (1969), 189-207.

15. I. Roizen and J. Pearl, A Minimax Algorithm better than Alphabeta? Yes and No, *Artificial Intelligence 21(2)*, (1983), 199-220.

16. D.E. Knuth and R.W. Moore, An analysis of alpha-beta pruning, *Artificial Intelligence 6(4)*, (1975), 293-326.