Is Best First Search Really Best?

Alexander Reinefeld
T.A. Marsland
Jonathan Schaeffer

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

ABSTRACT

Of the many minimax algorithms, SSS* consistently searches the smallest game trees. Its success can be attributed to the accumulation and use of information acquired while traversing the tree, allowing a best first search strategy. The main disadvantage of SSS* is its excessive storage requirements. This paper describes a class of search algorithms which, though based on the popular alpha-beta algorithm, also acquire and use information to guide their search. They retain their directional nature yet are as good as SSS*, even while searching random trees. Further, while 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 Natural Sciences and Engineering Research Council Grant A7902 made it possible to complete the experimental work described in this report.

[Re-created from files dated 2 November 1985 in /cshome/tony/Reports/TR85.16]

22 March 2013

Is Best First Search Really Best?

Alexander Reinefeld
T.A. Marsland
Jonathan Schaeffer

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

1. Introduction

An article entitled "A Minimax Algorithm Better than Alpha-Beta", introducing the **State Space Search** (SSS*) algorithm [Sto79], caused researchers to reinvestigate the efficiency of the widely used **alpha-beta** ($\alpha\beta$) algorithm [KnM75] for searching game trees. It was shown that SSS* expands smaller trees than $\alpha\beta$ by saving information during the search so that subtrees are always expanded in a *best first* fashion. Since the information maintenance requires significant time and space overhead, SSS*'s application is restricted to small trees.

Minimal window search techniques [Fis81, MaC82, Rei83] have also been found superior to $\alpha\beta$ in practical applications [Mar83] as well as in artificially constructed trees [CaM83, MuS85, Rei83]. Their left to right (*directional*) search strategy is different from SSS*'s best first approach. Instead of accumulating information in a large data structure, minimal window search tries to show that the current subtree is inferior to the best subtree visited so far. This is usually the case and the cost of demonstrating this is never more and usually much less than that required by $\alpha\beta$. 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 is not dominated by SSS*. That is, minimal window search is capable of building smaller trees than SSS* whereas $\alpha\beta$ cannot.

In this paper, new information acquisition methods to improve minimax search are presented. Minimal window search, modeled here by the **NegaScout** (NS) algorithm [Rei83], is enhanced to gather information during an initial search of a subtree, and to use it if a second search is required. Two variants are presented, **Partially Informed NegaScout** (PNS) and **Informed NegaScout** (INS) [RSM85], differing in the extent to which they gather information. INS uses all the available information to expand the smallest possible subtrees, whereas PNS is a compromise that trades storage for reduced node expansions in a cost effective way. The performance of these algorithms is compared on both random and *strongly ordered* [MaC82] trees of uniform width w. Experiments indicate that the new algorithms are comparable to SSS* in terms of tree size, but with significantly lower execution and storage overheads.

The performance of INS and PNS suggests that SSS* does not make best use of its information. A new algorithm, **DUAL***, is introduced that adds some directional properties to SSS*'s best first approach. A left to right search is done at the root by invoking the *dual* of SSS* at the next level. Thus, after searching the first subtree, the remainder are searched with a better bound than SSS* would use. Experiments indicate that DUAL* generally traverses smaller trees than SSS*, even in the random case. Other compromise strategies involving different combinations of best first and directional strategies are also considered.

Game playing practicioners have to weigh out the space, time and search complexity of the various algorithms for their specific application. For this purpose we provide a summary of the space and time requirements, and also extensive empirical data comparing the search performance on trees of different kinds.

2. Minimal Window Search

Minimal window search capitalizes on the fact that it is cheaper to prove a subtree inferior, than to finds its true value. *Aspiration* or *narrow* window search [Bau78] also employs this notion by seeking a value for the tree within tight limits. If these limits take on adjacent values, then one has a *zero-width* or minimal window. To find the minimax value, v, of a tree, the most efficient aspiration $\alpha\beta$ search would use the window $(v-1, v+1)$. If all the subtrees have distinct values then all but one would be refuted cheaply by using such a window. This raises the possibility of scanning the range of plausible values for the tree, successively moving a narrow window to eliminate subtrees until only one remains. Minimal window search algorithms use a similar idea to identify better subtrees.

2.1. Refutation Wall

It is a well-known property of $\alpha\beta$ search that the narrower the bounds on the search window, the smaller the tree that is traversed. However, the tree size is also strongly influenced by the proximity of the window to the minimax value. If the true value of a tree is v, how does the tree size change with different values s when a window of $(s, s+1)$ is used? The size partly depends on the difference $s - v$, which will be referred to as the *distance* of the minimal window from the true value.

An experiment was conducted to see how tree size varied with distance. Having determined the minimax value v for a given tree, it was then re-searched using 50 windows

$$(v + i, v + i + 1), \qquad i = -25, -24, -23, \ldots, 23, 24$$

to cover all distances from -25 to +24. For a variety of widths, depths, and orderings, 20 trees were searched and the tree sizes averaged. Figure 2.1 shows two plots of distance versus tree size (normalized to the largest tree). One graph is for a set of randomly generated trees and the other for strongly ordered trees [1].

All experimental data follows a similar pattern. As the minimal window increases towards the true value the trees gradually grow in size, finally peaking with the window $(v-1, v)$. However, when the window is increased to $(v, v+1)$ the tree size falls dramatically! The better the ordering, the larger the decrease. Figure 2.1 illustrates that it is easier to show that a tree has a value greater than v, than to show that it does not. This is easily explained since in the latter case all immediate descendants of the root must be examined, while the former case benefits form cut-offs. What is surprising is the magnitude of tree size differential that occurs when the window reaches v. This change will be referred to as the *refutation wall*.

A point of possible confusion needs to be clarified. The smallest trees are traversed when the search window at the root has a range which is strictly greater than the value of the tree. Similarly, for each of the w sub-trees at the next level after the root node, the smallest trees are visited when the window is below the value for that sub-tree. This follows from the minimax algorithm which has maximum and minimum operations at alternating depths of the tree.

The refutation wall is an experimental demonstration of the potential benefits of using minimal windows to refute sub-trees. By searching using windows on the low side of the wall, tremendous savings can be made. On occasion, re-searches will be necessary and it is the frequency with which these occur that largely determines the over-all success of the approach.

2.2. NegaScout

Pearl's **Scout** algorithm [Pea80] employs a form of minimal window search. After its performance was proven [Noe80], the ideas in Scout were reformulated and improved several times. The resulting algorithms included **palphabeta** (PAB) [CaM83, Fis81], **Principal Variation Search** (PVS) [Mar83, MaC82] and **NegaScout** (NS) [Rei83][2]. Figure 2.2 illustrates the NegaScout algorithm, formulated in a pseudo code based on the C programming language [KeR78]. It searches a tree of uniform width w and depth d. Using the Dewey decimal system (as suggested by Knuth and Moore [KnM75]), the immediate successors of an internal node p are named $p.1, \ldots, p.w$. They are formed by the application dependent function

[1] These trees have a 60% probability that the the left-most descendant leads to the highest minimax value (see Section 5.1).

[2] As a practical matter there is little difference in performance between these algorithms.

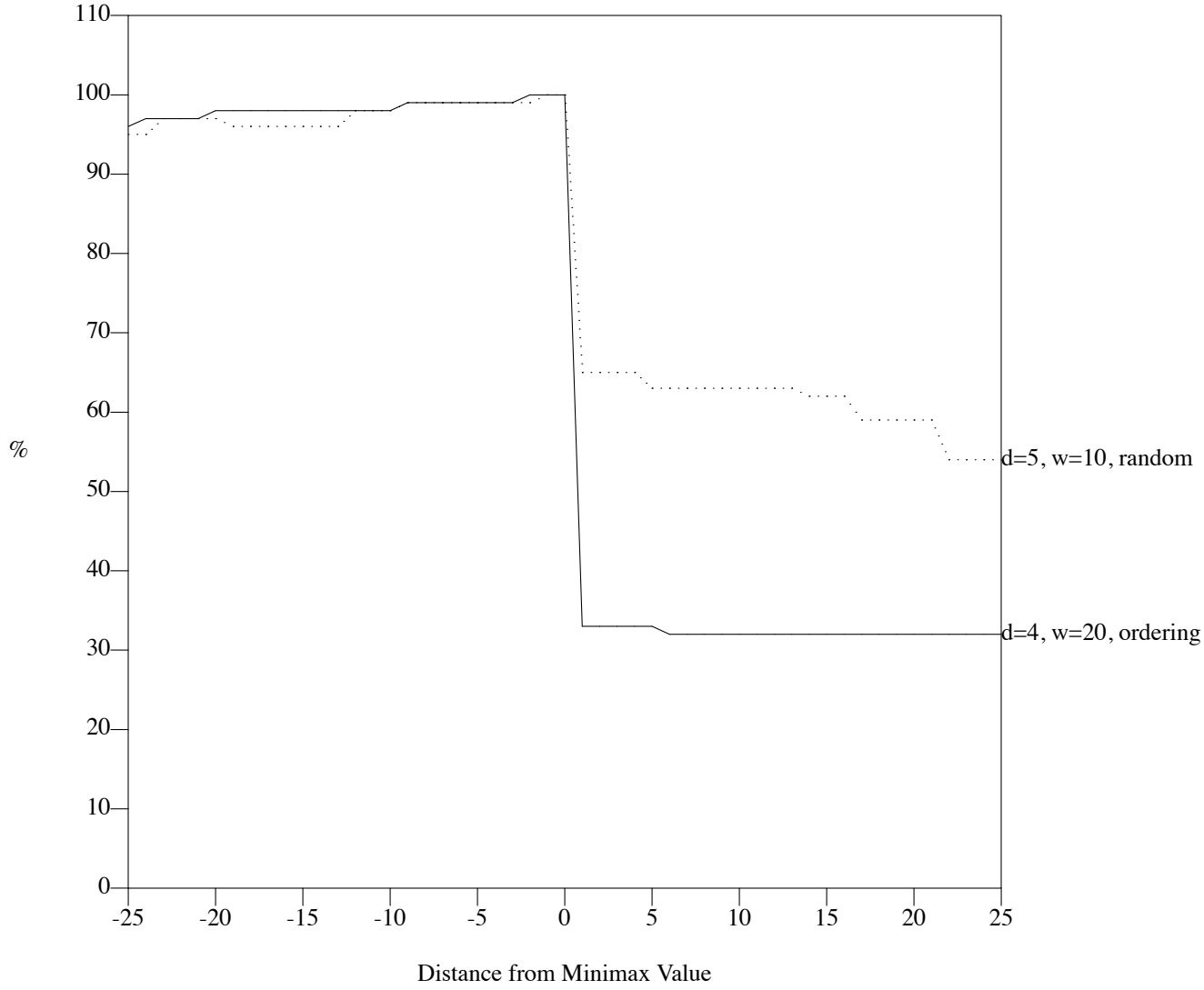


Figure 2.1: Minimax Wall

Generate(p) that also returns a count of p's successors. The static evaluation function, *Evaluate*(p), provides a numerical assessment of the quality of position p.

The minimax value of a tree may be determined by invoking

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

where p represents the root position, (α,β) the *search window*, and d the maximum search depth. As Figure 2.1 shows, after the expansion of the first successor with an appropriate window (α, β), all remaining successors are traversed with the minimal window $(a, a+1)$, where a represents the best available score $\geq \alpha$. Clearly, every minimal window search fails. If it fails low ($v \leq a$), then the subtree is inferior and can be ignored. If the search fails high ($v > a$), the same subtree must usually be re-searched with the opened window (v, β) to determine its exact value. Re-searches can be omitted whenever one of the following condi-

```
int NS ( p, α, β, depth)
position p; int α, β, depth;
{
   int i, v, w, a, b;

   if ( depth == 0 )
      return ( Evaluate(p) );

   w = Generate (p);
   if ( w == 0 )
      return ( Evaluate(p) );

   a = -∞;
   b = β;                          /* an open window for first branch */

   for ( i=1; i≤w; i++ )
   {
      make ( p.i );
      v = -NS ( p.i, -b, -Max( a, α ), depth-1 );

      if ( v > a )
         if ( i == 1 || v ≤ α || v ≥ β || depth ≤ 2 )
            a = v;
         else                                          /* re-search */
            a = -NS ( p.i, -β, -v, depth-1 );
      undo ( p.i );

      if ( a ≥ β )
         goto done;                                    /* β cut-off */

      b = Max ( a, α ) + 1;                    /* form minimal window */
   }
done:
   return (a);
}
```

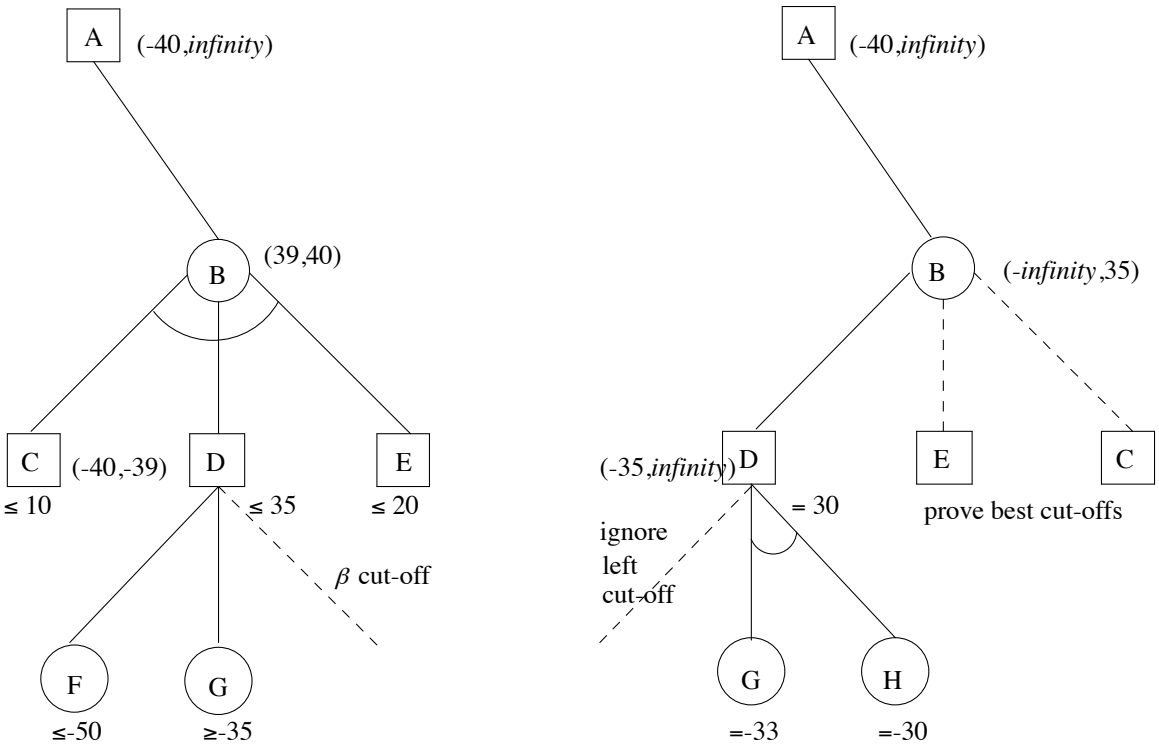
Figure 2.2: The NegaScout algorithm

tions holds true:

- i == 1: The first successor was searched with a full width window, thus the returned minimax value is correct.
- $v \le \alpha$: The v-value does not improve the α bound.
- $v \ge \beta$: The v-value will cause a cut-off, so a re-search to find its improved value is of no interest.
- *depth* ≤ 2: Backed up values from the two levels closest to the terminal nodes are always correct. For these shallow trees, the initialization of a to -∞ causes exact values to be returned even when they are lower than α. A discussion of other benefits of this *fail-soft* refinement can be found elsewhere [Fis83, MaC82].

2.3. 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. Since the minimax value will usually not be found in the first path, subsequent re-searches in deeper tree levels are also inevitable. Acquiring information about the bounds on the subtree value may simplify any necessary re-search.



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

Figure 2.3: Three cut-off types

In an initial search, one piece of information that is inexpensive to maintain is the *sub-variation* (path) to the leftmost terminal node that caused the failure. If a re-search occurs, the sub-variation can be the first path traversed. 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 Figure 2.3(a). Node F cannot possibly return a better value, because it did not stop the initial search and therefore is ignored in Figure 2.3(b).

Another piece of information that can be maintained is the score for each successor of nodes an odd depth away from the root of the re-search. At these nodes, such as node B in Figure 2.3(a), a β cut-off has not occurred and the scores 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. For example, in Figure 2.3(b), subtree D is re-expanded first and then E and C. Secondly, if a re-search of subtree D returns a value ≥ 20, then D is proved superior to both E and C, since it is already known from the initial search that their values can only be ≤ 20 or ≤ 10, respectively. Figure 2.3(b) shows that this *prove best* cut-off

eliminates nodes E and C. In case both D and E should 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 $(-35, \infty)$, perhaps returning the value $v_1 > -10$. Since v_1 is in the search window it is a lower bound on the true score, 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.

2.4. Informed NegaScout (INS)

Recursively saving and using prove best and ignore left information at all nodes in the tree is the basis for the Informed NegaScout algorithm. In essence, 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.

An examination of the INS algorithm presented in Figure 2.4 reveals that only β and prove best cut-offs are recognized, illustrating that ignore left cut-offs can be treated as a special case of prove best. 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 scores returned by all w successors are retained. SaveInfo and GetInfo access the data structure used to maintain information from initial searches of trees. The storage management issues are hidden by these routines. On a re-search at an ignore left node, the scores of the inferior successors $p.1,\ldots, p.i-1$ could be initialized to $-\infty$ with the remaining nodes having value $+\infty$. As a consequence of the sorting operation, the successor list becomes $p.i,\ldots, p.w, p.1,\ldots, p.i-1$, and a prove-best cut-off trims the inferior nodes instead.

With INS none of the node information is wasted if a re-search occurs, because only nodes that would be expanded a second time are stored. Also, INS knows when information is no longer relevant and can discard it. On small systems, memory availability usually limits the problem size. Unlike SSS*, INS is flexible enough to make efficient use of any storage size. For example, if the node information is maintained in a hash table (similar to the transposition tables used in chess programs [MaC82]) INS's storage requirements can be easily 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 in this paper, a tree-like linked data structure was used instead of a hash table, ensuring all information was retained and maximum cut-offs were achieved.

2.5. Partially Informed NegaScout (PNS)

Even a small amount of information is useful in reducing tree size. Partially Informed NegaScout keeps just enough information to enable ignore left and some prove best cut-offs. Ignore left cut-offs require the maintenance of a sub-variation which works analogously with the formation of a *principal variation* in game-playing programs. The sub-variation is built using a triangular matrix of size $\frac{d^2}{2}$ that retains the best path sequence for depths 1 through d [AkN77]. In addition, an integer array of size w is used to hold the upper bounds of the prove best cut-offs that are returned to the first level of the initial search.

3. State Space Search (SSS*)

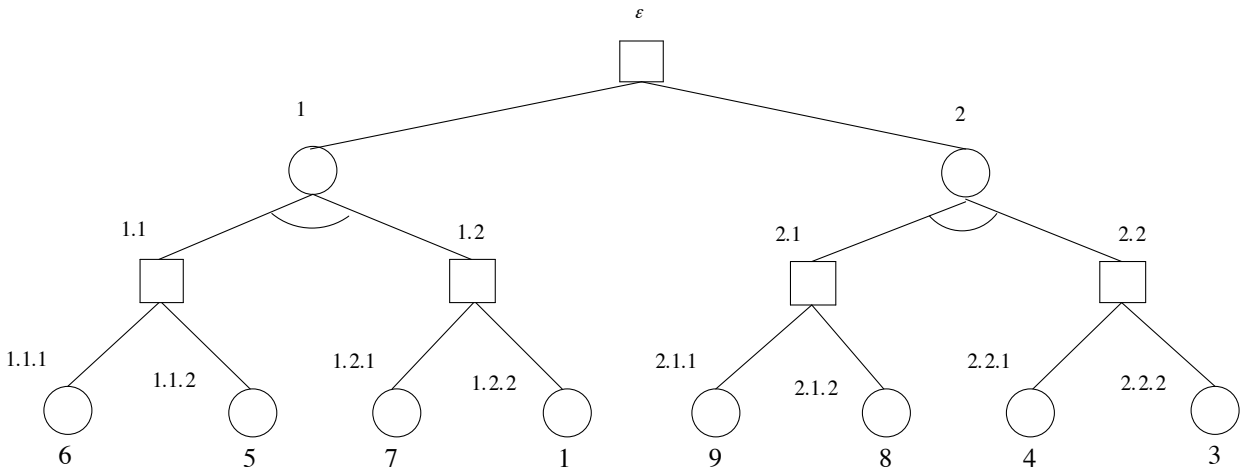
SSS* is a *non-directional* algorithm for searching AND/OR graphs in a best first manner similar to A* [Nil80]. SSS* expands simultaneously multiple paths in different regions of the graph and gains global information about the search space. When traversing game trees, which are a subset of AND/OR graphs, SSS* never searches more nodes than $\alpha\beta$, and usually considerably less [Sto79].

A game tree may be viewed as having alternating layers of MAX and MIN nodes, with a MAX node at the root. In the first phase of a search, SSS* expands all MAX successors and only one MIN successor providing an optimistic estimate on MAX's abilities. These are a subset of the nodes in the minimal game

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

   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, score[] );      /* get scores of successors */
      Sort ( succ[], score[] );                      /* and sort them */
   }
   else
      kind = PROVE_BEST;

   a = -∞;
   b = β;                              /* open window for 1st 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 == PROVE_BEST )
            a = v;                    /* searched with a narrow window */
         else                                          /* re-search */
            a = -INS ( succ[i], -β, -v, depth-1, TRUE );

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

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

Figure 2.4: Informed NegaScout (INS).

tree, so no expansions are wasted. In the second phase, SSS* seeks an optimal MAX solution tree by expanding the most promising nodes backwards from the terminal nodes to the root. Thus, SSS* may be regarded as a best first search strategy.



The OPEN list (stack) built by SSS* is shown below. Here # represents the bottom of the stack, and L and S refer to LIVE and SOLVED, respectively. Node 2.1.2 is searched by SSS*, but not by Negascout.

(ε, L, ∞) #
$(1, L, \infty)$ $(2, L, \infty)$ #
$(1.1, L, \infty)$ $(2, L, \infty)$ #
$(1.1.1, L, \infty)$ $(1.1.2, L, \infty)$ $(2, L, \infty)$ #
$(1.1.2, L, \infty)$ $(2, L, \infty)$ $(1.1.1, S, 6)$ #
$(2, L, \infty)$ $(1.1.1, S, 6)$ $(1.1.2, S, 5)$ #
$(2.1, L, \infty)$ $(1.1.1, S, 6)$ $(1.1.2, S, 5)$ #
$(2.1.1, L, \infty)$ $(2.1.2, L, \infty)$ $(1.1.1, S, 6)$ $(1.1.2, S, 5)$ #
$(2.1.2, L, \infty)$ $(2.1.1, S, 9)$ $(1.1.1, S, 6)$ $(1.1.2, S, 5)$ #
$(2.1.1, S, 9)$ $(2.1.2, S, 8)$ $(1.1.1, S, 6)$ $(1.1.2, S, 5)$ #

$(2.1, S, 9)$ $(1.1.1, S, 6)$ $(1.1.2, S, 5)$ #
$(2.2, L, 9)$ $(1.1.1, S, 6)$ $(1.1.2, S, 5)$ #
$(2.2.1, L, 9)$ $(2.2.2, L, 9)$ $(1.1.1, S, 6)$ $(1.1.2, S, 5)$ #
$(2.2.2, L, 9)$ $(1.1.1, S, 6)$ $(1.1.2, S, 5)$ $(2.2.1, S, 4)$ #
$(1.1.1, S, 6)$ $(1.1.2, S, 5)$ $(2.2.1, S, 4)$ $(2.2.2, S, 3)$ #
$(1.1, S, 6)$ $(2.2.1, S, 4)$ $(2.2.2, S, 3)$ #
$(1.2, L, 6)$ $(2.2.1, S, 4)$ $(2.2.2, S, 3)$ #
$(1.2.1, L, 6)$ $(1.2.2, L, 6)$ $(2.2.1, S, 4)$ $(2.2.2, S, 3)$ #
$(1.2.1, S, 6)$ $(1.2.2, L, 6)$ $(2.2.1, S, 4)$ $(2.2.2, S, 3)$ #
$(1.2, S, 6)$ $(2.2.1, S, 4)$ $(2.2.2, S, 3)$ #
$(1, S, 6)$ $(2.2.1, S, 4)$ $(2.2.2, S, 3)$ #
$(\varepsilon, S, 6)$ #

Figure 3.1: A Case where NegaScout Outperforms SSS*.

3.1. SSS* Compared to Minimal Window Search

Since SSS* is non-directional, the location of the solution path in the tree has little effect on the algorithm's efficiency. In contrast, the effectiveness of the minimal window methods depends heavily on the

solution path's location. Once the path has been found, the minimal window efficiently prunes the remaining subtrees, and in doing so never expands more nodes than SSS* [3]. In Figure 3.1, NegaScout expands only the left successor of node $p2.1$ and proceeds with node $p2.2$, where the refutation value $v(p2.2) = 4$ is found[4]. At the risk of eventually being forced to expand the subtree a second time, NegaScout does not evaluate the terminal node $p2.1.2$. In contrast, SSS* visits this node, thus showing that SSS* *does not dominate* NegaScout (nor PVS and Scout) in terms of terminal node evaluations. Of course, the domination of NegaScout over SSS* can also not be shown, because of the many cases where NegaScout must re-visit some subtrees.

3.2. The DUAL* Algorithm

SSS* is a powerful algorithm for searching random trees, because it is not directional. However, it does not see all the information it stores, especially in cases of real application where trees are well-ordered. To compensate for this shortcoming we propose **DUAL***, an algorithm which determines the tree value by employing a directional state space search strategy at the root node. The left to right search at the root is done by invoking the dual of SSS* (termed dual-SS* by Kumar and Kanal [KuK83]) at the next level. The dual of SSS* is formed 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. Clearly, dual-SS* and SSS* have similar performance characteristics. DUAL*, however, searches the successors at the root node in strict left to right order using dual-SS*. DUAL* is able to search an optimally sorted tree with minimal node expansions, is because the expansion of the root's right successors profits from bounds already established. Figure 3.2 illustrates the combination of the use of dual-SS* with a directional search, by outlining DUAL* in a C like pseudo code. Each node descriptor ($node$, $nodestatus$, h) represents one state of a partial MAX solution tree, and consists of an identifier $node$, a status $nodestatus$ (which is either *LIVE* or *SOLVED*) and a merit h. Descriptors of already traversed subtrees are maintained on the OPEN list in increasing order of their h-values. At the beginning, the root node descriptor ($root$, *LIVE*, $-\infty$) is pushed onto the open list. Thereafter, one descriptor is removed from the top of OPEN and the appropriate node expansions or reductions are done. This is repeated until the root node descriptor ($root$, *SOLVED*, h) is found, with h representing the value of the tree. The following functions are assumed to exist:

pop (p, s, h)	returns the top node descriptor from OPEN.
push (p, , s , h)	places the node descriptor at the top of OPEN.
insert (p, s, h)	puts the node descriptor behind all nodes of similar merit, but in front of nodes of equal merit which are located more to the right in the search tree.
purge (p)	deletes all node desriptors which are an ancestor of p.

4. Mixed Strategies

The *DUAL** algorithm added some directional properties to a best-first search. Many different mixed or *hybrid* strategies are possible covering the continuum from purely directional ($\alpha\beta$) to purely best-first (SSS*). Some methods capitalize on known properties of the application to gamble their way to a quick solution. Others, like staged-SSS* [CaM83], accept increased search overhead to limit storage requirements. One aim of a hybrid is to benefit from the best properties of each method, so that the combination is better than either alone. Alternatively, the aim might be to reduce the storage requirements of a fast algorithm, by doing more of the work with a slower method.

---

[3] Justification for this and other points of theory relating to DUAL* are the subject of a forthcoming report by A. Reinefeld of Hamburg.

[4] Because of NegaScout's negamax approach, for odd search depths the subtree value will be the negative of the minimax value computed by SSS*.

```
int DUAL* (root, bound)                    /* initial bound is -infinity */
{
  h = bound;
  for (j=1; j≤w; j++)
  {                                  /* use directional search at root */
     v = dualSS* (root.j, h);
     if (v > h)
        h = v;
  }
  return(h);
}
int dualSS*( root, bound )
{
   push ( root, LIVE, bound );                        /* save root node */
   while ( true )
   {
      pop ( node, nodestatus, h );          /* restore node description */
      if ( nodestatus == 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                                      /* nodestatus == 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 );
      }
   }
}
```

Figure 3.2: The DUAL* algorithm

4.1. DUAL-NS and DUAL-INS

Despite its internal best first search, DUAL* expands all root successors strictly from left to right. Thus the stronger the degree of ordering the more often the value of the leftmost subtree is superior to the remainder. In the optimal case where the first successor is best, DUAL* expands exactly the same nodes as

NS, PNS and INS, and the node information on the OPEN list is never used. Although DUAL* is a powerful means for determining the minimax value of a superior subtree, like SSS* its complex stack operations are too time consuming to prove the rest of the subtrees inferior. One can reduce this disadvantage by using minimal window search to prove sub-trees inferior and leave DUAL* to search the superior sub-trees. Two mixed strategies, **DUAL-NS** and **DUAL-INS**, have been examined. A normal minimal window search is used, but all searches with a wide window are done by DUAL*. If the right subtrees are inferior, DUAL* expands exactly the same nodes as the two hybrids. However, DUAL-NS and DUAL-INS do so using minimal window search, and thus avoid the time-consuming manipulation of the OPEN list of DUAL*. If any of the subtrees are found superior, DUAL* will traverse smaller trees than DUAL-NS and DUAL-INS.

```
int DUAL_NS ( p, α, β, depth)
position p; int α, β, depth;
{
   int i, v, w, a, b, bound;

   if ( depth == 0 )
      return ( Evaluate(p) );

   w = Generate (p);
   if (w == 0)
      return ( Evaluate(p) );

   a = -∞;
   b = β;                          /* an open window for first branch */

   for ( i=1; i≤w; i++ )
   {
      bound = Max( a, α );
      if ( b - bound > 1 )                    /* not a minimal window */
         v = -dualSS* ( p.i, -bound );
      else
         v = -DUAL_NS ( p.i, -b, -bound, depth-1 );

      if ( v > a )
         if ( i == 1 || v≤α || v≥β || depth≤2 )
            a = v;
         else                                          /* re-search */
            a = -dualSS* ( p.i, -a );

      if ( a≥β )
         goto done;                                    /* β cut-off */

      b = Max( a, α ) + 1;                    /* form minimal window */
   }
done:
   return (a);
}
```

Figure 4.1: The DUAL-NS algorithm

Figure 4.1 illustrates the DUAL-NS algorithm, placed in the Neagascout framework. DUAL-INS is slightly more complicated. Also, since DUAL* and INS expand identical nodes to prove a subtree inferior, DUAL-INS can be altered to save the node information from an initial search in a manner that is suitable for DUAL* to use in the possible re-search. Rather than using INS to re-search a new superior subtree,

DUAL* can resume its best first search with the aid of the data gathered by the initial INS search. Thus for DUAL-INS, the INS component would require some changes to its stored information, so that the best interface with DUAL* is provided.

5. Space, Time and Search Comparison

There are several bases for comparing algorithms. For tree searching, the usual measure is tree size. However, for practical purposes, this measure is inadequate because it fails to take into account the execution time and storage overhead. Accordingly, in this section, each algorithm considered in this paper is compared on the basis of tree size, storage requirements, and execution time.

5.1. Generating Trees

In practice, trees rarely have a random distribution of values at terminal nodes. Usually application dependent knowledge is available that allows the searching program to examine siblings of interior nodes in order of most to least promising. The stronger the ordering, the smaller the trees that are built. Many tree searching papers only consider the case of random trees. Although these give an adequate measure 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. This includes varying the tree width w, depth d and degree of ordering.

Several approaches for generating trees have appeared in the literature [MuS85] These methods usually suffer from inflexibility (for example are restricted to the random tree case) or excessive storage (the trees must be pre-computed [CaM83]). For the experiments reported in this section, a different method was used which is flexible with respect to ordering and uses modest storage. It works on the principle of having the value of a sub-tree derivable from information available from its parent node. Thus, the entire tree need not be generated to know its minimax value. That information makes it possible to build a tree from the root down, imposing whatever ordering criteria is desired at interior nodes, in a manner guaranteed to generate the correct minimax value. As such, it only requires $O(w \times d)$ storage.

Initially, the user specifies w *weights* reflecting the percentage of time at interior nodes that each of the w siblings will be the root of the sub-tree having the highest minimax value of all the siblings. These weights determine the degree of ordering for the tree, and obviously must total 100. For example, assigning equal weight to each of the siblings implies a random tree. Assigning a weight of 100 to the first sibling and zero to the remaining results in an optimal tree where the left-most descendant is always best.

At the root node r, a random number is used to select which of the w descendants will have the highest minimax value based on the pre-selected *weights*. Assume that descendant i has been selected. The minimax value v for this subtree is randomly generated in the range $-inf \ldots inf$. The value for sub-trees $1 \ldots i-1$ are randomly chosen from the range $-inf \ldots v-1$ and for sub-trees $i+1 \ldots w$, from $-inf \ldots v$ [5], and are stored on a stack. When at an interior node, $r.i$ for example, the value of the sub-tree can be retrieved from the stack. The best sibling j is again randomly chosen according to the weights and is assigned its pre-computed negamax value $-v$. This approach is applied recursively throughout the tree. Essentially, the values of a node's descendants are generated in a way consistent with the known minimax value of the parent, and the degree of ordering required for the tree.

The algorithm has the property that all random numbers are derived from an initial *seed* specified by the user. Hence, given the same w, d, *weights*, and *seed*, the same tree will always be generated. Changing any of the parameters results in an entirely different tree.

5.2. Nodes Searched

5.2.1. An Exhaustive Comparison

The terminal node order greatly influences the efficiency of any search algorithm. Therefore a useful performance evaluation should include a variety of node orderings from the best to the worst case. Table 5.1 shows an exhaustive comparison of the algorithms considered in this paper on trees of width 2 and

[5] Note that more than one sub-tree could have the same minimax value. Since sibling i has the highest value, sibling $1 \ldots i-1$ must have a value less than v because of left-to-right ordering.

depth 3. All $8! = 40320$ terminal node permutations with distinct scores have been searched by each algorithm.

leaf visits	DUAL*	SSS*	INS	PNS	$\alpha\beta$	NS
5	8064	8064	4032	4032	4032	4032
6	17024	18816	15232	15232	11648	10752
7	12032	4224	14272	13120	11072	4768
8	3200	9216	6784	5824	13568	5216
9	0	0	0	1664	0	5504
10	0	0	0	448	0	6240
11	0	0	0	0	0	3232
12	0	0	0	0	0	576
avg visits	6.26	6.36	6.59	6.68	6.85	7.79

Table 5.1: All trees of width = 2 and depth = 3.

DUAL* and SSS* traverse fewest terminal nodes twice as often as any other algorithm. This is because of the symmetry in SSS*'s search, for which it makes no difference whether the principal variation lies in the left or in the right subtree. On the other hand, SSS* expands one node more than NS in 10.5% of the trees. These trees are similar to that shown in Figure 3.1.

PNS and INS perform even better than NS[6]. In 15.7% of the trees they omit one node expansion that is done by SSS*. Each time, the principal variation lies in the right subtree, NS, PNS and INS must do at least one re-search. Because of the "depth ≤ 2" condition, this is the only case where re-searches occur. While NS must descend along the leftmost path of the right subtree, PNS has a sub-variation starting at either p2.1 or p2.2. In contrast, INS retains both sub-variations. In addition, PNS and INS never re-visit the first terminal node of the sub-variation. This explains the poorer overall performance of NS when searching these depth 3 trees.

This example illustrates that minimal window search has the potential to be more efficient than SSS*, even though re-searches are sometimes necessary.

5.2.2. Average Node Expansions

Figures 5.2 and 5.3 illustrate some representative experimental data. The graphs plot search depth versus tree sized expressed as a percentage of the minimal tree size. All data points were averaged over 20 sample trees. Two types of ordering are presented: random trees, and strongly ordered trees where the first successor has a 60% chance of being best and the remaining 40% equally distributed between the other $w - 1$ siblings. Data for the $w = 20$ trees beyond $d = 6$ was too computationally expensive to obtain.

Perhaps the most interesting result is the poor performance of SSS*. If the first successor proves best, SSS* usually expands more nodes than either DUAL* or INS. In addition, the best first search often misleads SSS* into jumping from one node to another in subtrees that are later cut off. On the other hand, a best first search can be a great advantage. Sometimes SSS* visits only one quarter of the nodes traversed by any other algorithm. This erratic behavior leads to a standard deviation of about 30%, which could not be reduced by averaging the data over more runs.

Despite the common basis for SSS* and DUAL*, they exhibit different search characteristics. The performance of DUAL* is similar to that of 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 approximately the same nodes as the minimal window techniques.

[6] Note that here NS made no use of Fishburn's *lalphabeta* refinement, which obviates the need to re-search the last branch.

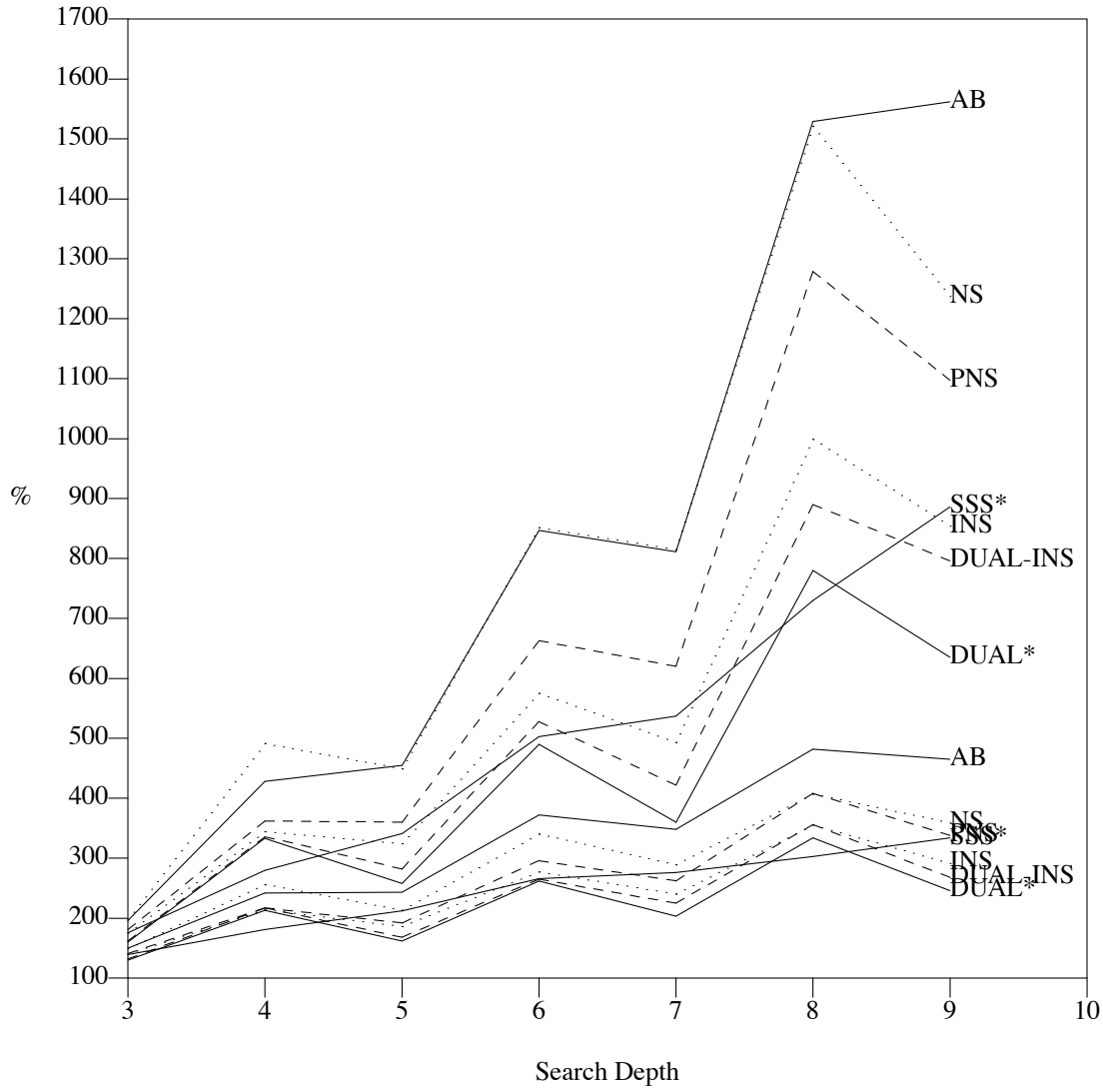


Figure5.2: Comparison to Minimal Tree (w=5)

On the other hand, DUAL* is able to search more efficiently than INS when the minimax value lies in a right subtree because re-searches of these subtrees are expanded best first.

Directional algorithms are handicapped, especially in even depth search trees, because they initially expand a solution tree to the left, and at a later time might expand some more to the right. Note that this affects any directional search algorithm every time a subtree is found to be superior. Thus $\alpha\beta$ and DUAL* are also influenced, but to a lesser extent than NS. In the $w = 20$ data, the SSS* performance is much lower in trees of even depths because SSS* depends on successor ordering in odd tree levels. There are $d/2$ levels in even depth trees where successor ordering is essential, and the same number for trees of odd depth (d+1).

In summary, DUAL*, INS and the intermediate version DUAL-INS exhibit a performance comparable to that of SSS*. They are consistently better than SSS* in odd tree depths, but are usually less efficient in even search depths. Examination of the odd ply data reveals a lower growth rate of the DUAL* and INS graphs than for SSS*. Although the graphs suggest an algorithm preference according to their performance, no strict dominance relationship can be established. As an extreme example, cases have been observed where the information saved by PNS, INS and SSS* mislead them into expanding more nodes than the simpler NS algorithm.

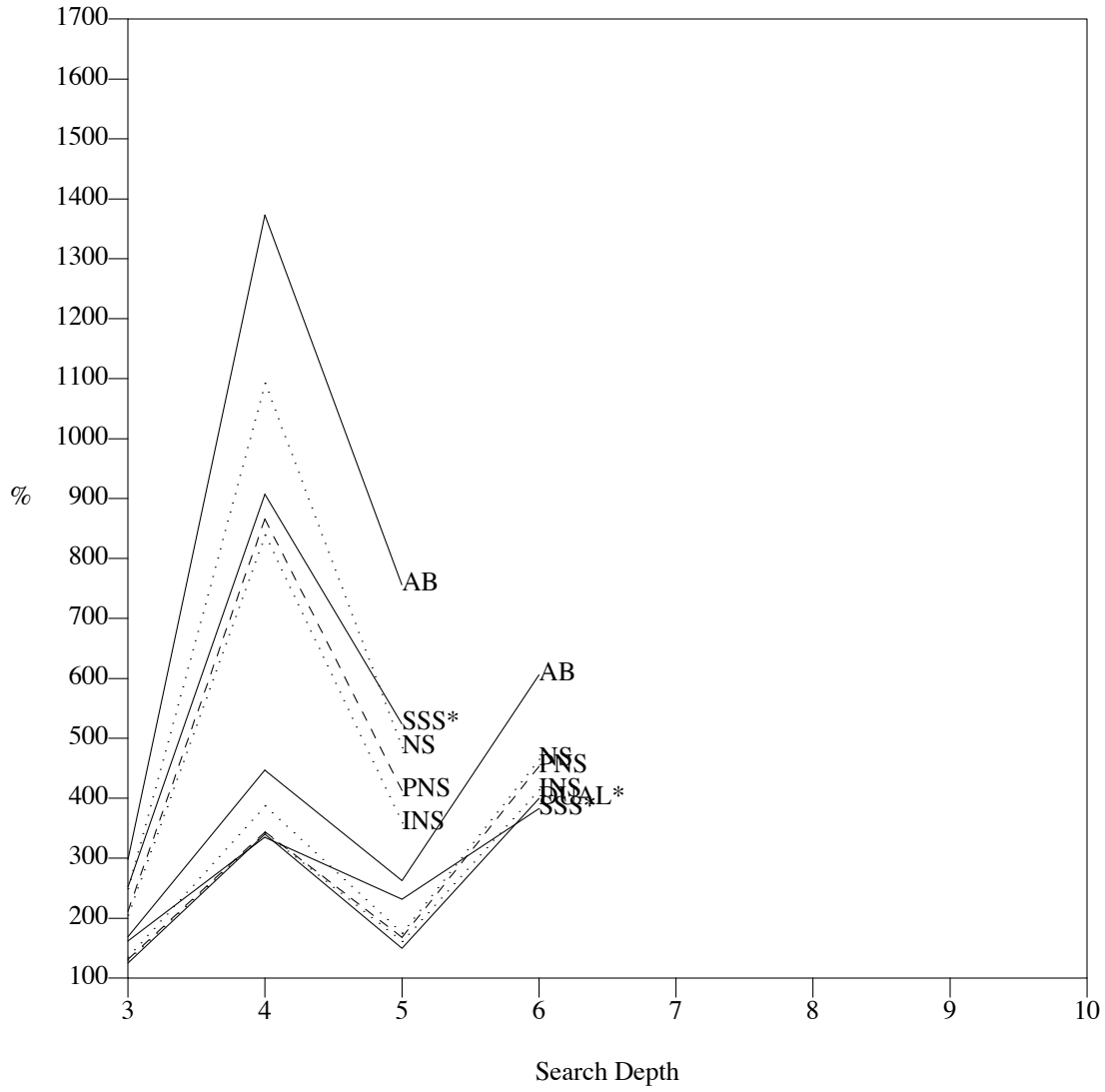


Figure 5.3: Comparison to Minimal Tree (w=20)

5.3. Storage Requirements

The space needs of the algorithms considered are summarized in Table 5.4. Note that a node descriptor, as used by SSS* and DUAL*, consists of a node identifier, the subtree status, its value and a pointer to the next descriptor. These four fields occupy three integer cells.

5.4. Elapsed CPU 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. Implementation dependencies complicate a direct CPU time comparison. One might argue that the C language is well suited to the arithmetic and logical operations used by $\alpha\beta$, NS and PNS, but not appropriate for the pointer structures used in INS, SSS* and DUAL*. The efficiency of various abstract data types for the OPEN list should also be carefully considered. For example, implementing the OPEN list as a partially ordered tree structure might simplify the insertion of new elements but slow down the removal of the top item, an action that occurs far more frequently. Despite these problems, a rough indication of the CPU time consumption is helpful.

SSS*:	$w^{\lceil \frac{d}{2} \rceil}$ node descriptors
DUAL*:	$w^{\lfloor \frac{d}{2} \rfloor}$ node descriptors
INS:	$\sum_{i=1}^{\lfloor \frac{d-1}{2} \rfloor} w^i + \sum_{i=1}^{\lfloor \frac{d-1}{2} \rfloor} w^i$ integers, for $d \geq 2$ (ignore left information) + (prove best information)
PNS:	$\frac{d^2}{2} + w$ integers, for $d \geq 2$ (ignore left information + prove best information)
DUAL-INS:	Max { DUAL*, INS }

Table 5.4: Storage requirements

Each algorithm has some overhead associated with it that affects the execution time of the program. This overhead can be very small (as in the case of NS) or very large (as for SSS*). Whether this overhead is significant or not depends on the cost of terminal node evaluations. For example, if evaluations are very expensive, then an algorithm with high overhead that builds small trees may be better than an algorithm with less overhead but builds larger trees; the total cost of the evaluations makes the algorithm overhead insignificant. Figure 5.5 shows how the terminal node evaluation time affects the total search time. Each datapoint represents an average over 40 trees, 10 each with w = 20 and d from 3 to 6. For example, if a terminal node evaluation costs 100 μsecs, $\alpha\beta$, NS and PNS require about the same search time, and INS about 25% longer. Off the graph is data showing that DUAL* is 5 times slower, and SSS* is more than 10 times slower than INS. Profiles of SSS* indicate that 90% of its time is spent in adding to and deleting from the OPEN list. DUAL* is also degraded by these operations but to a lesser extent because its OPEN list is always shorter.

The graphs suggest that in the long run INS is best, but PNS also deserves consideration when the time for a terminal node evaluation is less than 1000 μsecs. Both are better than NS and $\alpha\beta$, despite the slightly more time consuming node processing. Extrapolation of the results indicates that SSS* and DUAL* are not cost effective unless the terminal node evaluation time is very large.

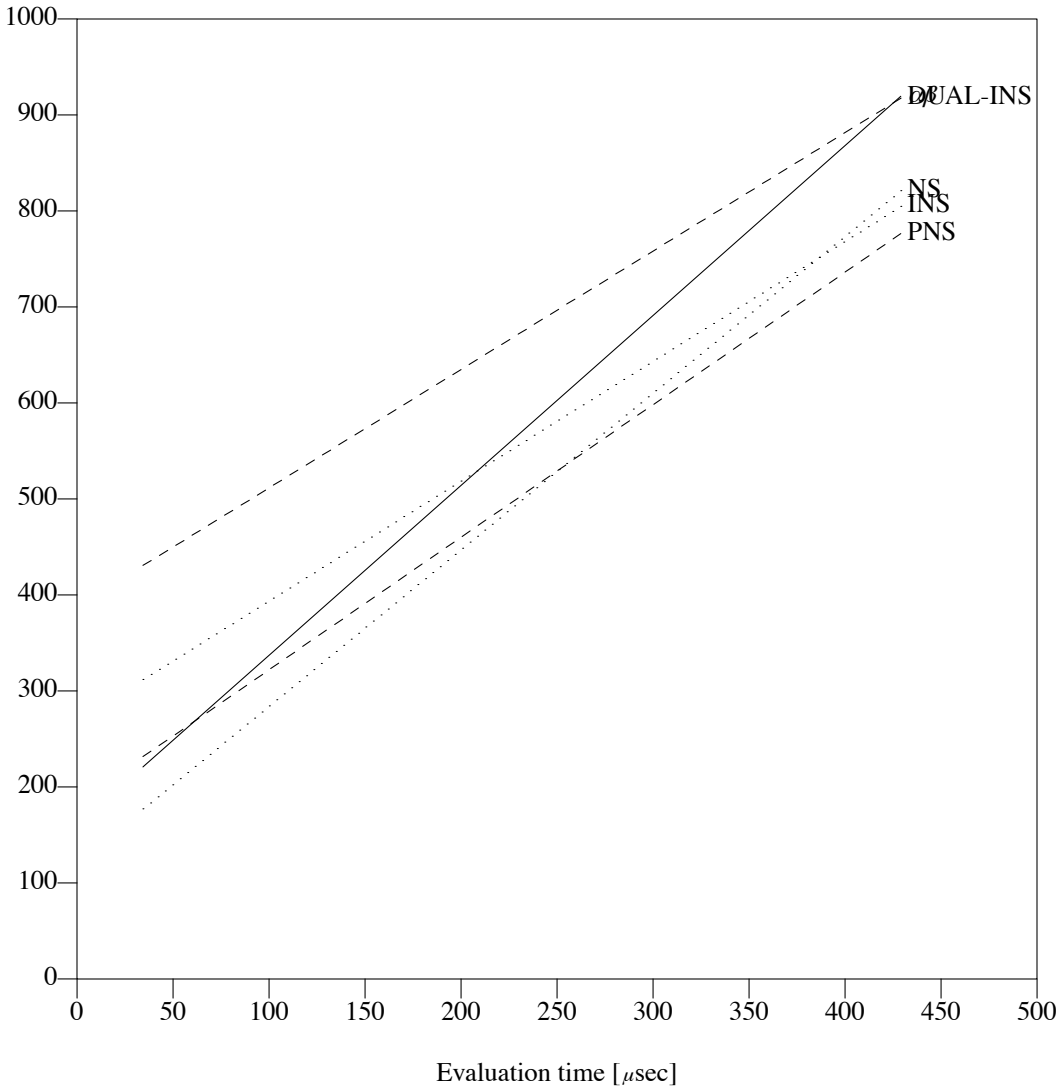


Figure 5.5: CPU comparison (w=20,d=3-6 trees)

6. Discussion

Various search strategies have been compared on tree size as well as time and space efficiency. Naturally, the simple minimal window strategies consume the least CPU time with NS about 10-15 times faster than SSS*. Gathering a small amount of information yields the PNS variant that reduces the tree size with little CPU time and space overhead. INS, on the other hand, uses more data to reduce further the leaf node visits, and can be implemented to use any size of additional storage.

Two things work in favour of minimal window techniques. First, an inferiority proof costs fewer node expansions than finding the value of a superior subtree and, secondly, the probability of a new superior subtree is small; especially under conditions of strong ordering. 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 value is found. Consequently, DUAL* usually expands fewer nodes than INS, which in turn also expands fewer nodes 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.

7. References

[AkN77] A. Akl and M. Newborn, "The Principal Continuation and the Killer Heuristic," *ACM Nat. Conf. Procs.*, Seattle, Oct. 1977, 466-473.

[Bau78] G. M. Baudet, The design and analysis of algorithms for asynchronous multiprocessors, Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, Apr. 1978.

[CaM83] M. S. Campbell and T. A. Marsland, ''A comparison of minimax tree search algorithms,'' *Artificial Intelligence* **20**(4), 347-367 (July 1983).

[Fis83] J. Fishburn, "Another optimization of alpha-beta search," *ACM Sigart Newsletter*, Apr 1983, 37-38.

[Fis81] J. P. Fishburn, Analysis of speedup in distributed algorithms, University of Wisconsin, Tech. Rep. 431, Madison, May 1981.

[KeR78] B. W. Kernighan and D. M. Ritchie, The C programming language, Prentice-Hall, New Jersey, 1978.

[KnM75] D. E. Knuth and W. Moore, ''An analysis of alpha-beta pruning,'' *Artificial Intelligence* **6**, 293-326 (1975).

[KuK83] V. Kumar and L. N. Kanal, Parallel Branch-and-Bound formulations for and/or tree search, Technical Report Tech. Rep. 83-14, 1983.

[Mar83] T. A. Marsland, "Relative efficiency of alpha-beta implementations," *8th Int. Joint Conf. on AI Conf. Procs.*, Karlsruhe, 1983, 763-766.

[MaC82] T. A. Marsland and M. S. Campbell, ''Parallel search of strongly ordered game trees,'' *ACM Computing Surveys* **14**(4), 533-552 (Dec. 1982).

[MuS85] A. Musczycka and R. Shinghal, ''An Empirical Comparison of Pruning Strategies in Game Trees,'' *IEEE Trans. on Systems, Man and Cybernetics* **SMC-15**(3), 389-399 (1985).

[Nil80] N. J. Nilsson, Principles of Artificial Intelligence, Tioga Publishing, Palo Alto, CA, 1980.

[Noe80] T. Noe, A comparison of the alpha-beta and Scout algorithms using the game of Kalah, UCLA-ENG-CSL-8017, Los Angeles, 1980.

[Pea80] J. Pearl, ''Asymptotic properties of minimax trees and game searching procedures,'' *Artificial Intelligence* **14**, 113-138 (1980).

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

[RSM85] A. Reinefeld, J. Schaeffer and T. A. Marsland, "Information acquisition in Minimal Window Search," *9th Int. Joint Conf. on AI Conf. Procs.*, Los Angeles, 1985, 1040-1043.

[Sto79] G. C. Stockman, ''A minimax algorithm better than alpha-beta,'' *Artificial Intelligence* **12**(2), 179-196 (1979).