A COMPARISON OF MINIMAX TREE SEARCH ALGORITHMS

M.S. Campbell
and
T.A. Marsland

Technical Report TR82-3

July 1982

A Comparison of Minimax Tree Search Algorithms

Murray S. Campbell*
and
T. A. Marsland

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

ABSTRACT

Although theoretic performance measures of most game-searching algorithms exist, for various reasons their practicality is limited. This paper examines and extends the existing search methods, and reports on empirical performance studies on trees with useful size and ordering properties. Emphasis is placed on trees that are strongly ordered, i.e. similar to those produced by many current game-playing programs.

1. Introduction

This paper attempts to provide a useful comparison between algorithms that search game trees. Asymptotic behavior [1, 7, 11] has limited relevance to practical situations, while more useful effort measures have typically been focused on random trees [5] (though [10] in particular attempted an analysis of alpha-beta on small trees with branch dependent scores). What is required is some indication of the relative merits of the various algorithms in cases of practical interest, i.e. for trees that are moderately or strongly ordered. This paper describes empirical studies of the search algorithms with the goal of finding the most efficient algorithms under different ordering assumptions. An extension of this study to the development of parallel search algorithms has been done [3].

The type of trees considered in this paper are those of games such as chess, which are classified as two-person, zero-sum games of perfect information. Given a position p in such a game, it is possible to represent all the potential continuations from p in the form of a *game tree*. The nodes of the tree correspond to game positions, while the branches represent the moves. The leaves of a game tree are called *terminal* nodes, and are assigned a value by a *static evaluation function*. All the remaining nodes are classified as *non-terminal*. The task in searching a game tree is to determine the *minimax* value of the root node.

---

*Present address: Computer Science Department, Carnegie-Mellon University, Pittsburgh, PA 15213, U.S.A

Intuitively, the minimax value of a node is the best score achievable from that node against an opponent who similarly chooses his best moves.

Some further terminology associated with game trees: A node is at *depth k* if it is k moves, or k *ply*, from the root. The number of branches leaving any particular non-terminal node is the *branching factor* of that node. A *uniform game tree* is one in which all non-terminal nodes have the same branching factor, and all terminal nodes are at the same depth in the tree.

2. Algorithm Descriptions

The **minimax algorithm** assumes there are two players called Max and Min, and assigns a value to every node in a game tree (and in particular to the root) as follows: Terminal nodes are assigned static values that represent the desirability of the position from Max's point of view. Non-terminal nodes can be given the minimax value recursively. If a non-terminal node p has Max to move, then the value of p is the maximum over the values of the successors of p. Similarly, if Min is to move he will choose the minimum over the values of the successors of p. The sequence of moves which minimax predicts as optimal for both sides is called the *principal variation.*

The **negamax algorithm** [7] is a variant of the minimax procedure which is often more convenient to use. In the negamax approach, the terminal nodes are assigned static values from the point of view of the side to move. This allows the value of non-terminal positions to be calculated as the maximum of the negatives of the values of their successors, i.e. the negamax algorithm applies the same operator at all levels in the tree. Negamax avoids having separate cases for Max to move and Min to move (this is done implicitly in the procedure staticvalue), and will be used throughout this paper. See Figure 2-1 for the negamax algorithm.

The **alpha-beta algorithm** is able to evaluate a game tree at reduced cost by ignoring subtrees that cannot affect the final value of the root node. Such subtrees are said to have been *cut off.* The program in Figure 2-2 implements this idea in an C/PASCAL-like language by maintaining two bounds, alpha and beta, for even and odd levels of the tree against which cutoffs can be made. The following functions are assumed to exist:

- terminal(p) - returns true if p is a terminal position.
- staticvalue(p) - returns an integer giving the value of p for the side to move.
- generate(p) - generates all the successors of p and returns the number of successors.

For most games it is usually not feasible to search the entire tree to the terminal nodes as specified by the rules of the game. Under these circumstances the function terminal can, using some arbitrary criteria (such as

```
negamax(p : position)
{
    m, i, t, w : integer;
    if (terminal(p))                /*  p is a terminal node  */
        return(staticvalue(p));

    w = generate(p);                /*  determine successors  */
                                    /*      p1 ... pw  */
    m = -∞;
    for i = 1 to w do
    {
        t = -negamax(pi);
        if (t > m)
            m = t;
    }
    return(m);
}
```

Figure 2-1: Negamax

```
1.     alphabeta(p : position; α, β : integer)
2.     {
3.         m, i, t, w : integer;
4.         if (terminal(p))                  /*  p is a terminal node   */
5.             return(staticvalue(p));
6.
7.         w = generate(p);                  /*  determine successors   */
8.                                           /*      p1 ... pw  */
9.         m = α;
10.        for i = 1 to w do
11.        {
12.            t = -alphabeta(pi, -β, -m);
13.            if (t > m)
14.                m = t;
15.            if (m ≥ β)
16.                return(m);             /*  cutoff  */
17.        }
18.        return(m);
19.    }
```

Figure 2-2: Alpha-beta

depth), assign nodes to be terminal. This requires staticvalue to be an estimate of a node value.

The interval enclosed by (α,β) is referred to as the alpha-beta *window*. For the alpha-beta algorithm to be effective, the minimax score of the root node must lie within the initial window. This can be ensured by setting the window to $(-\infty,+\infty)$ (where $+\infty$ $(-\infty)$ is some value larger (smaller) than any returned by *staticvalue*). Generally speaking, however, the narrower the initial window, the better the algorithm's performance. This provides the motivation for *aspiration searching* [6], in which the window is initialized to (V−e,V+e), where V is an estimate of the minimax value and e the expected error.

There are three possible outcomes of an aspiration search on a position p, depending on S, the minimax score of p [7].

1. if $S \leq V-e$, alphabeta(p,V−e,V+e) $\leq V-e$
2. if $S \geq V+e$, alphabeta(p,V−e,V+e) $\geq V+e$
3. if $V-e < S < V+e$, alphabeta(p,V−e,V+e) = S

Cases 1 and 2 are called *failing low* and *failing high* respectively [4]. Only in case 3 is the true score of p found. Searches that fail high or low must be repeated with a window that actually encloses S.

The aspiration search concept has spawned a number of variants on alpha-beta that attempt to employ the technique to improve search speed. Some of these alpha-beta modifications can profit from **falphabeta** [4], for "fail-soft-alphabeta". It has been noted that, when an aspiration search fails the search must be repeated with a more realistic window. If a search with window (V−e,V+e) fails high, for example, the window $(V+e,+\infty)$ is guaranteed to find the true score. Falphabeta can sometimes return a tighter bound on the score of the tree, and the second search can use this bound to advantage.

There are two differences between alphabeta and falphabeta. Line 9 in alphabeta becomes $m = -\infty$ (instead of $m = \alpha$), and line 12 becomes $t = -\text{falphabeta}(p_i, -\beta, -\max(m, \alpha))$. It has been proven [4] that, given a position p and window (a,b), f = falphabeta(p,a,b) obeys the following relation:

1. if $f \leq a$, negamax(p) $\leq f$
2. if $f \geq b$, negamax(p) $\geq f$
3. if $a < f < b$, negamax(p) = f

Thus a search that, say, fails high can use (f,∞) as the window for the second search, rather than (b,∞). Falphabeta is guaranteed to search the same nodes as alphabeta, and the only overhead of falphabeta is the

'max' operation in the recursive procedure call.

The most obvious application of aspiration searching has already been mentioned, namely guessing an initial window and repeating the search in case of failure high or low. Though this method is more effective if there is some prior knowledge about the score distribution, it is applicable even in the absence of such information. Aspiration searching benefits from the tighter bounds returned by falphabeta.

In the development of further optimizations of alpha-beta, the concept of a *minimal window* [4] or a *bound-testing procedure* [11] was introduced. Though the true score of a position p cannot be found by these methods, they do provide a bound on the score (i.e. determines whether or not negamax(p) > m), while making cutoffs that a full window search cannot. For example, if the score of one successor of p is found to exceed a bound m, an immediate cutoff can occur without searching the remaining successors of p for the one that exceeds m by the most. In many circumstances a bound of this type on a position is sufficient. Assuming scores can only take integer values, (m, m+1) is an example of a minimal window. With such a window, the search will necessarily fail high or low, but the direction of the failure will locate the score of p with respect to the bound m.

Lalphabeta [4], for "last-move-with-minimal-window alphabeta", is one application of this concept. The last successor of the root node is compared with the bound m, where m is the best score found so far. If the search fails low, the previously established best move still applies. If the search fails high the last move has been determined to be best, though the precise score is not known.

All the search algorithms discussed so far have been *directional*, i.e. there is some linear arrangement of the terminal nodes such that the algorithms never examine a node to the left of one previously examined [11]. All the remaining algorithms in this section are *non-directional*; no guarantee can be made about a 'left-to-right' examination of the terminal nodes.

Palphabeta [4], for "principal-variation alphabeta", is a generalized application of minimal window searching and is presented in Figure 2-3. If the first path to a terminal node is in fact the principal variation predicted by minimax, the balance of the tree is searched with a minimal window. However each time a minimal window search on a subtree fails high, the search is repeated with a wider window. Hence there is some risk, if the tree is poorly ordered, that palphabeta will visit more terminal nodes than alphabeta. There exist certain techniques, particularly *iterative deepening* [9, 15], which can provide a prefix to the actual principal variation with reasonable reliability. Such techniques increase the feasibility of palphabeta. An enhancement to palphabeta, used in the chess machine Belle**, continues searching with a minimal window,

** Ken Thompson, personal communication.

even after a fail high search. If there are no further fail high results, the best move has been located. In the event of a second fail high result, the window must be opened up to determine the new best move. As in lalphabeta, this technique does not always determine the score of a tree.

SCOUT [11] is a further generalization of palphabeta, where instead of calling alphabeta after a minimal window search fails high, a recursive call is made to SCOUT. As in palphabeta, the possibility for reexamining nodes in this case makes SCOUT non-directional. Figure 2-4 gives an adaptation of SCOUT, reformulated into the negamax approach. The original version of SCOUT did not employ the minimal window idea, but rather a similar procedure, TEST [11], which is applicable to both continuous and discrete score distributions. TEST could be used instead of the minimal window call to alphabeta in the body of the SCOUT procedure. TEST is given in Figure 2-5, also reformulated into the negamax framework. A potential disadvantage of SCOUT is that, unlike palphabeta, the bound that is returned by TEST (or a minimal window search) is not used to further reduce search.

SSS* [16] is a non-directional algorithm for determining the minimax value of AND/OR trees, of which game trees are a special case. It is claimed that SSS* *dominates* alpha-beta in terms of terminal nodes evaluated, that is "SSS* never scores a node that alpha-beta can ignore" [16]. For practical score distributions, SSS* can be expected to evaluate strictly fewer nodes than alpha-beta. This is achieved by means of a very large data structure, called the *OPEN list*, which simultaneously maintains a number of alternate solution paths throughout the tree. In uniform game trees of depth d and width w, the OPEN list is of order $w^{d/2}$ elements.

The description of SSS* relies on the following definitions, adapted from Stockman [16]. A game tree is an AND/OR tree whose root node is of type AND, all immediate successors of AND nodes are of type OR, and all immediate successors of OR nodes are of type AND. A *solution tree* T of a game tree S is a tree with the following characteristics:

- The root nodes of S is the root node of T.
- If a non-terminal node of S is in T, then all of its immediate successors are in T if they are of type AND, and exactly one of its immediate successors is in T if they are of type OR.

The *value* of a solution tree T is denoted as $f_T(p)$, and is defined as the minimum value of all terminal nodes in T. It can be shown that for a solution tree T of a game tree S, with root p, minimax(p) $\geq f_T(p)$, and for some solution tree T_0, minimax(p) = $f_{T_0}(p)$.

Let T be a potential solution tree of game tree S. A *state of traversal* of T is a triple (n,s,h) where n is a node of T, s is the status of the solution of n (either LIVE or SOLVED), and h is the merit of the state.

```
palphabeta(p : position)
{
    m, i, t, w : integer;
    if (terminal(p))
        return(staticvalue(p));

    w = generate(p);    /*  determine successors  */
                        /*       p1 ... pw        */
    m = -palphabeta(p1);
    for i = 2 to w do
    {
        t = -falphabeta(pi, -m-1, -m);
        if (t > m)
            m = -alphabeta(pi, -∞, -t);
    }
    return(m);
}
```

Figure 2-3: Palphabeta

```
scout(p : position)
{
    m, i, t, w : integer;
    if (terminal(p))
        return(staticvalue(p));

    w = generate(p);        /*  determine successors  */
                            /*      p_1 ... p_w       */
    m = -scout(p_1);
    for i = 2 to w do
    {
        t = -alphabeta(p_i, -m-1, -m);
        if (t > m)
            m = -scout(p_i);
    }
    return(m);
}
```

Figure 2-4: SCOUT

```
test(p : position; v : integer)
{
    i, w : integer;
    if (terminal(p))
        if (staticvalue(p) > v)
            return(TRUE);
        else
            return(FALSE);

    w = generate(p);        /*  determine successors  */
                            /*      p_1 ... p_w       */
    for i = 1 to w do
    {
         if  (not(test(p_i, -v))
            return(TRUE);
    }
    return(FALSE);
}
```

Figure 2-5: TEST

Now it is possible to give the SSS* algorithm:

1. Place the start state (ROOT, LIVE, $+\infty$) on a list called OPEN.

2. Remove from OPEN the state p=(n,s,h) with largest merit h. Since OPEN is kept in non-decreasing order of merit, p is first in the list.

3. If n = ROOT and s = SOLVED, terminate with h as the minimax value of the tree.

4. Expand state p by applying state space operator Γ, described in Table 2-1.

5. Go to 2.

The Γ operator requires the functions *type, first, next, parent* and *ancestor*, which are self explanatory.

It can be shown that the original version of SSS* does not necessarily dominate alpha-beta when equa valued terminal nodes are involved. Consider the tree in Figure 2-6 (where boxes represent AND nodes an circles OR nodes). Alpha-beta will do terminal evaluations on nodes c, g, and h only. However the OPEN lis in Figure 2-7 indicates that SSS* evaluates in addition nodes i and j.

To guarantee that SSS* dominates alpha-beta, the Γ operator [16] must be altered, as in Table 2-2. Th modification ensures that, when there are several states with equal merit, the *left-most* is always examine first. One beneficial side effect of the left-bias introduced here is improved searching performance if the tre being searched is strongly ordered, as is usually the case for practical applications of game tree search. Th modified version of SSS* will be used in the comparative studies in this paper.

To further improve searching performance, SSS* can draw upon the aspiration search idea used in alpha beta. Assume SSS* is given a window (a,b) over which to conduct the search. The use of the lower bound trivial; if at any point the top item in the OPEN list has score h $\leq$ a, the search can be terminated, returning as an upper bound on the actual position score. The use of the upper bound b is more interesting. Instead o initializing the OPEN list to (ROOT,LIVE,$+\infty$), (ROOT,LIVE,b) is used. As an example of how a searc reduction can take place, consider the subtree in Figure 2-8. Assume that an upper bound of 5 is supplied fo the tree score. The OPEN list, Figure 2-9, at step 4 is

(f,LIVE,5) (g,LIVE,5) ... #

Applying the appropriate operator in SSS*, the list becomes

(f,SOLVED,min(5,7)) (g,LIVE,5) ... #

or

(f,SOLVED,5) (g,LIVE,5) ... #

The next step gives

(d,SOLVED,5) ... #

Case of Operator Γ	Conditions satisfied by input state (n,s,h)	Action of Γ
1	s = solved n ≠ ROOT type(n)=OR	Stack (m=parent(n),s,h) on list. Then purge OPEN of all states (k,s,h) where m is an ancestor of k.
2	s=SOLVED n≠ROOT type(n)=AND next(n)≠NIL	Stack (next(n),LIVE,h) on OPEN list.
3	s=SOLVED n≠ROOT type(n)=AND next(n)=NIL	Stack (parent(n),s,h) on OPEN list.
4	s=LIVE n is terminal	Insert (n,SOLVED,min(h,value(n)) on OPEN list behind all states of greater or equal merit.
5	s=LIVE n is non-terminal type(first(n))=AND	Stack (first(n),s,h) on OPEN list
6	s=LIVE n is non-terminal type(first(n))=OR	n=first(n) while n≠NIL do stack (n,s,h) on OPEN n = next(n)

Table 2-1: The Γ Operator

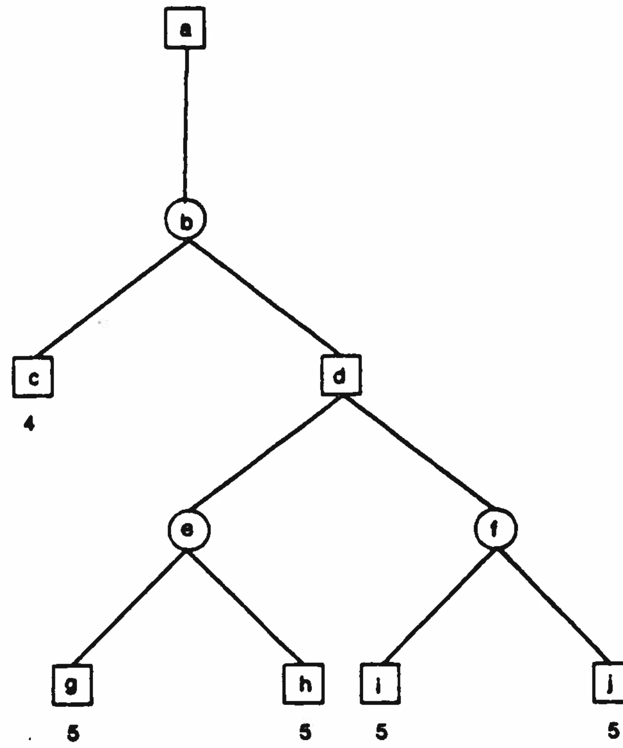

Figure 2-6: Tree illustrating non-dominance of SSS*

1. (a, L, ∞) #
2. (b, L, ∞) #
3. (c, L, ∞) #
4. (c, S, 4) #
5. (d, L, 4) #
6. (f, L, 4) (e, L, 4) #
7. (i, L, 4) (e, L, 4) #
8. (c, L, 4) (i, S, 4) #
9. (g, L, 4) (i, S, 4) #
10. (i, S, 4) (g, S, 4) #
11. (j, L, 4) (g, S, 4) #
12. (g, S, 4) (j, S, 4) #
13. (h, L, 4) (j, S, 4) #
14. (j, S, 4) (h, S, 4) #
15. (f, S, 4) (h, S, 4) #
16. (d, S, 4) #
17. (b, S, 4) #
18. (a, S, 4) #

Figure 2-7: OPEN list for Figure 2-6

Case of Operator Γ	Conditions satisfied by input state (n,s,h)	Action of Γ
4	s=LIVE n is terminal	Insert (n,SOLVED,min(h,value(n)) on OPEN list behind all states of greater merit, and in front of all states of equal merit where n is to the left of the given node.

6	s=LIVE
	n is non-terminal
	type(first(n))=OR
		n=last(n)
		while n≠NIL do
		  stack (n,s,h) on OPEN
		  n=previous(n)

Table 2-2: Modified cases of the Γ Operator

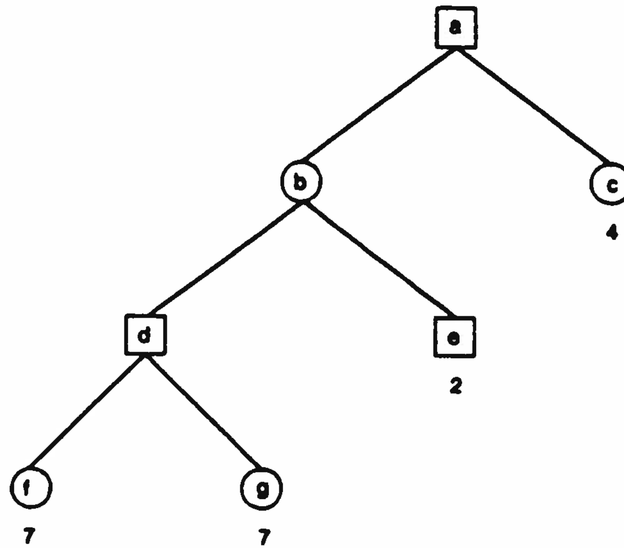

Figure 2-8: Tree illustrating aspiration SSS*

1. (a, L, 5) #
2. (b, L, 5) (c, L, 5) #
3. (d, L, 5) (c, L, 5) #
4. (f, L, 5) (g, L, 5) (c, L, 5) #
5. (f, S, 5) (g, L, 5) (c, L, 5) #
6. (d, S, 5) (c, L, 5) # node g is cut off
7. (e, L, 5) (c, L, 5) #
8. (c, L, 5) (e, S, 2) #
9. (c, S, 4) (e, S, 2) #
10. (a, S, 4) #

Figure 2-9: OPEN list for Figure 2-8

and node g is cutoff without evaluation. The intuitive justification of this is as follows: Since $f \geq 5$, and at d it is MAX to move, $d = \max(5,g)$ implies $d \geq 5$. This bound on d is sufficient to terminate search, since by the assumption that the score of the tree is less than 5, d cannot possibly be on the principal variation, regardless of the value of g. Aspiration SSS* can also be employed as a bound testing procedure, similar to TEST, by the use of a minimal window. The preference for the leftmost of equal merit states aids aspiration SSS* in case there is a fail high result, since the leftmost subtree is evaluated fully before later subtrees are examined.

SSS* performs particularly well, relative to alpha-beta, when the actual principal variation is to the right of the game tree. However in trees that are strongly ordered, the advantage is diminished. Since the SSS* algorithm utilizes more background computations than alpha-beta (for example, there is an insert into an ordered list), SSS* may only be practical if there is reason to believe that it will substantially outperform alpha-beta in number of positions examined, and even then the storage requirements limit the size of the tree examined.

Because of these problems, Stockman suggested that an amalgamation of SSS* and alpha-beta may be more practical. A first possibility would employ alpha-beta at the top levels of the tree, with SSS* used to reduce terminal node evaluations at deeper levels. An important point to note is that aspiration SSS* is very useful in such an algorithm. The nodes at the maximum search depth for alpha-beta will have a window which can be used to advantage by SSS*. An alternative would employ SSS* at top levels of the tree, with alpha-beta used to search deeper in the tree. Though it might appear that this method is better than alpha-beta/SSS*, particularly for random trees, there is a serious disadvantage present. This arises from the fact that SSS* has no lower bound on a node score comparable to the alpha value in the alpha-beta algorithm. Thus when alpha-beta is called from SSS*, alpha must be initialized to $-\infty$ to guarantee success. The effects of this will be seen in the empirical studies. However, both alpha-beta/SSS* hybrids can significantly reduce storage requirements.

Another variation on SSS* was proposed [3] which employs the algorithm to some fixed depth d, whereupon the 'terminal nodes' at d ply are evaluated by a further depth d SSS* search. The staging reduces storage requirements so that they are linear with search depth. This approach uses SSS* in layers, or stages, and is called staged SSS* [3]. As before, staged SSS* suffers from the fact that no lower bound is available on any given node's score.

3. Performance Comparison

3.1. Performance Measures

There are a number of alternative methods to measure the performance of algorithms that search game trees. These measures and their relative merits are now examined before presenting our search results.

Elapsed CPU time is excellent as a performance measure provided (1) a comparison between algorithms is done very carefully, and (2) the values are used only to determine relative performance. In other words, this measure is most useful when comparing the relative performance of a single algorithm on different types or sizes of trees. Relaxing the above restrictions reduces the validity of the measure. Comparing two different algorithms in this way is highly dependent upon the relative efficiency of the encodings of the algorithms. In addition, the values are machine dependent. It should be noted, however, that elapsed CPU time is the only performance measure discussed here which captures the idea that a more time-consuming algorithm is less desirable than a faster one, all else being equal.

NBP, for *Number of Bottom Positions* is a common measure of search performance [14]. By simply counting the number of terminal node evaluations, NBP provides a means of comparing algorithms' performances on trees of practical sizes. However NBP does not measure the amount of processing that must be done in order to choose nodes for evaluation, and makes the implicit assumption that terminal evaluations are the major cost in a tree search.

Total Nodes Visited is similar to NBP except it includes the cost of non-terminal nodes as well. The total nodes visited is rarely used as a performance metric for sequential programs, as it has a very similar character to the more easily calculated NBP.

The **asymptotic branching factor** as a cost measure can be defined as follows:

If $N_{w,d}$ is the number of terminal nodes examined by some algorithm A in searching a uniform tree of width w and depth d, then

$$\lim_{d \to \infty} (N_{w,d})^{1/d}$$

is called the *branching factor* of algorithm A [1].

This cost measure often has limited practical application, as depths of trees necessary to display the asymptotic properties may be computationally infeasible to search.

It is clear that CPU time is a very good performance measure for practical systems, since minimizing the real search time is often the main goal. However theoretical and empirical studies rarely use this measure, since it is so dependent on the application and the efficiency of the programs. Theoretical studies have concentrated on NBP and branching factor as performance measures, while empirical studies (including this one) usually measure NBP.

The theoretical performance characteristics of some of the previously discussed algorithms are now examined. For purposes of analysis it is convenient to compare searching performance on uniform game trees of depth d and width w.

The negamax algorithm visits all the nodes in a game tree, and in particular all w^d terminal nodes. Thus the branching factor of negamax is w. It has been shown that any tree searching algorithm must examine at least

$$w^{(d+1)/2} + w^{(d-1)/2} - 1 \quad \text{for odd d, and}$$
$$2w^{d/2} - 1 \quad \text{for even d}$$

terminal nodes [14]. Alpha-beta, under optimal conditions, attains this lower bound, as do palphabeta, SCOUT, and SSS*. In trees for which the w^d terminal nodes are independent identically distributed random variables with a continuous distribution function, general formulas for the average number of terminal positions scored by alpha-beta have been developed [5, 1]. In the worst case, however, alpha-beta must examine all w^d terminal nodes [14].

For continuous-valued trees, the branching factors of alpha-beta and SCOUT have been shown to be asymptotically optimal over all directional search algorithms [11, 12]. For discrete-valued trees, this result has been strengthened to optimality over all algorithms, both directional and non-directional, with branching factor $w^{1/2}$ in almost all cases [11]. SSS* also achieves the optimal branching factor in the discrete-value case. It has recently been reported [13] that alpha-beta is asymptotically optimal in continuous-valued trees as well.

Aspiration searching has been examined theoretically [2], and it is shown that for trees with typical game-playing characteristics, a speedup of between 15% and 25% can be expected. In other words, aspiration alpha-beta is, under normal circumstances, better than alpha-beta.

3.2. Algorithms Compared

At present, if it is desired to study searching performance on trees with varying types of ordering properties, only empirical methods are available. In the following study a number of algorithms will be compared:

- alphabeta (AB)

- palphabeta (PAB)
- SCOUT (SC)
- SSS* (SSS)
- staged SSS* (SS)
- SSS*/alphabeta hybrid (SAB)
- alphabeta/SSS* hybrid (ABS)

Lalphabeta, strictly speaking, is not as powerful as the other algorithms since the position score is not always determined, and will not be included here. Studies on checkers game trees [4] indicate that lalphabeta can produce a minimal savings over alphabeta (about 1.5%).

The trees searched were uniform with (width,depth) combinations: (8,2), (16,2), (24,2), (8,4), (16,4), (24,4) and (8,6). Terminal nodes were assigned values in the range (0,127), thus allowing possible duplicate scores. Increasing the branching factor of the trees did not significantly affect the relative searching performances of the various algorithms, and thus only the width 24 data is included here. For a complete tabulation of the data see [3].

It is also desirable that the trees exhibit a variety of ordering properties. The different sizes and ordering of the trees should give some indication of the strengths and weaknesses of the algorithms. In this study, trees are classified by the distribution of the location of the best move at any given node. The following distributions were employed:

Tree Order Type	Ordering Property
1	random
2	.5 first-move-best (moderately ordered)
3	geometric with parameter .5 (moderately ordered)
4	.8 first-move-best (strongly ordered)
5	geometric with parameter .8 (strongly ordered)
6	best-first (perfect) ordering

By studying trees with the above properties, it is possible to compare algorithms in situations of practical interest. Current chess programs, for example, are able to approach best-first ordering in their trees by means of various search enhancements [9]. Thus the emphasis is placed on strongly ordered trees.

The values in the following graphs were determined by the independent generation of 100 trees having the desired properties, and searching them with each applicable algorithm. 100 was chosen as the sample size for practical reasons, since generation of the larger trees is computationally expensive. Both absolute performance in terms of NBP, and performance relative to alpha-beta are shown.

4. Results

For the 2-ply case, only alphabeta, palphabeta, SCOUT, and SSS* can be compared, Figures 4-1 and 4-2, since staged SSS* and alpha-beta/SSS* hybrids are not applicable. A number of observations can be made. Firstly, all the algorithms are able to attain the minimum number of evaluations on perfectly ordered trees. Although not apparent from the data presented, in fact SSS* dominates alphabeta dominates palphabeta dominates SCOUT. This could have been predicted beforehand. SSS* is known to dominate alpha-beta. Both palphabeta and SCOUT, when carrying out their minimal window searches, repeat the search on failure high. The place where palphabeta and SCOUT try to compensate for this repetition, the failure low minimal window searches, evaluate exactly the same nodes as alphabeta in two ply trees. Thus alphabeta dominates palphabeta and SCOUT. Palphabeta dominates SCOUT because, after a failing high search, palphabeta examines all the terminal nodes of the given subtree once more, but each node that is better than its earlier siblings is examined twice more by SCOUT. In addition, palphabeta is sometimes able to use the tighter bound returned by the initial call to falphabeta to cutoff search earlier.

The values for alphabeta on random trees are found to be consistently lower than those calculated by Fuller's formula [10]. This is not unexpected, since that formula assumes distinct values for all terminal nodes. For discrete-valued trees, a search reduction is possible owing to the fact that cutoffs occur on equal scores.

When we consider 4-ply trees, Figures 4-3, and 4-4, it is possible to include staged SSS* and SSS*/alpha-beta hybrids. The SS algorithm used here has a stage depth of 2. SAB uses a two ply SSS* search on top of a two ply alpha-beta search, while ABS does the opposite. It is apparent from the data that neither SS nor SAB are able to attain the theoretical minimum NBP on perfectly ordered trees. This results from the previously mentioned fact that SSS* does not maintain a lower bound for position scores.

Some dominations can be shown to exist at 4 ply:

- SSS dominates AB
- SSS dominates SS
- SSS dominates ABS

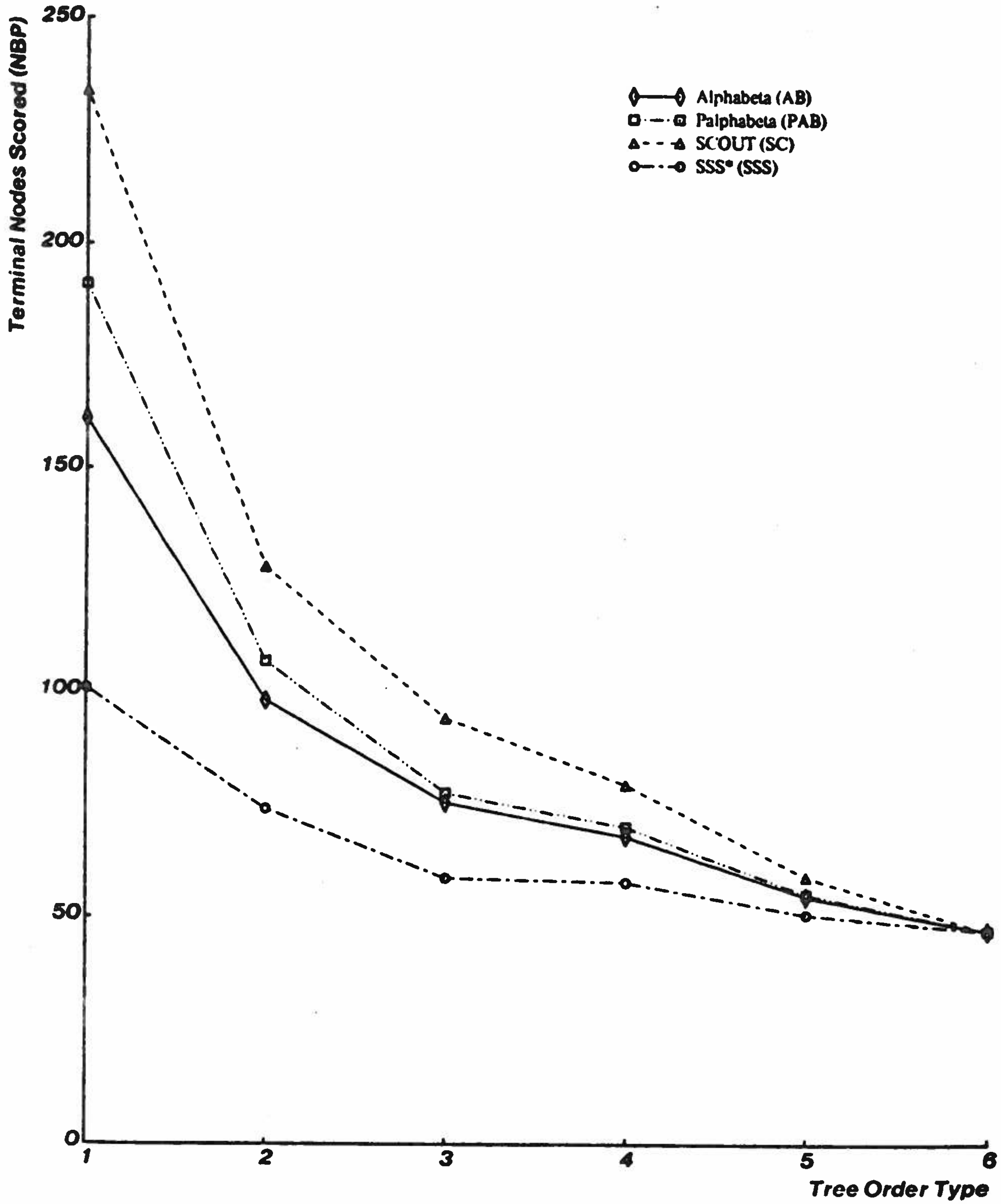

Figure 4-1: NBP - Tree width = 24, depth = 2

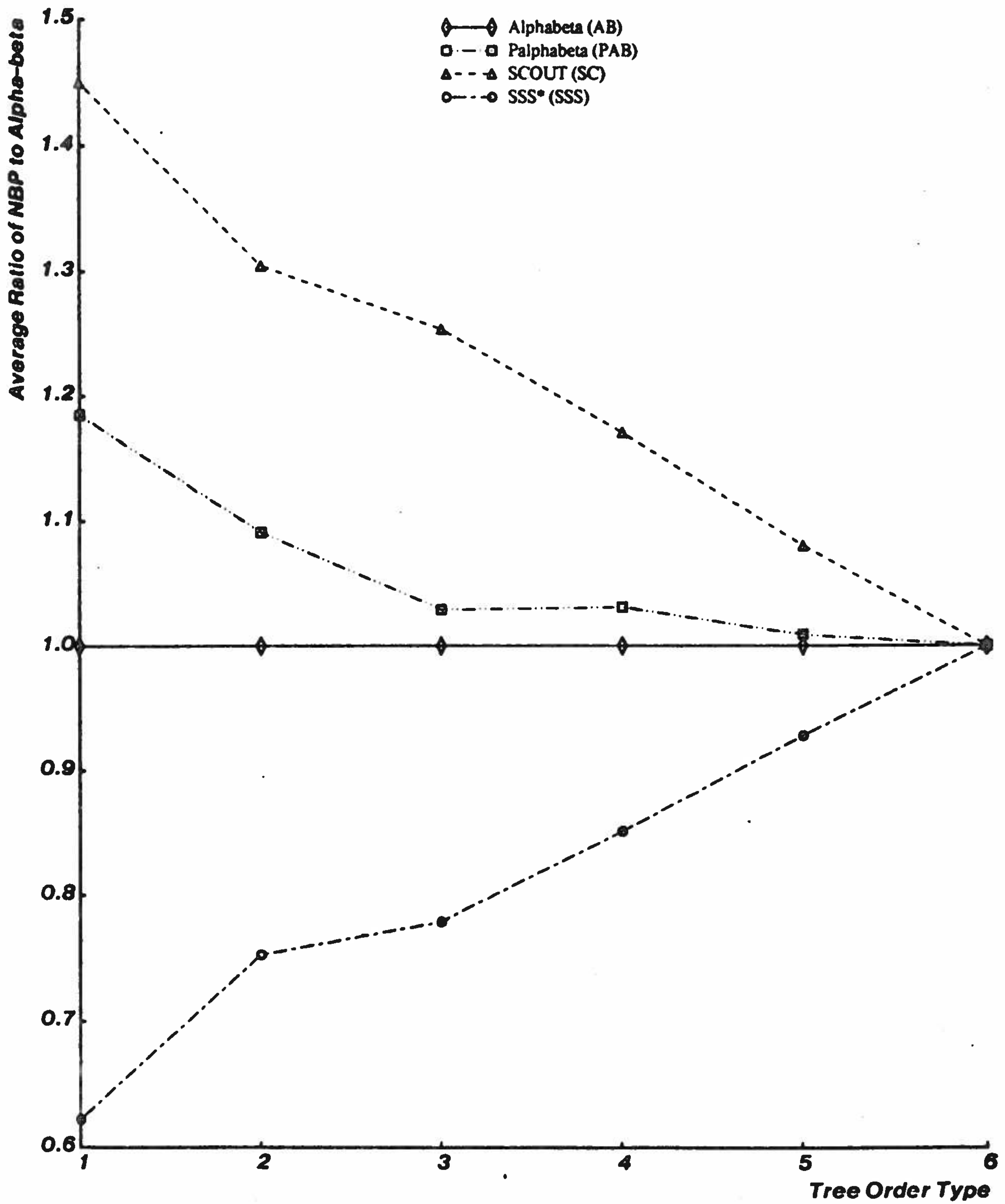

Figure 4-2: Relative performance - Tree width = 24, depth = 2

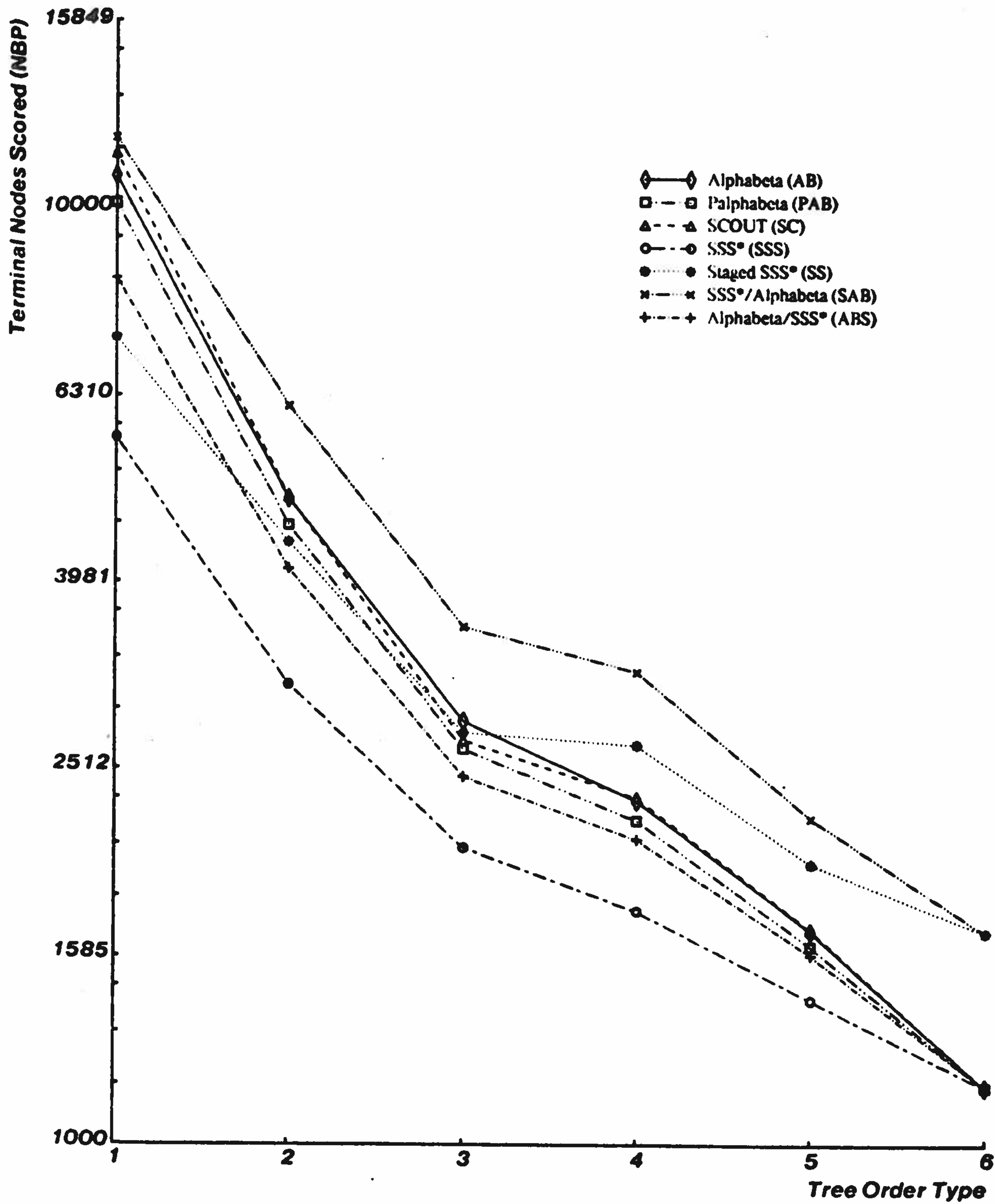

Figure 4-3: NBP - Tree width = 24, depth = 4

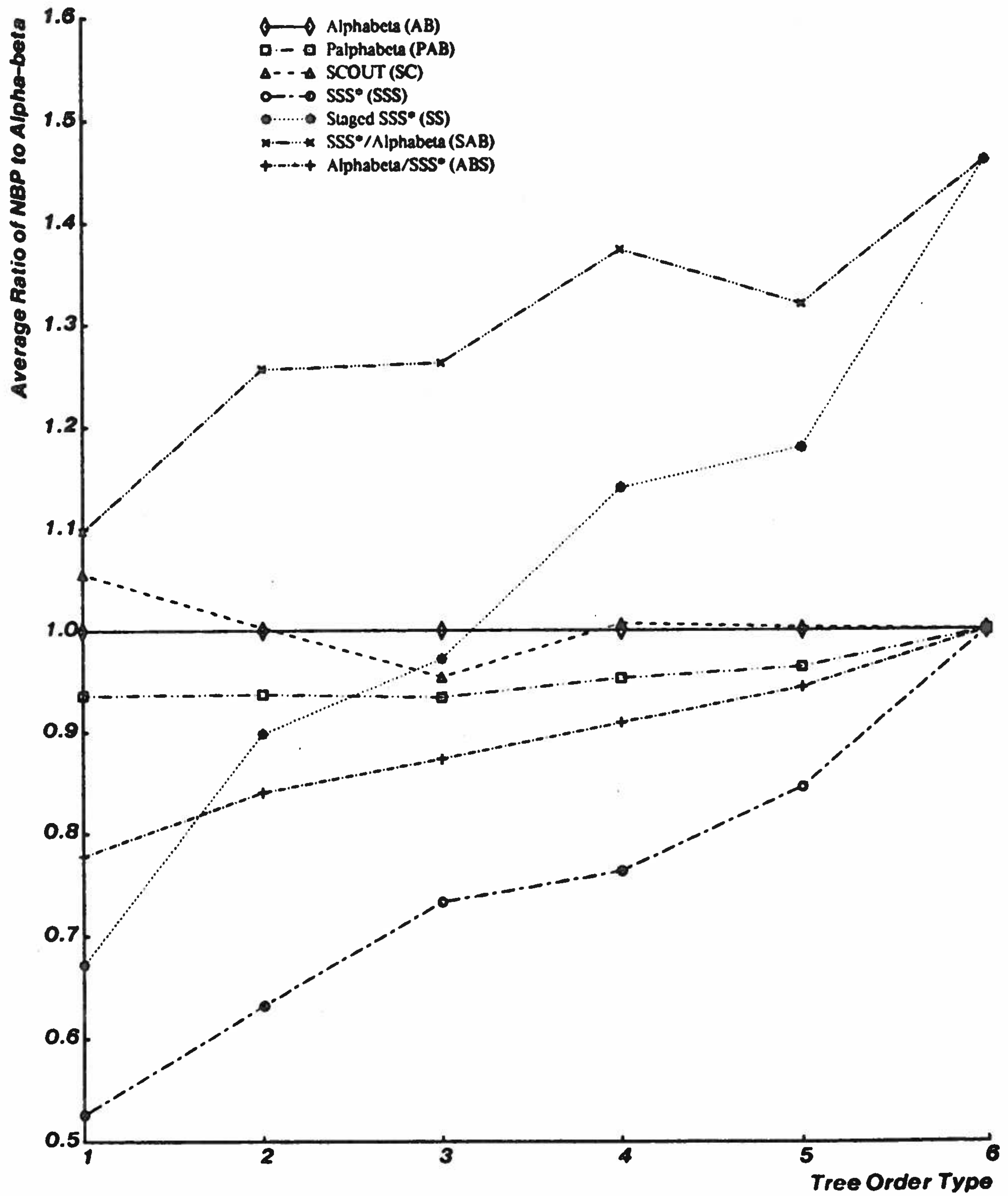

Figure 4-4: Relative performance - Tree width = 24, depth = 4

- ABS dominates AB
- SS dominates SAB

Some interesting cases of non-domination:

- PAB does not dominate SC
- SSS does not dominate SC or PAB
- SS does not dominate ABS, and vice versa

Examining the graphs, some conclusions can be drawn. For random trees, SSS, SS and ABS are noticeably superior in terms of NBP. However the standard deviations were:

- SSS - 843
- SS - 815
- ABS - 1737

This appears to indicate that SSS* and staged SSS* should be used for search of random trees. They perform well, with a relatively small standard deviation. Note that SSS* requires OPEN list storage of $24^2 =$ 576 entries, while staged SSS* only needs $24 \cdot 2 = 48$ entries.

As the trees become better ordered, PAB and SC begin to deserve consideration. For width 24 trees both methods are slightly better than AB, and under favorable conditions can beat SSS*. Both methods have a relatively large standard deviation compared to AB (524 vs. 912 and 961 are the width 24 values). ABS and SSS remain good choices for ordered trees from a node evaluation point of view. However their advantage over AB is proportionally less, and the more time-consuming algorithms become less attractive.

Some further tests were conducted, to model iterative deepening, so that palphabeta and SCOUT can be compared with alphabeta on trees which had the correct principal variation but were random otherwise. PAB and SC searched the same nodes, while AB searched about 40% more nodes (on trees of depth 4 and width 8).

6 ply data, not included here, provides further support for the conclusions drawn from the 4 ply data. Additional algorithms tested were:

- SS - staged SSS* with 3 stages of depth 2.
- SA2 - 2 ply SSS* over 4 ply alphabeta.

- SA4 - 4 ply SSS* over 2 ply alphabeta.
- AS2 - 2 ply alphabeta over 4 ply SSS*.
- AS4 - 4 ply alphabeta over 2 ply SSS*.

Again SSS*, staged SSS* and alphabeta/SSS* showed well on random trees, with palphabeta and SCOUT becoming relatively effective on ordered trees.

A final test was done to compare aspiration SSS* with aspiration alpha-beta. The test was run on random trees of depth 4 and width 8, with the tree score set to 64. SSS* was given an upper bound of 65, while alphabeta was tested with the window (63,65). It was found that in almost all cases identical nodes were examined. There were some exceptions where SSS* was able to make additional cutoffs when a particular node's score was equal to 64. Since this score can be within the alpha-beta window initially, alphabeta cannot make this cutoff.

5. Conclusions

Various algorithms for searching game trees have been presented and compared by performing searches on differing sizes and types of trees. The relative strengths of the algorithms depend not only on space/time considerations but also on the ordering properties of the trees being searched. In trees with random or poor ordering, SSS*, staged SSS* and alpha-beta/SSS* appear to be distinctly superior to the alternatives, with implementation considerations (e.g. space available) helping to choose further among these. For strongly ordered trees, common of those found in practical situations, the situation is less clear. SSS* and alpha-beta/SSS* remain competitive with alpha-beta in terms of NBP, but the more efficient (and more compact) alpha-beta, palphabeta, and SCOUT would be the practical choices. Experiments with actual game-playing programs would be useful to further compare the practical implementations of these algorithms.

ACKNOWLEDGEMENT

Financial support for this study was provided through grants from the Natural Sciences and Engineering Research Council of Canada.

References

[1] Baudet, G.
On the branching factor of the alpha-beta pruning algorithm.
Artificial Intelligence 10:173-199, 1978.

[2] Baudet, G.
The design and analysis of algorithms for asynchronous multiprocessors.
PhD thesis, Carnegie-Mellon University, 1978.

[3] Campbell, M.
Algorithms for parallel search of game trees.
Master's thesis, University of Alberta, 1981. (also Tech. Rep. 81-[illegible])

[4] Fishburn, J. and Finkel, R.
Parallel alpha-beta search on Arachne.
Technical Report 394, University of Wisconsin-Madison, 1980.

[5] Fuller, S., Gaschnig, J. and Gillogly, J.
Analysis of the alpha-beta pruning algorithm.
Technical Report, Carnegie-Mellon University, 1973.

[6] Gillogly, J.
Performance analysis of the Technology chess program.
PhD thesis, Carnegie-Mellon University, 1978.

[7] Knuth, D. and Moore, R.
An analysis of alpha-beta pruning.
Artificial Intelligence 6:293-326, 1975.

[8] Marsland, T.A., Campbell, M. and Rivera, A.
Parallel search of game trees.
Technical Report 80-7, University of Alberta, 1980.

[9] Marsland, T.A. and Campbell, M.
A survey of enhancements to the alpha-beta algorithm.
In *Proceedings of the ACM National Conference*, pages 109-114. ACM, 1981.

[10] Newborn, M.
The efficiency of the alpha-beta search in trees with branch dependent terminal node scores.
Artificial Intelligence 8:137-153, 1977.

[11] Pearl, J.
Asymptotic properties of minimax trees and game-searching procedures.
Artificial Intelligence 14:113-138, 1980.

[12] Pearl, J.
The solution for the branching factor of the alpha-beta pruning algorithm.
Technical Report UCLA-ENG-CSL-8019, University of California, Los Angeles, 1980.

[13] Pearl, J.
Game-searching theory: Survey of recent results.
SIGART Newsletter 80:70-75, April, 1982.

[14] Slagle, J. and Dixon, J.
Experiments with some programs that search game trees.
JACM 2:189-207, 1969.

[15] Slate, D. and Atkin, L.
CHESS 4.5 - The Northwestern University chess program.
In Frey, P. (editor), *Chess Skill in Man and Machine*, chapter 4. Springer-Verlag, 1977.

[16] Stockman, G.
A minimax algorithm better than alpha-beta?
Artificial Intelligence 12:179-196, 1979.