A Minimax Algorithm Better than Alpha-Beta?

G. C. Stockman

L.N.K. Corporation, 4321 Hartwick Road, Suite 321, College Park, Maryland 20740, U.S.A.

Recommended by Dr. Hans Berliner

ABSTRACT

An algorithm based on state space search is introduced for computing the minimax value of game trees. The new algorithm SSS* *is shown to be more efficient than α–β in the sense that* SSS* *never evaluates a node that α–β can ignore. Moreover, for practical distributions of tip node values,* SSS* *can expect to do strictly better than α–β in terms of average number of nodes explored. In order to be more informed than α–β,* SSS* *sinks paths in parallel across the full breadth of the game tree. The penalty for maintaining these alternate search paths is a large increase in storage requirement relative to α–β. Some execution time data is given which indicates that in some cases the tradeoff of storage for execution time may be favorable to* SSS*.

1. Introduction

The α–β procedure is a well-known procedure for computing the minimax value of a game tree. It does so by sequentially transversing the positions of the game tree. By carrying along upper and lower bounds it can ignore a significant number of positions which are irrelevant to the minimax computation and thus the α–β procedure is significantly better than a pure minimax strategy which considers all positions. The number of nodes, particularly tip nodes, which a minimax procedure can ignore is an important measure of its efficiency. Knuth and Moore [3] give an excellent history and mathematical development of α–β.

The efficiency of α–β has been studied by Baudet [1], Fuller et al. [2], Knuth and Moore [3], Newborn [4], and Slagle and Dixon [6]. Various probabilistic assumptions have been used for the assignment of tip values to the game tree and both simulation and closed form results have been reported. Knuth and Moore [3] hint that α–β may be as efficient an algorithm as can be obtained for doing minimax.

While α–β may be an optimal minimax algorithm under the assumption of sequential traver sal of the nodes of the game tree, this paper shows that α–β can be beaten in efficiency if subtrees of the game tree are traversed "in parallel". Since

α–β is condemned to explore the game tree in left-to-right order it may do more work than is necessary when the optimal path of moves is toward the right of the tree. The new algorithm introduced in this paper simultaneously develops multiple paths in all regions of the tree. As a result it obtains a better global perspective on the tree and can cut off search where α–β cannot. The new algorithm performs the parallel tree traversals via state space search, and hence is called SSS*. It is proven that SSS* is a correct minimax algorithm and that SSS* never explores a node that α–β can ignore. Moreover, for practical distributions of tip value assignments SSS* will explore strictly fewer game tree nodes than α–β.

Section 2 provides the foundations necessary for description and analysis of the new algorithm SSS*, which is then defined in Section 3. The efficiency of SSS* relative to α–β is treated in Section 4 and concluding remarks are given in Section 5.

2. Foundations

In what follows, a basic knowledge of AND/OR trees and the minimax and α–β procedures will be assumed. A good intuitive introduction to this material can be found in Chapters 4 and 5 of Nilsson's text [5]. The purpose of this section is to make fundamental definitions and to show that the minimax evaluation of a game tree can be defined as the maximum value of all solution trees when the game tree is viewed as an AND/OR tree. The definitions and concepts used here are more specific than necessary in an attempt to trade off generality for clarity.

2.1. AND/OR trees and solution trees

Definition 2.1. An AND/OR tree G is a (non-empty) tree where all immediate successors of a node in the tree are of the same type, AND or OR. The type of the root node may be either AND or OR.

An AND/OR tree G is actually a meta-tree in the sense that it can be used to generate many other trees T which are regarded as potential solution trees of the AND/OR tree. The root node models some problem which is to be solved and a potential solution tree models a recursive partitioning of the root problem into equivalent subproblems. Note that any non-empty subtree of G is also an AND/OR tree.

Definition 2.2. A *solution tree* T of an AND/OR tree G is a tree with the following characteristics:

(1) The root node of the AND/OR tree G is the root node of the solution tree T.

(2) If a nonterminal node of G 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.

(3) All the terminal nodes of T represent "solved problems".

For an example, consider Figs. 1 and 2. Fig. 1 shows a binary AND/OR tree.

Nodes of type AND are represented by boxes while those of type OR are represented by circles. Assuming that all terminals of the AND/OR tree represent solved problems there are 8 different solution trees with 4 terminals each. Two of the 8 solution trees are given in Fig. 2.

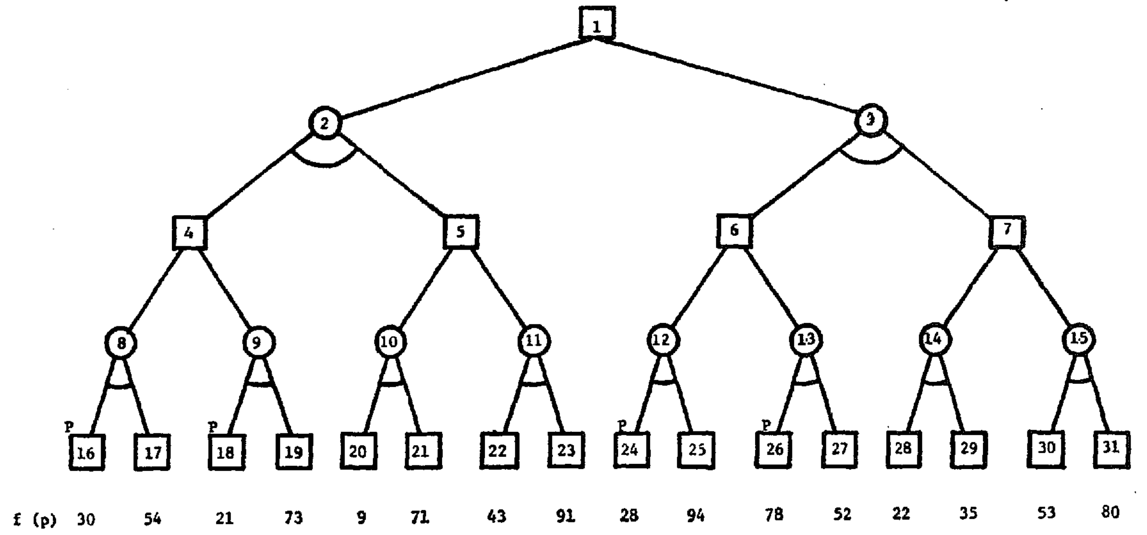

FIG. 1. An AND/OR tree with branching factor 2 and depth 4. Nodes are numbered in breadth first order with boxes denoting AND nodes and circles denoting OR nodes.

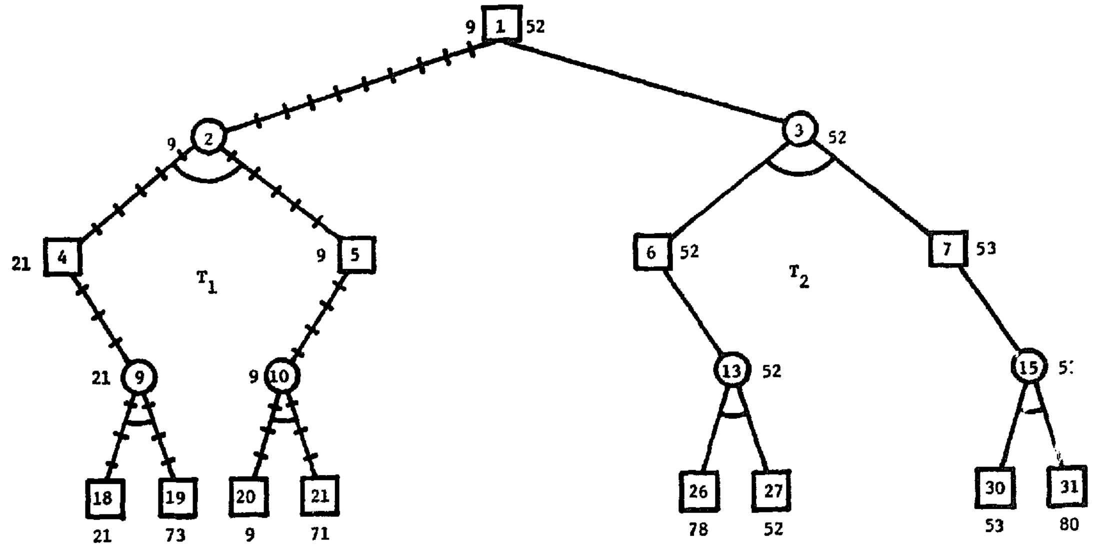

FIG. 2. Two different solution trees T_1 and T_2 of the AND/OR tree of Fig. 1. Evaluation $f_T(p)$ of the nodes p of T_1 and T_2 induced by terminal evaluations $f(p)$ are shown next to the nodes.

Since there generally will be many possible solution trees derivable from a given AND/OR tree it may be fruitful to evaluate the goodness of each solution and thus be able to rank them. We assume that an evaluation function $f(n)$ exists to assign a numerical value to terminal nodes of a solution tree. All that is needed then is a way of composing values from the terminals to assign a value to the tree. This can be done in several interesting ways. The approach taken below is oriented to the special interests of this paper.

Definition 2.3. If T is a solution tree of AND/OR tree G rooted in node p the *value of* T is denoted as $f_T(p)$ and is defined as the minimum value of all terminal nodes in T.

The best solution trees will be those whose value is maximum over the set of all possible solution trees. It will now be shown that this maximum value is the same as that computed by minimax definitions and procedures.

2.2. Minimax evaluation of game trees

Definition 2.4. Let G be an AND/OR tree with set of terminals V_T. A *minimax evaluation* on the nodes n of G is a real-valued function denoted as $g(n)$ or $g_G(n)$ and defined as follows.

(1) If n has immediate successors of type OR:

$g(n) = \max\{g(n_i)\}$ for all immediate successors n_i of n.

(2) If n has immediate successors of type AND:

$g(n) = \min\{g(n_i)\}$ for all immediate successors n_i of n.

(3) If n is terminal:

$g(n) = f(n)$ where f is a real-valued "static evaluation function" defined on V_T.

Note that g has a value for all nodes of the AND/OR tree. This has been designed for recursive definition and proof. Recall that any node of G is the root of an AND/OR subtree of G.

It is now shown that the minimax evaluation of any node of an AND/OR tree (computed as in Definition 2.4) cannot be less than the value of any solution tree rooted at that node (computed as in Definition 2.3).

Thereom 1. Let $T(p)$ be a solution tree rooted at node p of an AND/OR tree and let $g(p)$ be the minimax evaluation of p. Then

$$g(p) \geqslant f_T(p)$$

Moreover $g(p) = f_{T_0}(p)$ for some solution tree T_0.

Proof. By induction on the height of node p denoted by $H(p)$. The height of a node in a tree is defined as the maximum length of path from p to a terminal.

Suppose $H(p) = 0$. Then p is a terminal node and $p = T(p)$ represents a solved problem of value $f(p)$. By case (3) of Definition 2.4 we have $g(p) = f(p)$. But $f(p) = \min\{f(p)\}$ which gives $f_T(p)$ by Definition 2.3. Thus $g(p) = f_T(p)$ in case $H(p) = 0$.

Now it is assumed that $g(p) \geqslant f_T(p)$ for all nodes of G such that $H(p) < k$.

Let $H(p) = k > 0$. If node p has immediate successors of type OR in G then p has exactly one immediate successor p_1 in T and $H(p_1) < k$. Moreover the subtree of T rooted at p_1 is a solution tree for problem p_1: call it T_1. By the induction assumption we have

$$g(p_1) \geqslant f_{T_1}(p_1)$$

Since T and T_1 have the same terminals $f_{T_1}(p_1) = f_T(p)$ by Definition 2.3. Since p_1 is of type OR $g(p) \geqslant g(p_1)$ by case (1) of Definition 2.4. Thus $g(p) \geqslant g(p_1) \geqslant f_{T_1}(p_1)$ as desired.

If node p has n immediate AND successors $p_1, p_2, \ldots, p_n$ in G then p has n successors $p_1, p_2, \ldots, p_n$ in T. Again $H(p_i) < k$ so the induction hypothesis yields $g(p_i) \geqslant f_{T_i}(p_i)$ for all successors p_i in T.

For all successors p_i of p tree T_i has a subset of all terminals in tree T. Therefore it must follow that

$$f_{T_i}(p_i) \geqslant f_T(p) \quad \text{for all successors } p_i \text{ of } p$$

and hence $g(p_i) \geqslant f_{T_i}(p_i) \geqslant f_T(p)$ for all i.

Thus $\min_i \{g(p_i)\} \geqslant f_T(p)$. But by Definition 2.4 (2)$g(p) = \min_i\{g(p_i)\}$ and the inductive step is proven for AND successors of p. Thus the first part of the theorem obtains for all cases.

It now must be argued that $g(p)$ is actually realized as $f_{T_0}(p)$ for one of the possible solution trees T_0 of G. An optimal tree T_0 can actually be constructed as follows from G and g.

(1) Place the root node n of G in T_0.

(2) For any nonterminal node p in T_0 with OR successors in G place successor p_i in T_0 where $g(p_i)$ is not less than $g(p_j)$ for any siblings p_j.

(3) For any nonterminal node p in T_0 with AND successors in G place all immediate successors in T.

It is clear that the tree T_0 constructed above, if viewed as an AND/OR tree is such that $g_{T_0}(n) = g_G(n)$. All that need be shown is that $f_{T_0}(n) = g_{T_0}(n)$. By Definition 2.3 there is some terminal node p_0 in T_0 such that $f(p_m) \geqslant f(p_0)$ for all other terminals p_m in T_0.

Let $p_0, p_1, \ldots, p_n = n$ be the path from terminal p_0 to root node n in T_0. It is easy to show that $g_{T_0}(p_i) = f(p_0)$ for all nodes p_i on this path. Clearly this is so by Definition 2.3 and 2.4 for $p_i = p_0$. Suppose $g_{T_0}(p_k) = f(p_0)$ for some node p_k on the path. If p_k is of type AND it is clear that $g_{T_0}(p_j) \geqslant f(p_0)$ for all siblings p_j of p_k since $f(p_0)$ is the minimum of all subsets of terminals. Thus $g_{T_0}(p_{k+1}) = g_{T_0}(p_k) = f(p_0)$. If p_k is of type OR $g_{T_0}(p_{k+1}) = g_{T_0}(p_k)$ because by construction no node of AND/OR tree T_0 has more than one immediate OR successor. Since $g_{T_0}(p_k) =$

$f(p_0)$ implies that $g_{T_0}(p_{k+1}) = f(p_0)$ this equality must hold for all nodes p_i on the path. In particular $g_{T_0}(p_n) = g_{T_0}(n) = f(p_0)$. Thus it follows that $g_G(n) = f(p_0) = f_{T_0}(n)$ as desired.

It is now established that the minimal terminal value of some solution tree T of AND/OR tree G is the minimax value of G. Moreover, Theorem 1 states that the minimax value of G is the maximum value over all possible solution trees T of G. The algorithm to be developed later will essentially develop all solution trees T of G in parallel in best first order. Upper bounds established on $f_T(n)$ by examining some of the terminals of T are used to order the solution trees for development. Since the bounds on $f_T(n)$ are monotonically non-increasing as T is developed by successively adding terminals, development of that tree T with maximum upper bound is an admissible minimax strategy. This "development" of solution trees is defined in Section 3.

3. Development of the New Minimax Algorithm

Solution trees of the AND/OR tree will be developed in best-first order by state space search.

3.1. Traversal of solution trees of game trees

A tree traversal algorithm gives rules for "visiting" the nodes of a tree in a certain sequence. If each node is visited only once a linear ordering is induced on the set of nodes according to visitation sequence. Visitation will be viewed here as focusing the *state of processing* at a given node of the tree. In the following algorithm terminal nodes will each be visited once and nonterminals will be visited twice. Traversal is defined below in terms of changes of the state of processing from one node in the tree to another.

Definition 3.1. Let T be a potential solution tree of a game tree. A *state of traversal* of T is a triple

$$(n,s,\hat{h})$$

where n is a node of T; s is the status of solution of n and is either LIVE or SOLVED; and $\hat{h}$ is the merit of the state and an upper bound on $f_T(n)$ and $f_T(1)$ where 1 is the root of T.

The start state of any traversal of a game tree will be $(n = 1, s = \text{LIVE}, \hat{h} = +\infty)$ and the final state (if ever reached) will be $(n = 1, s = \text{SOLVED}, \hat{h} = f_T(1))$. $\hat{h}$ will sometimes be written as a function $\hat{h}(n)$ although this is technically a misuse of notation. $\hat{h}(n)$ is undefined for nodes n that are never visited. For nodes visited twice, once with $S = \text{LIVE}$ and once with $S = \text{SOLVED}$, $\hat{h}(n)$ will be taken to be the merit of the LIVE state (usually different from the merit of the SOLVED state).

Example. The following is a sequence of states of transversal for the solution tree on the left in Fig. 2.

(1, LIVE, $+\infty$) (2, LIVE, $+\infty$) (4, LIVE, $+\infty$) (9, LIVE, $+\infty$)
(18, LIVE, $+\infty$) (18, SOLVED, 21) (19, LIVE, 21) (19, SOLVED, 21)
(9, SOLVED, 21) (4, SOLVED, 21) (5, LIVE, 21) (10, LIVE, 21)
(20, LIVE, 21) (20, SOLVED, 9) (21, LIVE, 9) (21, SOLVED, 9)
(10, SOLVED, 9) (5, SOLVED, 9) (2, SOLVED, 9) (1, SOLVED, 9)

The value of the solution tree is 9 and takes its value from the node with minimum static evaluation $f(20) = 9$. This is not, however, the best solution tree.

3.2. Simultaneous generation and traversal using state space search

Once the st. tes of transversal of a potential solution tree of an N–K tree are defined and understood it is easy to simultaneously develop competing solution trees using state space search. Ordered state space search will be done so that the developing tree of highest merit will be the first to continue development. The basic ordered search algorithm is quite trivial—whatever problem dependent complexity exists is embedded in the next state operator. The algorithm given below is basically the A* algorithm given in Nilsson [5, Section 3.6].

SSS* Algorithm. Algorithm for state space search to find the minimax value of a game tree.

(1) Place the start state ($n = 1$, $s =$ LIVE, $\hat{h} = +\infty$) on a list called OPEN.

(2) Remove from OPEN state $p = (n, s, \hat{h})$ with largest merit $\hat{h}$. OPEN is a list kept in non-decreasing order of merit, so p will be the first in the list.

(3) If $n = 1$ and $s =$ SOLVED then p is the goal state so terminate with $\hat{h} = g(1)$ as the minimax evaluation of the game tree. Otherwise continue.

(4) Expand state p by applying state space operator Γ and queuing all output states $\Gamma(p)$ on the list OPEN in merit order. Purge redundant states from OPEN if possible. The specific actions of Γ are given in Table I

(5) go to (2)

A few notes are in order. First of all, the OPEN list will never be empty because the operator always produces a non-empty set of output Γ (p) for any non goal state p. Secondly, the algorithm is not searching for a least cost solution but a maximum solution contrary to what is usually assumed for state space search.

The operator Γ as defined in Table 1 requires the use of the functions *first*, *next*, *parent*, *ancestor*, and *type* which are easily defined once an encoding of nodes is chosen. Verify that $\Gamma(p)$ is always non-empty whenever p is not the goal state. In all cases but case 4, Γ will yield output states which may be pushed right back on the top of the OPEN list. This follows since case 4 is the only case where the value $\hat{h}$ of the output state is different from the value of the input state. Case 6 of operator Γ is the only case where there are multiple states output. This corresponds to the generation of N alternate solution trees due to immediate OR successors in the game tree. While case 6 shows that OR successors are searched for in parallel, cases 2 and 3 show how immediate AND successors are searched for sequentially. Case 6 guarantees that all potential solution trees can be developed and cases 2 and 3 produce an efficient sequential development (traversal) of individual solution trees. The careful queuing of equal merit states done in cases 4 and 6 is only necessary for comparison of SSS* and α–β which traverses trees left-to-right. The pruning operation in Case 1 should be noted. If state (m, SOLVED, $\hat{h}_1$) appears on the top of OPEN and m is an ancestor of k, then state ($k, s, \hat{h}_k$) can be purged from the interior of OPEN because it cannot possibly lead to a higher merit state (m, SOLVED, $\hat{h}_2$). Thus a best solution of node m is already at hand and other successors can be ignored.

TABLE 1. State space operations on state (n, s, $\hat{h}$) (just removed from top of OPEN list)

Case of operator Γ	Conditions satisfied by input state (n, s, $\hat{h}$)	Action of Γ in creating new output states
not applicable	s = SOLVED n = ROOT	Final state reached, exit algorithm with $g(n) = \hat{h}$.
1	s = SOLVED $n \neq$ ROOT type(n) = OR	Stack (m = parent (n), s, $\hat{h}$) on OPEN list. Then purge OPEN of all states (k, s, $\hat{h}$) where m is an ancestor of k in the game tree.
2	s = SOLVED $n \neq$ ROOT type(n) = AND next(n) $\neq$ NIL	Stack (next(n), LIVE, $\hat{h}$) on OPEN list.
3 (same as 1)	s = SOLVED $n \neq$ ROOT type(n) = AND next(n) = NIL	Stack (parent(n), s, $\hat{h}$) on OPEN list.
4	s = LIVE first(n) = NIL	Place (n, SOLVED, min $\{\hat{h}, f(n)\}$) on OPEN list (interior) behind all states of lesser or equal value.
5	s = LIVE first(n) $\neq$ NIL type(first(n)) = AND	Stack (first(n), s, $\hat{h}$) on (top of) OPEN list.

6	s = LIVE
	first(n) $\neq$ NIL
	type (first(n)) = OR
		Reset n to first (n).
		While $n \neq$ NIL
		  do
		    queue (n, s, $\hat{h}$)
		    on (top of) OPEN list;
		    reset n to next (n)
		    end

3.3. Example of a complete state space evaluation of a game tree

The complete OPEN list is given below for each loop through the algorithm for the game tree with terminal evaluations $f(p)$ given in Fig. 1. In the state encodings LIVE and SOLVED are abbreviated as L and S respectively.

Note that $\hat{h}(1) = \hat{h}(2) = \hat{h}(4) = \hat{h}(8) = \hat{h}(16) = +\infty$, $\hat{h}(27) = 78$, $\hat{h}(7) = 52$, etc.

# STATE	Case of Operator	Open List				
1	–	$(1, L, +\infty)$	#			
2	6	$(2, L, +\infty)$	$(3, L, +\infty)$	#		
3	5	$(4, L, +\infty)$	$(3, L, +\infty)$	#		
4	6	$(8, L, +\infty)$	$(9, L, +\infty)$	$(3, L, +\infty)$	#	
5	5	$(16, L, +\infty)$	$(9, L, +\infty)$	$(3, L, +\infty)$	#	
6	4	$(9, L, +\infty)$	$(3, L, +\infty)$	$(16, S, 30)$	#	
7	5	$(18, L, +\infty)$	$(3, L, +\infty)$	$(16, S, 30)$	#	
8	4	$(3, L, +\infty)$	$(16, S, 30)$	$(18, S, 21)$	#	
9	5	$(6, L, +\infty)$	(,,)	(,,)	#	
10	6	$(12, L, +\infty)$	$(13, L, +\infty)$	$(16, S, 30)$	$(18, S, 21)$	#
11	5	$(24, L, +\infty)$	$(13, L, +\infty)$	(,,)	(,,)	#
12	4	$(13, L, +\infty)$	$(16, S, 30)$	$(24, S, 28)$	$(18, S, 21)$	#
13	5	$(26, L, +\infty)$	(,,)	(,,)	(,,)	#
14	4	$(26, S, 78)$	(,,)	(,,)	(,,)	#
15	2	$(27, L, 78)$	(,,)	(,,)	(,,)	#
16	4	$(27, S, 52)$	(,,)	(,,)	(,,)	#
17	3	$(13, S, 52)$	(,,)	(,,)	(,,)	#
18	1	$(6, S, 52)$	$(16, S, 30)$	$(18, S, 21)$	#	
19	2	$(7, L, 52)$	$(16, S, 30)$	$(18, S, 21)$	#	
20	6	$(14, L, 52)$	$(15, L, 52)$	$(16, S, 30)$	$(18, S, 21)$	#
21	5	$(28, L, 52)$	(,,)	(,,)	(,,)	#
22	4	$(15, L, 52)$	$(16, S, 30)$	$(28, S, 22)$	$(18, S, 21)$	#
23	5	$(30, L, 52)$	(,,)	(,,)	(,,)	#
24	4	$(30, S, 52)$	(,,)	(,,)	(,,)	#
25	2	$(31, L, 52)$	(,,)	(,,)	(,,)	#
26	4	$(31, S, 52)$	(,,)	(,,)	(,,)	#
27	3	$(15, S, 52)$	(,,)	(,,)	(,,)	#
28	1	$(7, S, 52)$	$(16, S, 30)$	$(18, S, 21)$	#	
29	3	$(3, S, 52)$	$(16, S, 30)$	$(18, S, 21)$	#	
30	1	$(1, S, 52)$	#			

3.4. Correctness of SSS*

Now that the new algorithm for minimax computation has been defined and explained it must be verified as a correct minimax procedure. The proof of the following theorem achieves this.

Theorem 2. (Correctness). *Ordered search algorithm SSS* with next state operator* Γ *defined in Table* 1 *computes the minimax value of the root node of any game tree.*

Proof. It is clear that Γ can potentially generate all solution trees T of the game tree. From Definition 2.3 it is seen that the value $\hat{h}$ of any state of traversal $(n, s, \hat{h})$ of tree T is an upper bound on $f_T(1)$. By theorem 1 there exists some tree T_0 such that $f_{T_0}(1) = g(1)$, the minimax evaluation of the game tree rooted at node 1. The algorithm is admissible because in no case can it terminate with an inferior tree T_1 being fully developed because there must be some state of traversal $(n, s, \hat{h})$ of an optimal tree T_0 on the OPEN list with $\hat{h} \geqslant f_{T_0}(1) \geqslant f_{T_1}(1)$. (This is the same as Nilsson's proof of admissibility for $A*$ with $\hat{h}$ being an over-estimating heuristic.) Of course the algorithm will always terminate because there is always a large number of solution trees and it has just been shown that at termination the maximum solution tree evaluation is achieved.

4. The Efficiency of SSS* relative to α–β

In this section it is shown that the SSS* algorithm dominates the α–β algorithm in the sense that whenever SSS* must explore a node then so must α–β. It is also shown for practical distributions of tip values that SSS* is strictly superior to α–β in terms of average number of tip nodes examined. Prior to proving these results interesting examples are presented which motivated the search for those results. In addition, simulations performed before deriving the theorems are briefly discussed.

4.1. Interesting examples comparing α–β and SSS*

Very early in the study of SSS* example game trees were produced in which SSS* explored fewer nodes than α–β. An example particularly unfavorable to α–β is given in Fig. 1 and 3: SSS* ignores 11 nodes while α–β ignores only 3. α–β does not do well because the best path is toward the right of the tree.

Table 2 lists all possible (ranked) distinct tip assignments to a 2-level binary game tree and the number of tips that α–β and SSS* would explore. The average number of tips explored is 11/3 for α–β and 10/3 for SSS*. More important is the fact that SSS* never explores a node that α–β ignores.

4.2. Simulation of α–β versus SSS*

Examples such as those given above were at first studied with respect to an older and weaker version of SSS* (SSS). Trees were found on which α–β explored fewer nodes than the older SSS Algorithm. Simulation was done in order to test the hypothesis that on the average α–β explored more terminal nodes. Not only did the

TABLE 2. *SSS** domination of α–β in a 2-level binary game tree

Case	Tip assignments	Number of tips α–β evaluates	Number of tips *SSS** evaluates
1	1 2 3 4	4	3
2	1 2 4 3	4	3
3	1 3 2 4	4	3
4	1 3 4 2	4	3
5	1 4 2 3	4	3
6	1 4 3 2	4	3
7	2 1 3 4	4	3
8	2 1 4 3	4	3
9	2 3 1 4	3	3
10	2 3 4 1	4	4
11	2 4 1 3	3	3
12	2 4 3 1	4	4
13	3 1 2 4	4	4
14	3 1 4 2	4	4
15	3 2 1 4	3	3
16	3 2 4 1	4	4
17	3 4 1 2	3	3
18	3 4 2 1	3	3
19	4 1 2 3	4	4
20	4 1 3 2	4	4
21	4 2 1 3	3	3
22	4 2 3 1	4	4
23	4 3 1 2	3	3
24	4 3 2 1	3	3
Average		11/3	10/3

simulation work support that hypothesis but it also led to the development of SSS* which can be proven to dominate α–β. The final simulation results appear in Table 3. 1000 identical sets of distinct tip assignments were presented to α–β and SSS* and the behavior of the algorithms was recorded. Not only was there a smaller mean number of tips evaluated for SSS* but there was also a smaller standard deviation. The simulation program was then altered to check if α–β did better on any single set of tip assignments. 100 sets of tip assignments were checked for four different game trees as indicated in Table 3. None of the cases checked showed SSS* exploring a node that α–β could ignore.

4.3. Domination of α–β by SSS*

It will now be shown that SSS* is more efficient than α–β in terms of the average number of nodes explored. In order to do the comparison a mathematical characterization of α–β is required. The characterization of α–β used here is due to Baudet [1] and represents the maturation of a long path [6, 2–4] of research.

TABLE 3. Comparative performance of α–β and *SSS** on 1000 sets of distinct tip values

				Published Comparisons		α–β Performance		SSS* Performance		
Branching factor N	Number of AND node levels past the root K	Number of Tips in solution tree	Number of Solution trees	α–β Exact value from [3] or [1]	α–β Lower bound from [2][a]	Average number of tips evaluated and standard deviation	CPU seconds used	Average number of tips evaluated	CPU seconds used	Maximum number of states on OPEN
2	1	2	2	3.67[3]	3.41	3.65 $\sigma = 0.5$	1.76	3.33 $\sigma = 0.5$	8.69	2
3	1	3	3	7.44[3]	6.42	7.41 $\sigma = 1.3$	2.95	6.26 $\sigma = 1.1$	10.0	3
5	1	5	5	17.67[1]	13.7	17.6 $\sigma = 3.5$	6.80	13.7 $\sigma = 2.7$	15.7	5
10	1	10	10		37.4	56.1 $\sigma = 11.4$	27.1	41.3 $\sigma = 9.1$	41.0	10
20	1	20	20	177 [3]	102	174 $\sigma = 34.2$	119	127 $\sigma = 27.7$	147	20
2	2	4	8	12.04[1]		12.1 $\sigma = 2.2$	5.67	10.1 $\sigma = 1.9$	17.2	4
& 3	2	9	81	45.20[1]		45.0 $\sigma = 9.6$	22.1	34.2 $\sigma = 7.6$	45.6	9
& 5	2	25	5^6	227.35[1]	188	227 $\sigma = 46.6$	180	161 $\sigma = 36$	273	25
& 2	3	8	128	37.16[1]		37.3 $\sigma = 7.7$	22.1	28.7 $\sigma = 6.2$	51.0	8
& 3	3	27	3^{13}	248.56[1]	265	248 $\sigma = 51.5$	235	177 $\sigma = 40.6$	400	27
2	4	16	2^{15}		135	110 $\sigma = 23.1$	78.5	80.5 $\sigma = 17.7$	182	16

[a] lower bound for α–β without deep cutoffs
& SSS* dominated α–β for first 100 tip assignments

4.3.1 *Definition of a game tree*

The following definition of a game tree is used throughout this section. 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. While these restrictions are unnecessary for SSS* they are conventional assumptions for minimax and α–β.

4.3.2. *Encoding the nodes of the game tree*

Dewey decimal notation can be used to represent game tree nodes. Let the root be encoded as the sequence 1. Then if sequence J represents any nonterminal node of the tree, $J \cdot j; j = 1, 2, \ldots, n$ represent the n immediate successors (sons) of J. Fig. 3 gives an example of Dewey decimal encoding of nodes.

4.3.3. *Negamax evaluation of nodes of the game tree*

Negamax evaluation of nodes is made with respect to value to the player making the move rather than with respect to the player moving from the root position as with pure minimax. If node J has n sons then

$$v(J) \equiv \max\{-v(J \cdot i) \mid 1 \leqslant i \leqslant n\}.$$

If we assume that the root is an AND node then $v(J) = g(J)$ for AND nodes (player at J is player at root) and $v(J) = -g(J)$ for OR nodes (player at J is opponent of player at root) where g is the minimax value defined in Section 2.

4.3.4. *Legacy of elder brothers of a node*

For any node $J \cdot j$ of the game tree

$$c(J \cdot j) \equiv \max\{-v(J \cdot i) \mid 1 \leqslant i < j\}.$$

Since $J \cdot i$, $1 \leqslant i < j$ are the older brothers of $J \cdot j$, $c(J \cdot j)$ accounts for information provided to a node from the older brothers. If $j = 1$ the set of elder brothers is empty and by convention $c(J \cdot 1) = -\infty$. See Fig. 3 for examples.

4.3.5. *Static functions α and β*

Finally we define two values associated with a node J by the α–β procedure.

$$\alpha(J) \equiv \max\{c(a) \mid a = J \text{ or } a \text{ is an ancestor of } J \text{ of the same type as } J\}$$

$$\beta(J) \equiv -\max\{c(a) \mid a \text{ is an ancestor of } J \text{ of the opposite type from } J\}$$

$\alpha(1) = -\infty$ by definition, and $\beta(1) \equiv +\infty$ by convention. Examples are given in Fig. 3.

Baudet [1] has shown that the negamax α–β algorithm of Knuth and Moore [3] will explore node J of a game tree if and only if $\alpha(J) < \beta(J)$. Similar results were also obtained by Fuller et al. [2]. It is important to note that $\alpha(J)$ and $\beta(J)$ are *defined* for all nodes of the tree but will not in fact be computed by α–β for all nodes of the tree.

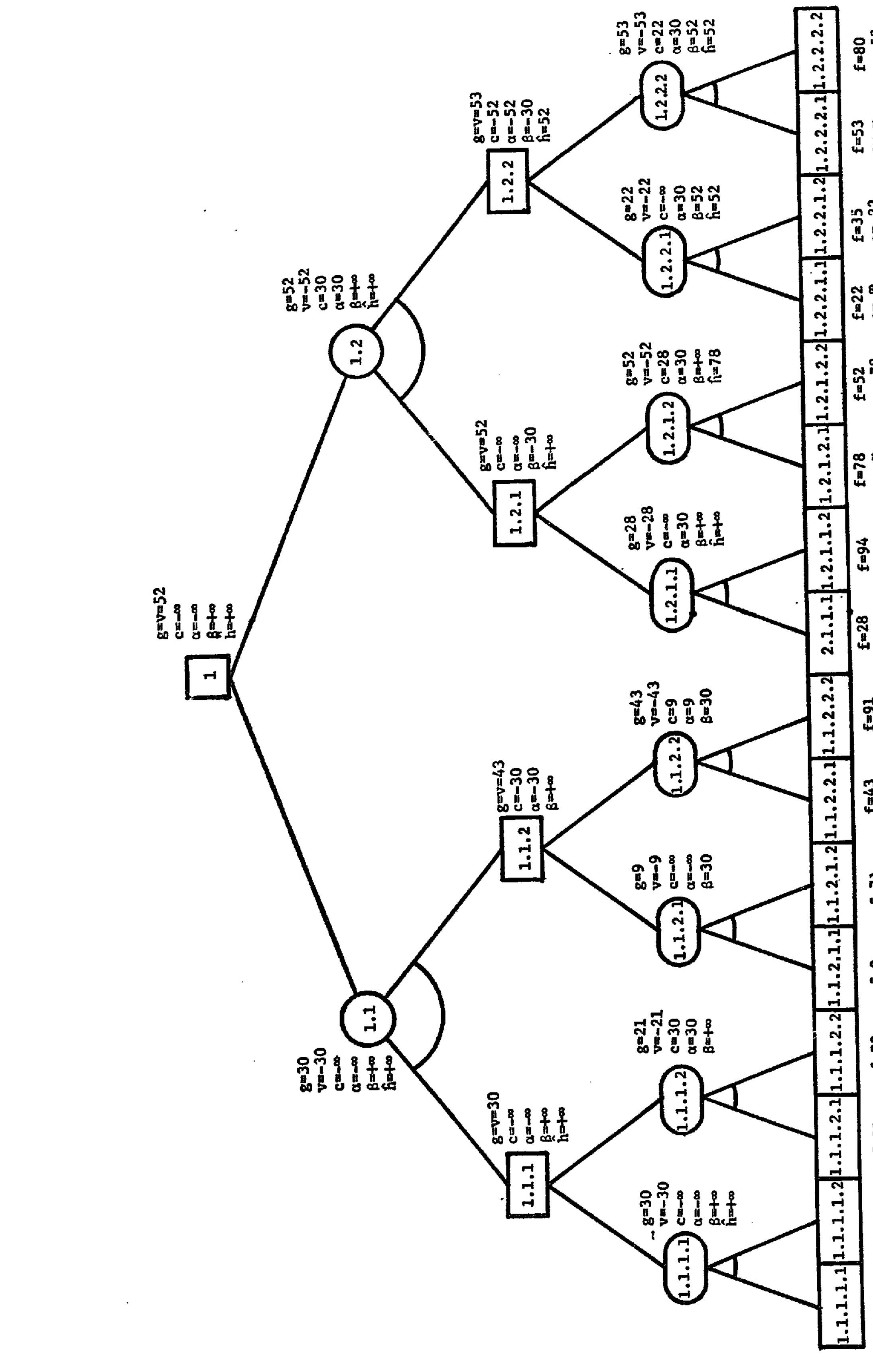

FIG. 3. Game tree on which SSS* explores fewer nodes than α–β. (Nodes not explored by α–β are 1.1.1.2.2, 1.2.1.1.2, 1.2.2.1.2.; Nodes not explored by SSS* are 1.1.1.1.2, 1.1.1.2.2, 1.1.2, 1.1.2.1, 1.1.2.2, 1.1.2.1.1, 1.1.2.1.2, 1.1.2.2.1, 1.1.2.2.2, 1.2.1.1.2, 1.2.2.1.2.)

4.3.6. *Relationship of $\hat{h}$ of* SSS* *to $\alpha(J)$ and $\beta(J)$*

Two lemmas are now established which characterize the relationship between the merit $\hat{h}$ of state $(J, L, \hat{h})$ when SSS* visits AND node J and the static values $\alpha(J)$ and $\beta(J)$.

Lemma 1. *For all* AND *nodes J explored by* SSS*: $\alpha(J) = -\hat{h}(J)$.

Proof. $\hat{h}(J)$ is defined since SSS* visits node J. Recall that $\hat{h}(J)$ is the merit assigned when J is first visited by SSS* with state $(J, \text{LIVE}, \hat{h})$. By definition

$\alpha(J) \equiv \max\{c(a) \mid a = J \text{ or } a \text{ is an AND ancestor of } J\}$.

$= \max\{\max\{-v(e) \mid e \text{ is an elder brother of } a\} \mid a = J \text{ or } a \text{ is an AND ancestor of } J\}$.

But for AND nodes the negamax value $v(e)$ is the same as the minimax value $g(e)$.

$\alpha(J) = \max\{-\min\{g(e) \mid e \text{ is an elder brother of } a\} \mid a = J \text{ or } a \text{ is an AND ancestor of } J\}$

$= -\min\{\min\{g(e) \mid e \text{ is an elder brother of } a\} \mid a = J \text{ or } a \text{ is an AND ancestor of } J\}$

$= -\hat{h}(n)$

because all the nodes e are part of the solution tree which SSS* has traversed up to state $(J, \text{LIVE}, \hat{h})$. For the notation to be meaningful in case there are no elder brothers of node a we define $\min\{\quad\} = +\infty$ just as we have defined $\max\{\quad\} = -\infty$.

Lemma 2. *If* SSS* *explores* AND *node n then $\hat{h}(n) > -\beta(n)$.*

Proof. Suppose that $J{\cdot}j$ is an arbitrary OR node ancestor of node n. It must be that $g(J{\cdot}i) \leqslant \hat{h}(n)$ for $i \neq j$. Suppose $g(J{\cdot}i) > \hat{h}(n)$ for some i. Then state $(J{\cdot}i, \text{SOLVED}, \hat{h}_0)$ would have appeared at the top of OPEN with $\hat{h}_0 \geqslant \hat{h}(n)$. Thus $(J, \text{SOLVED}, \hat{h}_0)$ would have been the next state placed on top of OPEN and all states $(J{\cdot}x, s, \hat{h})$ would have been purged from OPEN. Hence node n would not be explored by SSS*. Thus $g(J{\cdot}i) \leqslant \hat{h}(n)$ for $i \neq j$ and clearly $g(J{\cdot}i) < \hat{h}(n)$ for $i < j$, i.e. $J{\cdot}i$ is an elder brother of $J{\cdot}j$. Since $g(J{\cdot}i) \equiv -v(J{\cdot}i)$ we have $\hat{h}(n) > -v(J{\cdot}i)$ for all $i < j$ and for all OR ancestors $J{\cdot}j$ of node n. Thus $\hat{h}(n) > \max\{-v(J{\cdot}i) \mid i < j\} \equiv c(J{\cdot}j)$ for all OR ancestors $J{\cdot}j$ of node n. Hence $\hat{h}(n) > \max\{c(J{\cdot}j) \mid J{\cdot}j \text{ an OR ancestor of } n\} \equiv -\beta(n)$ as claimed. To touch up the proof in case $j = 1$ and there are no elder brothers of $J{\cdot}j$ note that $c(J{\cdot}j) \equiv -\infty$ and that $\hat{h}(n)$ being a minimum of $+\infty$ and finite tip values, is never $-\infty$.

4.3.7. *Proof of* SSS* *dominance*

It is now easy to conclude that SSS* will explore fewer nodes than α–β.

Theorem 3 (Domination). *If* SSS* *explores arbitrary node n of a game tree then so must α–β.*

Proof. Lemmas 1 and 2 yield a trivial proof in case n is an AND node. $\alpha(n) = -\hat{h}(n)$ by Lemma 1 and $-\hat{h}(n) < \beta(n)$ by Lemma 2. Thus $\alpha(n) < \beta(n)$ which from Baudets' theorem implies that α–β must explore node n.

Now assume that node $n = J{\cdot}j$ is of type OR and explored by SSS*. SSS* must have explored the AND node parent J and $\hat{h}(J{\cdot}j) = \hat{h}(J)$ from Table 1. In the proof of Lemma 2 it was shown that $\hat{h}(J) > c(J{\cdot}j)$. By the definition of α and β we have

$$\alpha(J{\cdot}j) = \max\{c(J{\cdot}j), -\beta(J)\} \qquad \beta(J{\cdot}j) = -\alpha(J).$$

Now since J is type AND and explored by SSS* it must be explored by α–β as well. Thus $\alpha(J) < \beta(J)$ or $-\beta(J) < -\alpha(J)$. Since $c(J{\cdot}j) < \hat{h}(J) = -\alpha(J)$ it follows that $\max\{c(J{\cdot}j), -\beta(J)\} < -\alpha(J)$ and so $\alpha(J{\cdot}j) < \beta(J{\cdot}j)$. Again from Baudets theorem it follows that α–β must explore node $J{\cdot}j = n$ and the proof is established.

It is now clear that SSS* is strictly superior to α–β in terms of nodes explored when reasonable probabilistic assumptions are made about tip value assignments. The following corollary gives an example.

Corollary. *Let f be an evaluation function assigning the tips of a game tree values from any nonsingleton set R of real numbers. If each of the possible realizations of f has non-zero probability, then* SSS* *can expect to explore fewer nodes than α–β on the average.*

Proof. Let v_1 and v_2 be two values from R such that $v_1 < v_2$. Consider the leftmost set of terminal AND nodes $\{J \mid J = 1 \cdot 1 \cdot 1 \cdot \ldots \cdot 1 \cdot i\}$. α–β must explore this entire set of nodes regardless of their value. However, if the value of the leftmost node is v_1 and the value of the game tree is v_2 or greater SSS* will explore only the leftmost node of the set. Thus SSS* would explore strictly fewer nodes than α–β in this case and in no case explore more nodes by Theorem 3. By assumption, this case has non-zero probability and therefore the expected number of nodes evaluated by SSS* would be less than for α–β.

Just how much better SSS* can do depends on the distribution of possible tip assignments and is left for future study. The simulation results give some indication for random distinct tip assignments.

4.3.8. *The storage efficiency of* SSS*

Let the game tree consist of K levels of AND nodes and K levels of OR nodes beyond the root and let the branching factor be N. Also assume that there is not a guaranteed win for the root player; i.e. $g(1) < +\infty$. Because SSS* starts with state $(1, \text{LIVE}, +\infty)$ and propagates N-fold with case 6 of Γ (Table 1), OPEN will receive N^K states of merit $\hat{h} = +\infty$ before SSS* can possibly terminate with $g(1) < +\infty$. For instance, by state 13 of Example 3.3 the fourth state of merit $+\infty$ appeared at the top of OPEN. There can never be more than N^K states on OPEN because AND nodes will be "solved" at the frontier without increasing the size of

OPEN, and sooner or later OR nodes above in the tree will become solved and OPEN will actually decrease in size. Requiring N^K OPEN entries, SSS* is very inefficient in storage in comparison to α–β which requires only $2K$ stack entries for the same game tree.

5. Discussion and Conclusions

This research originated not in game playing but in syntactic pattern recognition. The SSS procedure was constructed to produce ambiguous parses of waveforms modeled by a context-free grammar. Evaluation of terminals corresponded to curve fits of a mathematical model to a segment of waveform data. Terminal values were χ^2 values of the confidence in the fit. The states of SSS were actually entire partial parse trees and expansion of the states successively enforced the syntactic constraints of the grammar. The best fitting parses were developed first by the best-first order of SSS. Since we were interested in multiple interpretations no states were removed from OPEN unless their value fell below a threshold. In addition SSS was often operated in a bottom-up mode by initializing OPEN with states (J, SOLVED, $\hat{h}$) where J was a tip node and $\hat{h} = f(J)$. Despite quite large encodings of states the procedure was able to develop parses of significant pulse wave data in a few seconds with only a few thousand words of dynamic storage. Details are provided in [6] for the interested reader.

Knuth and Moore halted their search for a better algorithm [2, page 298] and devised an optimality theorem [2, page 307] for α–β. The definition of optimality in that theorem seems rather sterile in light of the results reported here. Once again a classical tradeoff between storage space and execution time has been achieved. α–β is bound to sequential left-to-right development of the game tree while SSS* sinks multiple paths to the leaves and then competitively develops alternative traversals. α–β loses efficiency when the best solution is toward the right of the tree. If there exists accurate ordering information for successors this loss will not occur because the best solution after reordering will be toward the left of the game tree. It may turn out to be more practical to use SSS*, however, than to order successors. SSS* may also be used to advantage on a system with true parallel processing.

For a game tree of depth $2K$ and branching factor N SSS* requires N^K cells of storage for the OPEN list. This presents no practical problem only for small N or K. The stack size of α–β varies only with depth and is independent of N so a meager $2K$ cells of storage are required. The steps of SSS* are slightly more complicated than those of α–β and thus SSS* will be slower than α–β when identical node sets are explored. The complication arises in maintenance of the ordered OPEN list. Some insight can be gained by reexamining Table 3 Columns 8 and 10. SSS* was implemented as quickly as possible as was not designed to be efficient according to any criteria other than number of tip nodes explored. SSS* consistently ran slower than α–β in the simulations. However, tip evaluation time was *not* considered. For instance, on the game trees with $N = 2$ and $K = 2$ SSS* took 12 seconds longer but evaluated 2000 fewer tip nodes. If the terminal evaluation function requires greater than 0.006 seconds, SSS* would in practice execute faster. The break even point is less than 0.002 seconds for $N = 5$ and $K = 2$. Redesign of the OPEN list structure in the implementation of SSS* would further enhance the practical comparison to α–β.

SSS* has been shown to be superior to α–β according to the criteria of number of nodes evaluated. Simulation results indicate that SSS* can in fact execute faster than α–β in certain practical cases. It seems likely that a hybrid SSS*–α–β procedure would be worthwhile. α–β could be used to a certain depth to avoid SSS* storage problems and then SSS* could be used effectively to save on tip evaluations. Alternatively SSS* could be used at the shallow level and α–β at deeper levels of the game tree. More work needs to be done in this regard. Hopefully other AI workers will further examine the tradeoffs involved and evaluate the potential of SSS* for their application. There are also theoretical questions left open. It is easy to see that SSS* has the same best case as α–β and thus is still an expensive search procedure even in the best situations. From the results of this paper any upper bound on α–β efficiency also applies to SSS*. Deep cutoffs are now known to have only a secondary effect on the character of α–β [1, 3]. What is the effective branching factor of SSS* and does it have a character significantly different from α–β?

ACKNOWLEDGEMENTS

This research was supported in part by the National Science Foundation under Grant NSF-Eng. 73-04099-A02 to the Lab. for Pattern Analysis, Department of Computer Science of the University of Maryland and in part by the 6570th Aerospace Medical Research Laboratory, Aerospace Medical Division, Air Force Systems Command, USAF, Wright-Patterson AFB, Ohio, under Contract F 33615-76-C-0521 to L.N.K. Corporation, Silver Spring, Maryland.

The author would like to thank Laveen Kanal and others of the Laboratory for Pattern Analysis for stimulating discussions on this topic and the reviewers for much constructive criticism.

REFERENCES

1. Baudet, G. M., On the Branching Factor of the Alpha–Beta Pruning Algorithm, *Artificial Intelligence* 10 (1978) 173–199.
2. Fuller, S. H., Gaschnig, J. G., and Gillogly, J. J., An analysis of the alpha–beta pruning algorithm, Dept. of Computer Science Report, Carnegie-Mellon University (July 1973).
3. Knuth, D. E., and Moore, R. N., An analysis of alpha–beta pruning, *Artificial Intelligence* 6 (1975) 293–326.
4. Newborn, M. M., The efficiency of the alpha–beta search on trees with branch-dependent terminal scores, *Artificial Intelligence* 8 (1977) 137–153.
5. Nilsson, N. J., *Problem-solving Methods in Artificial Intelligence*, (McGraw-Hill, New York, 1971).
6. Slagle, J. R., and Dixon, J. K., Experiments with some programs that search game trees, *Journal of the ACM* 2 (1969) 189–207.
7. Stockman, G., A problem-reduction approach to the linguistic analysis of waveforms, Ph.D. dissertation, University of Maryland Computer Science Technical Report TR-538, (May 1977).

Received 21 February 1798; revised version received 27 November 1978.