A Minimax Algorithm Better than Alpha-Beta?

resubmitted to

The Artificial Intelligence Journal

Oct 1978

by

G. C. Stockman

L.N.K. CORPORATION
4321 Hartwick Road, Suite 321
College Park, Maryland 20740
(phone 301-927-3223)
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*.
List of Figures

Figure 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.

Figure 2. Two different solution trees $T_1$ and $T_2$ of the AND/OR tree of Figure 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.

Figure 3. Game tree on which SSS* explores fewer nodes than $\alpha-\beta$.

(Nodes not explored by $\alpha-\beta$ 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.)
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.
1. Introduction

The \( \alpha-\beta \) procedure is a well-known procedure for computing the minimax value of a game tree. It does so by sequentially traversing 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 \( \alpha-\beta \) 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 \( \alpha-\beta \).

The efficiency of \( \alpha-\beta \) 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 \( \alpha-\beta \) may be as efficient an algorithm as can be obtained for doing minimax.

While \( \alpha-\beta \) may be an optimal minimax algorithm under the assumption of sequential traversal of the nodes of the game tree, this paper shows that \( \alpha-\beta \) can be beaten in efficiency if subtrees of the game tree are traversed "in parallel". Since \( \alpha-\beta \) 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 $\alpha-\beta$ 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 non-terminal 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 Figures 1 and 2. Figure 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 Figure 2.

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.
Figure 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.
Figure 2. Two different solution trees $T_1$ and $T_2$ of the AND/OR tree of Figure 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.
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
   \[ g(n) = \max \{ g(n_i) \} \text{ for all immediate successors } n_i \text{ of } n. \]

2. if n has immediate successors of type AND
   \[ g(n) = \min \{ g(n_i) \} \text{ for all immediate successors } n_i \text{ of } n. \]

3. if n is terminal
   \[ g(n) = f(n) \text{ where } f \text{ is a real-valued "static evaluation function" defined on } V_T. \]
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 Def. 2.4) cannot be less than the value of any solution tree rooted at that node (computed as in Def. 2.3).

**Theorem 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

$$g(p) \geq f_T(p)$$

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 $g(p) > f_T(p)$ 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

$$g(p_1) > f_{T_1}(p_1)$$

Since $T$ and $T_1$ have the same terminals $f_{T_1}(p_1) = f_T(p)$ by Def. 2.3. Since $p_1$ is of type OR $g(p) > g(p_1)$ by case (1) of Def. 2.4. Thus $g(p) > g(p_1) > f_{T_1}(p_1)$ 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 $H(p_1) < k$ so the induction hypothesis yields $g(p_1) > f_{T_1}(p_1)$ 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

$$f_{T_i}(p_i) > f_T(p)$$

for all successors $p_i$ of $p$

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

Thus \( \min \{ g(p_i) \} \geq f_T(p) \).

But by Def. 2.4 case (2) \( g(p) = \min \{ g(p_i) \} \) 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 non terminal 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 non terminal 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 \( f(p_m) \geq f(p_0) \) for all other terminals \( p_m \) in \( T_0 \).
Let \( p_o, p_1, \ldots, p_n = n \) be the path from terminal \( p_o \) to root node \( n \) in \( T_o \). It is easy to show that \( g_{T_o}(p_1) = f(p_o) \) for all nodes \( p_i \) on this path. Clearly this is so by Defs. 2.3 and 2.4 for \( p_i = p_o \). Suppose \( g_{T_o}(p_k) = f(p_o) \) for some node \( p_k \) on the path. If \( p_k \) is of type AND it is clear that \( g_{T_o}(p_j) \geq f(p_o) \) for all siblings \( p_j \) of \( p_k \) since \( f(p_o) \) is the minimum of all subsets of terminals. Thus \( g_{T_o}(p_{k+1}) = g_{T_o}(p_k) = f(p_o) \). If \( p_k \) is of type OR \( g_{T_o}(p_{k+1}) = g_{T_o}(p_k) \) because by construction no node of AND/OR tree \( T_o \) has more than one immediate OR successor. Since \( g_{T_o}(p_k) = f(p_o) \) implies that \( g_{T_o}(p_{k+1}) = f(p_o) \) this equality must hold for all nodes \( p_i \) on the path. In particular \( g_{T_o}(p_n) = g_{T_o}(n) = f(p_o) \). Thus it follows that \( g_{T_o}(n) = f(p_o) = f_T(n) \) as desired.

Q.E.D.

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.

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

Example

The following is a sequence of states of traversal for the solution tree on the left in Figure 2.

\[
(1, \text{LIVE}, + \infty) \ (2, \text{LIVE}, + \infty) \ (4, \text{LIVE}, + \infty) \ (9, \text{LIVE}, + \infty) \\
(18, \text{LIVE}, + \infty) \ (18, \text{SOLVED}, 21) \ (19, \text{LIVE}, 21) \ (19, \text{SOLVED}, 21) \\
(9, \text{SOLVED, 21}) \ (4, \text{SOLVED, 21}) \ (5, \text{LIVE, 21}) \ (10, \text{LIVE, 21}) \\
(20, \text{LIVE, 21}) \ (20, \text{SOLVED, 9}) \ (21, \text{LIVE, 9}) \ (21, \text{SOLVED, 9}) \\
(10, \text{SOLVED, 9}) \ (5, \text{SOLVED, 9}) \ (2, \text{SOLVED, 9}) \ (1, \text{SOLVED, 9})
\]

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 states of traversal 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 \( n = 1, s = \text{LIVE}, \hat{h} = +\infty \) 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 \( s = \text{SOLVED} \) 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 $\Gamma$ 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 $\Gamma$ are given in Table I.

(5) goto (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 $\Gamma(p)$ for any non-goal state $p$. Secondly, the algorithm is not searching for a least cost solution but a maximum value solution contrary to what is usually assumed for state space search.

The operator $\Gamma$ as defined in Table I requires the use of the functions $\text{first}$, $\text{next}$, $\text{parent}$, $\text{ancestor}$, and $\text{type}$ 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, $\Gamma$ 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 $h$ of the output state is different from the value of the input state. Case 6 of operator $\Gamma$ 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
Table I State space operations on state \((n,s,h)\)
(just removed from top of OPEN list)

<table>
<thead>
<tr>
<th>Case of operator (\Gamma)</th>
<th>Conditions satisfied by input state ((n,s,h))</th>
<th>Action of (\Gamma) in creating new output states</th>
</tr>
</thead>
<tbody>
<tr>
<td>not applicable</td>
<td>(s = \text{SOLVED}) (n = \text{ROOT})</td>
<td>Final state reached, exit algorithm with (g(n)=h).</td>
</tr>
<tr>
<td></td>
<td>(n \neq \text{ROOT})</td>
<td>(\text{SOLVED})</td>
</tr>
<tr>
<td></td>
<td>(\text{type}(n) = \text{OR})</td>
<td>Stack ((m=\text{parent}(n),a,h)) on OPEN list. Then purge OPEN of all states ((k,s,h)) where (m) is an ancestor of (k) in the game tree.</td>
</tr>
<tr>
<td>1</td>
<td>(s = \text{SOLVED}) (n \neq \text{ROOT})</td>
<td>(\text{SOLVED})</td>
</tr>
<tr>
<td></td>
<td>(\text{type}(n) = \text{AND})</td>
<td>Stack ((\text{next}(n),\text{LIVE},h)) on OPEN list.</td>
</tr>
<tr>
<td></td>
<td>(\text{next}(n) \neq \text{NIL})</td>
<td></td>
</tr>
<tr>
<td>2 (same as 1)</td>
<td>(s = \text{SOLVED}) (n \neq \text{ROOT})</td>
<td>(\text{SOLVED})</td>
</tr>
<tr>
<td></td>
<td>(\text{type}(n) = \text{AND})</td>
<td>Stack ((\text{parent}(n),a,h)) on OPEN list.</td>
</tr>
<tr>
<td></td>
<td>(\text{next}(n) = \text{NIL})</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(s = \text{LIVE}) (\text{first}(n) = \text{NIL})</td>
<td>Place ((n,\text{SOLVED},\text{min}(h,f(n)))) on OPEN list (interior) behind all states of lesser or equal value.</td>
</tr>
<tr>
<td>4</td>
<td>(s = \text{LIVE}) (\text{first}(n) \neq \text{NIL})</td>
<td>(\text{LIVE})</td>
</tr>
<tr>
<td></td>
<td>(\text{type}(\text{first}(n)) = \text{AND})</td>
<td>Stack ((\text{first}(n),a,h)) on (top of) OPEN list.</td>
</tr>
<tr>
<td>5</td>
<td>(s = \text{LIVE}) (\text{first}(n) \neq \text{NIL})</td>
<td>Reset (n) to (\text{first}(n))</td>
</tr>
</tbody>
</table>
|                           | \(\text{type}(\text{first}(n)) = \text{OR}\) | While \(n \neq \text{NIL}\) do
|                           |                                                   | queue \((n,s,h)\) on (top of) OPEN list; reset \(n\) to \(\text{next}(n)\) end |
comparison of SSS and α-β which traverses trees left-to-right. The pruning operation in case 1 should be noted. If state \((m, \text{SOLVED}, \hat{h}_1)\) 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 \((m, \text{SOLVED}, \hat{h}_2)\). Thus a best solution of node \(m\) is already at hand and other successors can be ignored.

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 Figure 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.
<table>
<thead>
<tr>
<th>State</th>
<th>Case of Operator</th>
<th>Open List</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>$(1,L,+\infty)$ #</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
<td>$(2,L,+\infty)$ $(3,L,+\infty)$ #</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>$(4,L,+\infty)$ $(3,L,+\infty)$ #</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>$(8,L,+\infty)$ $(9,L,+\infty)$ $(3,L,+\infty)$ #</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>$(16,L,+\infty)$ $(9,L,+\infty)$ $(3,L,+\infty)$ #</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>$(9,L,+\infty)$ $(3,L,+\infty)$ $(16,S,30)$ #</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>$(18,L,+\infty)$ $(3,L,+\infty)$ $(16,S,30)$ #</td>
</tr>
<tr>
<td>8</td>
<td>4</td>
<td>$(3,L,+\infty)$ $(16,S,30)$ $(18,S,21)$ #</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>$(6,L,+\infty)$ $(&quot;)$(&quot;) #</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>$(12,L,+\infty)$ $(13,L,+\infty)$ $(16,S,30)$ $(18,S,21)$ #</td>
</tr>
<tr>
<td>11</td>
<td>5</td>
<td>$(24,L,+\infty)$ $(13,L,+\infty)$ $(&quot;)$(&quot;) #</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>$(13,L,+\infty)$ $(16,S,30)$ $(24,S,28)$ $(18,S,21)$ #</td>
</tr>
<tr>
<td>13</td>
<td>5</td>
<td>$(26,L,+\infty)$ $(&quot;)$(&quot;)$(&quot;) #</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>$(26,S,78)$ $(&quot;)$(&quot;)$(&quot;) #</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
<td>$(27,L,78)$ $(&quot;)$(&quot;)$(&quot;) #</td>
</tr>
<tr>
<td>16</td>
<td>4</td>
<td>$(27,S,52)$ $(&quot;)$(&quot;)$(&quot;) #</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>$(13,S,52)$ $(111)$ $(111)$ $(111)$ #</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>$(6,S,52)$ $(16,S,30)$ $(18,S,21)$ #</td>
</tr>
<tr>
<td>19</td>
<td>2</td>
<td>$(7,L,52)$ $(16,S,30)$ $(18,S,21)$ #</td>
</tr>
<tr>
<td>20</td>
<td>6</td>
<td>$(14,L,52)$ $(15,L,52)$ $(16,S,30)$ $(18,S,21)$ #</td>
</tr>
<tr>
<td>21</td>
<td>5</td>
<td>$(28,L,52)$ $(&quot;)$(&quot;)$(&quot;) #</td>
</tr>
<tr>
<td>22</td>
<td>4</td>
<td>$(15,L,52)$ $(16,S,30)$ $(28,S,22)$ $(18,S,21)$ #</td>
</tr>
<tr>
<td>23</td>
<td>5</td>
<td>$(30,L,52)$ $(&quot;)$(&quot;)$(&quot;) #</td>
</tr>
<tr>
<td>24</td>
<td>4</td>
<td>$(30,S,52)$ $(&quot;)$(&quot;)$(&quot;) #</td>
</tr>
<tr>
<td>25</td>
<td>2</td>
<td>$(31,L,52)$ $(&quot;)$(&quot;)$(&quot;) #</td>
</tr>
<tr>
<td>26</td>
<td>4</td>
<td>$(31,S,52)$ $(&quot;)$(&quot;)$(&quot;) #</td>
</tr>
<tr>
<td>27</td>
<td>3</td>
<td>$(15,S,52)$ $(&quot;)$(&quot;)$(&quot;) #</td>
</tr>
<tr>
<td>28</td>
<td>1</td>
<td>$(7,S,52)$ $(16,S,30)$ $(18,S,21)$ #</td>
</tr>
<tr>
<td>29</td>
<td>3</td>
<td>$(3,S,52)$ $(16,S,30)$ $(18,S,21)$ #</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>$(1,S,52)$ #</td>
</tr>
</tbody>
</table>
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 \( \Gamma \) defined in Table I computes the minimax value of the root node of any game tree.

Proof: It is clear that \( \Gamma \) 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 \( f_T(n) \). By Theorem 1 there exists some tree \( T_0 \) such that \( f_{T_0}(n) = \hat{h}(n) \), the minimax evaluation of the game tree rooted at node \( n \). 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 \( \hat{h} \geq f_{T_0}(n) \geq f_{T_1}(n) \). (This is the same as Nilsson's proof of admissibility for A* with \( \hat{h} \) being an overestimating 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. Q.E.D.