Another view on the SSS* algorithm

Wim Pijls,
Arie de Bruin
Erasmus University Rotterdam
P.O.Box 1738, NL-3000 DR Rotterdam,
The Netherlands,
wimp@cs.eur.nl

September 26, 1995

Abstract

A new version of the SSS* algorithm for searching game trees is presented. This algorithm is built around two recursive procedures. It finds the minimax value of a game tree by first establishing an upper bound to this value and then successively trying in a top down fashion to tighten this bound until the minimax value has been obtained.
This approach has several advantages, most notably that the algorithm is more perspicuous. Correctness and several other properties of SSS* can now more easily be proven. As an example we prove Pearl's characterization of the nodes visited by SSS* [Pl].

1 Introduction

During the last two decades several algorithms have been developed for computing the minimax value of a game tree. The most famous one is the alpha-beta-algorithm [Kn]. Another well known one is the SSS*-algorithm [St]. In this paper a new version of the latter algorithm will be developed equivalent with SSS* in the sense that the same nodes are examined in the same order. Recursion plays a key role in this version.
The SSS* algorithm originates from Stockmann [St]. The explanation in this paper is rather opaque, the algorithm is presented as a (semi-) parallel search in the "state space" consisting of so called partial (min-) solution trees. Later papers by Kumar and Kanal [KK1, KK2] recognized this algorithm as a special case of Branch and Bound. In [KK2] the observation is made that there is a dual view on SSS*, namely as a (sequential) search over (max-) solution trees. We will expand on this view and give it some formal underpinning.
Several authors have studied SSS*, most notably Pearl [Pl] and Ibaraki [Ib1]. These papers give amongst others characterizations of the nodes which are visited by SSS*, and contain proofs of the superiority of SSS* over alpha-beta in

this respect. Due to the fact that in these papers the SSS* algorithm is specified essentially as a search these investigations are complicated.
In this paper we present a version of SSS* as a top down recursive algorithm. This leads to a more perspicuous description of the algorithm. We will use this presentation amongst others to present an alternative proof of the fact that SSS* surpasses alpha-beta [Ib1].
After some preliminaries, our version of SSS*, which will be called SSS-2, is introduced in section 3. In section 4 some properties are derived which will be needed for a comparison between SSS-2 and alpha-beta. Some of these results are similar to the ones published by Pearl and Ibaraki, also a few new results are presented. In section 5 we will make a few remarks on implementation issues, and we will sketch how SSS-2 can be transformed into the usual format.

2 Game trees and solution trees

Game trees are related to two person games with perfect information like Chess, Checkers, Go ,Tic-tac-toe, etc. Each node in a game tree represents a game position. The root represents a position of the game for which we want to find the best move. The children of each node n correspond to the positions resulting from one move from the position given by n. The terminals in the tree are positions in the game for which a real valued evaluation function f exists giving the so called game value, the pay-off of that position.
We assume that the two players are called MAX and MIN. A node n is marked as max-node or min-node if in the corresponding position it is max's or min's move respectively. We assume that MAX moves from the start position.
The evaluation function can be extended to the so called minimax function, a function which determines the value for each player in any node. The definition is:

$f(n) = \max \{f(x) | x \textit{ child of } n\}$, *if* x *is a max node,*
$\min \{f(x) | x \textit{ child of } n\}$, *if* x *is a min node,*

We adopt the convention that the minimax value of a game tree T, denoted by $f(T)$, is the minimax value of the root of this tree. In Figure 1 an example of a game tree is shown labelled with its f-values. The bold lines in this figure define a so called solution tree. This is an important notion in our version of SSS*.

Definition 1 *A (max-) solution tree S in a game tree T is defined as a subtree T with the property that for all non terminal nodes n in S we have:*

- *if n is a max node, then all its children are included in S.*
- *if n is a min node, then exactly one of its children is included in S.*

In [KK2] such a tree is called an OR solution tree. The standard explanation of the SSS* algorithm is based on the complementary notion, which we will call a min-solution tree. Such a tree can be defined by interchanging the restrictions for min and max nodes in the above definition. Given a game tree T and a node $n \in T$, we denote the set of all solution trees rooted in n by $\mathcal{S}_T(n)$, or $\mathcal{S}(n)$ if no confusion can occur. $\mathcal{S}_T$ stands for $\mathcal{S}_T(\textit{root of } T)$.
We adopt the convention that the minimax function in a solution tree will be denoted by the letter g.
Lemma 1 () *[(St]) Let S be a solution tree. Then for all $x \in S$ we have $g(x) \geq f(x)$.*

Proof. By induction on the height of S. □

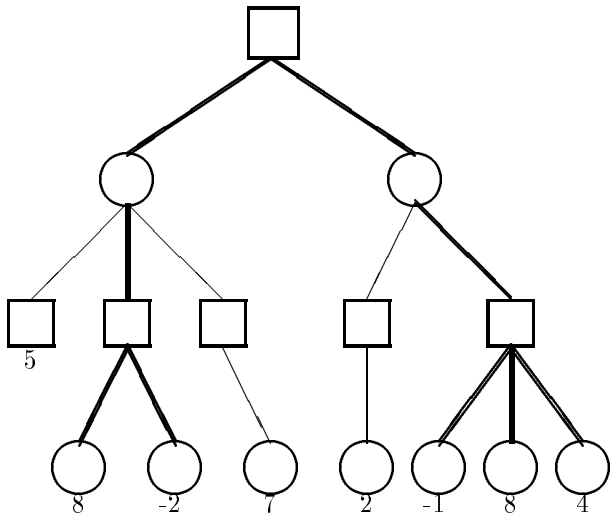

Figure 1: A game tree with some f-values.

Lemma 2 *For each game tree T there exists a solution tree S in $\mathcal{S}_T$ such that $f(T) = g(S)$.*

Proof. We give a construction of S. Firstly the root of T is put in S. Then proceed with the construction recursively: append to each min node $n \in S$ that is non terminal in T a child with minimal f-value, append to each max node in S that is not a terminal in T all its children. It can be shown by induction that $\forall x \in S : g(x) = f(x)$. □

In the sequel it will be useful to have a kind of genealogical ordering $\gg$ on game trees at our disposal. To be able to establish such an order we assume that in any non-terminal the child nodes have a fixed order, i.e. that we can establish whether one child is older than another one.

Definition 2 *Let n be a node in a game tree T. Let S and S' be two different solution trees in $\mathcal{S}_T(n)$. Because $S \neq S'$ n cannot be a terminal. We define $S \gg S'$ recursively as follows:*
If n is a max node, then consider the oldest child m of n such that the subtrees $\bar{S}$ and $\bar{S}'$, both rooted in m are different (because $S \neq S'$, such a subtree must exist). We define $S \gg S'$ if $\bar{S} \gg \bar{S}'$.
Suppose n is a min node. Let $\bar{S}$ be the subtree of S, rooted in the child m of n in S, and let $\bar{S}'$ be the subtree of S', rooted in the child m' of n in S'. We define $S \gg S'$ if either m is older than m' or $m = m'$ and $\bar{S} \gg \bar{S}'$.

Notice that for every n, we have that $\gg$ is a total ordering on the set $\mathcal{S}_T(n)$. Related to this ordering is an ordering $\gg$ on nodes which is an extension of the "older than" relation. We say that $m \gg n$ iff there exist two ancestors m' and n' of m and n respectively, such that m' and n' have the same parent and m' is

older than n'. An intuitive explanation of this ordering is as follows. The game tree may be viewed as a genealogy of a royal family or a dynasty. The root is the actual king. The ordering $\gg$ on the nodes in the tree corresponds to their priority in succeeding the king.

3 The new SSS* version

The new version of SSS*, which we call SSS-2, is based on the idea to first establish an upper bound for $f(T)$, the game tree under consideration, and after that to repeatedly transform this upper bound into a tighter one. This is repeated until the upper bound cannot be diminished any more, in which case we have determined the minimax value of T.
We can establish an upper bound on a game tree T by exploiting the following recursive property of such an upper bound. The bottom of the recursion occurs when T consists of a terminal node only. In that case the game value of this node is a good upper bound. If the root of T is a max node, then we can obtain an upper bound for $f(T)$ by establishing an upperbound for each of its children and taking the maximum of these bounds. If the root of T is a min node then we do not need to investigate all its children, an upper bound of any child of the root is also an upper bound for the root itself.
If we turn this description into an algorithm (this will be the procedure "expand" to be defined later on), then it is clear that in order to find an upper bound in this way, we have to construct a solution tree in $\mathcal{S}_T$. We will organize "expand" in such a way that it constructs the oldest solution tree. This will be realized by taking in a min node the oldest child instead of "any child".
Now if we want to tighten the upper bound related to this solution tree, say S, we can realize this by transforming S as follows. First of all, if S consists of only one terminal node, then it is not possible to generate a better upper bound: we have obtained the minimax value. Now if the root of S is a max node, we can obtain a better upper bound only if for all children c of the root with $g(c) = g(root)$ a solution tree in $\mathcal{S}_T(c)$ can be generated with a lower g-value than $g(c)$.
If, on the other hand, the root of S is a min node then there are more possibilities to generate a better upper bound. First of all, one can try to obtain (recursively) a better upper bound for the current child of the root of S. But it is also possible to select another child c' of the root, and to try to establish an upper bound for $f(c')$ by building a new solution tree in $\mathcal{S}_T(c')$ with g- value $< g(S)$. Finding such a new solution tree is in fact a generalization of the expand process mentioned before: the difference is that now we are not satisfied with any new solution tree, we want a solution tree with a g- value better than a given value (i.e. g(root)). We will therefore define a procedure $expand(n, v_{in}, v_{out})$ that generates the oldest solution tree in $\mathcal{S}_T(n)$ with g -value $< v_{in}$ (if possible). Notice that by taking $v_{in} = \infty$ the oldest solution tree is obtained.
Later in this section we will define the procedure *diminish* which is based on the description given above on how to obtain from a given solution tree a better one. This procedure has three parameters, a node n, an input value v_{in}, and an output parameter v_{out}. On call of this procedure it is supposed that there is a solution tree S rooted in n with $g(S) = v_{in}$. If possible, after execution there will be a new solution tree S', rooted in n, with the property that $g(S') = v_{out} < g(S)$ and furthermore that it is the oldest solution tree with that property.

The overall idea behind SSS-2 can thus be described as follows. Construct the oldest solution tree S in T. Repeatedly transform this solution tree S into another tree S' such that $g(S') < g(S)$ and $S' \ll S$ in such a way that if this transformation does not succeed, we have obtained the solution tree with the smallest g-value which is, by Lemma 1 and 2 the minimax value of T. The algorithm uses a global variable G which has as a value the current solution tree in $\mathcal{S}_T$.

We now give the main body of SSS-2.

begin
  root:= the start position of the game;
  expand(root, ∞, v_{out}, *S);*
  $G := S$;
  repeat
    [$v_{in} := v_{out}$;
      diminish(root, v_{in} , v_{out} *);*
    ]
  until $v_{in} = v_{out}$;
end.

In order for this program to work correctly the procedures *expand* and *diminish* must have the properties sketched above. In a moment we will give a precise specification for these procedures. But we need a definition first.

Definition 3 *Let* $n \in T$, $G \in \mathcal{S}_T$, v *a real number. The triple* $< n, v, G >$ *is called D-correct if* $n \in G, v = g(G_n)$, *where* G_n *is the subtree of* G *rooted in* n, *and for all subtrees* S *in* $\mathcal{S}_T(n)$ *older than* G_n *we have that* $g(S) > v$.

Specification of procedure *diminish(n,* v_{in}, v_{out}*)*: input-parameters: n, a node in the solution tree G, v_{in}, a real number
output-parameter: v_{out}, a real number.

If the call *diminish(n,* v_{in}, v_{out}*)*: is executed in a situation with global solution tree G such that $< n, v_{in}, G >$ is D-correct, then
either this call terminates with $v_{in} = v_{out}$ (no better solution tree found), in which case $v_{in} = v_{out} = f(n)$,
or the call terminates with $v_{in} > v_{out}$ (a better solution tree was found), in which case G has a new subtree G_n, rooted in n, with $g(G_n) = v_{out}$, such that G_n is the oldest solution tree with g-value $< v_{in}$.
Notice that in the latter case, on succesful termination of diminish, we apparently have that the triple (n, v_{out}, G) is again D-correct.
Specifications of the procedure *expand(n,* v_{in}, v_{out}, *S)* input-parameters: n, a node in T, v_{in},a real number
output-parameters: S, a solution tree rooted in n v_{out}, a real number.
The call *expand(n,* v_{in}, v_{out}, *S)*
either terminates with $v_{in} = v_{out}$ (no suitable tree found), in which case $f(n) \geq v_{in}$,
or terminates with $v_{in} > v_{out}$ (there is a suitable tree), in which case S is the oldest solution tree in $\mathcal{S}_T(n)$ with g-value $< v_{in}$, and furthermore $g(S) = v_{out}$.
We now give the code of *diminish* and *expand*.
procedure diminish(n, v_{in}, v_{out}*)*
begin
  if terminal(n) then $v_{out} := v_{in}$
  else if type(n)=max then

```
        [ for c:= firstchild(n) to lastchild(n) do
              [ if g(c) = v_in then diminish(c,v_in, v'_out);
                if v_in = v'_out then exit forloop;
              ]
          v_out := max(g-values of all children of n)
        ]
  else if type(n)=min then
        [ c := the single child of n in G;
          diminish (c, v_in, v_out) ;
          if v_in = v_out then
            for b:=nextbrother(c) to lastbrother(c) do
                [ expand(b, v_in, v'_out, S);
                  if v_in > v'_out then
                    [ detach in G from n the subtree rooted in c
                              and attach S to n;
                      v_out := v'_out;
                      exit forloop;
                    ]
                ]
        ]
end.
procedure expand(n, v_in, v_out , S);
begin
   if terminal(n) then
     [ if f(n) < v_in then S := the tree consisting only of node n;
       v_out := f(n);
     ]
   else if type(n) = max then
          [ for c := firstchild(n) to lastchild(n)
                [ expand (c, v_in, v'_out, S');
                  if v'_out ≥ v_in then
                    [ v_out := v'_out;
                      exit forloop;
                    ]
                ]
            S := the tree composed by attaching
                  all intermediate values of S' to n;
            v_out := max of all intermediate values of v'_out;
          ]
   else if type (n)=min then
          [ v_out := v_in;
            for c := firstchild(n) to lastchild(n)
                [ expand (c, v_in, v'_out, S');
                  if v'_out < v_in then
                    [ S := tree with S' attached to n;
                      v_out := v'_out;
                      exit forloop;
                    ]
                ]
          ]
end;
```

Lemma 3 *The procedures* diminish *and* expand *meet the specifications stated above.*

Proof (outline). It is sufficient to prove that the bodies of *diminish* and *expand* are correct, under the assumption that the inner calls of *diminish* and *expand* within these bodies confirm to the specifications. In essence this is a proof by induction on the number of nested calls. In order to prove correctness of the bodies it is, amongst others, needed to check whether before each call of diminish its precondition (D- correctness) is established. □

Lemma 4 *During execution of the SSS-2 algorithm we have that before each call of diminish the tripel* $< n, v_{in}, G >$ *is D-correct. Here* n *and* v_{in} *are parameters of the call and* G *is the global solution tree.*

Proof (outline). From Lemma 3 we can prove that before each call in the main program we have a D-correct situation. That this is also the case for inner calls can be established by the argument used in the proof of Lemma 3. □

Corollary 1 *For every finite tree* T *SSS-2 terminates with* $v_{in} = f(T)$.

Proof. There are only finitely many elements in $\mathcal{S}_T$. If *diminish* terminates with $v_{out} < v_{in}$ then, by the specification of *diminish*, we have that G has changed into a younger solution tree. Therefore the repeat loop in the main program must terminate. That $v_{in} = f(T)$ follows from Lemma 3 , Lemma 4 and the specification of *diminish*.

Corollary 2 *Consider an execution of SSS-2 on a game tree* T. *Suppose the main loop performs* n *iterations, and let* G_i *be the value of the global solution tree* G *before the* i*-th call of diminish in this loop. Then we have:*
-G_1 *is the oldest solution tree in* $\mathcal{S}_T$.
-G_{i+1} *is the oldest solution tree with g-value* $< g(G_i)$
$(i = 1, \ldots, n-1)$.

Proof. Immediate from Lemma's 3 and 4. □
We call $G_1, \ldots, G_n$ the solution tree sequence related to the execution of SSS-2 on T. Notice that Corollary 2 does not state anything on the final value $\bar{G}$ of G on termination of the algorithm. It is always true $g(G_n) = g(\bar{G})$, but it is not necessarily the case that $G_n = \bar{G}$.

4 Some properties of the new algorithm

In this section we will compare the efficiency of SSS-2 and the alpha-beta algorithm. Before doing so a few useful results will be derived. First a few definitions. For each node n in the game tree the quantities $\beta(n)$, $\alpha_L(n)$ and $\alpha_R(n)$ can be defined. These quantities play an important role in the well known alpha-beta-algorithm. Firstly we need the the notions AMIN, AMAX, AMIN-LC and AMAX-LC. The definitions are quite equal to those in [Pl]; in [Ib1] the same notions are used by a different name.

Definition 4 *For each node* n *the following quantities are defined:*
$AMAX(n) = \{\ x \mid x$ *is a max-node and* x *is a proper ancestor of* $n\ \}$
$AMIN(n) = \{\ x \mid x$ *is a min-node and* x *is a proper ancestor of* $n\ \}$
$AMIN\text{-}LC(n) = \{\ x \mid x$ *is a child of an element in* $AMIN(n)$ *and* $x \gg n\ \}$
$AMAX\text{-}LC(n) = \{\ x \mid x$ *is a child of an element in* $AMAX(n)$ *and* $x \gg n\ \}$
$AMAX\text{-}RC(n) = \{\ x \mid x$ *is a child of an element in* $AMAX(n)$ *and* $x \ll n\ \}$

See Figure 2 to illustrate these definitions.

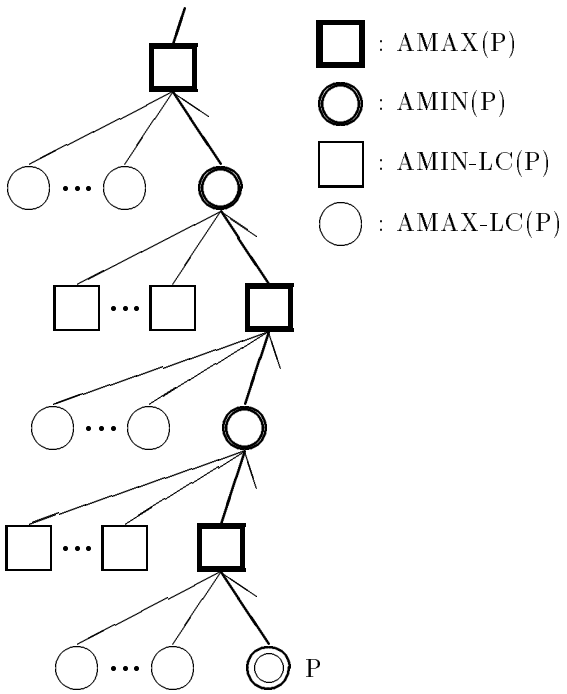

Figure 2: The sets AMAX, AMIN, AMAX-LC, AMIN-LC.

Definition 5 *Suppose n is a node in a game tree. Then we define:*
$\beta(n) = \min\{f(x) | x \in AMIN\text{-}LC(n)\ \}$
$\alpha_L(n) = \max\{f(x) | x \in AMAX\text{-}LC(n)\ \}$
$\alpha_R(n) = \max\{f(x) | x \in AMAX\text{-}RC(n)\ \}$
(we assume $\max(\emptyset) = -\infty$ *and* $\min(\emptyset) = \infty$*).*

Further we define the notion of β-ancestor of n. This node is the ancestor of n that is responsible for the value $\beta(n)$, i.e. this ancestor has a child in AMIN-LC with minimal f-value over AMIN-LC.

Definition 6 *Given a node n in a game tree. Suppose m' is a node such that m' in AMIN-LC(n) and $f(m') = \beta(n)$. Then the β-ancestor of n is defined as the father of m'. In case of ties the node closest to the root is chosen. If AMIN-LC(n)$= \emptyset$ we take the root of T as the β-ancestor. Notice that for every node n for which AMIN-LC(n)$\neq \emptyset$ we have that the β-ancestor of n is a min node.*

We now derive a few results on the execution of SSS-2. First, we establish a property of every node n that is subjected to a diminish call during this execution.

Lemma 5 *During execution of SSS-2, before each call* diminish(n, $v_{in}, \ldots,$) *we have that $\beta(n) > v_{in} > \alpha_L(n)$ and that $v_{in} \geq \alpha_R(n)$.*

Proof. It is sufficient to prove that if this property holds before a call of *diminish*, that it then also holds for each inner call of *diminish* in the body, cf. the remarks in the proof of Lemma 1. In order to prove this, Lemma's 1 and 2 must be used. □

We have a similar results for expand calls.

Lemma 6 *During execution of SSS-2, before each call* expand(n, $v_{in}, \ldots,$) *we have that $\beta(n) = v_{in} > \alpha_L(n)$ and that $\beta(n) \geq \alpha_R(m)$, where m is the β-ancestor of n.*

Proof. Similar to the proof of Lemma 5. In the proof of lemma 5 must be used as well. For the case that a call of *expand(n,* $v_{in}, \ldots$) is performed inside the body of *diminish* we use the fact that the β-ancestor of n is its parent. □

Now the superiority of SSS-2 (and SSS*) over alpha-beta follows from these lemma's and the following result by Pearl[Pl]:

Lemma 7 *A node n is examined by the alpha-beta algorithm if and only if $\beta(n) > \alpha_L(n)$.*

Proof. See [Pl]. Notice that a different proof can be given along the lines of Lemma 5 and 6. □

Theorem 1 *SSS-2 surpasses alpha-beta in the sense that SSS-2 visits no more nodes than alpha-beta does.*

Proof. From the algorithm it is clear that a node can only be visited by SSS-2 if it is created in an expand call. Now the theorem follows from Lemma's 6 and 7.

The next result that we want to derive is an extension of Lemma's 5 and 6: we want to establish that the property of n given in Lemma 6 is not only a necessary condition for a node to be expanded but a sufficient condition as well. This result gives us a precise characterization of the nodes generated by SSS*, similar to the result (see Lemma 7) for alpha-beta. We need an auxiliary lemma first.

Lemma 8 *Suppose during execution of SSS-2 a call of* expand(n, $v_{in}, \ldots$) *is performed with* $v_{in} = \beta(n)$. *Then during this call a recursive call* expand(m, $v_{in}, \ldots$) *will be generated for every child* m *of* n *for which* $\beta(m) = \beta(n)$ *and* $\beta(m) > \alpha_L(m)$ *holds.*

Proof. We treat only the case that n is a max node (the case that n is a min node uses a similar argument).
If n is a max node then we have for all children m of n that $\beta(m) = \beta(n)$ and that $\alpha_L(m)$ is the maximum of $\alpha_L(n)$ and the values $f(m')$ for m' the older brothers of m. By Lemma 6 we have that $\beta(m) = \beta(n) > \alpha_L(n)$. Moreover, one straightforwardly shows that an expand call will be generated for a child m of n only if all its brothers have an f-value $> \alpha_L(m)$.

Lemma 9 *For each node* n *in a game tree* T *we have that* n *is examined by SSS-2 if* $\beta(n) > \alpha_L(n)$ *and* $\beta(n) \geq \alpha_R(m)$, *where* m *is the* β*-ancestor.*

Proof. Choose such a node n. Without loss of generality we can assume that AMIN-LC(n) is not empty, i.e. that the we have a proper β-ancestor of n. This is the case if n is at least two levels deep in the game tree.
We will prove that n will be subjected to a call of *expand*, by constructing a solution tree $S \in \mathcal{S}_T$ that will be an element G_i in the solution tree sequence (cf. the remarks after Lemma 2), and furthermore showing that in the next iteration of the main loop of SSS-2 the call *diminish(root,...)* will generate, somewhere in the recursion, a call of *expand(n, ...)* .
Suppose the β-ancestor of n is m and consider the path from m upward to the root. This will be the backbone of the solution tree to be constructed. See Figure 3.
Attach to the max nodes in this path all its children p (which are therefore members of AMAX-LC(m) and AMAX-RC(m)). We will next describe what the subtrees S_p rooted in these nodes p should be.
For each node p in AMAX-LC(m) we have $f(p) \leq \alpha_L(m) \leq \alpha_L(n) < \beta(n)$. So there exists for each p in AMAX-LC(n) a solution tree S_p rooted in p with the property $g(T_p) \leq \beta(n)$. Attach the oldest solution tree with this property to p.
For each node $p \in$ AMAX-RC(m) we have $f(p) \leq \alpha_R(m) \leq \alpha_R(n) \leq \beta(n)$.
Attach to such a p the solution tree S_p defined as the oldest solution tree rooted in p with $g(S_p) \leq \beta(n)$. Because m is the β-ancestor of n, m has a child with f-value $= \beta(n)$. Let c_1 be the oldest child of m with that property. The child of m on the path from m to n is called c_2. Then we have that c_2 is younger than c_1, that the children of m older than c_1 have an f-value $> \beta(n)$, and that the children of m between c_1 and c_2 have f-value $\leq \beta(n)$. Since $f(c_1) = \beta(n)$, there exists a solution tree rooted in c_1 with g-value $= \beta(n)$. Attach the oldest solution tree with this property to m. We call this tree S_1.
This construction has now resulted in a solution tree $S \in \mathcal{S}_T$ with g-value $= \beta(n)$. Moreover for every $S' \gg S$, we have $g(S') > g(S)$. Therefore S is an element of the solution tree sequence discussed after Corollary 2.
Now consider the call *diminish(root,...)* in the main loop of the program which tries to diminish this S. The reader is invited to check that during this call, the procedure expand is activated with input parameters $n = c_2$ and $v_{in} = f(c_1)$.
For all nodes n' on the path from c_2 up to and including n we have the property that $\beta(n') = \beta(n) > \alpha_L(n) \geq \alpha_L(n')$, that n' and n both have m as β-ancestor, and thus that $\beta(n') \geq \alpha_R(m)$. By induction on the length of the path from c_2 to n we can prove, using Lemma 8, that each n' on this path will be expanded. So, n will be expanded. □

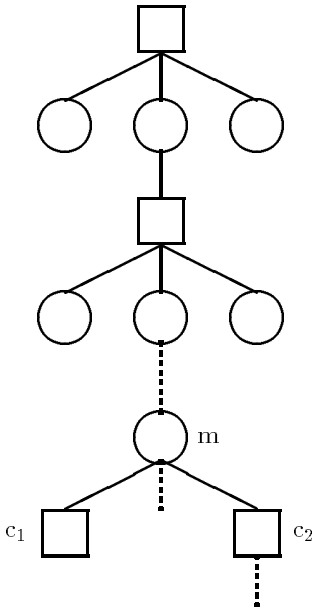

Figure 3: The tree constructed in the proof of Lemma 9.

As mentioned in Lemma 7 the nodes which are examined by alpha-beta are characterised by one simple condition. We have seen in the previous lemma that the characterisation of the nodes visited by SSS-2, needs one additional condition. In the next theorem we show a particular type of game tree where this additional condition is satisfied automatically for any node that satisfies the alpha-beta condition.

Theorem 2 *Suppose T is a game tree with the property that in each max node n the children $c_1, c_2, \ldots, c_n$ are ordered in such way that $f(c_1) \geq f(c_2) \geq \ldots \geq f(c_n)$. Then the alpha-beta-algorithm and SSS-2 examine the same set of nodes.*

Proof. For the given game tree T it holds for any node x: $\alpha_L(x) \geq \alpha_R(x)$.
Moreover in any game tree it holds for any two nodes x and y: if y is an ancestor of x, then $\alpha_R(x) \geq \alpha_R(y)$
Suppose that a node n is examined by alpha-beta; then by Lemma 6 $\beta(n) > \alpha_L(n)$. Now we may state: $\beta(n) > \alpha_L(n) \geq \alpha_R(n) \geq \alpha_R(m)$, where m is the β-ancestor of n. By Lemma 8 the node n will be examined by SSS*. For the given game tree T alpha-beta surpasses SSS*. As a consequence of Theorem 1 it follows that both algorithms examine the same set of nodes. □

The alpha-beta algorithm (see [Kn]) contains a recursive procedure with three input parameters: a node parameter n and two real parameters called alpha and beta respectively. The main body of the algorithm calls this procedure with $n =$ *root of* T (T is a game tree), alpha$=-\infty$ and $\beta = +\infty$. It can be shown that the alpha-beta procedure returns the value $f(T)$ also when the two parameters have values such that $alpha < f(t) < beta$ holds. The set of examined nodes becomes smaller as the distance between alpha and beta decreases. In the next

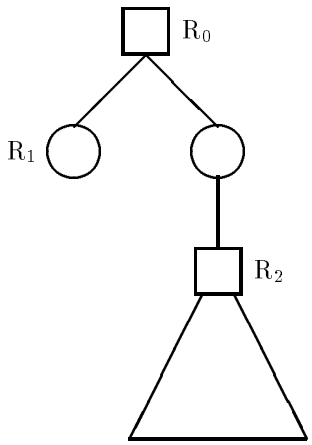

Figure 4: The extended tree in Theorem 3

theorem a smaller alpha-beta window is given that still yields a superset of the node set examined by SSS-2.

Theorem 3 *Given a game tree T such that $f(T) = v_0$. Suppose the procedure alpha-beta is executed with parameters alpha=$v_0 - \varepsilon$ and beta= ∞ where ε is any constant > 0. Then the set of nodes examined by the alpha-beta procedure is a superset of the set of nodes that is examined by SSS-2.*

Proof. We enhance the game tree T by some additional nodes in the way that is illustrated in Figure 4 . The node R_2 is the root of the original game tree T; we define $f(R_1) = v_0 - \varepsilon$. When SSS-2 would be applied to the new tree, the same descandants of R_2 would be examined. Calling the procedure alpha-beta with $n = R0$, alpha= $-\infty$ and beta= $+\infty$ causes a subcall with $n = R_2$, alpha=$v_0 - \varepsilon$ and beta= $+\infty$. So by Lemma 7 we conclude the theorem. □

5 Implementation issues

The global variable G should be implemented as a tree where the nodes are labelled by its g-value. In [KK2] it is observed that the list OPEN, used in the original SSS*, represents a OR tree by the set of its leaves. Careful observation reveals that it is possible to use this representation in our version as well.
The procedure *expand* can be adapted to this new representation in a straightforward fashion. The only difference is that the parameter S is now implemented as a list of its terminals. The prodecure *diminish* requires closer inspection. The execution of this procedure can be divided into three stages. During the first stage a terminal is selected in a top down fashion, using the g-value in the internal nodes of G. This terminal is the oldest one with maximal value. In the second phase it is attempted to obtain a better g-value for some ancestors of this terminal. This can be done by a strictly local search: backing up to a father and expanding a younger child. As soon as a better g-value has been found, the execution enters its third phase, which in essence amounts to updating g-values upwards until the root.
If we switch to a new representation the first stage can be reduced to selecting the oldest terminal with the maximum value. During the second stage we walk up the tree performing some expand calls on the way. The third phase can be

omitted, since there is no need to update g-values any longer.
Notice that this new description is closer in spirit to the original Stockman version. Notice also that we only deal with a list of terminals ordered with respect to their game value. There is no need for control information as it is done in the Stockman triples, e.g. Solved/Live and g-values.

6 Conclusions

In this paper a new version of SSS* has been presented which is more transparant and "better suited for understanding" whereas the original version is "more convenient for implementation" [Ib1]. This enabled us to prove several properties of the algorithm, e.g its correctness and its surpassing alpha-beta. Our intention is to extend these results towards related algorithms, e.g. the RSEARCH version of SSS*.

References

[Ib1] T. Ibaraki, *Generalisation of Alpha-Beta and SSS* Search Problems,* Artificial Intelligence 29 (1986) 73-117.

[Ib2] T. Ibaraki, *Searching Minimax Game Trees under Memory Space Constraint*, to appear.

[KK1] V. Kumar and L.N. Kanal, *A General Branch and Bound Formulation for Understanding and Synthesizing And/Or Tree Search Procedures*, Artificial Intelligence 21 (1983) 179-198

[KK2] V. Kumar and L.N. Kanal, *Parallel Branch and Bound Formulations for AND/OR Tree Search*, IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol PAMI-6 no. 6, november 1984.

[Kn] D.E.Knuth and R.W.Moore, *An Analysis of Alpha-Beta Pruning,* Artificial Intelligence 6 (1975), 293-326.

[Pl] I.Roizen and J. Pearl, *A Minimax Algorithm Better than Alpha-Beta? Yes and No.* Artificial Intelligence 21 (1983) 199-220.

[St] G.C. Stockman, *A Minimax Algorithm Better than Alpla-Beta?*, Artificial Intelligence 12 (1979) 179-196.