Searching informed game trees

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

Abstract

Well-known algorithms for the evaluation of the minimax function in game trees are alpha-beta [Knuth] and SSS* [Stockman]. An improved version of SSS* is SSS-2 [Pijls-1]. All these algorithms don't use any heuristic information on the game tree. In this paper the use of heuristic information is introduced into the alpha-beta and the SSS-2 algorithm. Extended versions of these algorithms are presented. The subset of nodes which is visited during execution of each algorithm is characterised completely.

1 Introduction

In this paper several methods are discussed to compute the minimax function on a game tree with heuristic information.
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:

$$\begin{aligned} f(n) = &\max \{f(c) \mid c \textit{ a child of } n\}, \textit{ if } n \textit{ is a max node,} \\ &\min \ \{f(c) \mid c \textit{ a child of } n\}, \textit{ if } n \textit{ is a min node.} \end{aligned}$$

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 labeled with its f-values. The bold lines in this figure define a so called solution tree, which is to be defined in Section 4.

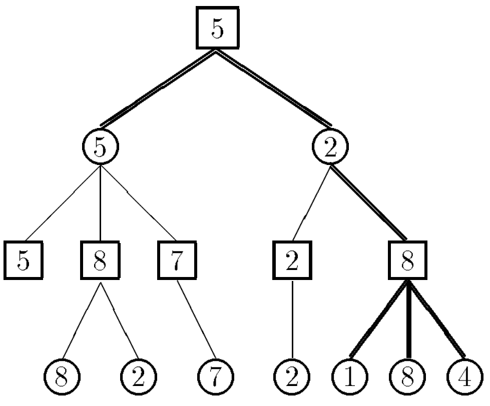

Figure 1: A game tree with f-values.

The value $f(n)$ in any node n (n not necessarily a max node) indicates the highest attainable pay-off for MAX in the position n, under the condition that both players will play optimally in the sequel of the game. In any node n the move for each player to optimize the pay-off is the transition to a child node c such that $f(c) = f(n)$. In this way, MAX tries to maximize and MIN tries to minimize the profit of MAX. Therefore, an optimal play will proceed along a *critical path*, which is defined as a path from the root to a leaf such that $f(n)$ has the same value for all nodes n in the path. All nodes in this path have a game value equal to the game value of the root.

For some game trees, heuristic information on the minimax value $f(n)$ is available for any node. This information can be expressed as a pair $H = (U, L)$ where U and L are heuristic functions mapping the nodes of the game tree into the real numbers, such that $U(n) \geq f(n) \geq L(n)$ for any node n and $U(n) = f(n) = L(n)$ for every terminal n. The heuristic functions thus denote an upper bound and a lower bound respectively of the minimax value. A heuristic pair $H = (U, L)$ is called consistent if $U(c) \leq U(n)$ for every child c of a given max node n, and $L(c) \geq L(n)$ for every child c of a given min node. From now, we assume that an input instance consists of a pair (G, H), called an informed game tree, where G denotes a game tree and H a pair of consistent heuristic functions. If heuristic information is discarded or is not available at all, we define $U(n) = +\infty$ and $L(n) = -\infty$ for every non-terminal node n.

In order to compute the minimax value of a game tree, several algorithms have been developed. The brute force approach would compute the minimax function in each node of the game tree according to the definition. Each feasible algorithm has its own method to avoid examining the entire tree.

The oldest algorithm is the so-called alpha-beta algorithm [Knuth]. Another important algorithm is called SSS* [Stockman]. The working of this algorithm is rather opaque. In [Pijls-1] an algorithm, called SSS-2, is presented, which traverses the game tree in the same order as SSS*. However, the underlying paradigm is much more perspicuous than in the case of SSS*. All these algorithm do not take into account any heuristic information. Ibaraki [Ibaraki] introduced the idea to exploit heuristic information for improving the efficiency of game tree algorithms. In our current paper we will generalise alpha-beta and SSS-2 in the sense that a heuristic pair H features in the algorithm. Furthermore, a complete characterisation is given of the set of nodes visited during execution. For the case that no heuristic information is taken into account, such a characterisation has been found for alpha-beta and SSS* by Baudet [Baudet] and Pearl [Pearl] respectively. However, when our characterisation is restricted to such a situation, we have simpler results. Because of the extensive use of recursion, our proofs differ completely from theirs.
Moreover, when the proofs in this paper are reduced to the case without heuristic estimates, we obtain improved versions with respect to those in [Pijls-1].

In Section 2, the extended alpha-beta algorithm is discussed. In Section 3 the characterisation is given for the nodes visited by alpha-beta. After introducing in Section 4 the notion of a solution tree, which plays a key role in SSS-2, the SSS-2 algorithm itself is presented in Section 5. In Section 6 we elaborate on the correctness of some procedures used in the algorithm. In Section 7 the characterisation is given for the nodes examined by SSS-2. It appears that SSS-2's set of nodes visited during execution is a subset of the corresponding alpha-beta set. Section 8 contains some notes on the implementation for a special case.

In this paper, correctness proofs will be given of the procedures featuring in our algorithms. This will be done in the following fashion. Consider a call $P(t_1, t_2, \ldots, t_n)$ of procedure P with parameters $t_1, t_2, \ldots, t_n$. First of all, we shall establish a specification of P, consisting of a precondition p and a postcondition q, describing relationships between the parameters of P and, if needed, the relevant global variables.
We will be concerned with partial correctness of P with respect to such a specification, defined as follows: whenever a call $P(t_1, t_2, \ldots, t_n)$ is executed starting in a situation, in which precondition p holds and given the fact that this call terminates, then on termination the postcondition q will be true [Hoare].
We shall often use recursion in our algorithms. In general, a call of a recursive procedure will generate a tree of nested calls. We will use the technique of *recursion induction* to prove correctness of recursive procedures. Like every induction proof, such a proof consists of two steps.
The basic step establishes the desired result for the case that the call does not generate inner calls, i.e., when only the *basic part* of the procedure body is executed. Thus the basic step establishes that all calls, which are leaves in a calling tree, are correct.
In the induction step we prove that a call meets a specification in case the *re-*

cursion part of the body is executed, i.e., when the call generates inner recursive calls. In the induction step we can use the induction hypothesis, i.e., the assumption that all inner calls meet the specification. Such an induction step is generally proven as follows. Suppose a call $P(t_1, t_2, \ldots, t_n)$, results into k inner calls $P(t_{11}, t_{12}, \ldots, t_{1n})$, ..., $P(t_{k1}, t_{k2}, \ldots, t_{kn})$. Suppose furthermore that the precondition $p(t_1, t_2, \ldots, t_n)$ holds before the outer call, that is before the body of P is executed. We then prove that before the first call $P(t_{11}, t_{12}, \ldots, t_{1n})$ the precondition $p(t_{11}, t_{12}, \ldots, t_{1n})$ holds. Using the induction hypothesis we can infer that if this inner call terminates, that afterwards the postcondition holds $q(t_{11}, t_{12}, \ldots, t_{1n})$. Using this result we then prove that before the second call $p(t_{21}, t_{22}, \ldots, t_{2n})$ holds. The induction hypothesis gives us that after the second call we have $q(t_{21}, t_{22}, \ldots, t_{2n})$. This trick is repeated and finally we have that after the last inner call $q(t_{k1}, t_{k2}, \ldots, t_{kn})$ holds. If we are able to derive from this fact that at the end of the procedure body the postcondition $q(t_1, t_2, \ldots, t_n)$ holds, we have completed our induction step. Summarizing, given the assumption that at the beginning of the body $p(t_1, t_2, \ldots, t_n)$ holds and that after each inner call the postcondition holds, our task is to prove that before each inner call the precondition holds and that at the end of the body the postcondition $q(t_1, t_2, \ldots, t_n)$ is true.
A proof by recursion induction is essentially a proof on the depth of the calling tree. If the depth is equal to 0, we derive correctness by the basic step of the induction. Correctness of calls generating a tree with depth $d + 1$ given the fact that calls with depth d are correct, can be derived by the induction step.
Notice that as a side effect of a proof by recursion induction of the fact that a procedure P is correct with respect to the precondition p and postcondition q, we have that before each nested call in the calling tree p holds and that after each such call q holds. Moreover, if the induction step establishes that at a certain place in the procedure body an assertion p is true (e.g. a loop invariant), then the same will be true for each inner call occurring in the calling tree. We will often make use of such results in the sequel of our paper.

Now we give a list of all definitions, use while discussing the properties of algorithms:

Definition 1.1 *For each node n the following quantities are defined (we assume* $\max(\emptyset) = -\infty$ *and* $\min(\emptyset) = \infty$*):*

$$\begin{array}{lcl}
ANC(n) & = & \{\ x \mid x \textit{ is a proper ancestor of } n\ \} \\
\bar{L}(n) & = & \max\{L(x) \mid x \in ANC(n) \textit{ and } x \textit{ a max node }\} \\
\bar{U}(n) & = & \min\{U(x) \mid x \in ANC(n) \textit{ and } x \textit{ a min node }\} \\
\hat{L}(n) & = & \max\{L(x) \mid x \in ANC(n)\} \\
\hat{U}(n) & = & \min\{U(x) \mid x \in ANC(n)\} \\
\hat{f}(n) & = & \max\{f(x) \mid x \in ANC(n)\}
\end{array}$$

A node x is called a left sibling of a given node n, if x is a child of a node m with $m \in ANC(n)$, and x is older than m', where m' denotes the child of m on the path from m to n.

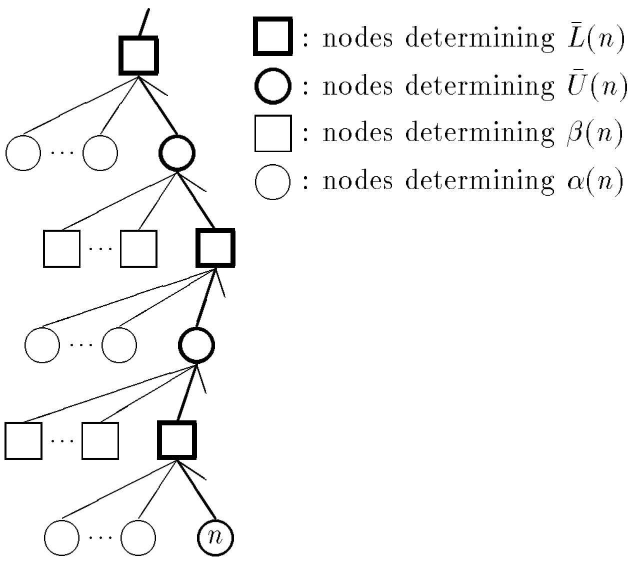
: nodes determining $\bar{L}(n)$
: nodes determining $\bar{U}(n)$
: nodes determining $\beta(n)$
: nodes determining $\alpha(n)$
n

Figure 2: Illustrating the definitions.

$$
\begin{aligned}
\alpha(n) &= \max\{f(x) \mid x \textit{ a left sibling of } n \textit{ and father}(x) \textit{ is max node}\} \\
\beta(n) &= \min\{f(x) \mid x \textit{ a left sibling of } n \textit{ and father}(x) \textit{ is min node}\} \\
\textit{Maxbroth}(n) &= \max\{f(n') \mid n' \textit{ an older brother of } n\}; \\
\textit{Minbroth}(n) &= \min\{f(n') \mid n' \textit{ an older brother of } n\}.
\end{aligned}
$$

These definitions are illustrated in Figure 2.

2 The alpha-beta algorithm

In this section we present the generalisation of the alpha-beta algorithm, in the sense that a heuristic pair is introduced.
The algorithm consists of one central recursive procedure, which is presented in Figure 3. For a given game tree with root r, the minimax value is computed by the call *alphabeta(r, alpha, beta, f)*, where *alpha* and *beta* are real numbers such that *alpha* $\leq f(r) \leq$ *beta.*

Specification of the procedure *alphabeta.*
The input parameters are n, a node in a game tree, and *alpha* and *beta*, two real numbers. There is one output parameter f, a real number.

$$
\begin{array}{lllll}
\textit{pre:} & \textit{alpha} < \textit{beta}, & & \\
\textit{post:} & \textit{alpha}<f<\textit{beta} & \Rightarrow & f = f(n), \\
 & f \leq \textit{alpha} & \Rightarrow & f(n) \leq f \leq \textit{alpha}, \\
 & f \geq \textit{beta} & \Rightarrow & f(n) \geq f \geq \textit{beta}.
\end{array}
$$

```
procedure alphabeta(in:  n, alpha, beta; out:  f);

basic part:
if alpha$\geq$U(n) or L(n)$\geq$beta or U(n)=L(n) then
   [ if L(n) $\geq$ beta then f:=L(n) else f:=U(n);
     exit procedure;
   ]

recursion part:
if type(n) = max then
   [ alpha':= max(alpha, L(n));
     for c := firstchild(n) to lastchild(n) do
          [ alphabeta(c, alpha', beta, f');
            if f'$>$alpha' then alpha':=f';
            if f'$\geq$ min(beta, U(n)) then exit for loop;
          ]
     f:=maximum of the intermediate f'-values;
   ]
if type(n)=min then
   [ beta':=min(beta, U(n));
     for c := firstchild(n) to lastchild(n) do
          [ alphabeta(c, alpha, beta', f');
            if f'$<$ beta' then beta':=f';
            if f' $\leq$ max(alpha, L(n)) then exit for loop;
          ]
     f:=minimum of the intermediate f'-values;
   ]
```

Figure 3: The procedure alphabeta.

It follows from the specification that, if we have the relation $alpha \leq f(n) \leq beta$ before the call *alphabeta*(n, *alpha*, *beta*, f), then the relation $f = f(n)$ holds on termination.

Theorem 2.1 *The procedure alphabeta meets the specification.*

Proof
The theorem is proven by recursion induction.
If the procedure halts by the exit statement in the basic part, we have three cases. Firstly we consider the case that the procedure halts, because the condition $alpha \geq U(n)$ is satisfied. Then it holds on termination that $f(n) \leq U(n) = f \leq alpha$. Secondly we have the case that $L(n) \geq beta$. Then it holds on termination that $f(n) \geq L(n) = f \geq beta$. In the third case, the equality $U(n) = L(n)$ holds. The procedure terminates with $f = U(n) = L(n)$ and thus $f = f(n)$. In all cases the specification is met. Notice that the third case does not exclude the two other cases.

Now, we consider the case that recursive calls are carried out. We suppose that n is a max node; (the proof in the alternate case is similar).

Before an iteration in which c is a parameter in a recursive call, the following three assertions hold:

$$\max(alpha, L(n)) \leq alpha' < \min(beta, U(n)) \tag{1}$$

$$\max(alpha, L(n)) = alpha' \Rightarrow Maxbroth(c) \leq max\{f'\} \leq alpha' \tag{2}$$

$$\max(alpha, L(n)) < alpha' \Rightarrow alpha' = \max\{f'\} = Maxbroth(c) \tag{3}$$

Notice that the value $\min(beta, U(n))$ is constant in the loop. Due to (1), the precondition of *alphabeta* holds before each iteration.

Before the first inner call, the first relation, (1), follows from the statements in the basic part. The second assertion, (2), holds trivially before the first iteration, since the values $Maxbroth(c)$ and $max\{f'\}$ are still equal to $-\infty$.
The third assertion, (3), holds, because the premiss in this assertion does not apply.
We will show that the three assertions remain valid after each next iteration.

If the inner call *alphabeta*(c, $alpha'$, $beta$, f') ends with $f' \leq alpha'$, we have, by the postcondition, that $f(c) \leq f'$. After this inner call, the three assertions remain valid.

If the inner call ends with $alpha' < f' < \min(beta, U(n))$, then it holds, due to the postcondition that $f' = f(c)$. The quantities $alpha'$, $\max\{f'\}$ and $Maxbroth(c)$ are updated and the assertions remain valid.

If the inner call ends with $f' \geq \min(beta, U(n))$, the loop is aborted.

Now, we show that the postcondition holds after termination of the procedure. The procedure halts, when the for loop terminates. There are two possibilities for the for loop to terminate.

Firstly, we suppose that the loop is aborted, due to the fact that a subcall *alphabeta*(c, *alpha'*, *beta*, f') ends with $f' \geq \min(\textit{beta}, U(n))$. Since $\min(\textit{beta}, U(n)) > \textit{alpha}'$ and thus $f' > \textit{alpha}'$, we have by the postcondition: $f(c) \geq f'$. The main procedure ends with $f = f'$, since all former calls have ended with $f' < \min(\textit{beta}, U(n))$. Since $f(n) \geq f(c)$ for max node n, we have the following inequality sequence: $f(n) \geq f(c) \geq f' = f$. If $f' \geq \textit{beta}$, it follows that $f(n) \geq f \geq \textit{beta}$. If $f' \geq U(n)$, it follows that $f(n) \geq f' \geq U(n)$ and, since $f(n) \leq U(n)$ in general, $f(n) = f$.

Secondly, we suppose that the inequality $f' \geq \min(\textit{beta}, U(n))$ never occurs. For this situation, we introduce a fictitious sentinel child $\bar{c}$, which is assumed to be younger than any proper child of n. Then (2) and (3) hold on termination of the for loop for $c = \bar{c}$ and the value $\textit{Maxbroth}(\bar{c})$ is equal to: $\max\{f(c) \mid c \in C(n)\} = f(n)$. Furthermore we have that $f = \max\{f'\}$.
If (2) holds, we have that $f(n) = \textit{Maxbroth}(\bar{c}) \leq \max\{f'\} = f \leq \textit{alpha}' = \max(\textit{alpha}, L(n))$. Since, by definition $L(n) \leq f(n)$, it follows that $L(n) \leq f(n) \leq f \leq L(n)$ and hence $f = f(n)$ or, $L(n) \leq f(n) \leq f \leq \textit{alpha}$.
If (3) holds, we have that $f = \max\{f'\} = \textit{Maxbroth}(\bar{c}) = f(n)$. □

Note
From our discussion in Section 1 on recursion induction, we can infer that the above proof has established that before each nested call *alphabeta*(n, *alpha*, *beta*,f), we have that the precondition of the specification holds, that after each call we have that the postcondition holds and that for each nested call the loop invariant (1), (2) and (3) applies.

3 The nodes, visited by alpha-beta

In this section we will give a characterisation of the nodes visited by the alpha-beta algorithm. The results in [Baudet] and [Pearl] will be generalised and presented in Theorem 3.2.

Theorem 3.1 *If a node n is parameter in a nested call alphabeta(n, alfa, beta,f) anywhere in the recursion, during the execution of the call alphabeta(r, $-\infty$, $+\infty$, f) with r the root of a game tree, then*

$$\textit{alpha} = \max(\alpha(n), \bar{L}(n)) \tag{4}$$

and

$$\textit{beta} = \min(\beta(n), \bar{U}(n)) \tag{5}$$

Proof
We give a proof by induction on the depth of n.

The theorem is true trivially if the depth of n is equal to 0, that is, n is the root. We assume that the theorem is true for any node n with depth d. We will prove that (4) and (5) hold for any child c of n. Then we know that the theorem also holds for all nodes with depth $d+1$.
We only consider the situation that n is a max node. (The alternate situation is similar). Then $\beta(n) = \beta(c)$ and $\bar{U}(n) = \bar{U}(c)$.
The parameters in the inner call are *alpha'* and *beta*. It follows from (2) and (3), that *alpha'* = max(*alpha, L(n), Maxbroth(c))*, and hence, *alpha'* = $\max(\alpha(c), \bar{L}(c))$.
The second equality, (5), holds trivially. □

Lemma 3.1 *For any n, $\hat{U}(n) = \min(\bar{U}(n), U(m))$ and $\hat{L}(n) = \max(\bar{L}(n), L(m))$, where m =father(n).*

Proof
Follows from the consistency property of the heuristic pair. □

Theorem 3.2 *Suppose the call alphabeta$(r, -\infty, +\infty, f)$ is executed where r denotes the root of a given game tree (G, H). A node n is parameter in a nested call alphabeta$(n$, alpha, beta, $f)$ if and only if*

$$\max(\alpha(n), \hat{L}(n)) < \min(\beta(n), \hat{U}(n)) \tag{6}$$

Proof
We give a proof by induction on the depth of n.
Again, the theorem is true trivially if the depth of n is equal to 0, that is, n is the root.
We assume that the theorem is true for any node n with depth d. We only consider the situation that n is a max node. (The alternate situation is similar). We can derive from (2) and (3) that, if c is visited

$$\max(\mathit{alpha}, \mathit{Maxbroth}(c), L(n)) < \min(\mathit{beta}, U(n)) \tag{7}$$

If n is visited and c is not visited, then $f(c') \geq \min(\mathit{beta}, U(n))$ for at least one older brother c' of c. If n is not visited then (6) does not hold and hence (7) does not hold either. It follows that c is visited if and only if (7) holds. By (4) and by Lemma 3.1, the left-hand side is equal to $\max(\alpha(c), \hat{L}(c))$. By (5) and by Lemma 3.1, the right-hand side of (7) can be rewritten as: $\min(\mathit{beta}, U(n)) = \min(\beta(n), \bar{U}(n), U(n)) = \min(\beta(n), \hat{U}(c)) = \min(\beta(c), \hat{U}(c))$. Hence (7) is equivalent to

$$\max(\alpha(c), \hat{L}(c)) < \min(\beta(c), \hat{U}(c))$$

□

Corollary 3.1 *Let $H_1 = (U_1, L_1)$ and $H_2 = (U_2, L_2)$ denote heuristic pairs on a tree G, such that $U_1(n) \leq U_2(n)$ and $L_1(n) \geq L_2(n)$ for any node n. Let S_1 and S_2 denote the set of nodes, that are visited during execution of the alphabeta procedure on G with H_1 and H_2 respectively. Then $S_1 \subseteq S_2$.*

Proof
Follows from Theorem 3.2. □

4 Solution trees

In the SSS-2 algorithm to be discussed in Section 5, the notion of a solution tree plays a central role. This notion was defined in [Stockman] in order to explain the working of the SSS* algorithm. The definition given now is more general, because heuristic functions are used.

Definition 4.1 *Given an informed game tree (G, H), a solution tree S is a subtree of G with the properties:*

- *for a max node n, either all children of n are included in S or no child is included;*
- *for a min node, either exactly one child is included in S, or no child is included.*

A node in S which has no children in S is called a tip node of S.

In figure 1 the bold edges generate a solution tree.
The set of all solution trees rooted in a node n is denoted by $\mathcal{M}(n)$. For a given game tree (G, H) the set of all max solution trees with the same root as G is denoted by $\mathcal{M}_G$.
If S is any solution tree and m is any node in S, then $S(m)$ denotes the subtree of S, rooted in m. For a node n in a solution tree, the minimax function $g(n)$ is defined as:

$$\begin{aligned} g(n) &= U(n), && \textit{if } n \textit{ is a tip node,} \\ &= \max\{g(c) \mid c \textit{ a child of } n\}, && \textit{if } n \textit{ is an inner max node,} \\ &= g(c) && \textit{if } n \textit{ is an inner min node and} \\ & && c \textit{ is the single child of } n. \end{aligned}$$

Similar to the minimax function f in a game tree G, we identify the minimax value $g(S)$ of a solution tree S with the minimax value of the root of S.

We give some Lemma's with respect to the minimax value of a solution tree.

Lemma 4.1 *Given an informed game tree (G, H), for every solution tree $S \in \mathcal{M}_G$, $g(S) \geq f(G)$.*

Proof
By induction on the height of S. □

Lemma 4.2 *For each informed game tree (G, H), there exists a solution tree S, with the same root as G, such that $g(S) = f(G)$.*

Proof
We give a construction of S. Firstly the root of G is included in S. Next, proceed with the construction recursively: append to each min node $n \in S$ that is a non terminal, a child with minimal f-value; append to each max node in S that is not a terminal, all its children.
In a terminal node n, it holds, by the definition of the g-function, that $g(n) =$

$U(n)$ and, by the definition of a terminal, that $U(n) = f(n) = L(n)$ and consequently, that $f(n) = g(n)$. It can be shown by induction on the height of n that $g(n) = f(n)$ for each node n in S. □

The solution tree constructed in the proof of Lemma 4.2 is a solution tree which contains a critical path of the entire game tree.

Theorem 4.1 *For each informed game tree (G, H), it holds that* $f(G) = \min\ \{g(S) \mid S \in \mathcal{M}_G\}$.

Proof
Follows immediately from Lemma's 4.1 and 4.2. □

In order to investigate the set of solution trees, we introduce the following linear ordering *'older'*, denoted by $\gg$, on this set. (In any non-terminal of a game tree a fixed order for the child nodes is assumed.)

Definition 4.2 *For two solution trees S and S' in $\mathcal{M}(n)$ with n a node in a game tree (G, H), the relation $\gg$ is defined recursively as follows:*

- *if n is a tip node in S and n is not a tip node in S', then $S \gg S'$;*
- *if n is a max node and n is not a tip node in S or in S', then consider the oldest child m of n, such that the subtrees $S(m)$ and $S'(m)$ are different (because $S \neq S'$, such a subtree must exist); if $S(m) \gg S'(m)$ then $S \gg S'$.*
- *if n is a min node and n is not a tip node in S or in S', then consider m and m', the children of n in S and S' respectively; we define $S \gg S'$, if either m is older than m' or $m = m'$ and $S(m) \gg S'(m)$.*

In the SSS-2 algorithm we pay special attention to those solution trees T in $\mathcal{M}(n)$ such that $g(T) < g(T')$ for any element $T' \in \mathcal{M}(n)$ with $T' \gg T$. A solution tree with this property is called a *milestone* in $\mathcal{M}(n)$. The following two Lemma's give a characterisation of a milestone.

Theorem 4.2 *For an informed game tree (G, H) and a node $n \in G$, a solution tree $S \in \mathcal{M}(n)$ is the oldest solution tree with g-value $\leq g_0$, if and only if one of the following statements holds:*

a) n is a tip node in S (i.e., S consists only of node n), and $U(n) \leq g_0$;

b) n has at least one child in S, $U(n) > g_0$ and $S(c)$ is the oldest solution tree in $\mathcal{M}(c)$ with g-value $\leq g_0$, for every child c of n in S; moreover, in case n is a min node, Minbroth(c) $> g_0$ for the single child c of n in S.

Proof
Proof of the *Only-if* part. Suppose that S is the oldest solution tree in $\mathcal{M}(n)$ with g-value $\leq g_0$.

Proof of a)

Suppose that n is a tip node. Since S consists of only node n, by definition, $g(S) = U(n)$. Since $g(S) \leq g_0$, $U(n) \leq g_0$.

Proof of b) (by contradiction)
Suppose that n has at least one child in S. If $U(n) \leq g_0$, then S is not the oldest solution tree with g-value $\leq g_0$, because the solution tree consisting of solely n is older and has g-value $\leq g_0$. Hence, the premiss is contradicted. We conclude that $U(n) > g_0$.

Assume that a child of n, say c_0, exists, such that $S(c_0)$ is not the oldest solution tree in $\mathcal{M}(c_0)$ with g-value $\leq g_0$. Let $S_1 \in \mathcal{M}(c_0)$ be a solution tree in $\mathcal{M}(c_0)$, such that $S_1 \gg S(c_0)$ and $g(S_1) \leq g_0$. We detach from S the subtree $S(c_0)$ and attach in c_0 the solution tree S_1. The transformed solution tree is an element of $\mathcal{M}(n)$ and is older than S and has g-value $\leq g_0$. Hence, S is not the oldest solution tree $\leq g_0$ in $\mathcal{M}(n)$, which contradicts the premiss. We conclude that the assumption is not correct.
Assume that, in case n is a min node, an older brother c_1 of c in G has the property that $f(c_1) \leq g_0$. It follows from Lemma 4.2 that a solution tree $S_1 \in \mathcal{M}(c_1)$ exists, such that $g(S_1) \leq g_0$. When S_1 is attached to n, a solution tree older than S is generated, which has a g-value $\leq g_0$. Hence, S is not the oldest solution tree $\leq g_0$ in $\mathcal{M}(n)$, which contradicts the premiss. We conclude that the assumption is not correct.

If part.
Proof of a)
By the definition of the $\gg$-relation, the oldest solution tree in $\mathcal{M}(n)$ consists of only n. Since $U(n) \leq g_0$, it follows that $g(S) \leq g_0$, and consequently, S is the oldest solution tree with g-value $\leq g_0$.

Proof of b) (by contradiction)
Assume that a solution tree $S_1 \in \mathcal{M}(n)$ exists, such that $g(S_1) \leq g_0$ and $S_1 \gg S$. Since $U(n) > g_0$, S_1 does not consist of only n as a tip node. We consider two cases.
Firstly, we assume that each child of n in S is also a child of n in S'. Notice that this condition holds trivially, if n is a max node.
Let c_0 be the oldest child, such that $S_1(c_0) \gg S(c_0)$; (such a child exists; otherwise, by the definition of the $\gg$ relation, S_1 is not older than S). Since $g(S_1(n)) \leq g_0$, it holds that $g(S_1(c_0)) \leq g(S_1(n)) \leq g_0$; in case n is a min node, the equality $g(S_1(c_0)) = g(S_1(n))$ holds. Hence, $S_1(c_0)$ is a solution tree $\gg S(c_0)$ in $\mathcal{M}(c_0)$ with g-value $\leq g_0$, which contradicts the premiss in part b).
Secondly, we assume that n is a min node, which has different children in S and S_1. The single child of n in S and S_1 respectively is called c and c_1. In that case c_1 is older than c. It follows that $f(c_1) \leq g(S_1(c_1)) = g(S_1) \leq g_0$, which contradicts the premiss in part b). $\square$

Theorem 4.3 *For an informed game tree (G, H), a node $n \in G$, a solution tree $S \in \mathcal{M}(n)$ is the oldest solution tree with g-value $< g_0$, if and only if one of the* *following statements holds:*

a) n is a tip node in S (i.e., S consists only of node n), and $U(n) < g_0$;

b) n has at least one child in S, $U(n) \geq g_0$ and $S(c)$ is the oldest solution tree in $\mathcal{M}(c)$ with g-value $< g_0$, for every child c of n in S; moreover, in case n is a min node, Minbroth$(c) \geq g_0$ for the single child c of n in S.

Proof
Similar to Theorem 4.2. □

5 The SSS-2 algorithm

Due to Theorem 4.1, the minimax value $f(G)$ of a game tree (G, H) is equal to the smallest g-value of the trees in the set $\mathcal{M}_G$. In this section we shall develop the SSS-2 algorithm, which computes $f(G)$ by determining the solution tree in $\mathcal{M}_G$ with the smallest g-value. First of all, SSS-2 constructs the oldest solution tree in $\mathcal{M}_G$. Then a loop is set up, in which in each iteration the next younger milestone is determined.
The algorithm will be built around two procedures, *diminish* and *expand*. Both procedures have an input parameter n, a node in the game tree, another input parameter g_1, a real number, and an output parameter g_2, a real number, which denotes the value of a solution tree. The *expand* procedure has a second output parameter, called S, which denotes a solution tree.
First we give the specification of each procedure and next we will show, how the procedures are embedded in the main program of SSS-2. We assume that, during the execution of SSS-2, a global variable, called T, contains a solution tree, rooted in the root of the game tree.

Specification of the procedure *diminish*(n, g_1, g_2).
The solution tree in the global variable T on call is denoted by T_1 and the solution tree on exit is denoted by T_2.

pre: $g(T_1(n)) = g_1$ *and* $T_1(n)$ *is the oldest solution tree in* $\mathcal{M}(n)$ *with* g*-value* $\leq g_1$.
post: $g_1 \geq g_2$ *and* $g(T_2(n)) = g_2$;
$g_1 > g_2 \Rightarrow T_2(n)$ *is the oldest solution tree in* $\mathcal{M}(n)$ *with* g*-value* $< g_1$;
$g_1 = g_2 \Rightarrow f(n) = g_1 = g_2$.

Specification of the procedure *expand*(n, g_1, g_2, S)

post: $g_1 > g_2 \Rightarrow g(S) = g_2$ *and*
S *is the oldest solution tree in* $\mathcal{M}(n)$ *with* g*-value* $< g_1$,
$g_1 \leq g_2 \Rightarrow f(n) \geq g_2 \geq g_1$ *(and S is undefined).*

Since $T_1(n)$ is the oldest solution tree with g-value $\leq g_1$, we conclude from the relation $g_1 \geq g_2$ on termination of *diminish*, that $T_1(n) = T_2(n)$ or $T_1(n) \gg T_2(n)$. Notice that it is possible on termination of *diminish*, that both $g_1 = g_2$ and

```
r := the root of the game tree;
expand(r, $\infty$, $g_2$, S);
T := S;
repeat
   [ $g_1$:= $g_2$;
     diminish(r, $g_1$, $g_2$);
   ]
until $g_1$= $g_2$;
```

Figure 4: The SSS-2 algorithm.

$T_1(n) \gg T_2(n)$ hold.
Briefly speaking, we can say that *diminish* looks for the next milestone in $\mathcal{M}(n)$ beyond T_1 and *expand* looks for the first milestone $S \in \mathcal{M}(n)$ such that $g(S) < g_1$. If $g_1 = g_2$ holds on termination of *diminish* or if $g_1 \leq g_2$ holds on termination of *expand*, then apparently a solution tree with g-value $< g_1$ does not exist, which implies that $f(n) \geq g_1$.

The code of *diminish* and *expand* can be found in Figure 5 and Figure 6 respectively. In the *diminish* code a procedure call *expandtip* features. This procedure is meant to have the same postcondition as *expand*. From the precondition of *diminish* and from inspection of the body of *diminish*, we can infer that *expandtip* has the precondition: $U(n) = g_1 > L(n)$. Therefore the body of *expandtip* can consist of only the recursion part of the *expand* body. We will not repeat this code for *expandtip*.
The code of the main program can be found in Figure 4. In the main program of SSS-2, the call *expand*$(r, +\infty, g_2, S)$ generates the oldest solution tree in $\mathcal{M}_G$ with finite g-value. In each iteration of the main loop, the next younger solution tree with a smaller g-value, if any, is constructed. If this construction fails, then apparently, the minimum value of the g-function in the set $\mathcal{M}_G$ has been obtained. The solution tree, constructed in each iteration, is stored into the global variable T.

Theorem 5.1 *The execution of SSS-2 terminates for any game tree* (G, H) *and, on termination,* $g_1 = f(G)$*, provided that the procedures expand and diminish meet their specification.*

Proof
We can prove by induction on the height of n in the game tree, that each call of *diminish*$(n, \ldots)$ and *expand*$(n, \ldots)$ will terminate.
The algorithm terminates when the relation $g_1 = g_2$ applies. Otherwise a younger solution tree is generated. There exists a finite number of solution trees and hence, the number of iterations in the main loop is finite.
Using the specification of *expand* and *diminish* one can prove that before each call of *diminish* in the main loop its precondition holds. The equality $g_1 = f(G)$ follows from the postcondition of *diminish*. $\square$

```
procedure diminish(in:  n, $g_1$; out:  $g_2$);

*basic part:*
if L(n) $\geq$ $g_1$  then [ $g_2$:=$g_1$;
                      exit procedure;
                    ]
if n is a tip node in T then [ expandtip(n, $g_1$, $g_2$', S);
                               if $g_1$> $g_2$' then attach S to n in T;
                               $g_2$:=$g_2$';
                               exit procedure;
                             ]
*recursion part:*
if type(n)=max then
   [ for c:= firstchild(n) to lastchild(n) do
         [ if $g_1$=g(T(c)) then diminish(c, $g_1$, $g_2$');
           if $g_1$=$g_2$' then exit for loop;
         ]
     $g_2$:= the maximum of g-values of all children;
   ]
if type(n)=min then
   [ c := the single child of n in T;
     diminish (c, $g_1$, $g_2$);
     if $g_1$= $g_2$ then
        for b:=nextbrother(c) to lastbrother(c) do
            [ expand(b, $g_1$, $g_2$', S);
              if $g_1$> $g_2$' then
                 [ detach in T from n the subtree rooted in c
                          and attach S to n in T;
                   $g_2$:= $g_2$';
                   exit for loop;
                 ]
            ]
   ]
```

Figure 5: The procedure diminish

```
procedure expand(in:  n, $g_1$; out:  $g_2$, S);

basic part:
if U(n) $<$ $g_1$  or L(n) $\geq$ $g_1$  then
        [ if U(n) < $g_1$
             then S := the tree consisting only of node n;
          if U(n)< $g_1$  then $g_2$:= U(n) else $g_2$:=L(n);
          exit procedure;
        ]
recursion part:
if type(n)=max then
   [ for c := firstchild(n) to lastchild(n) do
         [ expand (c, $g_1$, $g_2$', S');
           if $g_2$'$\geq$ $g_1$  then
              [ $g_2$:= $g_2$';
                exit for loop;
              ]
         ]
     $g_2$:= max of all intermediate values of $g_2$';
     if $g_2$< $g_1$  then S := the tree composed by attaching
                          all intermediate values of S' to n;
   ]
if type(n)=min then
   [ $g_2$:=$g_1$;
     for c := firstchild(n) to lastchild(n) do
         [ expand (c, $g_1$, $g_2$', S');
           if $g_2$'< $g_1$  then
              [ S := tree with S' attached to n;
                $g_2$:= $g_2$';
                exit for loop;
              ]
         ]
   ]
```

Figure 6: The procedure expand

6 The correctness of the procedures

In this section we prove the correctness of the procedures *expand* and *diminish*.

Theorem 6.1 *The procedures expand and expandtip meet their specification.*

Proof
We only consider the procedure *expand*. The proof for *expandtip* is similar. We give a proof by recursion induction.
First, we consider the situation that the procedure terminates, due to the *exit* statement in the basic part. If $U(n) < g_1$, then $g_2 = U(n)$ and thus $g_2 < g_1$, and by part a) of Theorem 4.3, S is the oldest solution tree in $\mathcal{M}(n)$ with g-value $< g_1$.
If $L(n) \geq g_1$, then S is undefined and the procedure ends with $g_2 = L(n) \leq f(n)$ and $g_1 \leq L(n) = g_2$, which conforms to the specification.

Next, we consider the situation that recursive calls are carried out.
If n is a max node, then, due to the *exit* statement, the for loop may be aborted. The for loop is continued, as long as each subcall ends with $g_1 > g'_2$. In that case we have by Theorem 4.1 $f(c) \leq g'_2$. We can conclude that the for loop has the following invariant: *Maxbroth*$(c)) < g_1$. If the for loop is not aborted, then the main call ends with $g_1 > g_2$. By the postcondition, we have after each call *expand*(c, g_1, g'_2, S') that S' is the oldest solution tree with g-value $< g_1$. By Theorem 4.3, S is the oldest solution tree with g-value $< g_1$.
If the for loop is aborted, then for a child, say c_0, the call *expand*(c_0, g_1, g'_2, S') ends with $g_1 \leq g'_2$. The main call ends with $g_1 = g_2$. By the postcondition, we have that $f(c_0) \geq g'_2$ and thus $f(n) \geq f(c_0) \geq g'_2 \geq g_1 = g_2$.

If n is a min node, again the for loop may be aborted. The for loop is continued, as long as each subcall ends with $g_1 \leq g'_2$. In that case we have by the postcondition of *expand* that $f(c) \geq g'_2 \geq g_1$. We can conclude that the for loop has the following invariant: *Minbroth*$(c)) \geq g_1$.
If, for a child, say c_0, the call *expand*(c_0, g_1, g'_2, S_0) ends with $g_1 > g'_2$, then *Minbroth*$(c_0) \geq g_1$ and, by the postcondition, S_0 is the oldest solution tree in $\mathcal{M}(c_0)$ with g-value $< g_1$. The main call ends with $g_1 > g_2$. It follows from Theorem 4.3 that S is the oldest solution tree with g-value $< g_1$.
If the for loop is not aborted, then $f(c) \geq g_1$ for each child c and hence $f(n) \geq g_1$. The procedures ends with $g_1 = g_2$. This conforms to the postcondition. □

Theorem 6.2 *The procedure diminish meets the specification.*

Proof
We give a proof by recursion induction.
If in the basic part the inequality $L(n) \geq g_1$ holds, then the procedure ends with $g_2 = g_1$. The solution tree in the global variable T is unaffected. We have the

following sequence of (in)equalities:

$$\begin{array}{ll} f(n) \geq L(n) & \\ L(n) \geq g_1 & \\ g_1 = g(T(n)) & \textit{(by the precondition of diminish)} \\ g(T(n)) \geq f(n) & \textit{(by Lemma 4.1)} \\ g_2 = g_1 & \textit{(by assignment)} \end{array}$$

It follows that $f(n) = g_1 = g_2 = g(T(n))$, in accordance with the specification.

If n is a tip node, then $g_1 = g(T(n))$ by the precondition, and $g(T(n)) = U(n)$ by the definition of the g-function. By the former *if*-clause, $g_1 > L(n)$. Hence the precondition of *expandtip* is satisfied. The subsequent call *expandtip*(n, g_1, g_2', S) cannot end with $g_1 < g_2'$, because in that case we would have the contradiction $f(n) \geq g_2' > g_1 = U(n)$. Hence this call ends with $g_1 \geq g_2'$. If $g_1 = g_2'$, we have $f(n) \geq g_2 = g_1 = U(n)$ and hence $f(n) = g_2$, which matches the specification. If $g_1 > g_2'$, then, after attaching S to T, we have $T_2(n) = S$. Due to the specification of *expandtip*, it holds that $g(S) = g_2$ and hence, $g(T_2(n)) = g_2$; furthermore, it holds that S is the oldest solution tree in $\mathcal{M}(n)$ with g-value $< g_1$.

Now we consider the situation, that recursive calls are performed. We have for each child c of n that is a parameter in a subcall *diminish*$(c,\ g_1,\ \ldots)$ that $g(T(c)) = g_1$. Since, by the precondition, $T_1(n)$ is the oldest solution tree with g-value $\leq g_1$, it follows from by the *only-if* part of Theorem 4.2, that, for each child c of n in T_1, $T_1(c)$ is the oldest solution tree in $\mathcal{M}(c)$ with g-value $\leq g_1$. We conclude that the precondition is met for each subcall *diminish*$(c,\ g_1,\ \ldots)$.

Suppose that n is a max node. If each subcall *diminish*(c, g_1, g_2') ends with $g_1 > g_2'$, then the main call ends with $g_1 > g_2$. For each child c' of n that was not parameter in a subcall $g_1 > g(T_1(c')) = g(T_2(c'))$. By the precondition and the *only-if* part of Theorem 4.2, $T_1(c')$ is the oldest solution tree with g-value $\leq g_1$ and hence also the oldest solution tree with g-value $< g_1$. Since $T_2(c') = T_1(c')$, $T_2(c')$ is the oldest solution tree with g-value $< g_1$. By the postcondition of *diminish*, after for each subcall *diminish*(c, g_1, g_2') $T_2(c)$ is the oldest solution tree in $\mathcal{M}(c)$ with g-value $< g_1$. It follows from Theorem 4.3 that $T_2(n)$ is the oldest solution tree in $\mathcal{M}(n)$ with g-value $< g_1$. By assignment $g_2 = \max\{g(T(c))\}$ for all children c of n and hence $g_2 = g(T_2(n))$.
If for a child, say c_0, the call *diminish*(c_0, g_1, g_2') ends with $g_1 = g_2'$, then we have by the postcondition that $f(c_0) = g_1$. By Theorem 4.1 it holds that $g_1 = g(T_1(n))$ $\geq f(n)$ and, since n is a max node, $f(n) \geq f(c_0)$. We conclude that $f(n) = g_1$.

Suppose that n is a min node. As mentioned above, by the precondition, $T_1(n)$ is the oldest solution tree with g-value $\leq g_1$. It follows from Theorem 4.2 that *Minbroth*$(c) > g_1$ where c is the single child of n.
If the subcall *diminish*(c, g_1, g_2) ends with $g_1 = g_2$, then the subsequent for loop of *expand* is continued as long as each subcall *expand*$(b,\ g_1,\ g_2',\ S)$ ends with $g_1 \leq g_2'$. For these calls, we have by the postcondition of *diminish* that $f(c) = g_2$,

and by the postcondition of *expand* that $f(b) \geq g_2' \geq g_1$. It follows that the for loop has the following invariant: *Minbroth*$(b) = g_1$.
If the for loop is not aborted, we can consider a fictitious child $\bar{c}$, younger than all real children. Due to the above invariant, we have *Minbroth*$(\bar{c}) = g_1$. It follows that $f(n) = g_1$.
If the subcall *diminish*(c, g_1, g_2) ends with $g_1 > g_2$ or a subcall *expand*$(b,\ g_1,\ g_2',\ S)$ ends with $g_1 > g_2'$, then the main procedure ends with $g_1 > g_2$. Let c_0 be the single child of n in $T_2(n)$. It follows from the postcondition of *diminish* or *expand* respectively, that $g_2 = g(T_2(c_0))$. Since $g(T_2(c_0)) = g(T_2(n))$, we have $g_2 = g(T_2(n))$. We stated above that *Minbroth*$(c_0) \geq g_1$. By the postcondition of *diminish* and *expand* respectively, we have that $T_2(c_0)$ is the oldest solution tree in $\mathcal{M}(c_0)$ with g-value $< g_1$. We conclude from Theorem 4.3 that $T_2(n)$ is the oldest solution tree in $\mathcal{M}(n)$ with g-value $< g_1$. $\Box$

7 The nodes, visited by SSS-2

Now we give the theorems, expressing the necessary and sufficient condition respectively for nodes to be visited by the SSS-2 algorithm.

Theorem 7.1 *A node n is parameter in a call diminish(n, g_1, g_2) and $S = T(n)$ at the moment of this call, if and only if S is the oldest solution tree in $\mathcal{M}(n)$ with $g(S) = g_1$ and*

$$\min(\beta(n), \hat{U}(n)) > g_1 > \max(\alpha(n), \hat{L}(n)) \tag{8}$$

and

$$g_1 \geq \hat{f}(n) \tag{9}$$

Proof
Notice that, since each *diminish* call satisfies the precondition of its specification, we have for each call *diminish*(n, g_1, g_2) with $S = T(n)$, that $g(S) = g_1$. Hence, the first of the *only-if* assertions has already been proven.
For the remainder of the theorem, we give a proof by induction on the depth of node n in the game tree. The proof is divided into an *only-if* and an *if* part respectively. For a given node c with depth $d+1$, the father (with depth d) is denoted by n.

Only-if part
If n has depth equal to 0, i.e., n is the root, then we have a *diminish* call in the main program (see Figure 4), and for such calls, $-\infty < g_1 < \infty$. Hence the *only-if* part is correct for the root.
Assume that a node c with depth $d+1$ is parameter in a call *diminish*(c, g_1, g_2). Then n is parameter in a call *diminish*(n, g_1, g_2). Since the *basic part* is passed, n is not a tip node in $T(n)$ and therefore, by Theorem 4.2, $U(n) > g_1$. Again, since the *basic part* is passed, $L(n) < g_1$.
If n is a max node, then for each older brother c' of c, a milestone $S' \in \mathcal{M}(c')$ exists with $g(S') < g_1$, (see the proof of Theorem 6.2). It follows that $f(c') < g_1$

for every older brother c' of c, and hence *Maxbroth*$(c) < g_1$. If n is a min node, then *Minbroth*$(c) > g_1$, due to Theorem 4.2.
By the induction hypothesis, (8) and (9) hold for n. Since $g(T_1(n)) = g_1$, we have that $f(n) \leq g_1$ and hence (9) also holds for c. Due to the fact that $U(n) > g_1$, $L(n) < g_1$ and, *Maxbroth*$(c) < g_1$ or *Minbroth*$(c) > g_1$ in a max or min node respectively, (8) also holds for c.

If part
If (8) holds for n with n the root, then g_1 is finite. It follows from the premiss of the *if* part, that a milestone in $\mathcal{M}_{\mathcal{G}}$ exists with g-value equal to g_1. Then, in the main program, a call *diminish*(r, g_1, g_2), with r equal to the root, is executed.
Now we prove the induction step. Suppose (8) and (9) hold for c. Then they also hold for n. In order to use the induction hypothesis, we need the existence of milestone in $\mathcal{M}(n)$ with g-value $= g_1$.
Since $f(n) \leq \hat{f}(c) \leq g_1$, there exists a milestone S' in $\mathcal{M}(n)$ with $g(S') \leq g_1$. We will prove that S' the required one. In case n is a min node, *Minbroth*$(c) < g_1$ due to (8) for c. Due to the fact that S is the milestone in $\mathcal{M}(c)$ with g-value $= g_1$, we conclude from Theorem 4.2, in case both n is a max node or a min node, that c is included in S' and S is a subtree of S'. Furthermore, it follows that $g(S') = g_1$. Hence S' is a milestone with g-value equal to g_1.
By the induction hypothesis, n is parameter in a call *diminish*(n, g_1, g_2) and $S' = T(n)$ at the moment of the call. It follows from (8) that $U(n) > g_1$ and $L(n) < g_1$. Hence the basic part is passed. We will show that c is parameter in a subcall.
Suppose n is a max node. It follows from (8) that for all older brothers c' of c, $f(c') < g_1$ and consequently, there exists a solution tree with g-value $< g_1$. Hence the for loop is not aborted after an inner call with c' as parameter, c' an older brother of c. Since S is a subtree in $S' = T(n)$, we conclude that $S = T(c)$ and hence $g(T(c)) = g_1$. It follows that c is parameter in a *diminish* call.
Suppose that n is a min node. Since c is included in S' and $S' = T(n)$, c is parameter in a *diminish* call. □

Definition 7.1 *Let n be a node in a game tree, which is not the root. Let m be a node in ANC(n) such that $U(m) = \min(\beta(n), \hat{U}(n))$ or such that m is a min node and the father of a left sibling m' of n with $f(m') = \min(\beta(n), \hat{U}(n))$. The node closest to the root with this property is called the β-ancestor of n.*

The value $\min(\beta(n), \hat{U}(n))$ is the minimum of the values $U(x)$ for x an ancestor of n and $f(x)$ for x a left sibling of n and x a child of a min node. Briefly speaking, we can say, that the β-ancestor m of n is the ancestor, in which or in whose children this minimum is achieved. A tie is solved in favour of the node closest to the root.

Theorem 7.2 *A node n is parameter in a call expand$(n,\ g_1,\ g_2, S)$, if and only if*
$$\min(\beta(n), \hat{U}(n)) = g_1 > \max(\alpha(n), \hat{L}(n)) \tag{10}$$

and

$$g_1 \geq \hat{f}(m), \quad m = \beta\text{-}ancestor(n) \tag{11}$$

Proof
We give a proof by induction on the depth of n.
If n has depth 0, then n is the root of the game tree. The root is parameter in an *expand* call, if and only if the call is executed in the main program of SSS-2. In that case $g_1 = +\infty$. It follows that the theorem holds for the root.
The induction step of the proof is divided into two separate parts.

Only-if part
Assume that c is parameter in a call *expand*(c, g_1, g_2, S). We distinguish three cases successively. The call *expand*(c, g_1, g_2, S) is a subcall in a call *expand*(n, g_1, g_2, S), or it is a subcall in a call *expandtip*(n, g_1, g_2, S) or this call is executed in the second for loop of the *diminish* body during the call *diminish*(n, g_1, g_2).

First, we discuss the case that n is parameter in an *expand* call. By the induction hypothesis, n satisfies (10). Since the basic part is passed in this call, $U(n) \geq g_1$ and $L(n) < g_1$. Since c is parameter in a recursive call, *Maxbroth*$(c) < g_1$ or *Minbroth*$(c) \geq g_1$, according to whether n is a max or min node respectively; (see the proof of Theorem 6.1). It follows that (10) holds for c. It also follows that n and c have the same β-ancestor. Since (11) holds for n, it also holds for c.

Second we discuss the case that n is parameter in a call *expandtip*(n, g_1, g_2, S). This call can only be executed in the basic part of a call *diminish*(n, g_1, g_2). By the precondition of *expandtip*, $U(n) = g_1 > L(n)$. For n as a parameter in a *diminish* call, (8) and (9) hold. If c is parameter in an inner *expand* call, then *Maxbroth*$(c) < g_1$ or *Minbroth*$(c) \geq g_1$, according to whether n is a max or min node respectively. Since $U(n) = g_1$, we conclude that n is the β-ancestor of c. It follows that (11) hold for c. Since $U(n) = g_1 > L(n)$ and *Maxbroth*$(c) < g_1$ or *Minbroth*$(c) \geq g_1$, according to whether n is a max or min node respectively, (10) for c follows from (8).

Third we have the case that the c is parameter in a call *expand*(c, g_1, g_2, S) in the second for loop of the *diminish* body. Then an older brother of c, say c_0, is also parameter in a *diminish* call, which ends with $g_1 = g_2$. By the postcondition, $f(c_0) = g_1$. Theorem 7.1 holds for c_0. All brothers c' of c between c_0 and c are parameter in an *expand* call which ends with $g_2 \geq g_1$. Therefore $f(c') \geq g_1$. We conclude that n is the β-ancestor of c. Since (9) holds for n, (11) holds for c. Since (8) holds for c_0 and since $f(c_0) = g_0$ and all brothers c' between c_0 and c satisfy $f(c') \leq g_1$, we conclude that (10) holds for c.

If part
Suppose that c satisfies (10) and (11). We distinguish two cases. Either the $=$ sign in (10) also holds for n or the $=$ sign in (10) must be changed for n into an $>$ sign. (Notice that a $<$ cannot hold for n.)
First, we assume that (10) also holds for n. Then n and c have the same β-

ancestor and (11) also holds for n. By the induction hypothesis, n is parameter in an *expand* call. Since (10) holds for c, $U(n) \geq g_1$ and $L(n) < g_1$ and thus the basic part in the *expand* call is passed. It follows from (10) for c that $\mathit{Maxbroth}(c) < g_1$ or $\mathit{Minbroth}(c) \geq g_1$, according to whether n is a max or a min node respectively. If the for loop was aborted before visiting c, we would have due to the postcondition of *expand*, that $f(c_1) \geq g_1$ or $f(c_1) < g_1$ for some older brother c_1 of c, according to whether n is a max or min node. We conclude that c is visited.

Second, we assume that a $>$ sign instead of an $=$ sign holds in (10) for n, i.e., $\min(\beta(n), \hat{U}(n)) > g_1 > \max(\alpha(n), \hat{L}(n))$. In that case, n is the β-ancestor of c. We distinguish two subcases, namely $U(n) = g_1$, or n is a min node and $f(c_0) = g_1$ for some c_0 older than c. (If several older brothers of c have game value equal to g_0, then choose as c_0 the oldest brother with this property).
If $U(n) = g_1$, the tree S' consisting of solely n is a milestone with g-value equal to g_1. By Theorem 7.1, n is parameter in a *diminish* call with $S' = T(n)$. It follows that n is a tip node in $T(n)$. Since (10) holds for c, $L(n) < g_1$ and consequently, the first *if* clause in the *diminish* body is passed and a call $\mathit{expandtip}(n, g_1, g_2)$ is executed. Again, since (10) holds for c, $\mathit{Maxbroth}(c) < g_1$ or $\mathit{Minbroth}(c) \geq g_1$, according to whether n is a max or a min node respectively. Similarly to the first case dealing with *expand*, we conclude that c is visited in the body of *expandtip*.
If n is a min node, $U(n) > g_1$ and $f(c_0) = g_1$, then a milestone $S' \in \mathcal{M}(n)$ with $g(S') = g_1$ is obtained, by appending to n the milestone in $\mathcal{M}(c_0)$ with g-value $= g_1$. By Theorem 7.1, n is parameter in a *diminish* call with $S' = T(n)$. Since $f(c_0) = g_1$ and $f(c') \geq g_1$ for all children c' between c_0 and c, c is parameter in an *expand* call. □

Theorem 7.3 *Let S_1 denote the set of nodes visited by the global alpha-beta algorithm applied to a game tree with heuristic pair $H_1 = (U_1, L_1)$. Let S_2 denote the set of nodes visited by the SSS-2 algorithm applied to a game tree with heuristic pair $H_2 = (U_2, L_2)$. Then $S_2 \subseteq S_1$, if for every node n, $U_2(n) \leq U_1(n)$ and $L_2(n) \geq L_1(n)$.*

Proof
Follows from Theorems 3.2 and 7.2. □

Notice that SSS-2 surpasses alpha-beta not only in the set of terminals, but in the set of nodes, regardless whether a node is a terminal or an internal node. Hence we have for SSS-2 a stronger result than for SSS*, which surpasses alpha-beta only in the set of terminals, visited during execution [Pearl].

SSS-2 is not heuristicly monotone. This is illustrated by the example in Figure 7. In each node x in the tree, except in b, we assume that $U_1(x) = U_2(x)$. In the non-terminals y we assume that $L(y) = -\infty$. H_1 is more accurate than H_2. In case of the heuristic pair H_1, node b is the β-ancestor of n, whereas e is the β-ancestor in the alternate case. The $\hat{f}$-values for the β-ancestors are equal

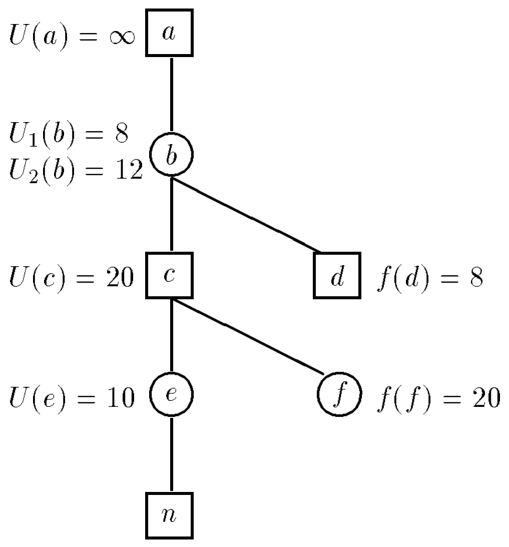

Figure 7: SSS-2 is not heuristicly monotone.

to 8 and 20 respectively. By Theorem 7.2, node n is visited in case H_1 applies, whereas it is not visited in case H_2 applies.

8 Notes on implementation

In this section, we discuss some ways to change the code of the SSS-2 algorithm, transforming it towards the code of the SSS* algorithm [Stockman]. We do not claim any mathematical rigour in this section.

First we will take a closer look at the call *diminish*(r, g_1, g_2) in the main program. This procedure call descends in T from the root along paths, such that $T(x) = g_1$ and $L(x) < g_1$ for each node x in the path. Notice that for each call there can be more than one path, due to the fact that a max node can have more than one child with g-value equal to g_1.
The following observation is relevant. If $L(x) = -\infty$ for all non-terminals x, then the condition in the first *if*-clause is equivalent to the condition: *if n is terminal.* This is argued as follows. By the precondition $g(T(n) = g_1$. If n is a terminal node, then n is a tip node and hence, by definition, $U(n) = g(T(n))$. Furthermore, $U(n) = L(n)$. It follows that $L(n) = g_1$. Conversely, if n is not a terminal, the condition $L(n) \geq g_1$ cannot be satisfied.
Therefore, the aforementioned procedure call descends in T from the root along paths with $T(n) = g_1$ up to tip nodes of T that are non-terminals, or to terminals of the game tree.
For each path, we can distinguish two cases. The path may end at a tip node n that is not a terminal. This node n is subject to a call *expandtip*(n, g_1, g_2', S). In the first case we assume that $g_2' < g_1$ and S is appended to n. In the second case

```
procedure diminish($g_1$,$g_2$);

lower:=true
while lower and a tip node (or terminal) t$\in$T satisfies U(t)=$g_1$ do
   [ m:=t;
     lower:=false;
     if t is a non-terminal then [ expandtip(t,$g_1$,$g_2$,S);
                                  if $g_1 >$ $g_2$then [ lower:=true;
                                                  remove t from T;
                                                  insert S;
                                                ]
                                ]
     while not lower and m $\neq$ root do
        [ m':=m;
          m:=father(m);
          if type(m)=min then
             for b:=nextbrother(m') to lastbrother(m') do
                 [ expand(b, $g_1$, $g_2$', S');
                   if $g_1$>$g_2$' then
                      [ remove all terminals of m' from T;
                        insert S' ;
                        lower:=true;
                        exit inner while loop;
                      ]
                 ]
        ]
   ]

if lower then $g_2$:=maximal g-value in the set of terminals
         else $g_2$:=$g_1$;
```

Figure 8: The new code for the procedure *diminish*.

we assume that this call terminates with $g_1 = g_2$ or that the end of the path is a terminal. In the second case we continue the procedure by expanding younger brothers of nodes c in T which are children of a min node in the path.

From now, we assume that $L(n) = -\infty$ for any non-terminal n in the game tree. Then any solution tree can be represented by a list of tip nodes and terminals in the solution tree, ordered according to their position in the solution tree, from left to right. Descending from the root up to a terminal or to a tip node that is a non-terminal, along a path with $g(T(n)) = g_1$ for every node n in the path is equivalent to selecting a terminal or tip node t with $g(t) = g_1$.
In the second case described above, 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, if the father is a min node. Finally, the g-value of the new milestone in T is determined as the maximum of the g-values of the terminals.

The new code can be found in Figure 8. The boolean *lower* indicates that a subtree containing a terminal with maximal value, has been replaced by another subtree containing solely terminals with lower value.
The procedures *expand* and *expandtip* can be adapted to the new representation of the solution trees in a straightforward fashion.

In the original *diminish*, applied to a game tree with $L(n) = -\infty$ for all non-terminals, the second if-clause is executed only if n is not a terminal. Assume that $U(n) = \infty$ for all non-terminals n. Then all tip node are terminals and consequently, the second *if*-clause is never executed. It follows that in the transformed code, the statement starting with the clause: *if tip node* ... can be deleted.

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.

References

[Baudet] G. M. Baudet, *On the branching factor of the alpha-beta pruning algorithm.* Artificial Intelligence 10 (1978), pp 173-199.

[Hoare] C.A.R. Hoare, An axiomatic basis for computer programming, Communications of the ACM, 12 (1969), pp 576-580.

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

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

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

[Pijls-1] W. Pijls and A. de Bruin, *Another view on the SSS* algorithm,* in: T. Asano, T. Ibaraki, H. Imai, T. Nishizeki (Eds.), Algorithms, International Symposium SIGAL '90, Tokyo, Japan, August 1990, Lecture Notes in Computer Science 450.

[Pijls-2] W. Pijls, *Shortest paths and game trees,* Ph.D.thesis, Erasmus University Rotterdam, 1991.

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