Solution Trees as a Basis for Game Tree Search

*Arie de Bruin, Wim Pijls, Aske Plaat**
Erasmus University, Department of Computer Science
P.O.Box 1738, 3000 DR Rotterdam, The Netherlands
plaat@theory.lcs.mit.edu

Abstract

A game tree algorithm is an algorithm computing the minimax value of the root of a game tree. Two well-known game tree search algorithms are alpha-beta and SSS*. We show a relation between these two algorithms, that are commonly regarded as being quite different.

Many algorithms use the notion of establishing proofs that the game value lies above or below some boundary value. We show that this amounts to the construction of a solution tree. We discuss the role of solution trees and critical trees [KM75] in the following algorithms: alpha-beta, PVS, SSS-2 and Proof number Search. A general procedure for the construction of a solution tree, based on alpha-beta and Null-Window-Search, is given.
Keywords: Game tree search, alpha-beta, SSS*, solution trees. proof-number search.

1 Introduction

In the field of game tree search the alpha-beta algorithm has been in use since the 1950's. It has proven quite successful, mainly due to the good results that have been achieved by programs that use it. No other algorithm has achieved the wide-spread use in practical applications that alpha-beta has. This does not mean that alpha-beta is the only algorithm for game tree search. Over the years a number of alternatives have been published. Among these are algorithms like PVS [CM83, Pea84], Proof-Number Search [AvdMvdH94], Best-First Minimax Search [Kor93], and SSS* [Sto79]. The last one, SSS*, has sparked quite some research activity. This may have been caused in part by the slightly provocative nature of the title of Stockman's original paper: "A Minimax Algorithm Better than Alpha-Beta?". This title alone has provoked a few reactions in the form of papers by Roizen and Pearl ("Yes and No" [RP83]), and Reinefeld ("A Minimax Algorithm Faster than Alpha-Beta" [Rei94]).
In the present paper we investigate the relation between alpha-beta, PVS, and SSS*. We confine ourselves to the basic algorithms, without enhancements like move-reordering, iterative deepening, or transposition tables (see e.g. [CM83, Sch89, ACH90]).

*Tinbergen Institute, Erasmus University, and Department of Computer Science, Erasmus University.

Alpha-beta, being a strictly depth-first algorithm, is generally regarded to be quite different in nature from best-first algorithms like SSS*. We will try to show in this paper how these algorithms are related. It turns out that SSS-2, an algorithm equivalent to SSS*, can be considered as a sequence of alpha-beta calls with narrow window.
At the center of our approach are solution trees—a notion that has been used in [Sto79] to prove the correctness of SSS*. By delving deeper into the nature of solution trees, and realizing that alpha-beta and PVS/NWS construct such trees as well, we have come to view solution trees as a unifying basis for alpha-beta and SSS*-like algorithms.

Preliminary Remarks

We assume that the reader is familiar with notions such as minimax, game tree (see e.g. [PdB93]). In this paper we assume a game tree to remain of fixed depth during the search for the best move, in the sense that we do not consider possible search extensions of the game tree. (See [Sch89] for an overview.)
In our figures, squares represent max nodes, circles min nodes. For a game tree G with root r, the minimax or game value of a node n is denoted by $f(n)$; the value $f(r)$ is also called the minimax value of G, denoted by $f(G)$. In this paper we will not apply *negamax*-like formulations: values of nodes will be conform the *minimax* rule, i.e., as seen by player MAX.
The minimax rule is based on the idea, that MAX tries to maximize and MIN tries to minimize the profit of MAX. Therefore, optimal play will proceed along a *critical path* (or Principal Variation), which is defined as a path from the root to a leaf such that $f(n)$ has the same value for all nodes n on the path. A node on a critical path is called *critical*.

Overview

We conclude this introduction with an outline of the rest of this paper. In section 2 we will show that in order to get a bound on the minimax value of a game tree, one has to construct a solution tree. A solution tree defining an upper bound is called a max solution tree. Likewise, a min solution tree defines a lower bound. In order to prove subsequently that the game value *equals* a certain value, say f, it is sufficient to find an upper bound *and* a lower bound with value f. In other words, a max and a min solution tree with this value are needed. The union of two such trees is called a critical tree. In section 3 we investigate the notion search tree, i.e., the subtree that consists of all nodes generated at a certain moment during execution of a game tree algorithm. ¿From a search tree, an upper and a lower bound on the game value of a node can be derived. We will investigate the relation between these bounds and the solutions trees embedded in the search tree.
In section 4 we will investigate how alpha-beta constructs a solution tree or, if the game value is determined exactly, a critical tree. In section 5 we show, how the working of PVS [FF80, Pea84] can be explained in terms of solution trees. Here, we use the fact that Null-Window-Search (=alpha-beta with a null window) generates solution trees. In section 6, the SSS*-variant SSS-2 [PdB92], which

also uses solution trees, is discussed. We show that SSS-2 consists of a number of alpha-beta calls, each with a null window. Finally, in section 7, we explain Proof-number search [AvdMvdH94] in terms of solution trees. So, viewing game tree search in terms of solution trees enables us to discover relations between two algorithms which where hitherto considered to be quite unrelated, viz. alphabeta, PVS and SSS* [Sto79, PdB90].

2 Solution Trees and Bounds

Solution trees occur in game tree literature mostly in relation to SSS* [Sto79, Kuma83]. In this section we will show that there exists a relation between solution trees and bounds on the game value. Further, we use solution trees to define the notion of *minimal* or *critical tree*.

Given a game tree, it generally takes a lot of effort to compute $f(n)$ for a node n. However, establishing an upper or a lower bound to $f(n)$ is a simpler task, as we will show. In a max node an upper bound is obtained, if an upper bound to each of the children is available. In that case the maximum of the children's bounds yields a bound to the father. If at least one child of a min node has an upper bound, this can also act as a father's upper bound. The above rules can be applied recursively. In a terminal, the game value is a trivial upper and lower bound. So, we need a subtree, rooted in n, of a particular shape. This subtree is constructed top-down, choosing all children in a max node and exactly one child in a min node. Such a subtree of a game tree is a called a max solution tree. Likewise, a min solution tree can be constructed to achieve a lower bound. We have the following formal definitions:

A max solution tree T^+ is a subtree of game tree G with the properties:

- *if an inner max node $n \in G$ is included in T^+, then all children of n are included in T^+;*
- *if an inner min node $n \in G$ is included in T^+, then exactly one child is included in T^+.*

A min solution tree T^- is a subtree of G with the properties:

- *if an inner min node $n \in G$ is included in T^-, then all children of n are included in T^-.*
- *if an inner max node $n \in G$ is include in T^-, then exactly one child is included in T^-.*

Notice, that every leaf of a solution tree is also a leaf in the game tree under consideration. However, the root of a solution tree is not necessarily the root of the game tree.
Given a max solution tree T^+, we compute an upper bound to $f(n)$ by applying bottom-up in T^+ the aforementioned rules. In fact, we apply the minimax function to T^+. It is easily seen that determining the minimax value of a node n in a max solution T^+ amounts to determining the maximum of the values $f(p)$ for all

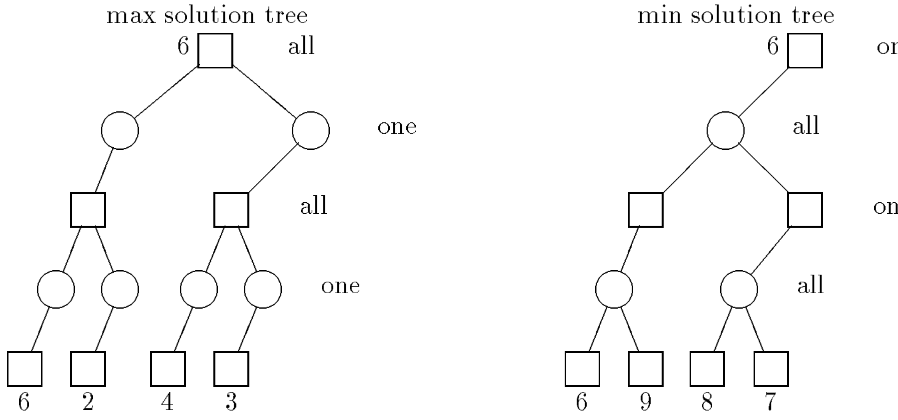

Figure 1: Solution Trees

terminals p in T that are descendants of n (see figure 1). Of course, analogous statements hold for a min solution tree.
The minimax function, restricted to a max or min solution tree T, is denoted by g. Analogous to $f(T)$, $g(T)$ denotes the g-value in the root of T. So, in a max solution tree T with root n, the fact that a max solution tree yields an upper bound, is expressed by the formula $g(n) \geq f(n)$ or alternatively, $g(T) \geq f(n)$. For a min solution tree T with root n, we may write $g(n) \leq f(n)$ or $g(T) \leq f(n)$.

Optimal Solution Trees

Having proved that a max solution tree delivers an upper bound, we now show that, in any game tree G, at least one solution tree has the same minimax value as G has. For instance, when a max solution tree T with the same root as the game tree is constructed, such that in every min node a child with the same f-value as the father is chosen, we have $g(n) = f(n)$ in every $n \in T$. (It can be shown that, in order to achieve at the root a g-value equal to the f-value, other construction methods are available as well.) Since we know that $g(T) \geq f(T)$ for any max solution tree T, we come to the following proposition. We also state its counterpart for min solution trees, which was given already by Stockman [Sto79].

> *Let a game tree G with root n be given. Then, the minimum of all values $g(T)$ with T a max solution tree rooted in n, is equal to $f(G)$. The maximum of all values $g(T)$, T a min solution tree rooted in n, is equal to $f(G)$.*

This statement will be referred to as *Stockman's theorem.* A solution tree with g-value equal to the game value is called an optimal or critical solution tree.

A max solution tree is a subtree, where in each node MIN's choice is known. Therefore, a max solution tree is also called a strategy of the MIN player, cf.

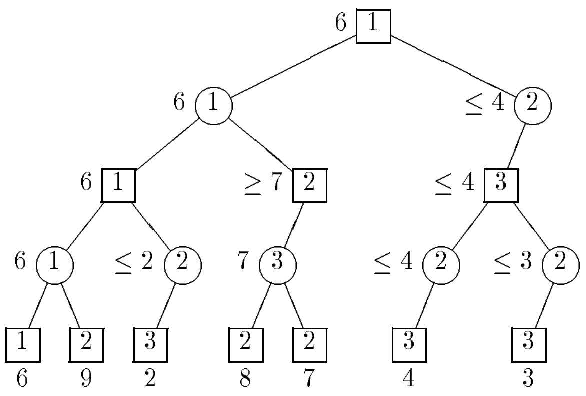

Figure 2: A Critical Tree with node types, values (f), and bounds (f^+, f^-)

[Kuma83]. Given a strategy of MIN or a max solution tree T, the pay-off attainable for MAX is equal to the greatest game value in the terminals of T. By definition, this value is equal to $g(T)$. Stockman's theorem actually states that the game value $f(G)$, the guaranteed pay-off for MAX, is equal to the smallest value among the values of MIN strategies. Of course this statement has a dual counterpart.

Critical Tree

The union of an optimal max and an optimal min solution tree is called a *critical* tree. Let $n_0, n_1, \ldots n_k$ be the nodes on the intersection path of T^+ and T^- where n_0 denotes the root. In T^+ we have:

$$f(n_i) \leq g(n_i) \leq g(n_0) = g(T^+), \quad i = 0, 1, \ldots k,$$

and in T^- we have:

$$f(n_i) \geq g(n_i) \geq g(n_0) = g(T^-), \quad i = 0, 1, \ldots k.$$

Since $g(T^+) = g(T^-) = f(G)$, also $f(n_i) = f(G)$ for $i = 0, 1, \ldots k$. We conclude that the intersection of T^+ and T^- is a critical path.
Figure 2 is an example of a critical tree. In [KM75] the notion critical tree is introduced as a minimal tree that has to be searched by alpha-beta in order to find the minimax value in a best first game tree. The numbers to the left of the nodes indicate what can be deduced from the tree about the game value of a node. The numbers inside the nodes represent Knuth & Moore's well known node types. For reasons of brevity we will not repeat their (quite complicated) definition of nodes types. Given our solution tree view of the critical tree, Knuth & Moore's type 1, type 2 and type 3 nodes can be given another interpretation. Type 1 nodes are in the intersection of the optimal solution trees for the player and its opponent—the critical path. Type 2 nodes are either min nodes of the max solution tree or max nodes of the min solution tree that are not in the critical

path. Type 3 nodes are max nodes in the max solution tree or min nodes in the min solution tree, that are not on the critical path.

3 Search Trees

In all game tree algorithms, the game tree is explored step by step. So, at each moment during execution of a game tree algorithm, a subtree has been visited. This subtree of the game tree is called a *search tree* [Iba86]. We assume that, as soon as at least one child of a node n is generated or visited, all other children of n are also added to the search tree. So, a search tree S has the property, that for every node $n \in S$ either all children are included in S or none.

On a search tree, we want to apply the minimax function tentatively. To that end, we define values in the *leaves* of S. We distinguish between so-called *open* and *closed* leaves in a search tree. A leaf that is not a terminal in the game tree, is always called open. A terminal is called closed or open, according to whether its final game value has been computed or not. Only values that surely are bounds, are chosen as tentative values for leaves in a search tree. This leads to two game trees derived from S, called S^+ and S^-, with game values f^+ and f^- respectively. We define $f^+(p) = +\infty$ and $f^-(p) = -\infty$ in every open leaf node p and $f^+(p) = f^-(p) = f(p)$ in every closed node. (Recall that the game value $f(p)$ is known in a closed node p.) In every node n of a search tree, we have $f^+(n) \geq f(n)$ and $f^-(n) \leq f(n)$, because this relation also holds in all children, if any, of n. (In the leaves of S, the relation holds trivially.)

Now, we are going to show that there exists a relation between solution trees in a game tree G and the values $f^+(n)$ and $f^-(n)$ of a node n in a search tree S of G.

Since a search tree S with minimax value f^+ or f^- can be viewed as a game tree, Stockman's theorem can be invoked. This tells us that $f^+(n)$ is equal to the minimum of all values $g(T)$ with g computed in S^+, T a max solution tree in S. Two kinds of solution trees are distinguished in S. A solution tree with at least one open leaf in S is called *open*, a solution tree with just closed terminals as leaves is called *closed*. A closed solution tree in S is also a solution tree in the entire game tree. We immediately see that $g(T)$ has a finite value, if and only if T is closed. Therefore, when applying Stockman's theorem to S, we only need to take closed max solution trees into account. We conclude that $f^+(n)$ is equal to the minimum of all values $g(T)$, T a max solution tree in G with root n and T is closed (without open terminals) in S. An analogous statement can be given for $f^-(n)$. There are no tighter bounds for $f(n)$, since it can be shown [PdB94] that every value between $f^+(n)$ and $f^-(n)$ can be made equal to the game value by constructing an appropriate extension of the search tree.

Almost every game algorithm builds a search tree and stops when $f^+(r) = f^-(r)$ and hence, $f^+(r) = f^-(r) = f(r)$, with r the root of the game tree. Due to the relationship, proved in the previous paragraph, between solution trees and f^+ and f^-, we may say that, on termination, a max and a min solution tree are obtained, which are optimal, i.e., $g(T^+) = g(T^-) = f(G)$. In the previous section, such a union has been called a *critical* tree. We conclude that the stop

```
procedure alpha-beta(n, α, β, v);
    if terminal (n) then  v := f(n);
    else if max(n) then
        v := −∞;
        c := firstchild(n);
        while v < β and c ≠ ⊥ do
            alpha-beta(c, α, β, v');
            v := max(v, v');
            α := max(α, v');
            c := nextbrother(c);
    else if min(n) then
        v := +∞;
        c := firstchild(n);
        while v > α and c ≠ ⊥ do
            alpha-beta(c, α, β, v');
            v := min(v, v')
            β := min(β, v');
            c := nextbrother(c);
```

Figure 3: Alpha-Beta

criterion is equivalent to the presence of a critical tree.

4 Alpha-Beta

The standard analysis of the alpha-beta algorithm is performed by Knuth & Moore in [KM75]. See figure 3 for the code of alpha-beta. The output value is stored into parameter v. In the present paper we will elaborate on Knuth & Moore's postcondition slightly by adding the notions $f^+(n)$ and $f^-(n)$, the notions that have been defined in the previous section. (This postcondition is derived from the one in [PdB94]. The accompanying precondition is $\alpha < \beta$.)

$$
\begin{aligned}
v \leq \alpha &\quad\Rightarrow\quad v = f^+(n) \geq f(n), v \text{ is the value of a max solution tree} && (1)\\
\alpha < v < \beta &\quad\Rightarrow\quad v = f(n), v \text{ is the value of a critical tree} && (2)\\
v \geq \beta &\quad\Rightarrow\quad v = f^-(n) \leq f(n), v \text{ is the value of a min solution tree} && (3)
\end{aligned}
$$

We will prove these implications, using induction on the height of node n. In this proof, we will analyse, how alpha-beta constructs solution trees.

(1). Every subcall to a child c has ended with $v'_c \leq \alpha$. By induction, we have that $v'_c = f^+(c)$ for every c. Since v, the output value, is equal to maximum of all value v'_c, the relation $v = f^+(n)$ holds. Using the theory of section 3, we conclude that n is the root of a max solution tree. We will analyse, how this solution tree is constructed. By induction, every c is the root of a max solution tree T_c with value $v'_c = g(T_c)$. Combining all trees T_c yields a new max solution

tree T^+.
(2). Notice that the *while* loop has only terminated, when $c = \perp$. Hence, every child has been parameter in a subcall, which has ended with $v' < \beta$. It follows that for every subcall, (1) or (2) applies. Therefore, for every c, $f^+(c) = v'_c$ and c is the root of at least a max solution tree T_c. Let c_0 be the child, whose output parameter v'_{c_0} performed the last update to v. So c_0 is the leftmost child, where $v' = v$ holds. Then it is easily seen that, for the subcall with parameter c_0, also (2) applies. By induction, we know that $f(c_0) = v'_{c_0}$ and c_0 is is the root of a critical tree. Appending this critical tree to n and appending all other max solution trees T_c to n yields a critical tree with root n. Since $f(c) \leq f^+(c) = v'_c \leq v_{c_0}$ for every c and $f(c_0) = v'_{c_0} = v$, we conclude $v = f(n)$.
(3). Let c_0 be the child that was parameter in the last subcall. Then the subcall with parameter c_0 has ended with $v'_{c_0} \geq \beta$. Every child c, older than c_0, has been parameter in a subcall ending with $v'_c < \beta$. It follows that $v = v'_{c_0}$. By induction, $v_{c_0} = f^-(c_0)$ and T_{c_0} is a min solution tree with value $v'_{c_0} = v$. This tree, appended to n is the desired solution tree. Since $v'_c < \beta \leq f^-(c_0) = v_{c_0} = v$ for every c older than c_0, and $f^-(c) = -\infty$ for every c younger than c_0, we conclude $f^-(c_0) = f^-(n) = v$.

In section 5 and 6, the alpha-beta procedure is invoked to construct solution trees, and to establish bounds to the game value.
In many implementations of algorithm, the cut-offs are recorded in the transposition table. So, the transposition table contains in a max node, where case (3), a beta-cut-off occurred, a pointer to the child, causing the cut-off. The proof of (3) actually shows that these pointers generate a min solution tree. Dually, the alpha cut-offs in min nodes give rise to a max solution tree. In case of (1) in a max node, the so called alpha bound is the same as $f^+(n)$ and is the g-value of a max solution tree.

5 PVS

PVS [FF80, CM83, Rei89] and the related algorithm SCOUT [Pea80, Pea84] are two well known algorithms based on the *minimal window search* [FF80] or *bound-test* [Pea80] idea.
PVS constructs a critical tree bottom up. At the start it descends via the left-most successors to the left-most leaf of the game tree. For the moment it is assumed that the path to this leaf, the principal leaf, is the critical path—the *Principal Variation* (PV) in PVS terms. (This is true for the tree in figure 2.) Suppose the value of this leaf is v. Then the assumption implies that the value of the root equals v. This assumption is then tested using a bounding procedure. If the parent of the leaf is a max node, then a proof must be established that no brother of the PV-node has a higher value. (If the parent of the leftmost leaf in the tree is a min node, then the dual procedure has to be performed.) In other words, for every brother a solution tree must be constructed yielding an upper bound on its value, which does not exceed v. If this succeeds we have built a critical tree rooted in the parent of the leaf at the end of the PV, proving that its game value is equal to v. If this is not possible, because some brother of the leaf

```
procedure NWS(n, γ, v);
    if terminal(n) then v := f(n);
    else if max(n) then
        v := −∞;
        c := firstchild(n);
        while v < γ and c ≠ ⊥ do
            NWS(c, γ, v′);
            v := max(v, v′);
            c := nextbrother(c);
    else if min(n) then
        v := +∞;
        c := firstchild(n);
        while v ≥ γ and c ≠ ⊥ do
            NWS(c, γ, v′);
            v := min(v, v′);
            c := nextbrother(c);
```

Figure 4: NWS

at the end of the PV has a higher value, the bounding procedure should prove this by generating a min solution tree defining a lower bound on the value of the brother that is higher than v, showing that the assumption is incorrect. In that case the path to this better brother then becomes the new PV-candidate. Since we have only a bound on its value, the game value of this PV-candidate must be found by re-searching the node.
Eventually the PV for the parent of the leftmost leaf is found. Its value is proven by the solution trees that bound the value of the brothers of the principal leaf. PVS has realized this by constructing a critical subtree for the current level of the game tree. It then backs up one level along the backbone, to start construction of a critical tree at a higher level, i.e., for the grandparent of the leftmost leaf in the game tree. This proceeds until the root has been reached and a critical tree below the root has finally been constructed.
The overall effect of this is that after termination, a critical path has been constructed consisting of the PV-nodes.

Assuming that leaf values are only integers, an upper bound $\leq \gamma$, γ a given constant, is achieved in PVS, by calling alpha-beta with a narrow window consisting of $\alpha = \gamma - 1$ and $\beta = \gamma$. (Dually, for a lower bound, a window with $\alpha = \gamma, \beta = \gamma + 1$ is used.) Such an alpha-beta call, which has one window parameter γ, will be referred to as *Null Windows Search* (NWS). For convenience, we will state the postcondition of NWS, which is the result of trivial substitutions in implications (1), (2), and (3).

$$\begin{aligned} &\alpha = \gamma - 1 \wedge \beta = \gamma \\ &\qquad v < \gamma \quad \Rightarrow \quad v = f^{+}(n) \geq f(n) \end{aligned} \tag{4}$$

$$v \geq \gamma \quad \Rightarrow \quad v = f^-(n) \leq f(n)$$

The code of NWS can be found in figure 4.

6 SSS-2

SSS-2 has been introduced in [PdB90] (cf. [Pij91, PdB92]) as an attempt to give an easier to understand, recursive description of SSS*. SSS-2 is equivalent to SSS*, in that the same *open* nodes (called *Live* nodes in the common SSS* description) are expanded in the same order. Here, expanding a node n means, that n gets closed in case n is a terminal, or its children are generated and added to the search tree. Bhattacharya & Bagchi have introduced another recursive version of SSS*, called RecSSS* [BB93]. However, their aim was different, viz. to obtain an efficient data structure implementing SSS*'s OPEN list.
In the current paper, we will present SSS-2 in yet another form. First, we present SSS-2 in a rudimentary form as a sequence of NWS calls. Next, we add a few enhancements to exploit in each call the information gathered in former calls. It turns out that SSS-2 in the original version [PdB90] is equivalent to the new version consisting of a sequence of NWS calls.

Our first version of the SSS-2 procedure is the following:

procedure SSS-2(n, v);
    $v := +\infty$;
    repeat
        $\gamma := v$;
        NWS(n, γ, v);
    until $v = \gamma$;

Similarly to alphabeta and NWS, n is an input parameter of type node and v is the output value. It follows from (4), that if $f(n) \leq \gamma$ on call, then on exit $f(n) \leq f^+(n) \leq v < \gamma$ or $f^-(n) = v = \gamma$. (The situation with $f^-(n) = v > \gamma$ cannot happen). The latter condition terminates the *repeat* loop. At the start of the first NWS call in SSS-2, we have $f(n) \leq \gamma = \infty$ trivially. We conclude that the *repeat* loop has the invariant $f(n) \leq \gamma$. The condition $f^-(n) = v = \gamma$ after a NWS call implies, in combination with the invariant $f(n) \leq \gamma$, that $v = f(n)$. Therefore, on termination of SSS-2, $v = f(n)$.

Like alpha-beta, an NWS call generates a search tree. Subsequent calls of NWS might use the information, contained in this tree, in order to avoid useless re-searches in nodes already examined. We saw earlier that for each NWS call in SSS-2 the output value v is the value of a max solution tree. embedded in the search tree. We will now argue that it is not necessary to preserve the full search tree between two NWS calls, but that this max solution tree suffices. When NWS is running, every nested call uses the same null window as the main call. Suppose such a nested call NWS(n, γ, v) ends with $v < \gamma$. If n is a max node, every child has been visited and every subcall has ended with $v' < \gamma$. If n is a min node, then n is the root of a min tree, containing the youngest (rightmost) child that was visited by a subcall. This subcall also ended with $v' < \gamma$. Every older child was visited by a subcall, which ended with $v' \geq \gamma$. Consequently, we can formulate a special property for the solution tree T^+, generated by a call NWS(n, γ, v): *in every min node m in T^+, we have for all older brothers b that a call NWS(b, γ, v), yields $v \geq \gamma$.* A formal proof is by induction on the height. We decide to store T^+ into a global variable T. The above special property of T is exploited in each next NWS call. Furthermore, we know that $g(T^+) = v = \gamma$ at the start of each call.

So we adapt our code of NWS(n, γ, v) for the calls inside the loop of SSS-2. The transformed code is called *T-NWS*. There are a few changes with respect to the original NWS code. In a max node, a call NWS(c, γ, v') to a child c that is already the root of max solution tree with g-value $< \gamma$, would return the same solution tree, as shown in [Bru94]. Therefore, in the new procedure, we don't visit children with $g(c) < \gamma$. Consequently, throughout the solution tree in T, only nodes with g-value $= \gamma$ are visited. In a min node n with single child c_0, each older brother c of c_0 has been parameter in a call *NWS*(c, γ, v') (with parameter γ larger than the current γ), which has ended with $v' \geq \gamma$. Since $g(n) = \gamma$, the call *NWS*(n, γ, v) will end with $v \leq \gamma$. Therefore, the older brothers of c_0 can be skipped in the *while* loop traversing the children of n. This statement is related to the observation in [Rein85, Wei92], that older brothers of a node generating a cut-off need not to be examined in subsequent alpha-beta calls. The children c of n younger than c_0 are still open and are subject to a regular NWS call.

If a new solution tree is delivered by a NWS call, the former and the new tree differ, in that the new tree has in a min node a younger child with smaller g-value than the former does in the same node. This phenomenon makes it possible to update the solution tree 'on the fly'.

The transformed code of NWS, called *T-NWS*, is shown in figure 5. Since the call T-NWS(c_0, γ, v) with c_0 the single child of a min node n ends with $v \leq \gamma$, the guard of the subsequent *while* loop contains $v = \gamma$ instead of $v \geq \gamma$, as was included in the orginal NWS code.

The *update* procedure consists of the following actions: a) the subtree with root c_0 (i.e. T_{c_0}) is detached from T, b) the solution tree produced by the most recent NWS call, which also aborts the *while* loop, is attached to T to replace T_{c_0}.

Since NWS is replaced by T-NWS, whenever parameter n is included in T, we also need new code of SSS-2. The new code is:

procedure SSS-2(n, v);
    $v := \infty$;
    NWS(n, γ, v);
    repeat
        $\gamma := v$;
        T-NWS(n, γ, v);
    until $v = \gamma$;

We assume that, after the first NWS call, the resulting solution tree is stored into the global variable T. The original SSS-2 description comprises two procedures,

```
procedure T-NWS(n, γ, v);
    if terminal(n) then v := f(n);
    else if max(n) then
        v := −∞;
        c := firstchild(n);
        while v < γ and c ≠ ⊥ do
            if g(c) = γ then T-NWS(c, γ, v′) else v′ := g(c) ;
            v := max(v, v′);
            c := nextbrother(c);
    else if min(n) then
        c0:=the single of n in T;
        T-NWS(c0, γ, v);
        c := nextbrother(c0);
        while v = γ and c ≠ ⊥ do
            NWS(c, γ, v′);
            if v < γ then update(T);
            v := min(v, v′);
            c := nextbrother(c);
```

Figure 5: T-NWS

diminish and *expand*. Apart from identifiers, their code is completely the same as the current procedures T-NWS and NWS respectively.

Narrower Window searches Surpass Wider Window searches
It is a well known feature that the narrower the α-β-window, the smaller the number of generated nodes [CM83, Pea84]. Therefore, NWS (= alpha-beta with a null-window) surpasses the alpha-beta algorithm. Since T-NWS expands less new nodes than NWS, T-NWS and also every sequence of T-NWS calls surpasses alpha-beta. Here, surpassing is used in the sense of Stockmann's paper on SSS* [Sto79], where the set of nodes, expanded at least once, is considered. Re-expanding or revisiting actions on such nodes are not taken into account. Since SSS-2 consists of a number of T-NWS calls, this algorithm surpasses alpha-beta. Here, we rediscover a result in [PdB90] extending a weaker result in [Sto79].

7 Proof-number search

Proof-number search is a new algorithm, published only recently in [AvdMvdH94]. A somewhat more elaborated formal explanation can be found in [Allis94]. It is applicable to game trees with outcomes *win*) and *loss*, or, equivalently, pay-offs 1 and 0. The original publication is built around the notions proof number and disproof number, which in [Allis94] are defined in terms of the notions proof and disproof set. Here, we will explain the underlying idea in terms of solution trees.

A *proof set* in a search tree is defined as a set of open leaves, which, if their game value is to 1, entails that the value of the root equals 1. The dual notion, disproof set, is obtained by taking 0 instead 1. A proof tree in a search tree for a node n is a min tree T^- with root n, in which each closed leaf p, if any, satisfies $f(p) = 1$. The set of open nodes in a proof tree is a proof set, because a proof tree is able to prove n, i.e., to make $f(n) = 1$. This is achieved, when, in a proof tree, each open node p in a proof tree takes the value $f(p) = 1$. In that case, the maximum value of the min solution trees is equal to 1, and consequently, by Stockman's theorem, $f(n) = 1$. Dually a disproof tree is a max tree T^+ with open and closed nodes, where the closed nodes has f-value = 0.

The proof number of a proof tree T^- (notation: *proof*(T^-)) is defined as the number of open nodes in T^-. The proof number a node n, denoted by *proof*(n) is defined as:

$$\mathit{proof}(n) = \min\{proof(T^-) \mid T^- \ \textit{a proof tree for} \ \ n\}$$

being the minimum number of open nodes, that must be closed with value 1 to achieve $f^-(n) = f(n) = 1$. Dually, we have the disproof number of n, defined as:

$$\mathit{disproof}(n) = \min\{\mathit{disproof}(T^+) \mid T^+ \ \textit{a disproof tree for} \ \ n\}$$

We assume that the minimum of an empty set of numbers is $+\infty$. It is easily seen, if n is a max node,

$$\mathit{proof}(n) = \min\{\mathit{proof}(c) \mid c \ \ \textit{a child of} \ n\} \tag{5}$$

and, if n is a min node:

$$\mathit{proof}(n) = \sum \mathit{proof}(c), c \ \ \textit{a child of} \ n \tag{6}$$

Of course, these formulas have dual counterparts for the *disproof*-function. We have the following key property for a node n

$$f^-(n) = 1 \Leftrightarrow \mathit{proof}(n) = 0 \Leftrightarrow \mathit{disproof}(n) = \infty \tag{7}$$

$$f^+(n) = 0 \Leftrightarrow \mathit{proof}(n) = \infty \Leftrightarrow \mathit{disproof}(n) = 0. \tag{8}$$

We only discuss a). The alternate case is dual.
First, we prove the left equivalence. $f^+(n) = 1$ holds iff there exists a min solution tree T^- such that $g(T^-) = 1$, iff there is a min solution tree with solely closed nodes with value 1, iff *proof*(n) = 0.
Now, we prove the right, second, equivalence. The property *disproof*(n) = ∞ is a formal way to state that no disproof tree for n exists. Obviously, in an arbitrary search tree, every max solution tree and every min solution tree have a common path from the root to a leaf. This has an important consequence. If at least one proof tree exists (a min solution tree with value 1 in each leaf), then every max solution tree has a leaf with value 1, and hence, a disproof tree cannot exist. We conclude that a proof number 0 implies a disproof number ∞. The inverse implication is more complex. The inverse implication says that, if every max tree

has at least one leaf with value 1, then a min solution tree with value = 1 in each leaf can be found. A proof is left out here. The reader is referred to [Bru94] for more details.

In [Allis94] the proof-number search algorithm is described as: as long as the root has proof and disproof number between 0 and ∞, perform an iteration, descending from the root to a leaf of the search tree, choosing in a max node a child with the same *proof*-value and in a min node a child with the same *disproof*-value. It is only informally indicated that this leaf has the desired properties, i.e., it is open and both in a minimal proof tree and disproof tree for root r. Using the above observations, we are now able to prove this.

We have the property that each node on the path is in a mimimal proof tree and in a minimal disproof tree, and has proof number and disproof number between 0 and ∞. Suppose n is a node on this path and these properties hold for n. Consider the child c of n chosen. Then c is a child with the same proof number as n, which is a value between 0 and ∞. But the right implication in (7) and (8) shows that also the disproof number of c is between 0 and ∞. Furthermore, if n is in the minimal disproof tree of root r (which is a max solution tree), then c is also in this minimal disproof tree. By the choice of c and the definition of proof number, c is in a minimal proof tree of n and thus in a minimal proof tree of the root. The fact that the leaf at the end of this path is open, follows from the observation that, if it was closed, it would have value both 0 and 1, being a member of both a proof and a disproof tree.

In each iteration, the leaf selected along the lines above described, is expanded. Subsequently, the proof and disproof numbers are updated bottom-up, using (5) and (6) together with their dual counterparts. The algorithms stops, when the root has either proof-number or disproof number = 0.

8 Conclusions

SSS* and SSS-2 are existing algorithms, built around solution trees. We have elaborated on the role of solution trees in alpha-beta and PVS/NWS. Due to the new insight into the working of alpha-beta and NWS, we were able to establish a clear link between alpha-beta and SSS-2/SSS*: SSS-2 is a sequence of NWS calls. The idea of a sequence of NWS-calls can applied in several ways. One can formulate several rules for determining the null window, that is to be used in the next iteration. NegaC* search [Wei90, Wei92] is an instance of an algorithm consisting of a sequence of NWS calls. Here, a null window is established in each iteration, in a way, different from SSS-2.
As shown in section 7, solution trees also have to do with Proof-number search. Unlike the original explanation, our description of this algorithm is entirely based on solution trees.

Acknowledgements

We would like to thank Jonathan Schaeffer for inspiration as well as critique. Talking with him has given us the opportunity to look from a different perspective at our work.

References

[ACH90] T. Anantharaman, Murray S. Campbell, and F.-H. Hsu, *Singular extensions: Adding selectivity to brute-force searching*, Artificial Intelligence **43** (1990), no. 1, 99–109.

[AvdMvdH94] L. Victor Allis, Maarten van der Meulen, and H. Jaap van den Herik, *Proof-number search*, Artificial Intelligence **66** (1994), 91–124.

[Allis94] L. Victor Allis, *Searching for Solutions in Games and Artificial Intelligence*, Ph. D. Thesis, Maastricht, NL 1994.

[BB93] Subir Bhattacharya and A. Bagchi, *A faster alternative to SSS* with extension to variable memory*, Information processing letters **47** (1993), 209–214.

[Bru94] A. de Bruin, W. Pijls and A.Plaat, *Solution Trees as a Basis for Game Tree Search*, Technical Report EUR-CS-94-04, Department Computer Science, Erasmus University Rotterdam.

[CM83] Murray S. Campbell and T. A. Marsland, *A comparison of minimax tree search algorithms*, Artificial Intelligence **20** (1983), 347–367.

[FF80] John P. Fishburn and Raphael A. Finkel, *Parallel alpha-beta search on arachne*, Tech. Report 394, Computer Sciences Dept, University of Wisconsin, Madison, WI, 1980.

[Iba86] Toshihide Ibaraki, *Generalization of alpha-beta and SSS* search procedures*, Artificial Intelligence **29** (1986), 73–117.

[Kuma83] 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.

[Kuma84] V.Kumar and L.N. Kanal, *Parallel Branch and Bound Formulations for AND/OR Tree Search*, IEEE Transactions on Pattern Analysis and Machine Intelligence, **PAMI-6** (1984) no. 6, pp. 768-778.

[KM75] Donald E. Knuth and Ronald W. Moore, *An analysis of alpha-beta pruning*, Artificial Intelligence **6** (1975), no. 4, 293–326.

[Kor93] Richard E. Korf, *Best-first minimax search: First results*, Proceedings of the AAAI'93 Fall Symposium, American Association for Artificial Intelligence, AAAI Press, October 1993, pp. 39–47.

[PdB90] Wim Pijls and Arie de Bruin, *Another view on the SSS* algorithm*, Algorithms, International Symposium SIGAL '90, Tokyo, Japan, August 16–18, 1990 Proceedings (T. Asano, T. Ibaraki, H. Imai, and T. Nishizeki, eds.), LNCS, vol. 450, Springer-Verlag, August 1990, pp. 211–220.

[PdB92] Wim Pijls and Arie de Bruin, *Searching informed game trees*, Tech. Report EUR-CS-92-02, Erasmus University Rotterdam, Rotterdam, NL, October 1992, Extended abstract in Proceedings CSN 92, pp. 246–256, and Algorithms and Computation, ISAAC 92 (T. Ibaraki, ed), pp. 332–341, LNCS 650.

[PdB93] Wim Pijls and Arie de Bruin, *SSS*-like algorithms in constrained memory*, ICCA Journal **16** (1993), no. 1, 18–30.

[PdB94] Wim Pijls and Arie de Bruin, *Generalizing alpha-beta*, Advances in Computer Chess 7, Maastricht (H.J. van den Herik, I.S. Herschberg, and J.W.H.M. uiterwijk, eds.), University of Limburg, Maastricht, The Netherlands, 1994, pp. 219–236.

[Pea80] Judea Pearl, *Asymptotical properties of minimax trees and game searching procedures*, Artificial Intelligence **14** (1980), no. 2, 113–138.

[Pea84] Judea Pearl, *Heuristics – intelligent search strategies for computer problem solving*, Addison-Wesley Publishing Co., Reading, MA, 1984.

[Pij91] Wim Pijls, *Shortest paths and game trees*, Ph.D. thesis, Erasmus University Rotterdam, Rotterdam, NL, November 1991.

[Rein85] A. Reineveld, J. Schaeffer and T. Marsland, *Information Acquisition in minimal window search*, Proceedings of the 9-th IJCAI, pp. 1040-1043.

[Rei89] Alexander Reinefeld, *Spielbaum suchverfahren*, volume Informatik-Fachberichte 200. Springer Verlag, 1989.

[Rei94] Alexander Reinefeld, *A minimax algorithm faster than alpha-beta*, Advances in Computer Chess 7 (H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk, eds.), University of Limburg, Maastricht, The Netherlands, 1994, pp. 237–250.

[RP83] Igor Roizen and Judea Pearl, *A minimax algorithm better than alpha-beta? yes and no*, Artificial Intelligence **21** (1983), 199–230.

[Sch89] Jonathan Schaeffer, *The history heuristic and alpha-beta search enhancements in practice*, IEEE Transactions on Pattern Analysis and Machine Intelligence **PAMI-11** (1989), no. 1, 1203–1212.

[Sto79] G. Stockman, *A minimax algorithm better than alpha-beta?*, Artificial Intelligence **12** (1979), no. 2, 179–196.

[Wei90] J.C. Weill, *Experiments with the NegaC* Search,* Heuristic Programming in Artificial Intelligence 2: the second computer olympiad (eds. D.N.L.Levy and D.F. Beal), 164–188. Ellis Horwood, Chicester, UK.

[Wei92] J.C. Weill, *The NegaC* Search,* ICCA Journal **15** (1992), no. 1, 3–7.