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

September 20, 1994

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: Principal Variation Search, alpha-beta, and SSS*. A general procedure for the construction of a solution tree, based on alpha-beta and Null-Window-Search, is given. Furthermore three new examples of solution tree based-algorithms are presented, which surpass alpha-beta—i.e., never visit more nodes than alpha-beta, and often less.
Keywords: Game tree search, alpha-beta, SSS*, solution trees.

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 minimal window 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]).

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.

---
[*]This paper is also registered as Technical Report EUR-CS-94-04
[†]Tinbergen Institute, Erasmus University, and Department of Computer Science, Erasmus University.

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.

As an important side effect of this insight we have found a number of new game tree search algorithms, all like SSS* in the sense that they search not more nodes than alpha-beta by exhibiting a best-first behavior. Also, these new algorithms are like SSS* in that they have a comparable space complexity, since they store solution trees in memory.

By doing more research into the behavior of these algorithms (e.g., memory requirements), we hope to show that they are of use for practical applications like game playing programs.

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 notion *critical* is introduced as follows: MAX tries to maximize and MIN tries to minimize the profit of MAX. Therefore, an 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 will investigate how alpha-beta, Principal Variation Search (PVS) [FF80, Pea84], and SSS-2 [PdB92] use solution trees to construct this critical tree. The relation between solution trees and Null-Window-Search is discussed. 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. PVS and SSS* [Sto79, PdB90].

In section 4 we give an enhanced version of alpha-beta that determines the game value of a node n. Unlike the standard version of alpha-beta found in many text books, this procedure also takes into account information that has been gathered in earlier visits of the subtree of the game tree rooted in n.

In section 5 we introduce three examples of algorithms that use our enhanced alpha-beta procedure to efficiently search game trees. From results derived in section 4 these algorithms search not more nodes than alpha-beta. The results of some preliminary tests on their behavior are presented.

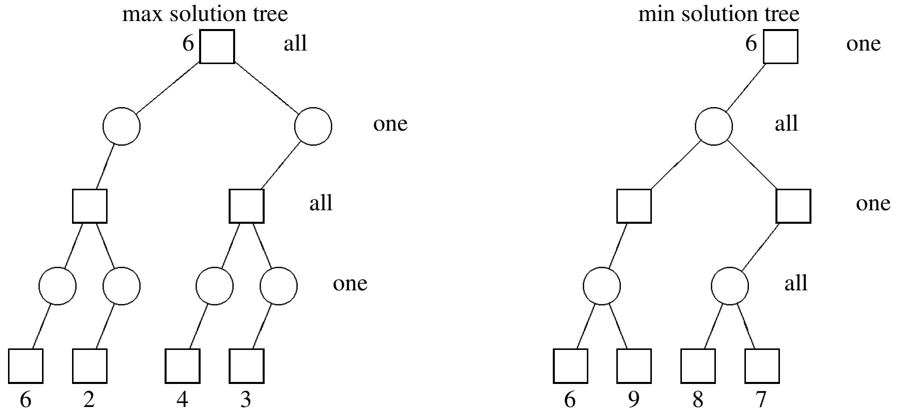

Figure 1: Solution Trees

2 Solution Trees and Bounds

In this section we will show that there exists a relation between solution trees and bounds on the game value, and we will show how solution trees and the *critical tree* relate.

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 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, which was made by Stockman before [Sto79]. We also state its counterpart for min solution trees.

> *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.

Search Tree

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 the solution tree in a given 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 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 trees 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,

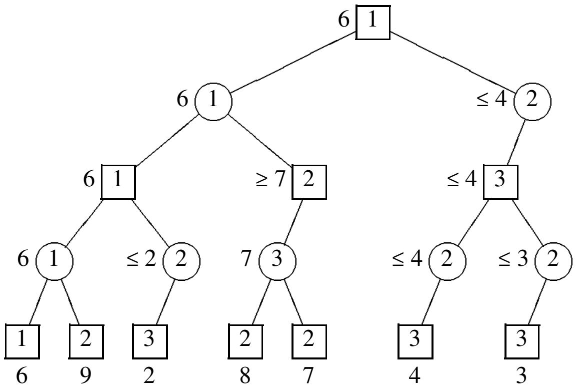

Figure 2: A Critical Tree with node types, values (f), and bounds (f^+, f^-)

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.

Critical 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, called T^+ and T^- respectively, are obtained, which are optimal, i.e., $g(T^+) = g(T^-) = f(G)$. 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. In [PK87, p. 462] an intuitive explanation of the concept "critical tree" is given in terms of optimal strategies for MAX and MIN. In [KK84] the link between an optimal strategy for MAX/MIN and an (optimal) min/max solution tree has been explained. The tree in figure 2 is the union of the solution trees of figure 1.

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

```
function alpha-beta(n, α, β) → v;
    if n = leaf then return eval(n);
    if n = open then generate all children of n;
    if n = MAX then
        v := −∞;
        c := firstchild(n);
        while v < β and c ≠ ⊥ do
            v := max(v, alpha-beta(c, α, β));
            α := max(α, v);
            c := nextbrother(c);
    if n = MIN then
        v := +∞;
        c := firstchild(n);
        while α < v and c ≠ ⊥ do
            v := min(v, alpha-beta(c, α, β));
            β := min(β, v);
            c := nextbrother(c);
    return v;
```

Figure 3: Alpha-Beta

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 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 Solution Trees in Game Tree Algorithms

In the preceding section the relation between solution trees and bounds on the minimax value of a node has been treated. An $f^+(n)$ is determined by a max solution tree, and an $f^-(n)$ by a min solution tree. In this section we will see how solution trees fit into a number of existing game tree algorithms. We will describe how alpha-beta, Principal Variation Search and SSS-2 use solution trees. To this end, we will first treat the simple case, and investigate how alpha-beta traverses solution trees.

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. 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 $\alpha < \beta$.)

$$v \geq \beta \quad \Rightarrow \quad v = f^-(n) \leq f(n) \tag{1}$$

$$\alpha < v < \beta \quad \Rightarrow \quad v = f^+(n) = f^-(n) = f(n) \tag{2}$$

$$v \leq \alpha \quad \Rightarrow \quad v = f^+(n) \geq f(n) \tag{3}$$

We can re-state these implications into the following twofold postcondition. Together with the theory of the preceding section—the relation between bounds and solution trees—we find immediately that this postcondition can be enhanced with statements on solution trees.

$$v > \alpha \quad \Rightarrow \quad v = f^-(n) \leq f(n) \text{ and } v \text{ is the value of a min solution tree} \tag{4}$$

$$v < \beta \quad \Rightarrow \quad v = f^+(n) \geq f(n) \text{ and } v \text{ is the value of a max solution tree} \qquad (5)$$

Looking carefully at the code of alpha-beta (figure 3), we can see clearly how the solution trees of implications (4) and (5) are generated. We will show how a solution tree is generated in a alpha-beta call with parameter n. This is done by induction on the height of n. We only consider the case that n is a max node (the alternate case is similar).

Suppose the procedure call ends with $v < \beta$. Then the while loop of line 7 has not been aborted and each child c has returned a value $v_c < \beta$. By the induction hypothesis, each child is the root of a max solution tree. Appending all these trees to n, a max solution tree with root n is obtained.

We will denote the input value of parameter alpha (the initial value of α) by α_0. Suppose the procedure call ends with $v > \alpha_0$. The while loop starting at line 7 has the folowing invariant: $v = \alpha > \alpha_0$ or $v \leq \alpha = \alpha_0$. Let c_0 be the node parameter in the latest subcall that increases v in line 8. The return value of this subcall also determines the return value v of the main call. When $v \leq \alpha = \alpha_0$ before this subcall, then, since the return value of the main call is $> \alpha_0$, we must have that the return value of this subcall exceeds $\alpha = \alpha_0$. When $v = \alpha > \alpha_0$ before this subcall, then the return value of this subcall, since it causes an update in line 8, must be larger than v and hence larger than α. In both cases we have by induction that the subcall alpha-beta(c_0, α, β) has generated a min solution tree. Appending this tree to n yields a min solution tree, rooted in n.

Note that if, in the latter case, we have in addition to $v > \alpha$ that $v < \beta$, also a max solution tree is constructed, i.e., a critical tree.

Now that we have seen how alpha-beta generated solution trees, we will investigate how more involved algorithms like PVS and SSS-2 use solution trees and bounds to construct the ultimate critical tree that proves the minimax value of the root.

Principal Variation Search

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. Although there is a difference between the two, the following description of PVS holds for SCOUT as well.

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 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.

As was noted before, alpha-beta can be used to construct a solution tree and return a bound by having it search a window of zero size. To achieve this, the search window $\langle \alpha, \beta \rangle$ is reduced to a null-window by substituting $\alpha = \gamma - 1, \beta = \gamma$ or $\alpha = \gamma, \beta = \gamma + 1$, for some γ. This window is called a null-window because it cannot contain an integer-valued leaf value. The call to alpha-beta is in effect transformed to a one-parameter call. We will state the postconditions for these cases for convenience, although they are the result of trivial substitutions in implications (1), (2), and (3).

$$\begin{array}{lcl} \alpha = \gamma - 1 \wedge \beta = \gamma & & \\ \qquad\qquad v < \gamma & \Rightarrow & v = f^{+}(n) \geq f(n) \\ \qquad\qquad v \geq \gamma & \Rightarrow & v = f^{-}(n) \leq f(n) \end{array}$$

$$\begin{array}{lcl} \alpha = \gamma \wedge \beta = \gamma + 1 & & \\ \qquad\qquad v \leq \gamma & \Rightarrow & v = f^{+}(n) \geq f(n) \\ \qquad\qquad v > \gamma & \Rightarrow & v = f^{-}(n) \leq f(n) \end{array}$$

The overall effect of this is that after termination, a critical path has been constructed by the PV-nodes. This critical path is the backbone of the critical tree. Because NWS as an alpha-beta instance constructs solution trees to prove a bound, PVS constructs a critical tree, consisting of the backbone together with solution trees.

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*. 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.

SSS-2 (see figure 4) works by establishing successive sharper upper bounds for the root, starting with an upper bound of $+\infty$. In the code in the figure this upper bound is the value of a max solution tree which is stored in the global variable T. In each call to *diminish* the solution tree in T is manipulated such that its value g, an upper bound of f, is lowered (or, if a lower value cannot be found, the algorithm terminates with $g(T) = f(G)$, the value of the solution tree equal to the minimax value of the game tree).

The algorithm is built around two procedures, *expand* and *diminish*. A call of expand(n, γ) tries to establish an upper bound to the game value of an open node n which is smaller than γ. *Expand* realizes this by building a max solution tree with value $< \gamma$. If this is not possible, a min solution tree with value $\geq \gamma$ has been traversed. (In the present formulation(figure 4) the handling of the min solution tree is not made explicit: only max solution trees are manipulated explicitly.)

The procedure *diminish* tries to refine an upper bound by transforming a max solution tree into a better one by searching for suitable open nodes in this tree and performing an *expand* on these nodes. After an initial call to *expand* to construct the first max-solution tree, the SSS-2 algorithm performs a sequence of calls of *diminish* applied to the root to obtain sharper max solution trees, until finally this is no longer possible: no lower upper bound can be found, so the optimal upper bound f^+ has been established. The last *diminish* proves this failure because it establishes a min solution tree with lower bound f^- greater than or equal to the previous upper bound. But this means that $f^- = f^+$, and therefore the algorithm has generated a max solution tree as well as a min solution tree of the same value, i.e. a critical tree.

In PVS the procedure NWS is used to construct a bound on the value of the brothers of the PV. (For this proof, children of a MAX node are upper bounded, children of a MIN node

```
function SSS-2(r) → v;
  (T, g) := expand(r, +∞);
  {T is a global variable containing the current max solution }
  repeat
    γ := g;
    g := diminish(r, γ);
  until g = γ;
  return g;

{precondition: γ = f⁺(T) }
function diminish(n, γ) → g;
  if n = leaf then return(n, γ);
  if n = MAX then
    for c := firstchild(n) to lastchild(n) do
      if γ = g(c) then
        g′ := diminish(c, γ);
        if γ = g′ then exit for loop;
    g := the maximum of g-values of all children of n;
  if n = MIN then
    c := the single child of n in the search tree T;
    g′ := diminish(c, γ);
    if γ = g′ then
      for b := nextbrother(c) to lastbrother(c) do
        (S, g′) := expand(b, γ);
        if g′ < γ then
          detach in T from n the subtree rooted in c and attach S to n in T;
          exit for loop;
  return g′;

function expand(n, γ) → (S, g);
  if n = leaf then return eval(n);
  if n = open then generate all children of n;
  if n = MAX then
    g := −∞;
    c := firstchild(n);
    while g < γ and c ≠ ⊥ do
      (S′, g′) := expand(c, γ);
      g := max(g, g′);
      c := nextbrother(c);
    if g < γ then S′ := the tree composed by attaching all intermediate values of S′ to n;
  if n = MIN then
    g := γ;
    c := firstchild(n);
    while g ≥ γ and c ≠ ⊥ do
      (S′, g) := expand(c, γ);
      c := nextbrother(c);
    if g < γ then S′ := the tree with S′ attached to n;
  return (S′, g);
```

Figure 4: SSS-2, an explicit-solution-tree version of SSS* [PdB90]

lower bounded.) In SSS-2 an *expand* constructs an upper bound, and returns a max solution tree that proves the value of this upper bound. Inspection of the code shows that NWS for upper bounds and *expand* traverse the same nodes, and are in fact equivalent in that respect. (For the dual case—min solution tree and lower bound—a procedure very similar to *expand* can be formulated. This procedure is equivalent to NWS for lower bounds.)

The difference between NWS and *expand* is that *expand* returns the max solution tree defining the f^+ explicitly, whereas NWS, being an alpha-beta instance, just traverses it, without returning anything but the value. So *expand* is equivalent to a solution tree-returning NWS.

Conclusion

Here we find one of the relations between alpha-beta and SSS*-like algorithms (in particular SSS-2). First we have shown that it is easy to see how the alpha-beta procedure generates solution trees. Furthermore we have seen how PVS uses the null-window variant of alpha-beta to construct solution trees, and how *expand* generates solution trees to determine an upper bound in each iteration of the SSS-2 algorithm. The two solution tree generating procedures turn out to be in fact equivalent.

The other procedure of SSS-2, *diminish*, will be discussed in the next section. In that section we will see the other link between alpha-beta and SSS-2: SSS-2 can be formulated as a sequence of null-window searches (a sequence of S-NWS calls, to be precise).

4 Using Bounds in the Search Tree

In the previous section the algorithms PVS and SSS-2 were discussed, together with their solution tree-traversing procedures NWS and *expand.* These procedures are called by these algorithms on open nodes.

As has been noted in [MRS87] it would be nice if the information that was gathered during a subtree-traversal could be used by the algorithm again. To this end they proposed INS, or Informed NegaScout, an enhanced version of PVS that remembers more information from a call to NWS than just the value of the bound. Going even further, we can save the entire solution tree that proves the value of the bound. This is done by *expand*, which returns the solution tree. This tree is then used by *diminish*, that transforms the input tree to one defining a sharper bound, that is subsequently returned together with the new tree. So, where NWS/alpha-beta and *expand* were called on an empty search tree, *diminish* is called on a non-empty search tree (more precisely, on a solution tree).

Where *expand* and NWS/alpha-beta are called on open nodes, we will now discuss a version of alpha-beta that can be called, like *diminish*, on existing nodes.

Alpha-Beta for Non-Empty Search Trees

Alpha-beta can only be called on *open* nodes n. In figure 5 we show a version of alpha-beta that can be called on the root of a *non-empty* search tree S. This procedure will be useful for the algorithms to be discussed in the next section. Similar versions of alpha-beta are used in game playing programs that use tables to prevent search overhead caused by transpositions, and store search results of previous iterations. See e.g. [Mar86, p. 14]. We will see how the standard version of alpha-beta can be changed so that it will be able to perform this new task. The procedure will be called S-alpha-beta, since it can be called on a search tree S.

In [PdB92] a version of alpha-beta is presented which can be applied to so-called *informed* game trees [Iba86]. These are trees for which in all internal nodes n a heuristic upper and lower bound to $f(n)$ is available. The idea is that the values $f^+(n)$ and $f^-(n)$ derived from the search tree rooted in n can be used as these heuristic bounds. (In a leaf n of the game tree $f^+(n) = f^-(n) = f(n)$ holds.) The precondition ($\alpha < \beta$) and postcondition (implications (1), (2), and (3)), as well as the correctness proof of S-alpha-beta are the same

```
function S-alpha-beta(n, α, β) → v;
    if n = leaf then n.f⁺ := n.f⁻ := eval(n);
    if α ≥ n.f⁺ or n.f⁻ ≥ β or n.f⁺ = n.f⁻ then
        if n.f⁻ ≥ β then return n.f⁻;
        else return n.f⁺;
    if n = open then attach all children to n with f⁺ = +∞, f⁻ = −∞;
    if n = MAX then
        α′ := max(α, n.f⁻); β′ := min(β, n.f⁺);
        g := −∞;
        for c := firstchild(n) to lastchild(n) do
            g := max(g, S-alpha-beta(c, α′, β′));
            α′:= max(α′, g);
            if g ≥ β′ then exit for loop;
    if n = MIN then
        α′ := max(α, n.f⁻); β′ := min(β, n.f⁺);
        g := +∞;
        for c := firstchild(n) to lastchild(n) do
            g := min(g, S-alpha-beta(c, α′, β′));
            β′ := min(β′, g);
            if g ≤ α′ then exit for loop;
    update (n.f⁻, n.f⁺);
    return g;
```

Figure 5: S-alpha-beta

as for the heuristic bounds version of alpha-beta [PdB92]. S-alpha-beta is nothing more than a version of alpha-beta that uses the bounds f^+ and f^- that may exist in the search tree below n.

The version of NWS that consists of S-alpha-beta called with a window of $(\gamma - 1, \gamma + 1)$ (sometimes $(\gamma - 1, \gamma)$ or $(\gamma, \gamma + 1)$) will be called S-NWS in the rest of this paper.

S-alpha-beta and SSS-2

Given a search tree that contains max solution trees, one gets a sharper max solution tree by calling S-alpha-beta($root, \gamma - 1, \gamma$) where $\gamma \in \langle f^-(root), f^+(root)]$, by the postcondition of (S-)alpha-beta.

Each iteration of SSS-2 yields a max solution tree, which is input to the *diminish* call with γ-parameter = $g(T)$ in the next iteration. T is the (explicit) max solution tree between two iterations of SSS-2. (Of course, T is part of the search tree at that time. This (implicit) search tree is denoted by S.) For every combination of n and c_0 with n a min node in T and c_0 its single child in T, we have a remarkable property. Every child c of n to the left of c_0 has been parameter in a former call to *expand* or *diminish* with parameter $\gamma > g(T)$, which has ended with $g = \gamma$. Every child to the right of c_0 has not been visited yet by an *expand* or *diminish* call. This implies for S that $f^-(c) > g(T)$ for every c to the left of c_0, and $f^+(c_0) = g(c_0) \le g(T)$. To the right of c_0 we have infinite values (see [PdB92] for more details). For every max node n in T, we have $f^+(n) = g(n) \le g(T)$. This relation can be proved bottom-up.

Applying S-NWS with parameter $\gamma = g(T)$ to S has the following effect. The nodes along paths with constant f^+-value ($f^+(n) = g(T)$) are parameter in a subcall of *S-alpha-beta*. These paths belong to T. For each such path, the open children of the min nodes are visited, from left to right in the search tree, to find out, whether at least one child c can take a value $f^+(c) < g(T)$. This is equivalent to looking for a sub-max solution tree T', rooted in c with $g(T') < g(T)$.

We find that S-alphabeta works on S in the same way as *diminish*. We can replace *diminish* calls by S-alpha-beta calls in the code of SSS-2. Since *expand* is equivalent to NWS (which is equivalent to S-alpha-beta on open nodes), we can replace the former code of SSS-2 by the following:

function SSS-2(n) $\rightarrow v$;
  $g := +\infty$;
  repeat
    $\gamma := g$;
    $g :=$ S-alpha-beta($n, \gamma - 1, \gamma$);
  until $g = \gamma$;
  return g;

Note that this formulation of SSS-2 makes construction of the dual of SSS-2 almost trivial: $g := -\infty$; $g :=$ S-alpha-beta($n, \gamma, \gamma + 1$). Also, it should be noted that ∞ should be read as a finite number outside the range of leaf values. Hence "$\infty - 1 < \infty$" in the S-NWS call with $\gamma = +\infty$ in the SSS-2 code above makes sense.

A minor point to note is that there is a small point on which *diminish* and S-NWS differ. Although both take a solution tree as input and produce a (sharper) solution tree as output, they do not always expand the same nodes. Because the original (diminish) definition of SSS-2 in figure 4 does not use lower bounds, it searched, in some exceptional cases, more nodes than the present formulation (just like SSS* searches more nodes). This point of difference is discussed in [PdB93, p. 29, figure 10]. Following the terminology of that article, the present (S-alpha-beta) version of SSS-2 should really be called "Maxsearch."

Narrower Window searches Surpass Wider Window searches

It is a well known feature that the narrower the α-β-window, the more cut-offs occur, and hence, the smaller the number of generated nodes is [CM83, Pea84]. The alpha-beta version in [PdB92], suited for game trees with heuristic bounds in each node, expands less nodes as the heuristic bounds become tighter or the α-β gets narrower. (See the proof ibid.) Accordingly, a call to S-alpha-beta expands less new nodes, when (a) the size of the existing search tree S is greater, or (b) the input window is narrower. Therefore, every alpha-beta call with a null-window surpasses the alpha-beta algorithm. (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 S-alpha-beta does not re-expand nodes when searching an existing search tree, it generates less nodes than when called on an open node. It follows that every sequence of S-alpha-beta calls with null windows surpasses the alpha-beta algorithm in the sense that it never generates more nodes.

Since SSS-2 consists of a number of S-NWS calls, this algorithm surpasses alpha-beta. Here, we rediscover a result in [PdB90] extending a weaker result in [Sto79]. (See also [Rei89].)

5 Three New Solution Tree Algorithms

Drawing on the knowledge of solution trees and how they are used in other algorithms, we will present in this section a few examples of new ways of using solution tree generating procedures. We do not wish to say that these three algorithms are the only possible ways of using procedures like S-alpha-beta to create new algorithms. We mention these instances only as examples of interesting new algorithms. At the end of this article, the results of some experiments to determine the performance of these algorithms will be discussed.

In the algorithms of this section we need a procedure that not only establishes upper or lower bounds but one that in some cases constructs a critical tree as well—cf. implication (2), the middle part of alpha-beta's postcondition—because we want to test the hypothesis that the game value equals γ. In this case we can choose $\alpha = \gamma - 1$ and $\beta = \gamma + 1$, yielding the following postcondition:

$$\begin{aligned} v < \gamma \quad &\Rightarrow \quad v = f^+(n) \geq f(n) \\ v = \gamma \quad &\Rightarrow \quad v = f^+(n) = f^-(n) = f(n) \\ v > \gamma \quad &\Rightarrow \quad v = f^-(n) \leq f(n) \end{aligned}$$

First Guess—SSS-0

SSS-2 performs a sequence of calls to establish sharper max solution trees in each iteration. It can be described as repetitive S-alpha-beta($n, \gamma - 1, \gamma$) (see previous section).

A natural variation of the idea of starting the search at $+\infty$ is to start the search at a value that we expect to be closer to the game value. We might save ourselves searching some nodes by starting closer (hopefully) to our target. The idea to start at some other value than $+\infty$ can be regarded as a generalization of SSS-2. The resulting generalization will be called *SSS-0*. The evaluation of a position may be a good candidate for use as such an approximating first guess needed by SSS-0.

function SSS-0(n, g) $\rightarrow v$;
    repeat
        $\gamma := g$;
        $g :=$ S-alpha-beta($n, \gamma - 1, \gamma + 1$);
    until $g = \gamma$;
    return g;

If, after the first call to S-NWS, the first bound g turns out to be lower than our initial guess, then we have (by the postcondition) $g = f^+$. We can then call S-alpha-beta($n, \gamma - 1, \gamma$) to lower the upper bound. If, on the other hand, the first bound is higher than our guess, then we have $g = f^-$. We can find the game value by increasing the lower bound, which comes down to calling S-alpha-beta($n, \gamma, \gamma + 1$). These two cases have been combined into the call S-alpha-beta($n, \gamma - 1, \gamma + 1$) in the code, to make SSS-0's code look more like SSS-2. It would be slightly more efficient to have the original two cases of S-alpha-beta calls. We have not shown that code here, since we feel it is less clear.

SSS-0 may save work because it starts closer to f, and can probably reach f in fewer steps than SSS-2 (which starts at ∞). Given the smaller number of steps, it is hoped that SSS-0's steps expand about the same number of nodes as SSS-2's steps—in other words, that the solution trees are about the same size. If this is true, we would have found a profitable way to make use of heuristic knowledge in the form of a first guess. Please refer to section 6 for some test results.

Stepwise State Space Search—SSS-4

Another possibility to exploit the idea of "bigger steps get you home sooner" is to lower S-alpha-beta's input parameter γ in bigger steps than from upper bound to upper bound as in SSS-2—keeping $f^-(r) \leq \gamma \leq f^+(r)$, since searching outside the root-window is useless.

function SSS-4(n) $\rightarrow v$;
    $g := +\infty$;
    repeat
        $\gamma := g$;
        $g :=$ S-alpha-beta($n, \gamma - 1, \gamma + 1$);
        $g := \max(g - \mathit{STEPSIZE}, f^-(n))$;
    until $g = \gamma$;
    return g;

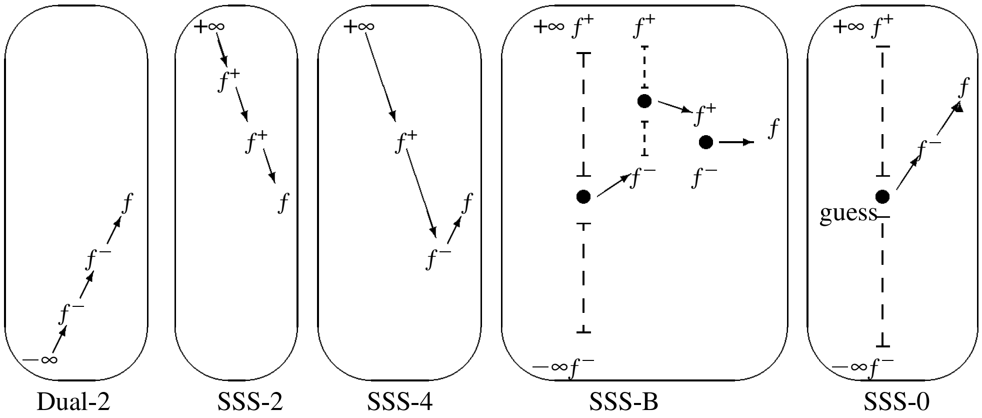

Figure 6: SSS-like algorithms

This idea should be compared to SSS-0. SSS-0 can be used to speed-up the search when we have some idea of f beforehand. SSS-4 can be used to get closer to f in less steps. A danger inherent to SSS-4 is that it can overshoot the target if the steps are too large.

A variation might be to resort to a wide window call to S-alpha-beta, when the bounds have become relatively close: v := S-alpha-beta($n, f^-(n), f^+(n)$). Another variation might be to have a variable STEPSIZE. To achieve good performance some application dependent fine tuning will probably be necessary.

Bisection—SSS-B

A third possibility is to use a well-known idea from approximation algorithms: bisection. During the search for the game value two bounds exist, f^+ and f^-. Instead of calling each S-NWS with a γ equal to one of the bounds, we can choose any value in between, for instance the average of the two bounds. After a call to S-NWS we get a sharper upper or lower bound. This new bound (say: the upper) can then be used in conjunction with the other bound (say: the lower) to compute the next pivot value for γ.

function SSS-B(n) $\rightarrow v$;
    repeat
        γ := average($f^-(n), f^+(n)$);
        g := S-alpha-beta($n, \gamma - 1, \gamma + 1$);
    until $g = \gamma$;
    return g;

Concluding Remarks

Figure 6 summarizes the behavior of these algorithms with respect to bounds and the input parameter. It gives an impression how each algorithms jumps from one bound to the next bound after each iteration.

The algorithms presented in this section are built around a sequence of S-NWS calls, and hence, they surpass alpha-beta($n, -\infty, +\infty$). The idea that this could lead to good algorithms is not new. Alpha Bounding also uses this idea. See e.g. [Sch86, p. 87] for more on this algorithm.

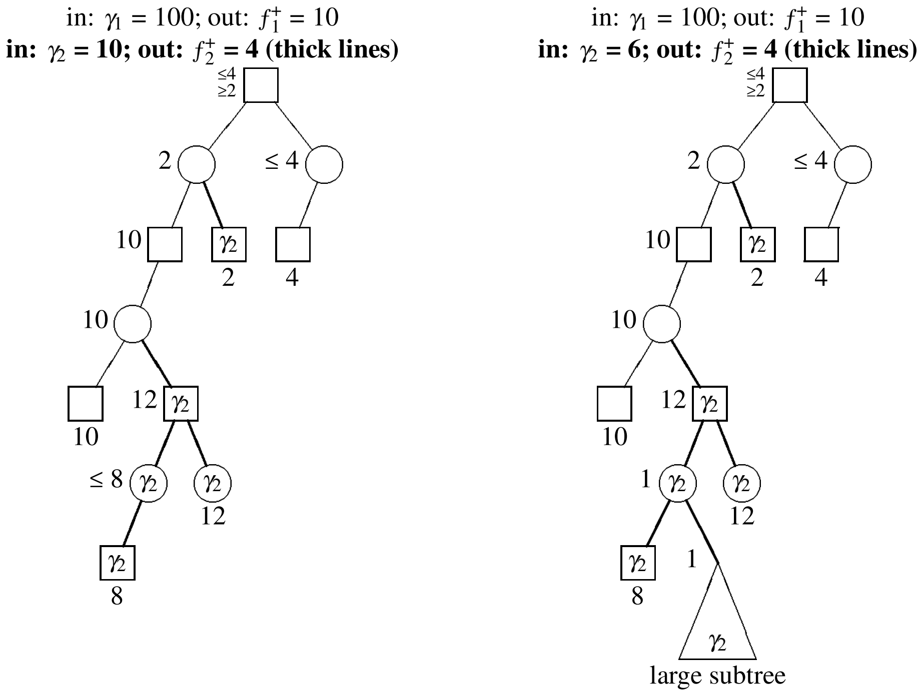

Figure 7: Two sequences of S-NWS calls.

6 First Test Results

Whether the last three algorithms, SSS-0, SSS-4, and SSS-B, expand fewer nodes than SSS-2/SSS* depends on the question whether a null-window call that starts near the game value expands fewer nodes than a call with say $+\infty$. In other words, does a "bigger steps get you home sooner" approach work? (Bigger in the sense of big steps in return value g, *not* big solution trees.) Although this may seem obvious at first, some care is needed here, since it can be shown that there exist game trees in which S-NWS with $f + x_1$ as input parameter expands fewer nodes than S-NWS with $f + x_2$ as input parameter where $x_1 > x_2$ and f is the game value. See figure 7 for an example of such a counter-intuitive tree. In the first iteration S-NWS(n, γ_1) the empty nodes are expanded. In the second iteration S-NWS(n, γ_2) the nodes marked with γ_2 are expanded. The values shown next to the nodes are the resulting values after the respective two S-NWS calls.

To get an impression of the average case performance we have conducted some experiments. We have called the algorithms on a number of artificially constructed uniform game trees. We only report for trees of width 5 and depth 9, although we believe that the results hold for wider trees as well (e.g., $w = 20, d = 5$). We have generated 180 different trees, using a procedure based on [MRS87, Hsu90]. The uniformly distributed leaf values ranged from 0 to 999. The results shown are the average of 20 different random seeds. In figure 8 the number of newly expanded nodes (inner nodes plus leaves) is shown relative to nine levels of node ordering (the probability that the first successor is best). We see that, as we would expect, in a perfectly ordered tree every algorithm expands the same number of nodes. In the graph the performance of "old" algorithms Alpha-beta, PVS, SSS-2 and Dual is shown, together with the "new" ones, SSS-4 with two stepsizes (50 and 100), and SSS-B. It turns out that in these tests, on these trees, Dual-2 performs the best. To reduce unwanted effects we have taken for each of the 20 seeds different game values (e.g., a game

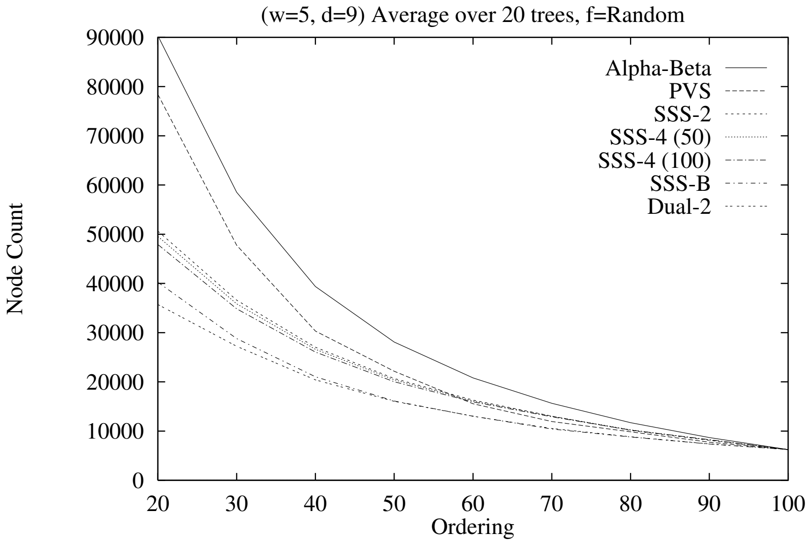

Figure 8: Node Count

value close to 0 would give Dual an advantage, a game value close to 500 would favor SSS-B (since the average of 0 and 999 is around 500), and a game value close to 999 would favor SSS-2—see also figure 10 and its text). More testing is needed to see whether these results hold for other trees as well.

Concerning SSS-0, there is a small complication, since it is difficult to think of a good way to emulate the performance of an evaluation function to provide the first guess on these artificial trees. Therefore, we have not shown SSS-0 on the trees with random game value. Instead, we have decided to show the best-case performance of SSS-0 in figure 9. In this graph the game value of all trees is equal to 504. This value is then used as the first guess for SSS-0. As can be seen—and expected, despite the tree in figure 7—it performs better than Dual. The actual performance of SSS-0 will vary between the lines for SSS-2, Dual, and SSS-0. Just how it varies is shown in figure 10.

The figures shows a phenomenon that has been noted before in [MRS87], viz. that Dual expands less nodes than primal SSS-2 on odd-depth trees. (On even-depth trees Dual does not perform better.) This can also be seen in figure 10, which shows the node count of SSS-0 for 6 different levels of node ordering relative to the first guess, or the start value for the search. On the x-axis the values for the input parameter to SSS-0 are tabulated. It can be seen that the left-hand side (first guess of 0, equivalent to $-\infty$ or Dual) is lower than the right-hand side (first guess of 999, equivalent to $+\infty$ or primal SSS-2). Figure 10 indicates that on average starting the search closer to f is a good idea, contrary to the counter-intuitive tree of figure 7. Figure 10 should not be confused with the "refutation wall" graph in [MRS87], which is a graph of *single* calls to NWS. The nodecount in figure 10 refers to calls to SSS-0, which consists of *a number of* calls to S-NWS. Figure 10 shows that the closer a search starts to the minimax value of a game tree, the less nodes are expanded, on average. The gain in performance is less in ordered trees. This implies that the "bigger steps get you home sooner" idea of SSS-0, SSS-4, and SSS-B may work in principle, although the actual gains will depend on the tree-characteristics of the application at hand.

If the results of these preliminary experiments on artificial trees hold for "real" trees, encountered in actual application domains, then it would appear that SSS-0, SSS-4, and SSS-B, are preferable over SSS-2/SSS*. Whether this is the case is the subject of ongoing research.

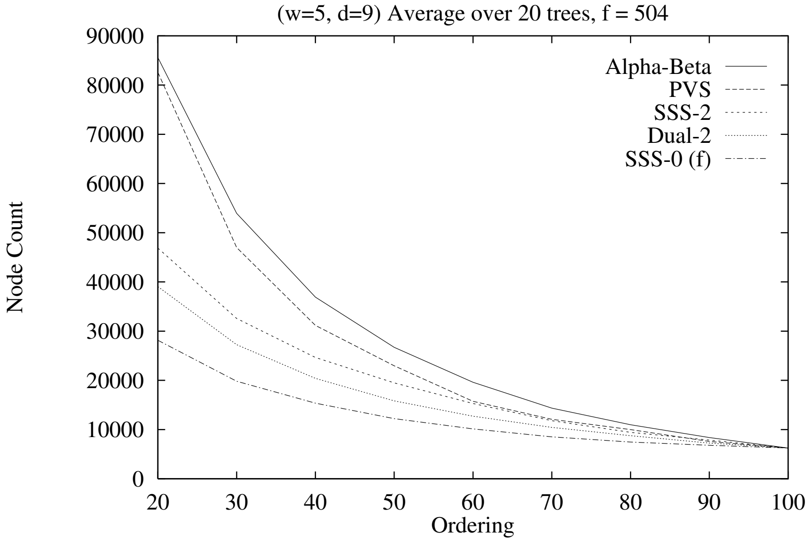

Figure 9: Node Count

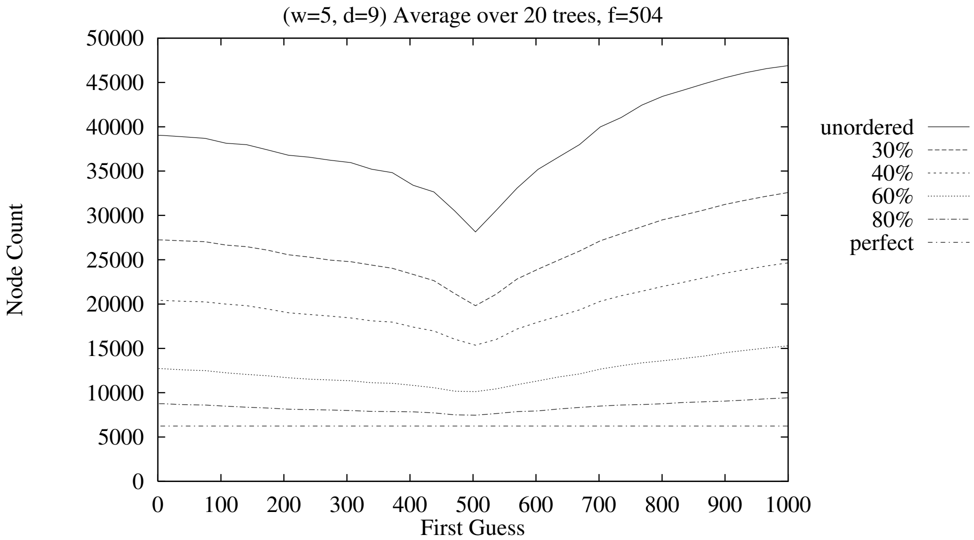

Figure 10: Node Count SSS-0

7 Future Work

We will try to find more, and also more interesting algorithms. For instance, we have experimented with an algorithm "second best search" that finds the best successor to the root by trying to establish a proof that it is better than the other root-successors. (In spirit close to Alpha Bounding, as we found out later.) This can be done by applying S-alpha-beta to a successor of the root with current highest f^+-value, however with a γ-parameter equal to the f^+-value of the *second highest* successor of the root. Here the algorithm stops as soon as one root-successor is better than the other, just as in Alpha Bounding and B* [Ber79].

Also, the space complexity of S-alpha-beta must be addressed. By deleting irrelevant parts of the search tree, or by using schemes like Rsearch [Iba86] or Staged SSS* [CM83], it should be possible to manage S-alpha-beta's memory usage. A transposition table approach, i.e., a scheme comparable to hash tables common in chess programs, might be advantageous in this respect. ([BB86] discusses similar issues for RecSSS*.)

We plan to look into the effect of tree characteristics on the relative performance of the preceding algorithms. Furthermore we will try to extend this research to parallel versions of the algorithms.

We believe that the relation of our work to search enhancements like iterative deepening and transposition tables deserves attention. Finally, the relation with other algorithms like Proof Number Search [AvdMvdH94], Best-First Minimax Search [Kor93], B* [Ber79], and H* [Iba87] is worth investigating.

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.

We thank Henri Bal of the Department of Computer Science of the Free University Amsterdam for generously letting us use their equipment to run our tests.

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.

[BB86] Subir Bhattacharya and A. Bagchi, *Making best use of available memory when searching game trees*, AAAI-86, 1986, pp. 163–167.

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

[Ber79] Hans J. Berliner, *The B* tree search algorithm: A best-first proof procedure*, Artificial Intelligence **12** (1979), 23–40.

[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.

[Hsu90] Feng-Hsiung Hsu, *Large scale parallelization of alpha-beta search: An algorithmic and architectural study with computer chess*, Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA, February 1990.

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

[Iba87] Toshihide Ibaraki, *Game solving procedure H* is unsurpassed*, Discrete Algorithms and Complexity (D. S. Johnson et al., ed.), Academic Press, Inc., 1987, pp. 185–200.

[KK84] Vipin Kumar and Laveen 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, 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.

[Mar86] T. Anthony Marsland, *A review of game-tree pruning*, ICCA Journal **9** (1986), no. 1, 3–19.

[MRS87] T. A. Marsland, Alexander Reinefeld, and Jonathan Schaeffer, *Low overhead alternatives to SSS**, Artificial Intelligence **31** (1987), 185–199.

[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.

[PK87] Judea Pearl and Richard E. Korf, *Search techniques*, Annual Reviews Computer Science **2** (1987), 451–467.

[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.

[Sch86] Jonathan Schaeffer, *Improved parallel alpha-beta search*, Fall Joint Computer Conference, FJCC (Washington, DC), IEEE Computer Society, 1986, pp. 519–527.

[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.