Solution Trees as a Basis for Game Tree Search

Arie de Bruin, Wim Pijls, Aske Plaat[*]
Technical Report EUR-CS-94-04, May 1994
Erasmus University, Department of Computer Science
P.O.Box 1738, 3000 DR Rotterdam, The Netherlands
plaat@cs.few.eur.nl

Abstract

A game tree algorithm is an algorithm computing the minimax value of the root of a game tree. Many algorithms use the notion of establishing proofs that this 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 in the following algorithms: Principal Variation Search, alpha-beta, and SSS-2. A general procedure for the construction of a solution tree, based on alpha-beta and Null-Window-Search, is given. Furthermore two new examples of solution tree based-algorithms are presented, that surpass alpha-beta—i.e., never visit more nodes than alpha-beta, and often less.
Keywords: Game tree search, alpha-beta, solution trees, algorithms.

1 Introduction

Game trees are related to two person zero sum games with perfect information like Tic-Tac-Toe, Checkers, Chess, and Go. Each node in a game tree represents a game position. The root represents a position of the game, for which we want to find the best move. The children of each node n correspond to the positions resulting from one move from that position. The leaves in the tree are positions in the game for which an integer valued evaluation function f exists giving the so called game value, the pay-off in that position. A game tree is assumed to remain unchanged during the search for the best move, in the sense that we do not look into search enhancements like iterative deepening [Sch89].

We assume that the two players are called MAX and MIN. A node n is marked as max-node or min-node, if in the corresponding position it is max's or min's turn to move respectively. We assume that MAX moves from the start position.

The evaluation function can be extended to the so called *minimax function*, a function which determines the value for each player in any node. The definition is:

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

In Figure 1 an example of a game tree is shown labeled with its f-values. The squares represent max nodes, the circles min nodes. For a game tree G with root r, the value $f(r)$ is also called the minimax value of G, denoted by $f(G)$. A game tree algorithm is an algorithm computing the root successor with the highest pay-off for MAX—the best move for MAX—or the minimax value of a game tree, from which we can easily infer the best move.

The value $f(n)$ in any node n (n not necessarily a max node) indicates the highest attainable pay-off for MAX in the position n, under the condition that both players will play optimally in the sequel of the game according to the evaluation function f. In any node n

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

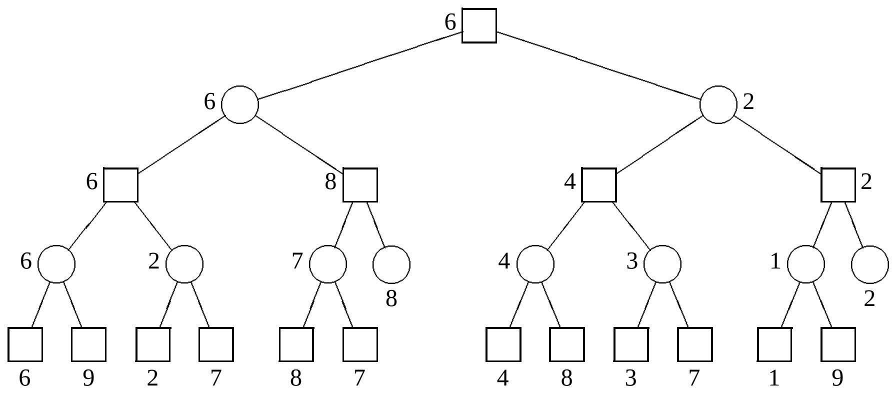

Figure 1: A game tree with f-values.

the move for each player to optimize the pay-off is the transition to a child node c such that $f(c) = f(n)$. In this way, MAX tries to maximize and MIN tries to minimize the profit of MAX. Therefore, an optimal play will proceed along a *critical path*, which is defined as a path from the root to a leaf such that $f(n)$ has the same value for all nodes n on the path. All nodes on this path have a game value equal to the game value of the root. A node on a critical path is called *critical*.

Overview

We conclude this section 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. 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 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.

In section 4 we introduce an alpha-beta based bounding procedure for the game value of a node n. Unlike alpha-beta, this procedure also takes into account information that has been gathered in earlier visits of the subtree of the game tree rooted in n. 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 5 we introduce two examples of algorithms that use our bounding procedure to efficiently search game trees. By the theory developed in section 4 these algorithms search no more nodes than alpha-beta. The results of some preliminary tests on their behavior are presented.

2 Solution Trees

In all game tree algorithms, the game tree is explored step by step, i.e., in each step a new node of the tree is visited. So, at each moment during execution of a game tree algorithm, there is a set of nodes which has been visited up to that moment. This subtree of the game tree will be called the *search tree*. We postulate that for every node in a search tree either all children are included or none. For the leaves in the search tree that are inner nodes in the game tree two tentative values are available, $-\infty$ and $+\infty$. Since the computation of $f(n)$, n a leaf, may be expensive, the tentative values $-\infty$ and $+\infty$ are also allowed for leaves in the search tree that are also leaves in the game tree. However, its game value $f(n)$ is allowed as well. So, in a leaf in the search tree we have either $f(n)$ or two tentative values. A leaf node of a search tree with tentative minimax values $+\infty$ or $-\infty$ is called *open*.

A search tree can be viewed as a some kind of game tree, so we can apply the minimax rule to this tree. If all open nodes in a search tree rooted in a node n are given tentative values $-\infty$, then the minimax value of the root is a lower bound to the game value. This value is denoted by $f^-(n)$. If the value $+\infty$ is assigned to all open nodes, then we obtain an upper bound $f^+(n)$. Moreover these bounds are the sharpest ones that can be derived from the search tree, cf. [Iba86].

Suppose we have a node n in a game tree, and we want to generate a search tree which establishes a non-trivial upper bound $f^+(n) \not= +\infty$. If n is a leaf in the game tree, we can simply take $f^+(n) = f(n)$. If n is a max node, we need non-trivial bounds in all children of n. Therefore the search tree should contain all children, none of which can be open. If n is a min node a non-trivial bound is needed for only one of its children, to lower $f^+(n)$ below $+\infty$.

In summary, to have a non-trivial upper bound f^+, we have to expand all successors of max nodes, and only one successor of min nodes. By recursively applying this prescription down to the leaves of the game tree, taking in the leaves p $f^+(p) = f(p)$, we get a minimal search tree generating a non-trivial upper bound. Such a tree is called a max solution tree.

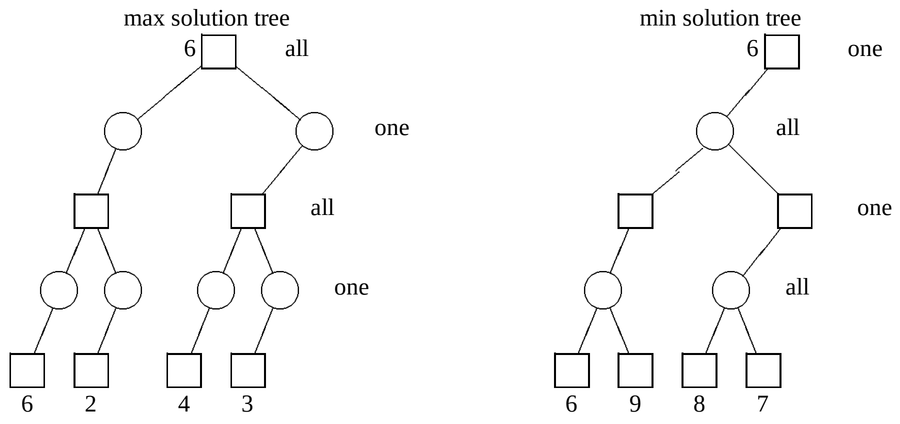

Figure 2: Bounds and Solution Trees

By analogy, we get a min solution tree by expanding *one* successor of max nodes, and *all* successors of min nodes. This yields a min solution tree defining a non-trivial lower bound f^- on $f(r)$. Figure 2 illustrates this idea. It features a max solution tree and a min solution tree which are subtrees from the game tree of figure 1 (open nodes are not shown). They define respectively an upper bound and a lower bound which is by construction in both cases equal to 6. (We have chosen the values in figure 2 to match with those in figure 3)

The notion of solution trees in game trees is analogous to solution trees in AND/OR trees [KK84, Sto79].) A max solution tree corresponds to an OR solution tree; a min solution tree to an AND solution tree.

Now suppose we have a search tree S with root n for which $f^+(n)$ is not trivial, i.e., $-\infty < f^+(n) < +\infty$. By induction we will show that S must contain at least one subtree which is a max solution tree with value $f^+(n)$ and that all other max solution trees in S define an upper bound $\geq f^+(n)$.

The basic case is that n is a leaf in S. Now $-\infty < f^+(n) < +\infty$ indicates that n is a leaf in the game tree and that $f^+(n) = f(n)$. Moreover, the only solution tree in S is S itself.

We now treat the induction step. Suppose n is a max node. Then for each child c of n we have $f^+(c) \le f^+(n)$ and there exists a child c' with $f^+(c') = f^+(n)$. By induction it is possible for each child c to find a max solution tree rooted in c with value equal to $f^+(c)$. Then the tree consisting of n and all these solution trees is a max solution tree with value $f^+(n)$.

Consider an arbitrary max solution tree through n. This tree contains for all children c of n a max solution tree rooted in c. The induction hypothesis tells us that for all these trees we have a value $\ge \max(f^+(c)) = f^+(n)$.

Finally, suppose n is a min node. Then $f^+(n) = \min\{f^+(c) \mid c \text{ child of } n\}$, so there is a child c' with $f^+(c') = f^+(n)$. By induction there is a max solution tree rooted in c' with value $f^+(c') = f^+(n)$. The solution tree obtained by appending n to this tree is the desired one.

Consider an arbitrary max solution tree through n. This tree contains one child c of n. By induction we have that every max solution tree through c defines an upper bound $\ge f^+(c) \ge f^+(n)$. Thus every solution tree through n has the same property.

Combining the above result with the observation that every game tree with value f is also a search tree with $f^+ = f^- = f$, we now obtain the following result from [Sto79]: For each game tree with value f there exists a max solution tree defining an upper bound equal to f. Such a max solution tree will be called optimal. (Of course, analogous results hold for lower bounds and min solution trees.)

Critical Trees

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. In [PK87, p. 462] an intuitive explanation 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. All this brings us to the following definition:

> *A critical tree is the union of an optimal max solution tree and an optimal min solution tree.*

Notice that, if the value of the game tree equals f, we have that an optimal max solution tree establishes an upper bound on the game value equal to f, and an optimal min solution tree establishes a lower bound of f. This shows that indeed a critical tree proves that the game value is equal to f.

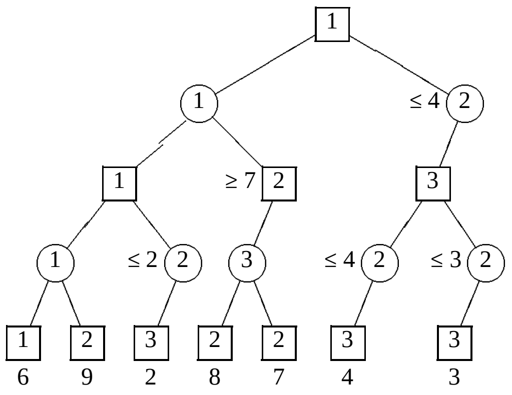

Figure 3: A Critical Tree with Node Types

Without proof we state the property that the intersection of an optimal max solution tree and an optimal min solution tree is a critical path.

Figure 3 is an example of a critical tree. It is the union of the solution trees of figure 2. 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.

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 on the critical path. As was noted before, only one successor is needed at these nodes to get a non-trivial bound. 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. All successors are needed at these nodes to get a non-trivial bound.

3 Hypothesis Testing in Game Tree Algorithms

Principal Variation Search

PVS [FF80, CM83, Rei89] and SCOUT [Pea80, Pea84] are two well known algorithms based on the *minimal window search* [FF80] or *bound-test* [Pea80] idea. The following description of PVS holds just as well for SCOUT.

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. 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 has a higher value. 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 max solution tree rooted in the parent of the leaf at the end of the PV, proving that its game value does not exceed 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 generates a min solution tree defining a lower bound on the value of the brother that is higher than v, and the proof fails. 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. (If the parent of the leftmost leaf in the tree is a min node, then the dual procedure has to be performed.)

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 PV. 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 bounding procedure that PVS uses is called *Null-Window-Search* or NWS. It is essentially a call to alpha-beta with $\alpha = \beta - 1$, which, assuming integer-valued leaves, assures that no leaf value can fall within the search window of $\langle \alpha, \beta \rangle$. The fact that alpha-beta not only finds the minimax value of a game tree, but can also be used for proving bounds on it, is obvious when we look at alpha-beta's postcondition. (See e.g. [PdB92, PdB93b].) The postcondition of $v \leftarrow$ alpha-beta(n, α, β) is

$$\begin{aligned} v \le \alpha \quad &\Rightarrow \quad \alpha \ge v = f^{+}(n) \ge f(n) \\ \alpha < v < \beta \quad &\Rightarrow \quad v = f^{+}(n) = f^{-}(n) = f(n) \\ v \ge \beta \quad &\Rightarrow \quad \beta \le v = f^{-}(n) \le f(n) \end{aligned}$$

If the first part of the postcondition holds, by the theory of the previous section, alpha-beta must have built a max solution tree. If the middle part holds a critical tree must have been built, and if the third part is established a min solution tree must have been constructed. By calling alpha-beta with $\alpha = \beta - 1$ only the first and last part of the postcondition can occur, which results in either an upper bound $f^+(n)$ or a lower bound $f^-(n)$ on n. The hypothesis that n, the current PV-candidate, is best, is thus either proven or refuted.

SSS-2

SSS-2 has been introduced in [PdB90, Pij91, PdB92] as an attempt to give an easier to understand, recursive description of SSS*. Bhattacharya & Bagchi 1993 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 works by establishing successive sharper upper bounds for the root, starting with an upper bound of $+\infty$. 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 $\geq \gamma$ will have been constructed.

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. The algorithm performs a sequence of calls to *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.

Some remarks on alpha-beta, NWS and Expand

In this subsection we look at the relation between alpha-beta, Null-Window-Search, and *expand*, the bounding procedure of SSS-2. 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. 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.

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

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

PVS and SCOUT are not interested in knowing whether the game value of the node investigated equals the input parameter γ, but only want to know whether $v < \gamma$ or $v \geq \gamma$. When NWS is applied to the brothers of a max PV-candidate we are interested to see whether there are any better brothers—here better means lower: they have a common min parent. (In case of a min PV-candidate, $v \leq \gamma$ or $v > \gamma$ is needed.) With NWS, this can be realized in the former case by calling alpha-beta($n, \gamma - 1, \gamma$), in the latter case by calling alpha-beta($n, \gamma, \gamma + 1$), as can be observed from the postconditions given above.

Now we turn our attention to SSS-2. Since SSS-2 performs successive calls to get a lower $f^+(r)$ after each call, it does not need the full generality of the three-part postcondition.

SSS-2's bounding procedure *expand* has the same postcondition as the first (max brother-) NWS call: $v < \gamma$ or $v \geq \gamma$. *Expand* and NWS have the same pre- and postcondition, both are called on empty search trees, and both expand nodes in a left-to-right order. Close inspection of the code of these procedures revealed that they are in fact identical, in the sense that they traverse the same nodes.

The dual of SSS-2 performs successive calls, starting with $-\infty$, to get a higher $f^-(r)$ after each call. *Dual-expand* has in this case the same postcondition as the second (min brother-) NWS: $v \leq \gamma$ or $v > \gamma$. *Dual-expand* does the same as NWS for brothers of a min PV-candidate.

In the algorithms of section 5 a postcondition of either a lower bound or an upper bound is not sufficient. We need a procedure that constructs a critical tree as well—cf. the middle part of alpha-beta's postcondition—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:

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

In the rest of this paper we will sometimes abbreviate a call to alpha-beta($n, \gamma - 1, \gamma + 1$) to alpha-beta($n, \gamma$).

4 A Bounding Procedure

Given the need for a procedure to construct solution trees to deliver bounds for proofs, we will focus our attention in this section on how to construct such a procedure. Luckily this turns out to be quite straightforward.

A Bounding Procedure for Non-Empty Search Trees

Alpha-beta can only be called on *open* nodes n. In figure 4 we propose 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. We will investigate how alpha-beta should 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, PdB93a, PdB93b] 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, as well as the correctness proof of S-alpha-beta are the same as for the heuristic bounds version of alpha-beta [PdB92].

In the previous section we noted the equivalence of expand(n, γ) and alpha-beta($n, \gamma - 1, \gamma$). An interesting observation is that diminish(n, γ) and S-alpha-beta($n, \gamma - 1, \gamma$) have this same postcondition. The difference is that diminish and S-alpha-beta are applied to non-open nodes. The analogy can be extended: if both procedures are applied to the same solution trees, they traverse the same nodes as well. We might say that diminish(n, γ) is to S-alpha-beta($n, \gamma - 1, \gamma$) what expand(n, γ) is to alpha-beta($n, \gamma - 1, \gamma$). We will formalize and prove this claim in a later paper.

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 expanded nodes is [CM83, Pea84]. The alpha-beta version

```
function S-alpha-beta($n$, $\alpha$, $\beta$) $\rightarrow$ $v$;
    if $\alpha \geq f^{+}(n)$ or $f^{-}(n) \geq \beta$ or $f^{+}(n) = f^{-}(n)$ then
        if $f^{-}(n) \geq \beta$ then return $f^{-}(n)$;
        else return $f^{+}(n)$;
    if $n$ = open then attach all children to search tree $S$;
    if $n$ = MAX then
        $\alpha' \leftarrow \max(\alpha, f^{-}(n))$;
        $g \leftarrow -\infty$;
        for $c \leftarrow$ firstchild($n$) to lastchild($n$) do
            $g \leftarrow \max(g$, S-alpha-beta($c$, $\alpha'$, $\beta$));
            $\alpha' \leftarrow \max(\alpha', g)$;
            if $g \geq \min(\beta, f^{+}(n))$ then exit for loop;
    if $n$ = MIN then
        $\beta' \leftarrow \min(\beta, f^{+}(n))$;
        $g \leftarrow +\infty$;
        for $c \leftarrow$ firstchild($n$) to lastchild($n$) do
            $g \leftarrow \min(g$, S-alpha-beta($c$, $\alpha$, $\beta'$));
            $\beta' \leftarrow \min(\beta', g)$;
            if $g \leq \max(\alpha, f^{-}(n))$ then exit for loop;
    update ($f^{-}(n)$, $f^{+}(n)$);
    return $g$;
```

Figure 4: S-alpha-beta

in [PdB92, PdB93a, PdB93b], 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 and re-expanding or revisiting actions on such nodes are not taken into account.) It follows that every sequence of S-alpha-beta calls with null windows surpasses the alpha-beta algorithm. Since SSS-2 consists of a number of NWS calls, this algorithm surpasses alpha-beta. Here, we rediscover a result in [PdB90] extending a weaker result in [Sto79]. Also, [Rei89, p. 99] shows that PVS surpasses alpha-beta for the same reason.

In the next section we present other algorithms built around NWS calls, and hence, surpassing alpha-beta.

5 Two New Solution Tree Algorithms

Drawing on the knowledge of bounding procedures and how they are used in other algorithms, we will present in this section a few examples of new ways of using bounding procedures. We do not wish to imply that these two algorithms are the only possible ways of using bounding procedures like S-alpha-beta to create new algorithms. We mention these instances only as examples of what is possible.

At the end of this paper, the results of some experiments to determine the performance of these algorithms will be discussed.

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($n, \gamma - 1, \gamma$). This can be formalized in the following pseudo-code fragment, which is adapted from [PdB90].

```
function SSS-2($n$) $\rightarrow$ $v$;
```

```
    $g \leftarrow +\infty$;
    **repeat**
        $\gamma \leftarrow g$;
        $g \leftarrow$ 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: $g \leftarrow -\infty$; $g \leftarrow$ S-alpha-beta$(n, \gamma, \gamma + 1)$.

A natural extension 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*, for 0 might be a good first approximation.

```
**function** SSS-0$(n, g) \to v$;
    **repeat**
        $\gamma \leftarrow g$;
        $g \leftarrow$ S-alpha-beta$(n, \gamma - 1, \gamma + 1)$;
    **until** $g = \gamma$;
    **return** $g$;
```

Apart from the different initialization of g and a choice for α and β yielding a three-part postcondition, the code for SSS-0 is the same as SSS-2.

If, after the first call to the bound-procedure, the first bound $g = f^+(n)$ or $g = f^-(n)$ turns out to be lower than our initial guess, then we have (by the postcondition) an f^+. We can perform SSS-2 (lower the upper bound) by calling S-alpha-beta. If, on the other hand, the first bound is higher than our guess, then we have an f^-. We can find the game value by performing the dual of SSS-2 (increase the lower bound), which comes down to calling S-alpha-beta as well.

SSS-0 may save work because it starts close 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 SSS-0 performs less steps of which the size is about the same, we would have found a profitable way to make use of heuristic knowledge in the form of a first guess.

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's bottom up live window is useless.

```
**function** SSS-4$(n) \to v$;
    $g \leftarrow +\infty$;
    **repeat**
        $\gamma \leftarrow g$;
        $g \leftarrow$ S-alpha-beta$(n, \gamma - 1, \gamma + 1)$;
        $g \leftarrow \max(g - STEPSIZE, f^-(n))$;
    **until** $g = \gamma$;
    **return** $g$;
```

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. This way of attack resembles somewhat the bisection method used to solve an equation $f(x) = 0$.

A variation might be to resort to a wide window call to S-alpha-beta, when the bounds have become relatively close: $v \leftarrow$ 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.

Average Time Complexity

Whether the last two algorithms, SSS-0 and SSS-4, 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-alpha-beta$(n, \gamma = f + x_1)$ expands fewer nodes than S-alpha-beta$(n, \gamma = f + x_2)$ where $x_1 > x_2$.

To get an impresssion of the average case performance we have conducted some experiments. We have called SSS-0 with different parameters on a number of artificially constructed uniform game trees, each with game value equal to 504. 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 100 different trees, using a procedure based on [MRS87, Hsu90]. The uniformly distributed leaf values ranged from 0 to 999. The set is made up of 20 different random seeds, whose results are averaged. We show 5 different levels of node ordering (the probability that the first successor is best): from $1/w$ (unordered), 25%, 50%, 90% to 100% (or perfectly ordered). On these 100 trees, we have called SSS-0 with 32 different values for *g*, the input parameter. On the *y*-axis the number of expanded nodes (inner plus leaves) is shown for different levels of node ordering. On the *x*-axis the values for the input parameter to SSS-0 are tabulated. (Figure 5 should not be confused with the "refutation wall" graph in [MRS87], which is a graph of *single* calls to NWS. The nodecount in figure 5 refers to calls to SSS-0, which consists of *a number of* calls to NWS/S-alpha-beta.)

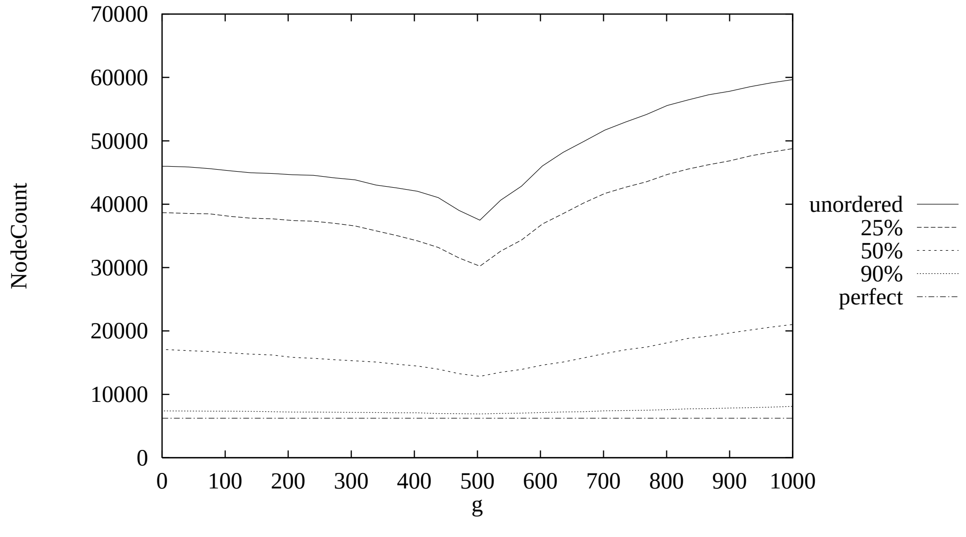

Figure 5: Node count of SSS-0, relative to input parameter

To the left of the graph you find the performance of Dual SSS-2, to the right primal SSS-2. Figure 5 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 and SSS-4 may work in principle, although the actual gains 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, and probably SSS-4, are preferable over SSS-2/SSS*. Whether this is the case is the subject of ongoing research.

6 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. 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 value of the *second highest* successor of the root.

Also, the space complexity of S-alpha-beta must be addressed. By deleting irrelevant parts of the search tree 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. Our hope is to base a taxonomy of (parallel) game tree algorithms on the way they generate and handle solution trees and critical trees.

We believe that the relation of our work to search enhancements like iterative deepening and transposition tables deserves some attention. Finally, the relation with other algorithms like Proof Number Search [AvdMvdH94], 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 game tree search.

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

[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.

[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.

[PdB93a] Wim Pijls and Arie de Bruin, *A framework for game tree algorithms*, Tech. Report EUR-CS-93-03, Dept. of Computer Science, Erasmus University, Rotterdam, The Netherlands, 1993.

[PdB93b] Wim Pijls and Arie de Bruin, *Generalizing alpha-beta*, Advances in Computer Chess 7, Maastricht (H.J. van den Herik, ed.), July 1993.

[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.

[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.