A framework for game tree algorithms

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

Abstract

A unifying framework for game tree algorithms is GSEARCH, designed by Ibaraki [Ibaraki 86]. In general, a relatively great deal of memory is necessary for instances of this framework. In [Ibaraki 91A] an extended framework, called RSEARCH, is discussed, in which the use of memory can be controlled.
In this paper variants of above frameworks are introduced, to be called Gsearch and Rsearch respectively. It is shown that, in these frameworks, the classical alpha-beta algorithm is the depth-first search instance and H* is a best first search instance. Furthermore two new algorithms, Maxsearch and Minsearch, are presented, both as best-first search instances. Maxsearch is close to SSS* [Stockman] and SSS-2 [Pijls-2], whereas Minsearch is close to dual SSS*.

1 Introduction

A game tree algorithm is an algorithm computing the minimax value of a game tree. We will recall a few well-known facts about game trees, search trees and algorithms defined on them.
A *critical path* in a game tree is a path from the root to a terminal such that the minimax value is constant along this path. Hence, in a game tree with minimax-value f, $f(x) = f$ for each node x on a critical path. A critical path represents an optimal strategy for each player. A node on a critical path is called *critical.*
In all game tree algorithms, the tree is explored step by step, i.e., in each step a new node of the tree is visited or generated, depending on whether the tree is given explicitly or defined implicitly by some rule. So, at each moment during execution of a game tree algorithm, there is a set of nodes which has been generated up to that moment. This subtree of the game tree will be called the *search tree.* In the sequel we will only consider search trees with the property that for each node either all its children are included in the search tree or none. All algorithms in this paper will generate such search trees.

When a part of the game tree has been explored, an upper and a lower bound

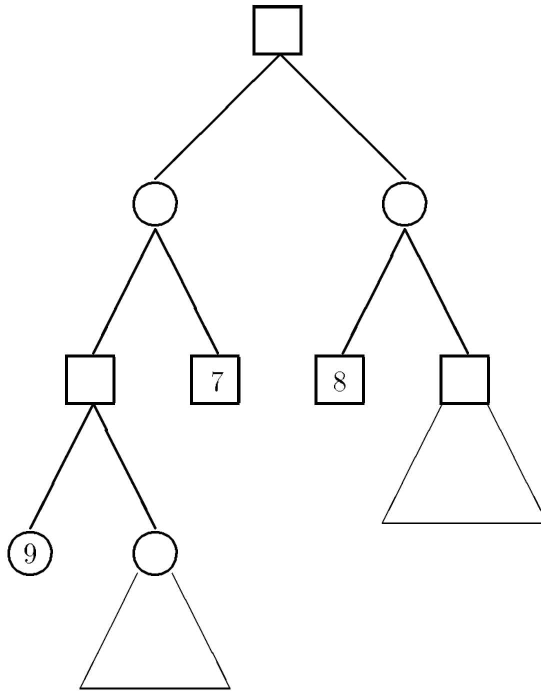

Figure 1:
A search tree with boundary values for the root.

value may be available for the value of some node already explored. Referring to the fragment of Figure 1, we see that we have obtained some knowledge on the game value of the root. It can be concluded that this value is smaller than or equal to 8 and greater than or equal to 7. In the figure, the normal conventions hold; squares are max nodes and circles are min nodes. The trapezia stand for subtrees not yet generated.

For some game trees, heuristic information on the minimax value $f(n)$ is available for any node n. This information can be expressed by two functions, U and L mapping the nodes of the game tree into the real numbers, such that $U(n) \geq f(n) \geq L(n)$ for any node n. The heuristic functions thus denote an upper bound and a lower bound respectively of the minimax value. In a terminal node n of the game tree, we require $U(n) = f(n) = L(n)$. Conversely, whenever $U(n) = f(n) = L(n)$, this node must be a terminal.
We impose the assumption that $U(c) \leq U(n)$ for every child c of a given max node n, and $L(c) \geq L(n)$ for every child c of a given min node. If heuristic information is discarded or not available at all, we define $U(n) = +\infty$ and $L(n) = -\infty$ for every non-terminal node n.

Suppose for each leaf node p in a search tree an upper bound $U_b(p)$ and a lower bound $L_b(p)$ for the game value $f(p)$ is given. Then these functions can be extended to the other nodes in the tree.

Definition 1.1 *Given values $U_b(n)$ and $L_b(n)$ for every leaf node n in a search tree, the values $U_b(n)$ and $L_b(n)$ are defined for every inner node n of the search tree as:*

$$\begin{aligned} U_b(n) &= \min(U(n), \max\{U_b(c) \mid c \in C(n)\}), && \textit{if } n \textit{ is an inner max node,} \\ &= \min(U(n), \min\{U_b(c) \mid c \in C(n)\}), && \textit{if } n \textit{ is an inner min node,} \end{aligned}$$

$$\begin{aligned} L_b(n) &= \max(L(n), \max\{L_b(c) \mid c \in C(n)\}), && \textit{if } n \textit{ is an inner max node,} \\ &= \max(L(n), \min\{L_b(c) \mid c \in C(n)\}), && \textit{if } n \textit{ is an inner min node.} \end{aligned}$$

where $C(n)$ denotes the set of children of n.

It is easily seen that for any node x in the search tree:

$$L_b(x) \leq f(x) \leq U_b(x), \tag{1.1}$$

provided of course that this property holds in the leaves.

We now give an important characteristic of the algorithms to be presented. When a node p is appended, the values $L_b(p)$ and $U_b(p)$ are set to $-\infty$ and $+\infty$. Such a node will be called *open*. In a later stage an open node may be *developed*, which implies the computation of $U(p)$ and $L(p)$ and the generation of the children of p, if any. So the computation of $U(p)$ and $L(p)$ and the generation of the children of p are coupled, i.e., performed in consecutive steps. Consequently, all open nodes p are leaves in the search tree with $U_b(p) = +\infty$ and $L_b(p) = -\infty$. If a leaf p in the search tree is not open, it has already been developed and it must be a terminal of the game tree.

For such search trees, the formula in Definition 1.1 can be simplified. Since $U(n) \geq U(c)$ for every child c of a given max node n, we have, if n is a max node with only non-open children, that

$$U_b(n) = \max\{U_b(c) \mid c \in C(n)\} \tag{1.2}$$

and similarly, if n is a min node with only non-open children, that

$$L_b(n) = \min\{L_b(c) \mid c \in C(n)\}. \tag{1.3}$$

We conclude this section with an outline of this rest of this paper. In Section 2 we will present a framework, called *Gsearch*. In general Gsearch requires a lot of memory. Therefore, a second framework, called *Rsearch*, is presented in Section 3. In this framework, the use of memory is controlled. Gsearch and Rsearch can be seen as special cases of each other. Both frameworks are actually algorithms

including non-deterministic statements. By specifying these statements in different ways, different instances of the framework can be obtained. In Section 4, we show several fashions to specify the non-deterministic statements of Gsearch or Rsearch respectively. Section 5 shows that the well-known alpha-beta algorithm [Knuth] is the depth-first instance of both Gsearch and Rsearch. Hence our frameworks are generalizations of alpha-beta. Section 6 presents Maxsearch, a best-first instance of Gsearch and Rsearch. This algorithms strongly resembles SSS* [Stockman]. Section 8 concludes this paper with mostly historical remarks.

2 The Gsearch framework

In this section we first define a criterion for a node to be pruned. Next, this criterion leads us to a general game tree algorithm, called *Gsearch*.
During execution of the algorithm, enough information may be obtained to conclude that a node cannot be critical. If the intersection of $[L_b(c), U_b(c)]$ and $[L_b(n), U_b(n)]$, c a child of root n, is empty, then $f(c) \neq f(n)$, and hence c is not critical. If a child c of n is critical, i.e., if $f(c) = f(n)$, then at any time during execution $f(c)$ lies in the intersection of the intervals $[L_b(c), U_b(c)]$ and $[L_b(n), U_b(n)]$ and thus in the interval with end points $\max(L_b(c), L_b(n))$ and $\min(U_b(c), U_b(n))$. Similarly, we can indicate for a child d of c an interval, which is the intersection of three intervals. So, at any time during execution for each node m of the search tree, rooted in n, we have an intersection set, such that, if m is critical, $f(m)$ lies in this intersection set. After these observations, we come to the following definition.

Definition 2.1 *The functions $\mathcal{U}$ and $\mathcal{L}$ are defined for a node n in the search tree as:*

$$\begin{aligned} \mathcal{U}(n) &= \min\{U_b(m) \mid m \in ANC(n) \vee m = n\} \\ \mathcal{L}(n) &= \max\{L_b(m) \mid m \in ANC(n) \vee m = n\}, \end{aligned}$$

where $ANC(n)$ denotes the set of (proper) ancestors of n.

It follows from Definitions 1.1 and 2.1 that, for each child c of a max node n, $\mathcal{U}(n) \geq \mathcal{U}(c)$ and $\mathcal{L}(n) = \mathcal{L}(c)$. Moreover, for at least one child c of n, $\mathcal{U}(n) = \mathcal{U}(c)$. Similarly, for each child c of a min node n, $\mathcal{U}(n) = \mathcal{U}(c)$, $\mathcal{L}(n) \leq \mathcal{L}(c)$ and for at least one child c of n, $\mathcal{L}(n) = \mathcal{L}(c)$.

We have a few theorems with respect to these new quantities. Theorem 2.1 is the main theorem in our theory on game tree algorithms.

Lemma 2.1 *Let a search tree T be given with root n. For every value $f \in [L_b(n), U_b(n)]$, the search tree can be extended such that $f(n) = f$.*

Proof
This lemma is proved by induction on the height of the search tree.
If n has height $= 0$, then n is a leaf in T and hence, either n is open, or n has been developed. In the former case, $L_b(n) = -\infty$ and $U_b(n) = \infty$, and we

define that n is a terminal with $f(n) = f$. In the latter case, we already have $f(n) = L_b(n) = U_b(n)$.
Now, we consider the situation that n is not a leaf. Assume n is max node; (the alternate case is similar). There is at least one child of n, say c_0, such that $U_b(c_0) \geq U_b(n)$. For every child c of n we have $L_b(c) \leq L_b(n)$. By the induction hypothesis, the search tree can be extended such that $f(c_0) = f$ and, for any child $c \neq c_0$, $f(c) = L_b(c)$. It follows that $f(n) = f$. □

Theorem 2.1 *Let a node m be given with $\mathcal{L}(m) \leq \mathcal{U}(m)$ in a search tree T with root n. For every value f we have: $f \in [\mathcal{L}(m), \mathcal{U}(m)]$ if and only if the search tree can be extended such that m is critical and $f(m) = f$.*

Proof
if part
For every node q on the path from n through m we have in T: $f(q) = f$ and $L_b(q) \leq f(q) \leq U_b(q)$. Hence the result follows.

only-if part
Consider path P from root n to m. For every child c of a node q on P with c outside P, $L_b(c) \leq L_b(q) \leq \mathcal{L}(m)$, in case q is max node, and $U_b(c) \geq U_b(q) \geq \mathcal{U}(m)$ in case m is a min node. By Lemma 2.1, we can extend the subtree rooted in c, c a child of a node on P and c outside P, such that, $f(c) = L_b(c)$ in case q is max node and $f(c) = U_b(c)$ in case q is a min node. By Lemma 2.1, we can extend the subtree rooted in m such that $f(m) = f$. In the extended subtree, we prove for every node q on P, by induction on the length of path from q to m, that $f(q) = f$. □

Note
Lemma 2.1 can be enhanced: if $f \in (L_b(n), U_b(n))$ then there is a unique critical path. Likewise, Theorem 2.1 can be enhanced: if $f \in (\mathcal{L}(m), \mathcal{U}(m))$, then m lies on a unique critical path.

Due to Theorem 2.1, the minimax value f for the root cannot be obtained, as long as at least one node m in the search tree satisfies $\mathcal{L}(m) < \mathcal{U}(m)$. It follows that every game tree algorithm should aim at achieving a search tree such that $\mathcal{L}(m) \geq \mathcal{U}(m)$ for every node m in the search tree.

Now, we present some dynamical results, dealing with updating processes.

Lemma 2.2 *Let p be an open node in a search tree T_1 and let n be an ancestor of p. When p is developed and the new tree is called T_2, then*

$$U_{b1}(n) \geq U_{b2}(n)$$

and

$$L_{b1}(n) \leq L_{b2}(n),$$

where the subscripts 1 and 2 refer to search tree T_1 and T_2 respectively.

Proof
This Lemma can be proven by induction on the length of the path from n to p. □

Lemma 2.2 states that, in a given node, the U_b-function is non-increasing and the L_b-function is non-decreasing, while building the search tree. An analogous statement holds for the $\mathcal{U}$- and $\mathcal{L}$-function.

Lemma 2.3 *Suppose that an open node of a given search tree T is developed. If the U_b-value changes in a series of nodes $m_0, m_1, \ldots, m_k$, m_i the father of m_{i-1} for $1 \leq i \leq k$, then:*

$$[U_{b2}(m_{i-1}), U_{b1}(m_{i-1})] \supseteq [U_{b2}(m_i), U_{b1}(m_i)], \quad 1 \leq i \leq k,$$

where U_{b1} and U_{b2} denote the old and the new value respectively. If the L_b-value changes in a series of nodes $m_0, m_1, \ldots, m_{k'}$, m_i the father of m_{i-1} for $1 \leq i \leq k'$, then:

$$[L_{b1}(m_{i-1}), L_{b2}(m_{i-1})] \supseteq [L_{b1}(m_i), L_{b2}(m_i)], \quad 1 \leq i \leq k',$$

where L_{b1} and L_{b2} denote the old and the new value respectively.

Proof
We give a proof for the U_b-function. (The alternate case is similar.) Suppose that m_i, $1 \leq i \leq k$, is a max node. Notice that, apart from m_{i-1}, no other child of m_i can be open, because, in that case, we would have $U_{b1}(m_i) = U_{b2}(m_i) = U(m_i)$. We distinguish two cases. In case m_{i-1} is open and m_{i-1} is developed, then $U_{b1}(m_{i-1}) = \infty$ and $U_{b1}(m_i) = U(m_i)$. In case m_{i-1} is not open, (1.2) holds. If $U_b(m_i)$ changes, when a proper descendant of m_{i-1} is developed, we must have $U_{b1}(m_i) = U_{b1}(m_{i-1})$. In both cases, after the updates, we have by (1.2) that $U_{b2}(m_i) \geq U_{b2}(m_{i-1})$. Hence the result follows.
Suppose that m_i, $1 \leq i \leq k'$ is a min node. We have $U_{b1}(m_i) \leq U_{b1}(m_{i-1})$ by definition and, since $U_b(m_i)$ changes, $U_{b2}(m_i) = U_{b2}(m_{i-1})$. Hence the result follows. □

Lemma 2.4 *Suppose that an open descendant of a node n in a search tree T is developed. If $U_b(n)$ and $\mathcal{U}(n)$ change, then afterwards $\mathcal{U}(n) = U_b(n)$. If $L_b(n)$ and $\mathcal{L}(n)$ change, then afterwards $\mathcal{L}(n) = L_b(n)$.*

Proof
We give a proof only for the lower bound functions. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ denote the $\mathcal{L}$-function before and after developing respectively.
If $L_b(q)$ does not change for a node $q \in ANC(n)$, then afterwards $L_b(q) \leq \mathcal{L}_1(n) < \mathcal{L}_2(n)$. If $L_b(q)$ changes for a node $q \in ANC(n)$, then afterwards, by Lemma 2.3, $L_b(q) \leq L_b(n)$. It follows that $\mathcal{L}_2(n) = L_b(n)$. □

Theorem 2.2 *Let a node n be given with $\mathcal{L}(n) < \mathcal{U}(n)$ and let a descendant m of n be given with $\mathcal{L}(m) \geq \mathcal{U}(m)$. When a descendant of m is developed, $\mathcal{L}(n)$ and $\mathcal{U}(n)$ are unaffected.*

Proof
Suppose $\mathcal{L}(n)$ is affected. Let the boundary values before developing be marked with subscript 1 and after developing with subscript 2. Due to the premiss, we have:

$$\mathcal{L}_1(n) < \mathcal{U}_1(n) \tag{2.1}$$

By the assumption that $\mathcal{L}(n)$ is affected, we have

$$\mathcal{L}_1(n) < \mathcal{L}_2(n) \tag{2.2}$$

Let the path from m to n be given by $m = m_0, m_1, \ldots, m_k = n$. If $L_b(n)$ does not change, then $\mathcal{L}(n)$ would not change. Hence $L_b(n)$ changes and consequently for $0 \leq i \leq k$, $L_b(m_i)$ changes. By Lemma 2.4, $L_{b1}(m_i) \leq L_{b1}(m_k)$ for $0 \leq i \leq k$. We conclude:

$$\mathcal{L}_1(m) = \mathcal{L}_1(n) \tag{2.3}$$

Again, by Lemma 2.3, $L_{b2}(m_i) \geq L_{b2}(m_k)$, $0, \leq i \leq k$. Furthermore, $L_{b2}(m_i) \leq f(m_i) \leq U_{b1}(m_i)$, $0 \leq i \leq k$. We conclude:

$$L_{b2}(n) \leq L_{b2}(m_i) \leq U_{b1}(m_i), \;\; 0 \leq i \leq k \tag{2.4}$$

Combining this result with Lemma 2.3, we have:

$$\mathcal{L}_2(n) = L_{b2}(n) \leq U_{b1}(m_i), \;\; 0 \leq i \leq k \tag{2.5}$$

Using (2.5) and the definition of $\mathcal{U}$-function, we come to:

$$\mathcal{U}_1(m) \geq \min(\mathcal{U}_1(n), \mathcal{L}_2(n)) \tag{2.6}$$

It follows from (2.1) and (2.2) and (2.6), that $\mathcal{U}_1(m) > \mathcal{L}_1(n)$. Combining this result with (2.3), we obtain $\mathcal{U}_1(m) > \mathcal{L}_1(m)$. We conclude that $\mathcal{L}(n)$ can only be affected, if $\mathcal{L}_1(m) < \mathcal{U}_1(m)$.
A similar reasoning holds, when we suppose that $\mathcal{U}(n)$ is affected. □

As mentioned earlier, we aim at achieving $\mathcal{L}(m) \geq \mathcal{U}(m)$ for every node m in the search tree. Theorem 2.2 shows that for open nodes p with $\mathcal{L}(p) \geq \mathcal{U}(p)$, it makes no sense to be developed any more. Realize that, when the inequality $\mathcal{L}(m) \geq \mathcal{U}(m)$ is ever reached for a node m, this inequality will stay valid, since the $\mathcal{L}$-function is non-decreasing and the $\mathcal{U}$ is non-increasing during execution. Therefore we introduce the following notion.

Definition 2.2 *A node n is called live if $\mathcal{L}(n) < \mathcal{U}(n)$.*

We showed earlier that, if a node is not a leaf in the search tree, there is at least one child c of n such that $\mathcal{L}(c) = \mathcal{L}(n)$ and $\mathcal{U}(c) = \mathcal{U}(n)$. It follows, that open live nodes are available as at least one node is live anywhere in the search tree. We conclude from Theorem 2.2, that, in order to make sure that no node in the search tree is live any more, we must develop open live nodes. The foregoing theory results into the game tree algorithm *Gsearch*, which is presented in Figure 2. The guard in the while loop can be replaced by another equivalent guard: *any*

```
procedure Gsearch(in n:node; out f:real);
S:=the tree consisting of only node n;
develop(n);
while Lb(n)< Ub(n) do
      [ select non-deterministically an open live node p of S;
        develop(p);
        update Ub(q) and Lb(q) for each q in ANC(p);
        update 𝒰(m) and ℒ(m) for each m in S;
      ]
f:=Ub(n);

procedure develop(p);
Ub(p):=U(p);
Lb(p):=L(p);
if p is not a terminal then
   [ for c := firstchild(p) to lastchild(p) do
         [ append c to S;
           Ub(c):= +∞;
           Lb(c):= -∞;
         ]
   ]
```

Figure 2: The Gsearch procedure.

node in the search tree is live.

We conclude this section with a theorem on updating boundary functions, which simplifies updating the $\mathcal{L}$- and $\mathcal{U}$-function.

Theorem 2.3 *Suppose a node n in a search tree T_1 is given. Suppose that T_1 is a subtree of another search tree T_2, such that all nodes in $T_2 \backslash T_1$ are descendants of n. Then $\mathcal{L}_2(n) = \max(L_{b2}(n), \mathcal{L}_1(n))$ and $\mathcal{U}_2(n) = \min(U_{b2}(n), \mathcal{U}_1(n))$, where the subscripts 1 and 2 refer to search tree T_1 or T_2 respectively.*

Proof
If T_2 is obtained from T_1 by developing only one descendant of n, we have $\mathcal{L}_1(n) = \mathcal{L}_2(n)$ or otherwise, due to Lemma 2.4, $\mathcal{L}_2(n) = L_{b2}(n)$. In that case, the result holds. If T_2 is obtained from T_1 by a sequence of steps, each developing one descendant of n, then the theorem is proved by induction on the number of steps. □

3 The Rsearch algorithm

For many instances the size of the search tree is too large to be contained in memory. In this section *Rsearch* is introduced as an alternative procedure for

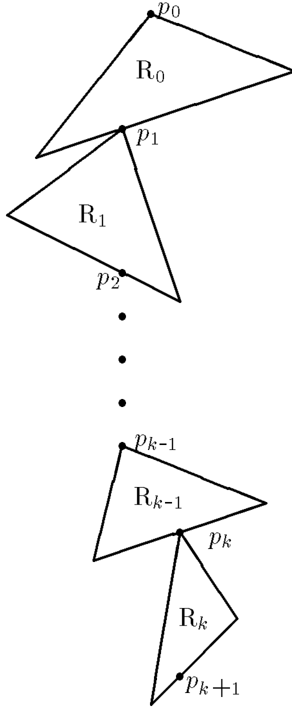

Figure 3: The memory use of Rsearch

Gsearch. For Rsearch the use of memory can be controlled. Some nodes in the game tree must be appointed to be so-called *pseudo-terminals*. It is assumed that a criterion exists, for deciding whether a node is pseudo-terminal or not. For example, it is possible to establish that each node in the tree with depth $d, 2d, 3d, \ldots$, where d is a given constant, is a pseudo-terminal. The Rsearch algorithm is an instance of Gsearch, in which pseudo-terminals play a special role. When a pseudo-terminal p is selected, in the next iterations only descendants of p are selected, until p is not live any more. In case a descendant p' of p is also a pseudo-terminal, the same rule holds for p'. This concept suggests to implement Rsearch recursively. So, we will write a recursive procedure *Rsearch*$(p, \ldots, L_b, U_b)$, where L_b and U_b are output parameters with the property $U_b = U_b(p)$ and $L_b = L_b(p)$ on termination.

During execution of the recursive call *Rsearch(p, ...)*, the subtree rooted in p with (pseudo)-terminals as leaves is called the *local search tree*. A local search tree is denoted in general by R. Assuming that, after a call *Rsearch*$(p \ldots)$, the local search tree with root p, is deleted from memory, the global search tree in memory can be depicted schematically according to Figure 3. This figure is taken from [Ibaraki 91A]. The nodes $p_1, p_2, \ldots, p_{k-1}, p_k, p_{k+1}$ are assumed to be pseudo-terminals. The local search trees rooted in p_i, $0 \leq i \leq k$ are denoted by R_i.

```
procedure Rsearch(in n:node, lower, upper:real; out Lb, Ub:real);
R:=the tree consisting only of node n;
develop(n);
while max(Lb(n),lower) < min(Ub(n),upper) do
      [ select non-deterministically an open live node of R;
        if p is a pseudo terminal
                then [ lower':=ℒ(p);
                       upper':=𝒰(p);
                       Rsearch(p, lower', upper', Lb, Ub);
                       Ub(p) := Ub;
                       Lb(p) := Lb;
                     ]
                else develop(p);
        update Ub(q) and Lb(q) for each q in ANCR(p);
        update 𝒰(m) and ℒ(m) for each m in R;
      ]
Ub:=Ub(n);
Lb:=Lb(n);
```

Figure 4: The procedure Rsearch.

As mentioned above, during Rsearch, in each iteration a descendant of a pseudo-terminal under consideration is selected. Therefore, a call *Rsearch*$(n, \ldots)$ with n the root of a game tree, can be viewed as a special case of *Gsearch* call. For a recursive call of *Rsearch* with p_i, $1 \leq i \leq k$, as node parameter, we have two other input parameters, called *upper* and *lower* respectively. On call, these values are equal to $\mathcal{U}(p_i)$ and $\mathcal{L}(p_i)$ respectively, where the $\mathcal{U}$- and $\mathcal{L}$-function are defined with respect to the global solution tree. However, we introduce a new definition for the $\mathcal{U}$ and $\mathcal{L}$-function and the *live* notion, such that the new definition only use local quantities, i.e., quantities which can be taken from the local search tree.

Definition 3.1 *During execution of Rsearch, the functions* $\mathcal{U}$ *and* $\mathcal{L}$ *are defined for a node* n *in a local search tree* R *as:*

$$\begin{aligned} \mathcal{U}(n) &= \min(upper, \min\{U_b(m) \mid m \in ANC_R(n) \vee m = n\}) \\ \mathcal{L}(n) &= \max(lower, \max\{L_b(m) \mid m \in ANC_R(n) \vee m = n\}), \end{aligned}$$

where $ANC_R(n)$ *denotes the set of (proper) ancestors of* n *in* R*. A node* n *is called live if* $\mathcal{L}(n) < \mathcal{U}(n)$*.*

Using Theorem 2.3, we can prove that $\mathcal{U}$ equals the same quantity regardless whether Definition 3.1 is applied in a local search tree, or Definition 2.1 is applied in the global search tree. A similar statement holds for $\mathcal{L}(n)$.

The procedure *Rsearch* is described in Figure 4. The $\mathcal{L}$- and $\mathcal{U}$-function in this procedure refer to Definition 3.1. Note that the guard in the while loop of the

body of Rsearch is equivalent to the condition: $\mathcal{L}(n) < \mathcal{U}(n)$.
It follows immediately from the guard that the *Rsearch* procedure has the following postcondition, provided that the precondition $lower < upper$ is satisfied:

Postcondition
$lower < L_b = f(n) = U_b < upper$ or
$L_b \leq f(n) \leq U_b \leq lower$ or
$upper \leq L_b \leq f(n) \leq U_b$,
and (in all cases) $L_b = L_b(n)$ and $U_b = U_b(n)$.

For a given game tree with root n, the minimax value is computed by the call $Rsearch(n, -\infty, +\infty, L_b, U_b)$. On exit, we have $U_b = f(n) = L_b$. Hence Gsearch can be viewed as a special instance of Rsearch. If Rsearch is invoked with finite values for the input parameters *lower* and *upper*, we obtain the exact minimax value only if this value lies in the open interval (*lower, upper*).
If the leftmost open live node is selected in each iteration of Rsearch, the order in which nodes are selected is not influenced by the occurrence of pseudo-terminals. So the instance of Rsearch which selects the leftmost open live node in each iteration, selects the same nodes in the same order, independently of the occurrence of pseudo-terminals.

In the description of Rsearch in Figure 4 the parameters $upper'$ and $lower'$ of a subcall *Rsearch(p, lower', upper', ...)* are set to $\mathcal{U}(p)$ and $\mathcal{L}(p)$. However, it is possible to use alternative input values for such a subcall. The input values for the recursive subcall in Figure 4, are determined according to Definition 3.1 or, equivalently, to, Definition 2.1. These values will be denoted by $upper'_1$ and $lower'_1$.
Alternative input values, denoted by by $upper'_2$ and $lower'_2$ are

$upper'_2 := \min(U_b(m) \mid m \in ANC(p)$ *and* m *is a min node*)
$lower'_2 := \max(L_b(m) \mid m \in ANC(p)$ *and* m *is a max node*)

In general, $upper'_1 \leq upper'_2$ and these values differ, if the minimum in $\min\{U_b(m) \mid m \in ANC(p)\}$ is achieved in a max node. If any node c is not open and n, the father of c,is a max node, then $U_b(c) \leq U(c) \leq U(n)$. It follows that, since p is open, the minimum in $\min\{U_b(m) \mid m \in ANC(p)\}$ can be achieved in a max node, say m_0, only if m_0 is the father of p. So, $upper'_1$ and $upper'_2$ only differ if $upper'_1 = U_b(m_0)$, where m_0 is a max node and the father of p. Since $U(p) \leq U(m_0)$ by assumption, and p is open, we have $U(p) \leq U(m_0) = U_b(m_0) = upper'_1$. After the develop procedure in the subcall $Rsearch(p, \ldots)$ we have $\mathcal{U}(p) = \min(U_b(p), upper') = \min(U(p), upper')$. The last value is equal to $U(p)$, if $upper'_1$ and $upper'_2$ differ. We conclude that the difference between $upper'_1$ and $upper'_2$ is no longer relevant, after p has been developed.
A similar reasoning holds for the $lower'$ parameter.

4 Instances of the framework

In this section we will consider several rules for selecting the node, which is to be developed. Each selection rule corresponds to an instance of the Gsearch or Rsearch respectively.

During Gsearch or Rsearch, we can search in a top down fashion the next open live node, to be expanded in the main loop. Starting from the root, we descend into the tree to an open node along a path consisting of only live nodes. We determine the successor node in this path according to some rule which we follow consistently while descending along the path. We consider four possibilities for choosing the next node in the path leading from the root to an open live node. Each possibility defines an instance of Gsearch or Rsearch. In each instance the choice may differ for a max and a min node. The following scheme lists a number of practical choices. The name of each instance is explained below.

instance	max node	min node
alpha-beta	leftmost	leftmost
Maxsearch	same $\mathcal{U}$-value	same $\mathcal{U}$-value
Minsearch	same $\mathcal{L}$-value	same $\mathcal{L}$-value
H**	same $\mathcal{U}$-value	same $\mathcal{L}$-value

Ties are assumed to be solved in favor of the leftmost node. Since the children of a live min node or the children of a live max node have the same $\mathcal{U}$-value or the same $\mathcal{L}$-value respectively as their fathers, a tie occurs in every min node during Maxsearch and in every max search during Minsearch. Therefore for Maxsearch in a min node and for Minsearch in a max node, the above choice can be replaced by *leftmost*.

The Rsearch instance which, descending top-down to search an open live node, chooses the leftmost live node, selects the leftmost open live node of both the local and the global search tree. In Section 3, we argued that this instance runs in the same way, independently from the place of the pseudo-terminals in the game tree. Since the leftmost open live node of the global search tree is selected, this Rsearch instance is equivalent to the Gsearch instance selecting the leftmost open node. In Section 5 we shall show that the well known alpha-beta algorithm is the Rsearch instance, where every node is a pseudo-terminal and the leftmost open live node is selected in each iteration. We conclude that the alpha-beta algorithm can be characterised as the Gsearch instance selecting the leftmost open live node, or the Rsearch instance selecting the leftmost open live node. The Rsearch characterisation holds, regardless the place of pseudo-terminals in the game tree.
Moreover, if every node in a game tree is a pseudo-terminal, then alpha-beta is the same algorithm as Maxsearch, Minsearch and H**.
The H** algorithm is a variant of H*(see [Ibaraki 87]), which in its turn is a variant of B* [Berliner]. Note that H** descends along a path with constant

window.
Beside alpha-beta and B*, another well-known game tree algorithm is SSS*. In [Pijls-2] an improved version, called SSS-2 is introduced. Similar to dual SSS*, also dual SSS-2 can be defined. Maxsearch and Minsearch strongly resemble SSS-2 and dual SSS-2 respectively. Maxsearch has been introduced in [Pijls-1].

5 The alpha-beta algorithm

In this section, we will construct the instance of Rsearch, for which every node in the game tree is a pseudo-terminal and in which open live nodes are selected from left to right. It will appear that the Rsearch instance constructed in this section is identical to the alpha-beta algorithm, presented in [Pijls-2]. This new version of alpha-beta is an extended version of the original alphabeta algorithm[Knuth], since it does take into account heuristic functions whereas the old alphabeta algorithm does not. When the heuristic functions are discarded, the original alphabeta algorithm is recovered.

Like any Rsearch instance, the instance of Rsearch to be constructed, has input parameters n and *lower* and *upper* and ouput parameters U_b and L_b. The main loop of Rsearch is started only if parameter n is live after being developed. In the code to be presented in this section, explicit occurrences of the *develop* procedure are omitted and appending new nodes to the local search tree is implemented implicitly. Before the main loop starts, the following code is executed:

if max(lower, $L(n)$) $\geq$ min(upper, $U(n)$) then
  [$U_b := U(n)$*;*
    $L_b := L(n)$*;*
    exit procedure;
  ]

Due to these statements, at least one iteration is performed, when the main loop starts. For the main loop of Rsearch we have distinct pieces of code for the case that parameter n is a max and n is min node respectively. Let us consider the situation that n is a max node. The code for this situation is shown below. When the main loop starts, n is assumed to be developed and therefore we have at that time, $\mathcal{U}(n) = \min(U(n), upper)$. Realize that $U_b(n)$ and and hence $\mathcal{U}(n)$ is unaffected, as long as at least one child of n is open. Only $\mathcal{L}(n)$ is updated after each iteration. For any open child c of n, $\mathcal{L}(c) = \mathcal{L}(n)$ and $\mathcal{U}(c) = \mathcal{U}(n)$. We use in the recursive calls the $lower_2$ and $upper_2$ parameters, defined in Section 3. As argued at the end of Section 3, after n has been developed, $\mathcal{L}(n) = \max(L(n), lower_2)$. In the following code, variable $\mathcal{L}$ denotes $\mathcal{L}(n)$ at any time.

```
if type(n) =max then
  [ ℒ := max(L(n), lower);
    for c:=firstchild(n) to lastchild(n) do
        [ Rsearch(c, ℒ, upper, L'_b, U'_b) ;
          if L'_b > ℒ then ℒ := L'_b; (1)
          if L'_b = U(n) or L'_b ≥upper then exit for loop;(2)
        ]
  ]
```

In (1), $\mathcal{L}(n)$ is updated. In (2), it is checked whether n preserves its liveness. The liveness can be lost due to $L'_b \geq \mathcal{U}(n)$, which is equivalent to $L'_b \geq$*upper* or $L'_b = U(n)$. (Notice that $L'_b > U(n)$ cannot happen.)

When the loop has terminated, $U_b(n)$ and $L_b(n)$ must be computed, in order to determine the output values. For this computation, we distinguish four cases.

a) Suppose that $\mathcal{L} = L(n) >$*lower* at the start of the *for* loop and suppose that every inner call in the for loop ends with $L'_b \leq \mathcal{L}$. So the variable $\mathcal{L}$ is never updated. Due to the postcondition of Rsearch, the condition $L'_b \leq \mathcal{L}$ implies $U'_b \leq \mathcal{L}$. Hence, every inner call ends with $U'_b \leq \mathcal{L}$, Consequently, since $\mathcal{L} < U(n)$ also $U'_b < U(n)$. We conclude that on termination of the for loop

$$U_b(n) = \textit{maximum of all intermediate } U'_b\textit{-values} \quad (5.1)$$

Since for every inner call $U'_b \leq \mathcal{L}(n) = L(n)$, we have: $U_b(n) \leq L(n) \leq L_b(n)$ and hence

$$L_b(n) = f(n) = U_b(n). \quad (5.2)$$

Notice that we also have *lower*$< L(n) = f(n) <$*upper*.

b) Suppose that $\mathcal{L} =$*lower*$\geq L(n)$ at the start of the *for* loop and suppose that every inner call in the for loop ends with $L'_b \leq \mathcal{L}$. Again, due to the postcondition of Rsearch, $L'_b \leq \mathcal{L}$ implies $U'_b \leq \mathcal{L}$ and hence $U'_b \leq$*lower*$<$ $U(n)$ for every inner call. Again (5.1) holds and further *lower*$\geq U_b(n) \geq L_b(n)$.

c) Suppose $L'_b > \mathcal{L}$ for at least one inner call and $L'_b <$*upper* for every inner call. So, the for loop may be aborted due the relation $L'_b = U(n)$, but anyhow for every call $L'_b \leq U(n)$. Then for inner calls ending with $\mathcal{L} < L'_b <$*upper*, we also have $L'_b = U'_b <$*upper*. For the other calls we have $L'_b \leq U'_b \leq \mathcal{L} < U(n)$. Hence (5.1) and (5.2) hold on termination. Like in a), we also have *lower*$< f(n) <$*upper*.

d) If $L'_b \geq$*upper* for one inner call, then the *for* loop is aborted and, taking into account that *upper*$> L(n)$, we have:

$$L_b(n) = \textit{maximum of all intermediate } L'_b\textit{-values} \quad (5.3)$$

and $U_b(n) \geq L_b(n) \geq$*upper*.

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

if alpha≥U(n) or L(n)≥beta or U(n)=L(n) then
   [ if L(n) ≥ beta then f:=L(n) else f:=U(n);
     exit procedure;
   ]
if type(n) = max then
   [ alpha':= max(alpha, L(n));
     for c := firstchild(n) to lastchild(n) do
         [ alphabeta(c, alpha', beta, f');
           if f'≥ min(beta, U(n)) then exit for loop;
           if f'>alpha' then alpha':=f';
         ]
     f:=maximum of the intermediate f'-values;
   ]
if type(n)=min then
   [ beta':=min(beta, U(n));
     for c := firstchild(n) to lastchild(n) do
         [ alphabeta(c, alpha, beta', f');
           if f' ≤ max(alpha, L(n)) then exit for loop;
           if f'< beta' then beta':=f';
         ]
     f:=minimum of the intermediate f'-values;
   ]
```

Figure 5: The procedure alphabeta.

The above analysis suggests that having two output parameters is an overkill. One output parameter f suffices, and in that case a suitable postcondition is:

Postcondition:
$lower < f = f(n) = U_b(n) = L_b(n) < upper$ or
$L_b(n) \leq f(n) \leq U_b(n) = f \leq lower$ or
$upper \leq f = L_b(n) \leq f(n) \leq U_b(n)$.

So, we transform our code. The inner calls will have one output parameter f'. The inequality $L'_b > \mathcal{L}$ on termination of an inner call translates to $f' > \mathcal{L}$. If the relations $L'_b \geq upper$ or $L'_b = U(n)$ hold on termination of an inner call, then also $L'_b > \mathcal{L}$. In that case L'_b is represented by f'. Therefore these relations are equivalent to $f' \geq upper$ and $f' = U(n)$ respectively.
In above cases a), b) and c), (5.1) is applied only when $U'_b < upper$ for every inner call. It follows that U'_b in the old code is denoted by f' in the new code. In these three cases, U_b must be represented by f. Therefore we can replace (5.1) by

$$f = maximum\ of\ all\ intermediate\ f'\text{-}values \tag{5.4}$$

Notice that, for the cases a) and c), $U_b(n) = f(n) = L_b(n)$, and, since $f = U_b(n)$, the new postcondition is satisfied. Notice that the new postcondition also holds in case b). In case d), the output value L'_b in the last inner call is $\geq$*upper* and is represented by f'. It follows that L_b must be represented by f. In all former inner calls, $L'_b \leq U'_b <$*upper*. We conclude that we can replace (5.3) by (5.4).

We have studied the situation that n is a max node. The situation with n a min node can be treated similarly to a max node. The situation that no children of n are developed must also be modified according to the new postcondition. The resulting code for our algorithm is presented in Figure 5.
This code contains, for historical reasons, some other changes: the procedure name *alphabeta*, the identifiers *lower* and *upper* are called *alpha* and *beta* and the identifier $\mathcal{L}$ is replaced by *alpha'*.

6 Maxsearch

Now we will give a practical description of Maxsearch, in which all steps are refined into detail. The algorithm consists of three procedures, viz. the main procedure, the procedure *diminish* and the procedure *expand*. Similarly to the previous section, the code of this section does not contain explicit occurrences of the *develop* procedure.

We first discuss some features of the procedure *diminish*. The input parameters are n, a node in the game tree, and *upper*, a real number equal to $\mathcal{U}(n)$ on entry. The output parameters are U_b and L_b, which are equal to $U_b(n)$ and $L_b(n)$ respectively on termination. Since the $\mathcal{U}$-values are non-increasing along a path from the root to an open node, $\mathcal{U}(n)$ stays equal to *upper*, as long as at least one open live node has $\mathcal{U}$-value equal to *upper*. The working of the procedure *diminish* can be described informally in the following way. *As long as at least one open live node p satisfies* $\mathcal{U}(p)$=*upper, develop this node p.* So, the procedure *diminish* contains a number of iterations of the main loop of Gsearch. Notice that, after developing a node p with $\mathcal{U}(p)$ =*upper* in *diminish*, the children may have the same $\mathcal{U}$-value. The specification of *diminish* implies that such children are developed during the same *diminish* call. On termination, we can have two cases. First, n satisfies $\mathcal{U}(n) <$*upper*. Then, by Lemma 2.4, $U_b(n) = \mathcal{U}(n)$. Every open live node p satisfies $\mathcal{U}(p) <$*upper*. Second, n satisfies $U_b(n) \geq \mathcal{U}(n)$ =*upper*. Then n is not live any more, since, if it was live, an open live node p had been left with $\mathcal{U}(p)$ =*upper*. Therefore, it follows that if $\mathcal{U}(n)$ =*upper* on termination, then n is no longer live and hence $\mathcal{U}(n) = \mathcal{L}(n)$. Using Theorem 2.3, we can show that $L_b(n)$ =*upper* in that case. We conclude that $U_b(n) \geq$ *upper* implies $L_b(n) =$ *upper*.

The procedure *expand* has the same parameters as *diminish*, here parameter n is assumed to an open live node. This procedure *expand* develops this open live node and all its descendants, as long as they are live and they have $\mathcal{U}$-value equal to *upper*. The following relation between *diminish* and *expand* exists. The

```
procedure Maxsearch(in:  n, lower, upper; out:  Lb, Ub);
if max(lower, L(n)) ≥ min(upper, U(n)) then
   [ Ub:=U(n);
     Lb:=L(n);
     exit procedure;
   ]
upper:=min(upper, U(n));
diminish(n, lower, upper, Lb, Ub)
while Ub>Lb and Ub>lower do
   [ upper :=Ub;
     diminish(n, lower, upper, Lb, Ub)
   ]
```

Figure 6: The procedure Maxsearch.

procedure *diminish* with input parameter n searches the open live descendants p of n with $\mathcal{U}(p) = \mathcal{U}(n)$ and calls the procedure *expand* to develop p and its descendants. The procedure *expand* also has the property that $U_b(n) \geq upper$ implies $L_b(n) \geq upper$.

Now we will discuss in detail the bodies of the main procedure of Maxsearch and the procedure *expand* and *diminish* respectively.
The main body of *Maxsearch* is presented in Figure 6. Using Lemma 2.4, we can prove that whenever the guard of the while loop is tested, we have

$$upper \geq \mathcal{U}(n) = U_b(n) \tag{6.1}$$

This relation is an invariant of the while loop. The guard in the main loop is equivalent to the test *live*(n). If the guard holds and a new iteration is performed, we obtain $upper = \mathcal{U}(n)$.

Now, we elaborate on the procedure *expand*. The procedure *expand* is described in Figure 7.
For a node n, which is developed, the values $U(n)$ and $L(n)$ are computed. In order to check whether a child of n should be selected or not, two conditions must be tested. Firstly, we have the condition: $U(n) <upper$. If this test succeeds the children of n have a smaller $\mathcal{U}$-value than n and hence, they should not be selected in the current *diminish* call. Secondly, If $L(n) \geq upper= \mathcal{U}(n)$, then n is no longer live and the procedure is aborted. If both tests fail, we have that $upper= \mathcal{U}(n) = U(n) > \mathcal{L}(n)$. Hence n is live. In the code of Figure 7, the value $\mathcal{U}(n)$ is replaced by the input value *upper*.
The successive activation of the children is realized by a for loop. This loop can be interrupted, as we shall show. Suppose that a subcall *expand*$(c, upper, L'_b, U'_b)$, where n is a max node, ends with $U'_b \geq upper$. We have in that case

```
procedure expand(in:  n, lower, upper; out:  Lb,Ub);
if pseudoterminal(n)
         then [ Maxsearch(n, lower, upper, Lb, Ub);
                exit procedure; ]
         else [ develop(n);
                if L(n) ≥ upper or U(n) < upper then
                   [ Ub:=U(n);
                     Lb:=L(n);
                     exit procedure;]
              ]
if type(n)= max then
   [ lower':= max(lower, L(n));
     for c := firstchild(n) to lastchild(n) do
         [ expand(c, lower', upper, Lb', Ub');
           if Lb'>lower' then lower':=Lb';
           if Ub' ≥ upper then exit for loop; ]
   ]
if type(n)=min then
   [ for c := firstchild(n) to lastchild(n) do
         [ expand(c, lower, upper, Lb', Ub');
           if Ub' < upper then exit for loop;]
   ]
Ub:=Ub(n);
Lb:=Lb(n);
```

Figure 7: The procedure expand.

$U_b(c) \geq$*upper* and, as shown earlier, $L_b(c) \geq$*upper*. Consequently, n is no longer live, and developing the children is aborted. If $U_b(c) <$*upper*, then, provided that open children of n are left, we have *upper*$= \mathcal{U}(n) > \mathcal{L}_b(n)$ and hence n is still live.
Suppose that a subcall *expand*(c, *upper*, U_b, L_b), where n is a min node, ends with $U_b' <$ *upper*. We have then for any younger brother c' of c that $\mathcal{U}(c') \leq U_b(n) \leq U_b(c)$; these inequalities follow directly from the definition of the $\mathcal{U}$- and the U_b-function. We conclude that $\mathcal{U}(c') <$ *upper*. Therefore, c' should not be developed, because its $\mathcal{U}$-value is too small.
Next we discuss the determination of the output values. In the code of Figure 7 the output values U_b and L_b are determined according to Definition 1.1. However, in some cases there is a shortcut. If a max node n is live after termination of the for loop, it follows that no child of n is open and consequently, (1.2) can be applied.
If a min node n is live on termination of the for loop, then this loop must have been interrupted. Let c_0 be the node parameter in the last subcall. For all older brothers c of c_0, the subcall *expand*(c, *upper*, L_b', U_b') has ended with $U_b' \geq$ *upper*,

```
procedure diminish(in:  n, lower, upper; out:  Lb, Ub);

if open(n) then [ expand(n, upper, Lb, Ub);
                  exit procedure; ]
if type(n)=max then
   [ lower':=max(lower, Lb(n));
     for c:= firstchild(n) to lastchild(n) do
         [ if Ub(c)=upper or open(c) then
              [ diminish(c, lower', upper, Lb', Ub');
                if Lb' > lower' then lower':=Lb';
                if Ub' ≥ upper then exit for loop;]
         ]
   ]
if type(n)=min then
   [ c:=the youngest non open child,
        or,(if all children are open,) the oldest child;
     if Ub(c) > Lb(c) then
        [ diminish(c, lower, upper, Lb',Ub');
          if Ub' < upper then skip next for loop;]
     for c:= nextbrother(c) to lastchild(n) do
         [ expand (c, upper, Lb', Ub');
           if Ub' < upper then exit for loop;]
   ]
Ub:=Ub(n);
Lb:=Lb(n);
```

Figure 8: The procedure diminish.

and the subcall *expand*(c_0, *upper*, L_b', U_b') has ended with $U_b <$ *upper*. Since at least one child of n has been expanded, the basic part in the code of *expand* has been passed and hence, $U(n) \geq$*upper*. Each younger brother c of c_0 is open and hence $U_b(c) = \infty$. Therefore, the call *expand(n, upper, ...)* ends with

$$U_b(n) = U_b', \tag{6.2}$$

where U_b' is the output value of the last subcall.

The third procedure which we discuss is *diminish*. In the main procedure of Maxsearch, we see that *diminish* is called, only if the search tree has been generated by an *expand* or a former call of *diminish*. Such search trees have special properties. For a non-open live max node n in a search tree resulting from an expand call either all children of n are open or no child is open. For a non-open live min node n in a search tree resulting from an expand call and c_0 be the youngest non open child of n, the children older than c_0 are not live, and $U_b(c_0) = U_b(n)$. If at least one child has been subjected to an *expand* call, then all children will be

```
procedure Maxsearch(in:  n, out:  f);
expand (n, ∞, Lb, Ub)
while Ub>Lb do
   [ upper :=Ub;
     diminish(n, upper, Lb, Ub)
   ]
f:=Ub;
```

Figure 9: Alternative code for Maxsearch as a Gsearch instance.

subjected to such a call, unless the loop is aborted. In that case, n is no longer live, as we showed earlier. We argued above that, if n is live, then $U_b(n) = U_b(c_0)$, where c_0 denotes the the last child, which has been expanded, i.e., the youngest child which is not open.

Now, we discuss the way, in which *diminish* searches the open live nodes with maximal $\mathcal{U}$-value. Walking top-down, we make the following choices in a node n with at least one non-open child. If n is a max node, no child is open, and hence, (1.2) holds. We choose child nodes c of n, such that $U_b(n) = U_b(c)$. If n is a min node, we first choose the youngest non-open child. As shown above for the tree constructed by *expand*, this child, say c_0, satisfies $U_b(c_0) = U_b(n)$. We conclude that, as long as *diminish* does not arrive at an open node, each node, which is visited by *diminish*, has a U_b-value, equal to the U_b-value of the root.
In order to ensure that each node on the path from the root to a leaf node is live, we have to check in each node n, while choosing a child c, that $L_b(c) < \mathcal{U}(c)$, or equivalently, because $U_b(c) = U_b(n) = \mathcal{U}(n) = \mathcal{U}(c)$, that $L_b(c) < U_b(c)$. This inequality always holds, if n is a live max node and $U_b(n) = U_b(c)$. Therefore, only in a min node n, this check must be performed explicitly.

The discussion of the for loop and of the computation of the output values is similar to that for *expand*. Hence, the properties of the search tree resulting from *expand* also hold for one, resulting from a *diminish* call.

7 Comparing Maxsearch with SSS* and SSS-2

When Maxsearch is considered as instance of Gsearch, the *lower* parameter and all related operations can be deleted consistently anywhere in the code. The main body can be replaced by the code of Figure 9. A similar replacement is not possible in Maxsearch as a Rsearch instance, because, in that case, we would have that, for a pseudo-terminal n, *Maxsearch* invokes *expand* first and *expand* invokes *Maxsearch* first. So infinite recursion occurs.
The new Maxsearch code has a great resemblance to SSS-2 [Pijls-2]. A search tree, resulting from a *diminish* or *expand* procedure, resembles a solution tree, as

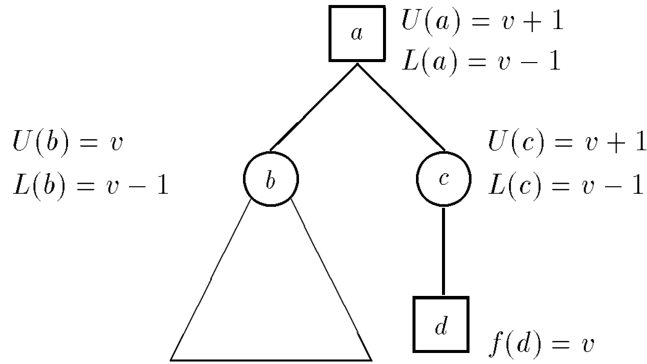

Figure 10:
The difference between Maxsearch and SSS-2.

used in SSS-2. The function U_b corresponds to the g-function for a solution tree T in SSS-2, which in turn corresponds to the merit in SSS*. Notice that $U_b > L_b$ on termination of *expand* or *diminish* implies *upper*$>$ U_b. This last inequality corresponds to the guard in the main loop of SSS-2.
The parameter L_b does not occur in SSS-2. It establishes an essential difference between Maxsearch and SSS-2. At the start of the *diminish* procedure in SSS-2, the test $g(n) > L(n)$ is executed, in order to decide whether the procedure is continued or not. In Maxsearch, the parameter L_b features in the tests *while* $U_b(n) > L_b(n)$ *do* ... in the main procedure, and *if* $U_b(c) > L_b(c)$ *then* ... in the *diminish* body, in order to ascertain that the node parameter in the subsequent *diminish* call is live.
The difference between Maxsearch and SSS-2 is illustrated by the search tree in Figure 10. In Maxsearch a is no longer live and no descendants of a will be considered. In SSS-2 we have that the tree, consisting of a, b, c and d is a so-called *milestone*. A call *diminish(a, v,...)* may be performed and the subtree, rooted in b, is expanded. It also appears that SSS-2 is not an instance of Gsearch, because SSS-2 visits nodes, which are not live.
SSS-2 can be regarded as a Gsearch-like instance, which takes the L-function instead of the L_b-function into account.

However, the idea behind Rsearch is not feasible for SSS* or SSS-2, since no information on lower bounds is passed top-down. See Figure 11. When d has been visited, e is expanded neither in the Maxsearch nor in SSS* and SSS-2. Assume that c is pseudoterminal and we apply SSS* or SSS-2 recursively to c in order to compute $f(c)$ or an upperbound for $f(c)$. Since the recursive call with parameter c does not have an appropriate parameter *lower*, the recursive call will visit e with descendants. It appears that, if the idea behind Rsearch is applied

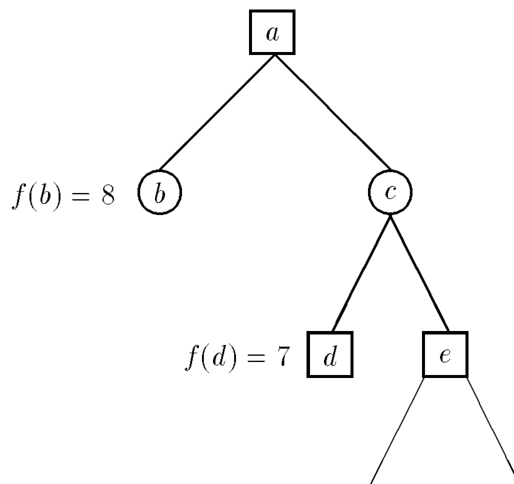

Figure 11:
A recursive SSS-call is inefficient

to SSS* or SSS-2, an inefficient algorithm results.

8 Concluding remarks

Each feasible game tree algorithm tries to avoid visiting the entire game tree and to prune this tree substantially. In their reviews on game-tree pruning [Marsland] and parallel alpha-beta search [Bal], the authors give an adequate overview of the existing techniques, but do not attempt to unify them. In [Ibaraki 86], a general principle has been formulated, underlying pruning techniques in game tree algorithms. Based upon this principle a non-deterministic game tree algorithm has been constructed, called GSEARCH, in which only the pruning criterion is given explicitly. GSEARCH demands in worst case that the entire game tree is in memory. The principle of recursive computation of the minimax value of a game tree, applied in alpha-beta, is exploited in a variant of GSEARCH, called RSEARCH, which stands for Recursive Search.
In our paper, related frameworks, Gsearch and Rsearch are presented, which resemble Ibaraki's ones. We showed in Section 5 that the pruning criterion of alphabeta can be viewed as an application of the Gsearch and Rsearch pruning method. Moreover, alphabeta has a specification similar to Rsearch, Hence alpha-beta is generalized.

Selecting an open node by a rule, which prescribes how to trace this open node in a top-down manner, is applied first in the definition of Maxsearch. Maxsearch was discussed earlier in [Pijls-1]. Classifying the instances of Gesearch and Rsearch according to a rule, which prescribes how to select top-down an open live node,

is an idea from Ibaraki [Ibaraki 91B]. His characterisation of SSS* in this sense is completely different from our Maxsearch characterisation, although the Maxsearch code is close to SSS-2 and SSS*.

Acknowledgments
This paper is the result from discussions with Toshi Ibaraki. The authors are grateful to him for his suggestions and contributions.
Furthermore, a lot of ideas have arisen in a working group on game trees at the Erasmus University Rotterdam, in which, apart from the authors, two students Patrick van der Laag and Theo Klein Paste participated, while preparing their Master's Thesis.

References

[Bal] H. E. Bal and R. van Renesse, *A summary of Parallel Alpha-Beta Search results,* ICCA Journal, 9 (1986), nr. 3, pp 146-149.

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

[Ibaraki 86] T. Ibaraki, *Generalization of Alpha-Beta and SSS* Search Problems,* Artificial Intelligence 29 (1986), pp 73-117.

[Ibaraki 87] T. Ibaraki, *Game solving procedure H* is unsurpassed,* in: Discrete Algorithms and Complexity, Academic Press 1987, pp 185-200.

[Ibaraki 91A] T. Ibaraki and Y. Katoh, *Searching Minimax Game Trees under Memory Space Constraint*, Annals of Mathematics and Artificial Intelligence, 1 (1991), pp 141-153.

[Ibaraki 91B] T. Ibaraki, *Search Algorithms for Minimax Game Trees,* Conference paper at *Twenty years NP-completeness*, Sicily, June 1991.

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

[Marsland] T.A. Marsland, *A Review of Game-Tree Pruning,* ICCA Journal, 9 (1986), nr.1, pp 3-19.

[Pijls-1] W. Pijls, *Shortest paths and game trees,* Ph.D.thesis, Erasmus University Rotterdam, 1991.

[Pijls-2] W. Pijls and A. de Bruin, *Searching informed game trees,* Report EUR-CS-92-02, Department Computer Science, Erasmus University Rotterdam. An extended abstract has been published in: T. Ibaraki, Y. Inagaki, K. Iwama, T. Nishizeki, M Yamashita (Eds.) Algorithms and Computation, Third International Symposion, ISAAC'92 Nagoya 1992, Lecture Notes in Computer Science series, vol. 650, pp 332-341.

[Pijls-3] W. Pijls and A. de Bruin, *SSS*-like Algorithms in Constrained Memory,* in: ICCA-Journal, 16 (1993), nr.1, pp.18-30.

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