Trends in Game Tree Search

Arie de Bruin and Wim Pijls

Erasmus University, Dept. Comp. Science, H4-29, P.O. Box 1738,
3000 DR The Netherlands
email {adebruin, pijls}@few.eur.nl.

Abstract. This paper deals with algorithms searching trees generated by two-person, zero-sum games with perfect information. The standard algorithm in this field is Alpha-Beta. We will discuss this algorithm as well as extensions, like transposition tables, iterative deepening and NegaScout. Special attention is devoted to domain knowledge pertaining to game trees, more specifically to solution trees.
The above mentioned algorithms implement depth first search. The alternative is best first search. The best known algorithm in this area is Stockman's SSS*. We treat a variant equivalent to SSS* called SSS-2. These algorithms are provably better than Alpha-Beta, but it needs a lot of tweaking to show this in practice. A variant of SSS-2, cast in Alpha-Beta terms, will be discussed which does realize this potential. This algorithm is however still worse than NegaScout. On the other hand, applying a similar idea as the one behind NegaScout to this last SSS version yields the best (sequential) game tree searcher known up till now: MTD(f).

1 Introduction

In this paper we give an overview on algorithms that search *game trees*, as defined by perfect-information, two-person, zero-sum games like chess, checkers and the like. Let the reader be warned that the presentation is certainly not exhaustive and also rather broad. Most of the algorithms presented here admit refinements, game trees are not so regular as suggested in this paper, many details and all proofs are skipped. The aim of this paper is to give the reader a feeling for what is going on in the field. For more details one should consult the references.

The term 'zero-sum game' indicates that the gain of one player equals the loss of her opponent. This excludes games studied in the field of classical mathematical game theory, such as the prisoner's dilemma. We furthermore exclude random events, the game should proceed completely deterministically. This means that a game like backgammon is not studied here either.

The games in the above described class proceed by the adversaries taking turns in making a move. A move alters the current *position*, e.g. the contents of a chess board, or the number and the size of heaps of matches in Nim-like games. Such games can be represented in abstract terms as *game trees*, the nodes of which correspond to positions, while an edge between two nodes describes the move that transforms the position corresponding to the source node into the position associated with the target node. Because the players take turns, such

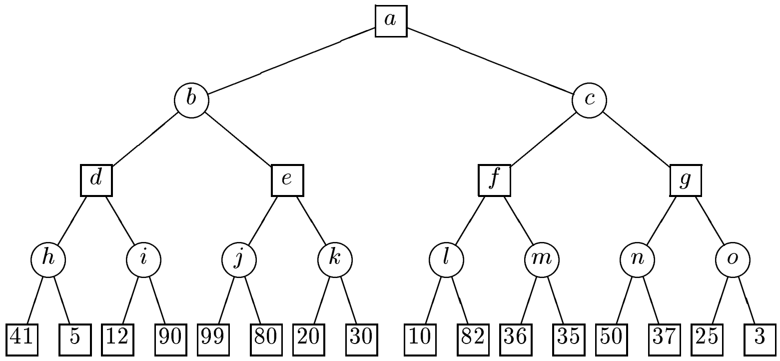

Fig. 1. A Game Tree

a tree is layered, in the sense that all nodes at the same depth are of the same type, e.g. MAX nodes (denoted by a square), in which the first player (usually called MAX) has the move. At the next level in the tree we only see MIN nodes, depicted as circles, where the second player (called MIN) should move. The root of the tree corresponds with a given position of the game, and the leaves are nodes from which no move can be made. In these final positions the game is decided, the value of such a position is known, e.g. 'win for MAX', 'loss for MAX', 'draw', or maybe a more subtle score. It is customary to represent a score as an integer which denotes the value that the position has for MAX. For instance, a win for MAX could be denoted by $+1$, a draw by 0 and a win for MIN by -1. An example of a game tree is given in Fig. 1.

A game tree search algorithm determines the value $f(\text{root})$ of the root node of a game, e.g. 'forced win for MAX' or 'forced win for MIN' or 'one of the players can force a draw'. Moreover the algorithm also determines the best move from the root position.

A few remarks are in order at this moment. First of all one should be aware of the fact that most games do not generate trees but *game graphs* instead. For instance, due to repetitions of moves the game tree for chess is infinite, while the game graph is finite, because there are only finitely many valid positions in chess. For efficiency reasons most game tree algorithms exploit a *transposition table* in which positions already encountered are stored together with relevant information thereof (value, best move, etc.). One gets the impression that transposition tables have been added as an afterthought, or as a coding trick. For instance one still speaks of game tree search instead of game graph search. More importantly, we have the impression that not all information stored in a transposition table is used to good effect by the standard algorithms. In fact, a few of the more recent results discussed in this paper are related to a fuller exploitation of the data hidden there. For other examples the reader is referred to [11, 13].

A second remark is related to the fact that trees for nontrivial games are far too big to be searched fully (which is one of the reasons such games are interesting for us humans). Therefore approximation techniques have to be used. One standard approach is to consider only a finite part of the tree, e.g. only a restricted number of levels (*plies* or half-moves). Notice that Fig. 1 can also be considered as such a partial game tree. This means that not in all leaves of such a reduced tree an exact evaluation of the position is possible (because if we would have such a gadget, there would be no need to search a full tree of variations...). We have to rely on an evaluation function 'eval' that can only approximate the true value of a position. It is hoped however that the inexactness of this approach is leveled by the fact that the tree is searched to a sufficient depth.

The value of a position is determined by the so called *minimax rule*. In a leaf n of the tree the value is given by eval(n). In non-leaf positions the MAX player will choose the move which leads to the best position, i.e. the position with the highest value. If it is MIN's move then she will choose the move resulting in the best position from her point of view, that is, the one with the smallest value. This mechanism can be applied recursively which yields the following definition.

$$\begin{aligned} &\mathrm{minimax}(n) = \\ &= \begin{cases} \mathrm{eval}(n) & \text{if } n \text{ is a leaf} \\ \max\{\mathrm{minimax}(c) \mid c \text{ is a child of } n\} & \text{if } n \text{ is a MAX node} \\ \min\{\mathrm{minimax}(c) \mid c \text{ is a child of } n\} & \text{if } n \text{ is a MIN node} \end{cases} \end{aligned} \tag{1}$$

This rule straightforwardly translates into the algorithm given in Fig. 2. A few remarks on the 'language' in which the algorithms appearing in this paper are written must be made. The **return** statement is like its C–counterpart, that is, the return value of the function is determined and an exit from the function is performed. Statement grouping is expressed by using indentation, obviating the need for pascal **begin**...**end** pairs or C-like {}-brackets. We will use a genealogical way of expressing relations between nodes in our game trees. For instance, the FirstChild of a node n or the OldestBrother of one of its children c is the leftmost child of n in the tree.

2 Alpha-Beta

The Minimax algorithm from the previous section visits all nodes in the game tree. Because this number is exponential in the depth d of the tree, and also because branching factors w can be rather large (for instance in the middle game of chess a node has about 30–40 children), one can be confronted with impressive numbers of nodes to be visited ($O(w^d)$ for *uniform trees*, i.e. trees in which all non-leaf nodes have the same number of children). It is only natural to search for ways to avoid having to visit all nodes. The well known Alpha-Beta algorithm uses, like Minimax, a depth first search but, unlike Minimax, finds a way to *cut off* nodes which are not relevant for the outcome. Alpha-beta has a rich history, a good overview of the early stages is presented in [5].

Suppose the Minimax algorithm of Fig. 2 searches the tree from Fig. 1 and suppose the inner call 'minimax(d)' executed within the call 'minimax(b)' has

```
function minimax(n) → g;
    if n is a leaf then g := eval(n);
    else if n is a MAX node then
        g := −∞; c := FirstChild(n);
        while c is well defined do
            g := max(g, minimax(c)); c := NextBrother(c);
    else /* n is a MIN node */
        g := +∞; c := FirstChild(n);
        while c is well defined do
            g := min(g, minimax(c)); c := NextBrother(c);
    return g;
```

Fig. 2. The Minimax algorithm

just returned the value 12. The next step is now to determine the minimax value of e, a call 'minimax(e)' is executed which in its turn leads to a call 'miminax(j)' which delivers the value 80. Therefore we now know that e, being a MAX node, will have a minimax value of at least 80. But from this we can already infer the minimax value of b, because this will be the minimum of 12 (d's value) and a number not smaller than 80. So the value of k cannot influence b's value any more and there is no need to investigate k or its descendants. This phenomenon, that a node needs not be visited because an older brother (in this case j) has a value worse than an uncle (here d) is called a *shallow cutoff*.

This is the main idea exploited in Alpha-Beta. It is implemented by giving the recursive search procedure 'alphabeta', apart from the node to be investigated, two other parameters, a lower bound α and an upper bound β using which information is transferred about the history of the computation previous to this call. In our case, node e as well as node j will feature in a call 'alphabeta' with parameters $\alpha = -\infty$ and $\beta = 12$. These parameters define the *$\alpha\beta$-window* $(-\infty, 12)$, using which it is communicated to the procedure that only return values within this window matter for the environment. Using this information, in the body of the call 'alphabeta($e, -\infty, 12$)' it can *locally* be decided that there is no need to investigate k after j's value has become known.

Let us return to the execution of Minimax and Alpha-Beta on the tree in Fig. 1. Once it is determined that the value of b equals 12, control returns to node a, after which the subtree rooted in c should be investigated. Now a similar line of reasoning applies: for node a, having already a child with value 12, only values greater than 12 are of interest. This translates into parameter values in the call for node c of $\alpha = 12$ and $\beta = +\infty$. This will lead to an inner call for nodes f and l with the same parameters and once it is determined that the oldest child of l has value smaller than 12 there is no need to visit the other child. Notice however that this cutoff is not of the type discussed earlier, there is no 'older brother – uncle' relationship. This type of cutoff is called a *deep cutoff* and the early versions of Alpha-Beta did not recognize such ones.

```
function alphabeta(n, α, β) → g;
    if n is a leaf then g := eval(n);
    else if n is a MAX node then
        g := −∞; α′ := α; c := FirstChild(n);
        while (g < β) and (c is well defined) do
            /* g is max{return values of children seen so far} */
            g := max(g, alphabeta(c, α′, β));
            α′ := max(g, α′); c := NextBrother(c);
    else /* n is a MIN node */
        g := +∞; β′ := β; c := FirstChild(n);
        while (g > α) and (c is well defined) do
            /* g is min{return values of children seen so far} */
            g := min(g, alphabeta(c, α, β′));
            β′ := min(g, β′); c := NextBrother(c);
    return g;
```

Fig. 3. The Alpha-Beta algorithm

It is also possible to have finite values for both the α and the β parameter. Consider node c. Once the call 'alphabeta$(f, 12, +\infty)$' has terminated with return value 35, node g will be called with $\alpha\beta$-window $(12, 35)$. There is no need for return values ≤ 12 because a does not need them (in that case a will choose b instead) and there is no need for a return value ≥ 35 because c does not need this (c will choose f instead).

The same parameters will feature in the recursive call for n. This latter call will return with a *high failure*, i.e. value 37, which will cause a cutoff (a *β-cutoff*) of node o.

The mechanism must be clear by now. If a child of a MAX node returns with a high failure, a return value $\geq \beta$, then the remaining younger children can be cut off. Dually, if a child of a MIN node returns with a *low failure*, a return value $\leq \alpha$, then we again have a cutoff, called an *α-cutoff*. Using this explanation the code of Fig. 3 should be clear.

The idea behind the Alpha-Beta function applied to a node n is that it needs only return the right value, the game value $f(n)$, when this value lies within the input $\alpha\beta$-window. It is interesting to see what value is returned if the function returns with a low or a high failure. We first consider the case that n is a MAX node with only leaves as children. From the code in Fig. 3 we see that on low failure all children have been investigated, and the return value is the highest value among the children. Thus in this case also the exact minimax value of n is returned. On the other hand if a high failure occurs, it is not necessary that all children of n have been visited. The procedure returns the value of the oldest child with value $\geq \beta$. This means that now not the exact value $f(n)$ is returned but only a lower bound to the game value of n. A dual observation can be made for the case that n is a MIN node.

This again entails that on low failure in a MAX node with height 2 (where 'height' is defined as the distance to the leaves), Alpha-Beta will not return the game value but in general only an upper bound thereof. This is due to the fact that the same holds for all its children. The next lemma states that this observation generalizes.

Lemma 1. *Consider a call* alphabeta(n, α, β)*, returning a value* g*. If on entry to this function the precondition* $\alpha < \beta$ *holds, then on exit we have the following postcondition*

$$g \leq \alpha \Rightarrow g \geq f(n), \text{ (low failure)} \tag{2}$$

$$\alpha < g < \beta \Rightarrow g = f(n), \text{ (success)} \tag{3}$$

$$g \geq \beta \Rightarrow g \leq f(n), \text{ (high failure)} \tag{4}$$

The proof (cf. [1, 8, 9]) is a formalization of the above line of reasoning, using recursion induction, which is in essence induction on the height of n.

It is interesting to notice that the earliest variants of Alpha-Beta would return the input value α on low failure and β on high failure, thus discarding information gathered during the call. On first sight the output value is indeed irrelevant on high or low failure because high enough up the game tree it will be discarded anyhow. There are reasons however to state the contrary. The most important ones will surface later when we will discuss Alpha-Beta variants of best first game tree search, but one argument can be given already now. Suppose a transposition table is used in order to apply the results of a previous investigation of a node n when this node turns out to be a transposition. In general, the $\alpha\beta$ input window will not be the same as in the previous call. This suggests that it is useful to store results in the transposition table that are as informative as possible. We return to this issue later.

Earlier we saw that Alpha-Beta searches less nodes than Minimax in essence by exploiting the input $\alpha\beta$-window to prune nodes. It seems that the effect is that making the input window smaller decreases the number of nodes to be visited. Closer examination of the code shows that this is indeed the case. Stated more precisely, we have

Lemma 2. *Suppose two calls* alphabeta(n, α, β) *and* alphabeta(n, α', β') *are executed on the same node in the same game tree and suppose* $\alpha \leq \alpha'$ *and* $\beta \geq \beta'$*. Then we have that the set of nodes visited by the second call is a subset of the set of nodes visited by the first call.*

This lemma can be proven by exploiting a characterization of the nodes visited by Alpha-Beta, given in [1, 3] amongst others. It can also be proven straightforwardly using recursion induction. This result will turn out to be very useful in the sequel while comparing the efficiency of Alpha-Beta based game tree search algorithms.

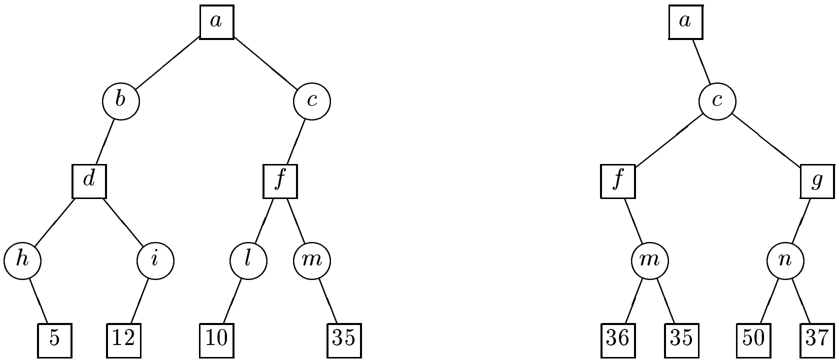

Fig. 4. An Optimal Max (left) and Min (right) Solution Tree

3 Solution Trees

In the previous section we indeed realized our goal, Alpha-Beta searches less nodes than Minimax (the pathological case that Alpha-Beta searches the whole tree is possible but unlikely). In this section we approach the question whether we can do better from a theoretical point of view, by introducing the notion *solution tree*.

Suppose, for the tree of Fig. 1, we want to find a subtree from which we can derive an upper bound for the game value 35 of the root. We can do so if we have an upper bound for both children, b and c. This follows from the minimax rule. So a, b and c must be included in the tree to be built. For b and c, being MIN nodes, there is a less precious way to establish an upper bound, an upper bound for only one child will do (this follows again from the minimax rule). Let us include d and f. Now the story repeats. For these MAX nodes we have to include both children, i.e. h, i, l and m. Finally we have to add one child for each of these 4 nodes and we choose the leaves labeled 5, 12, 10 and 35.

The subtree we have built is the left one in Fig. 4. It is called a *max solution tree*, the shape of which is defined by the rule that the root should be included, all children should be included for a non-leaf MAX node, and exactly one child should be included for a non-leaf MIN node. A max solution tree exhibits a *strategy for the MIN player* because in all possible variations in the tree the choice of MIN is fixed [4, 15].

The minimax value of the tree is an upper bound to the game value of the root. Because in MIN nodes of this tree the minimax rule determines the minimum of a singleton set, there is only maximization left, and we find that the minimax value equals $\max\{l \mid l \text{ leaf in the tree}\} = 35$. This value is in fact the best we can get, and therefore this solution tree is called *optimal*.

By reversing the 'construction rules' we define the dual notion *min solution tree*. The rightmost tree in Fig. 4 is a min solution tree. It is also optimal because its minimax value, the minimum of the values of its leaves equals $f(a)$.

When we take the union of an optimal min solution tree and an optimal max solution tree we obtain a so called *critical tree*. The intersection of these trees is the *principal variation*, in our example the path from the root to node 35. From the arguments given above we infer that a critical tree establishes both an upper bound and a lower bound equal to the value of the root, that is it establishes the game value. We summarize.

Lemma 3. [17] *The minimax value of a min/max solution tree is a lower/upper bound to the game value of the root. Moreover, for each game tree there exists at least one optimal min/max solution tree.*

Lemma 4. [3, 5, 10] *In order to establish the game value of the root an algorithm must have visited at least a critical tree.*

The reader is invited to check that both optimal solution trees for the tree of Fig. 1 are unique. Lemma 3 suggest that this need not be so. In fact, assigning to leaf 41 in Fig. 1 the value 11 makes the max solution tree with leaf set $\{11, 12, 10, 35\}$ optimal as well.

It is instructive to study how Alpha-Beta finds the critical tree of Fig. 1. The leaf set visited by Alpha-Beta is $\{41, 5, 12, 90, 99, 80, 10, 36, 35, 50, 37\}$ which is certainly bigger than the critical leaf set $\{5, 12, 10, 36, 35, 50, 37\}$. The overhead is due to the fact that Alpha-Beta determines that 12 is the game value of node b, while an upper bound of 12 would have been sufficient. Notice furthermore that on high failure, e.g. in e, a min solution tree has been constructed defining the return value 80 which is a lower bound. For these results we will therefore use the notation f^-. Similarly a low failure, e.g. in l, returns an upper bound f^+ defined by a max solution tree.

In [5] it is investigated whether it is possible that Alpha-Beta indeed searches only the critical tree. This will occur if the tree is *perfectly ordered*, i.e. for each node n, the oldest child has the best game value (maximal if n is a MAX node, minimal otherwise) of all its children. In that case the optimal solution trees will be the leftmost ones in the game tree. The reader is invited to reorganize the tree of Fig. 1 so that the above property holds, and to check how Alpha-Beta traverses this tree.

Notice that if we are able to organize the search so that we need only to search the critical tree we will have made a big progression because the size of this tree is $O(w^{d/2})$. Or, stated in other words, with the same amount of work we will be able to search trees twice as deep as Minimax can.

4 Enhancing Alpha-Beta

In this section we will discuss three reasons why it makes sense to add a transposition table to the Alpha-Beta algorithm [11, 16]. The first one has already appeared in the Introduction, one wants to avoid a recalculation when a transposition is found. Instead, the results stored in the transposition table should

```
function TTalphabeta(n, α, β) → g;
    if n in transposition table then
        if n.f⁻ = n.f⁺ then return n.f⁻;
        if n.f⁻ ≥ β then return n.f⁻;
        if n.f⁺ ≤ α then return n.f⁺;
        /* Info in transposition table did not cause a shortcut */
        α := max(α, n.f⁻); β := min(β, n.f⁺);
    if n is a leaf then g := eval(n);
    else if n is a MAX node then
        g := −∞; α' := α; c := FirstChild(n);
        while (g < β) and (c is well defined) do
            /* g is max{return values of children seen so far} */
            g := max(g, TTalphabeta(c, α', β));
            α' := max(g, α'); c := NextBrother(c);
    else /* n is a MIN node */
        g := +∞; β' := β; c := FirstChild(n);
        while (g > α) and (c is well defined) do
            /* g is min{return values of children seen so far} */
            g := min(g, TTalphabeta(c, α, β'));
            β' := min(g, β'); c := NextBrother(c);
    if n not in transposition table then
        put n into transposition table;
        n.f⁻ := −∞; n.f⁺ := +∞;
    if (g < β) or (n is a leaf) then n.f⁺ := g;
    if (g > α) or (n is a leaf) then n.f⁻ := g;
    return g;
```

Fig. 5. Alpha-beta with a transposition table

be used. From Lemma 1 we have that there are three types of outcome possible: high failure yielding a lower bound f^- to the game value, low failure yielding an upper bound f^+, and success yielding the game value f, combining an upper and a lower bound $f^+ = f^- = f$.

In Fig. 5 the code for Alpha-Beta using a transposition table is given. It is the same as in Fig. 3 except that code has been added that handles the transposition table. At the end of the procedure we store results, and at the beginning we try to profit of them. Information from the transposition table can cause an immediate cutoff, or it can lead to sharper bounds, worthwile to be adopted in view of Lemma 2.

A second advantage of transposition tables is that one can achieve a better move ordering. This is realized for instance by algorithms applying *iterative deepening* (ID). This technique was introduced for another reason though, i.e. to avoid that programs would overstep a time limit. Often only a limited time is available to search a position, and therefore it makes sense to organize the search as an *anytime algorithm*, i.e. one must be able to interrupt it at an arbi-

```
function NegaScout(n, α, β) → g;
    if n is a leaf then g := eval(n);
    else
        c := FirstChild(n); g := NegaScout(c, α, β); c := NextBrother(c);
        if n is a MAX node then
            α' := α;
            while (g < β) and (c is well defined) do
                /* g is max{return values of children seen so far} */
                α' := max(g, α'); t := NegaScout(c, α', α' + 1);
                if (t > α') and (t < β) then t := NegaScout(c, t, β);
                g := max(g, t); c := NextBrother(c);
        else /* n is a MIN node */
            β' := β;
            while (g > α) and (c is well defined) do
                /* g is min{return values of children seen so far} */
                β' := min(g, β'); t := NegaScout(c, β' − 1, β');
                if (t < β') and (t > α) then t := NegaScout(c, α, t);
                g :=min(g, t); c := NextBrother(c);
    return g;
```

Fig. 6. The NegaScout algorithm

trary moment and still obtain a sensible answer. The idea is to let the program search bigger and bigger trees. One starts with the game tree truncated to a limited depth d, which will be searched using an evaluation function defined at the nodes at depth d. The tree is then iteratively extended one level deeper, and re-searched. After an interation, the time spent in the earlier iterations will be negligeable to the time spent in the last one because search time grows exponentially with the depth.

The value eval(n) of a node at depth d shall in general be a good estimate of its minimax value determined by the evaluation function of its children. This means that the best son of an internal node, as calculated by a depth d search, will probably remain so when calculated by a depth $d+1$ search. It is therefore useful to store the identity of a node's best child in the transposition table. In the next iteration this node can then be searched before the others. This idea turned out to be quite successful, cf. [16], it caused an appreciable speedup, which should not surprise us in the light of the discussion at the end of the previous section.

Having achieved a good move ordering, the Alpha-Beta extension called *NegaScout* [6] tries to make advantage of that. The idea is to search suitable nodes using a *null window*, a window of the form $(f, f+1)$ which cannot contain a game value. This means that a null window call can only end in high or low failure. The NegaScout algorithm, given in Fig. 6 is indeed a more efficient version of Alpha-Beta:

Lemma 5. *A call* NegaScout(n, α, β) *returns the same value as* alphabeta(n, α, β) *and every node it visits must also be visited by* alphabeta.

We give an example of the ideas behind NegaScout, again using the game tree from Fig. 1. Suppose a call NegaScout$(a, -\infty, \infty)$ is performed. First of all the value of b will be determined through NegaScout$(a, -\infty, \infty)$. This returns 12, which it should according to the above lemma. Now Alpha-Beta would proceed with a call alphabeta$(c, 12, \infty)$, but let us try to be smart. If the tree is sufficiently ordered the odds are high that b is better than c, meaning that alphabeta$(c, 12, \infty)$ will fail low. But then any window $(12, x)$ will generate a low failure, so why not choose $(12, 13)$ promising the smallest amount of work. This leads to a call NegaScout$(c, 12, 13)$.

Now the risk of gambling is that one might lose, and this is exactly what happens in our case (remember, the tree of Fig. 1 is rather badly ordered.) The NegaScout call returns the value 35, again obeying the above lemma. We now are forced to perform a re-search to obtain the true value of c using a wide window, in our case $(35, \infty)$. Notice the value 35 here, we try to exploit as much information from the previous search as possible. This call will return 35.

Not intimidated by our previous experience, we try to perform the null window trick again. We again assume that f is the better child of c, so we do not issue a call with window $(-\infty, 35)$ on g but we do NegaScout$(g, 34, 35)$ instead. Notice that we shifted the window 1 to the left, because we now expect g to have a value ≥ 35. In this case we are succesful, the return value is 37 and no re-search is needed.

It is instructive to execute manually the recursion chain generated by the call NegaScout$(a, -\infty, \infty)$ on the tree in Fig. 1. A chain of inner calls will lead us to leaf 41, and to the hypothesis that the value of h equals 41. Null window search on leaf 5 immediately refutes this, and a re-search will lead to the conclusion $f(h) = 5$ and the hypothesis $f(d) = 5$. A null window search on i will invalidate this, so again we have to do a re-search on i, showing $f(d) = 12$, leading to the hypothesis $f(b) = 12$. The call NegaScout$(e, 11, 12)$ justifies this, leading to the hypothesis $f(a) = 12$, and so on.

The leaf set visited by NegaScout on a is $\{41, 5, 12, 90, 99, 80, 10, 36, 35, 50, 37\}$, unfortunately the same set as visited by Alpha-Beta. In order to show that NegaScout is better than Alpha-Beta the tree should be better ordered. On the other hand, if the tree would be perfectly ordered again NegaScout and Alpha-Beta would search the same subtree, i.e. the critical tree...

Our examples are rather unfortunate. Some of the advantages of NegaScout over Alpha-Beta can be observed by changing the tree of Fig. 1 a little bit: give leaf 10 the value 15, and leaf 50 the value 10. Now Alpha-Beta will generate calls alphabeta$(l, 12, \infty)$ and alphabeta$(m, 15, \infty)$ and it will visit the subtree under m. On the other hand, NegaScout will generate a call NegaScout$(l, 12, 13)$ and not visit m. Re-search in f will not be needed because NegaScout$(g, 12, 13)$ will fail low.

It turns out in practice that for reasonably well ordered trees NegaScout performs consistently better than Alpha-Beta, cf. [11, 12]. The overhead caused

```
program SSS-2;
    G := {root};
    g := expand(root, ∞);
    repeat
        g' := g;
        g := diminish(root, g');
    until g ≥ g';
```

Fig. 7. The SSS-2 algorithm, main loop

by re-searches turns out to be low, especially when the evaluation function is relatively expensive compared to tree traversal. This of course assumes that we store leaf values in a transposition table, which is an indication of the third advantage of transposition tables: it provides useful information for re-searches within the same iteration of iterative deepening. We will return to this issue in a later section.

5 The SSS* Family

Because in Alpha-Beta the search is organized in a depth first left to right manner the algorithm easily suffers from bad move orderings in the tree. The SSS* algorithm [17] tries to direct the search to those parts of the tree which are likely to be in the critical tree, in that sense it can be characterized as a best first search. In this section we will not discuss SSS* but SSS-2 instead [7]. The reason is that the latter algorithm is more perspicuous and it is equivalent with SSS* in a rather strong sense: it searches the same nodes in the same order and it shows almost all weaknesses that SSS* has. A similar analysis as given here, but based on SSS* can be found in [11, 12].

The main body of SSS-2 is given in Fig. 7. The idea is to find the best max solution tree. This is realized by first generating the leftmost max solution tree and then successively refining it into a better one until this is no longer possible. The program manipulates a global variable G in which the current max solution tree is stored.

The first max solution tree is generated by the call 'expand(a, ∞)'. It finds the leftmost max solution tree, assigns it to G and returns its value g. The max solution tree is then refined by a call 'diminish(a, g')'. If successful, this procedure returns a sharper upper bound g and stores in G the max solution tree defining this bound. Failure is indicated by a return value $g \geq g'$. When executed on the tree of Fig. 1, the expand call will return the max solution tree with leaf set $\{41, 12, 10, 36\}$, and the leaf sets of the max solution trees built by the diminish calls will be $\{5, 12, 10, 36\}$ and $\{5, 12, 10, 35\}$.

We now give a more detailed description of 'diminish', cf. Fig. 8 and 'expand', cf. Fig. 9. Consider the second time the main body of SSS-2 executes a diminish

```
function diminish(n, v) → g;
/* may only be called if subtree rooted in G has value v */
    if n is a leaf then
        /*By virtue of precondition game value of n equals v */
        return v;
    else if n is a MAX node then
        /* n has at least one child c with c.g = v */
        for c := FirstChild(n) to LastChild(n) do
            if c.g = v then v' := diminish(c, v);
            if v' ≥ v then /* no tighter upper bound available for n */
                PURGE all descendants of n from G;
                return v';
        /* loop terminated normally, tighter upper bound available for n */
        v' := max{c.g | c child of n};
        n.g := v'; return v';
    else /* n is a MIN node, the only child of n in G has c.n = v */
        c := the single child of n in G;
        v' := diminish(c, g);
        if v' < v then
            n.g := v'; return v';
        /* no tighter bound for c available, children of c in G have already
                been removed */
        remove c from G;
        for c := NextBrother(c) to LastChild(n) do
            add c to G; v' := expand(c, v);
            if v' < v then
                n.g := v'; return(v');
            /* arrive here only if v' ≥ v */
            remove c from G;
        return g; /* returns the sharpest lower bound, not strictly necessary */
```

Fig. 8. The SSS-2 algorithm, procedure diminish

call. At that moment we have a tree in G with leaf set $\{5, 12, 10, 36\}$. Accordingly, the input parameter g' will be 36. The tree G defines an upper bound of 12 for b and of 36 for c. Refining G so that the upper bound of b will become sharper is of no use at this stage as long as we are not able to tighten $g(c)$. Therefore a recursive call 'diminish$(c, 36)$' is generated. In order to efficiently find out which child to choose, the algorithm stores the g-value of each node, as defined by G, also in G. Because c is a MIN node there is only one child, f, in G. We first try to make the subtree in f better, i.e. we issue a call 'diminish$(f, 36)$'. This generates a call 'diminish$(m, 36)$' and an inner call 'diminish$(36, 36)$' returning 36, i.e. failure. There is one possibility left to make a better $g(m)$, the other child must be investigated. Because this child has not been visited before, a call of 'expand' is in order. However, we will only be satisfied with a return value smaller than 36, and that is why we add this value as a second parameter:

```
function expand(n, v) → g;
    /* G should contain n but no descendants of n */
    if n is a leaf then
        v' := eval(n); n.g := v'; return v';
    else if n is a MAX node then
        for c := FirstChild(n) to LastChild(n) do
            add c to G; v' := expand(c, v);
            if v' ≥ v then
                PURGE all descendants of n from G;
                return v';
            /* loop terminated normally; bound for n, tighter than v, available */
        v' := max{c.g | c child of n};
        n.g := v'; return v';
    else /* n is a MIN node */
        v'' := ∞;
        for c := FirstChild(n) to LastChild(n) do
            add c to G; v' := expand(c, v);
            if v' < v then
                n.g := v'; return v';
            /* arrive here only if v' ≥ v */
            remove c from G; v'' :=min(v', v'');
        /* loop terminated normally, i.e. there is no better bound than v */
        return v''; /* sharpest lower bound, v' would have worked as well */
```

Fig. 9. The SSS-2 algorithm, procedure expand

'expand$(35, 36)$'. This returns successfully, and the whole recursive chain winds up with value 35.

The next call 'diminish$(a, 35)$' in the main loop should fail. Recursive calls for c, f and m are issued, and the reader is invited to check how failure of the inner call for m is computed. Because f is a MAX node and m has failed there is no way to obtain a better bound for f. This means that the subtree of G in f can be destroyed. This is realized by the PURGE operation (notice that f itself will be removed at one level higher). The body of the call for c will now generate a call 'expand$(g, 35)$' and this will return the failure value 37 (please check, notice that expand also executes a PURGE). Now c itself fails, a PURGEs all descendants and the computation terminates.

The code given in Figs. 7, 8 and 9 must be clear. Some care has been taken to return on failure the best lower bound which can be deduced for the node. This is not strictly necessary, for the caller any value $\geq g$ will do.

Notice that this computation must have visited the min solution tree with value 35, by virtue of Lemma 4. This is indeed the case because the algorithm has issued a diminish call for nodes g and 36. These nodes are not on the critical path (from a to 35) themselves, but they are children of MIN nodes on the critical path. The second arguments of these diminish calls have been values

≥ 35. The calls have reported failure and each of them must therefore have seen a min solution tree with value ≥ 35.

The SSS-2 algorithm differs in two respects from original SSS*. The first difference is of conceptual nature. SSS* has been set up to find the best *min* solution tree. It achieves this by searching min solution trees from left to right. At each moment more than one tree is under investigation. The search proceeds in an interleaved way. Each tree is characterized by the last node visited, together with the best (minimal) value seen so far in this tree. The algorithm is organized in such a way that at each moment these end nodes form the leaf set of a max solution tree. Because the search of a min solution tree will also visit interior nodes of the game tree, it is possible that the corresponding max solution tree does not descend all the way down to the leaves of the game tree. This corresponds with points in time where SSS-2 is busy expanding new children of a MIN node.

The second difference with SSS* is the data structure used. Where SSS-2 uses a max solution tree, SSS* uses an OPEN list, which is the list of endpoints of this tree. The working of SSS* can roughly be described as a loop with body: search the maximal element in OPEN; perform local operations (like searching younger brothers) until you have found a better value or you recognize failure. This loop is repeated until the search is exhausted. SSS* also needs the PURGE operator, every now and then (corresponding to the points where SSS-2 would do a PURGE) it also discards all descendants from the OPEN list.

The original paper [17] proved that SSS* was more efficient than Alpha-Beta in the sense that the set of nodes visited by SSS* is always a subset of the node set visited by Alpha-Beta. For instance, for Fig. 1 SSS-2 visits the leaves $\{41, 5, 12, 10, 36, 35, 50, 37\}$. Notice that this is more than the critical tree which does not contain 41. SSS-2 suffers here from a left to right effect. On the other hand Alpha-Beta also visits the nodes 90, 99 and 80. This is due to the fact that Alpha-Beta has to evaluate b fully, because there is as yet no indication that another part of the tree is better.

So, SSS* seems to be the better algorithm, but this idea was challenged in the paper [15] which critized the algorithm both on theoretical and on practical grounds. From the theoretical side it was argued that in many cases the superiority of SSS* over Alpha-Beta was not as big as expected. Both algorithms search the same nodes for perfectly ordered trees, for perfectly unordered trees, as well as for trees where 'eval' yields only two values, e.g. 'win' and 'loss'. A statistical analysis indicated that for practical values of the depth of the game tree Alpha-Beta never searches more than 3 times the number of nodes that SSS* would.

From the practical point of view there was the observation that Alpha-Beta hardly needs memory space ($O(d)$ for the stack if the depth of the tree is d), while the OPEN list in SSS* would take room $O(w^{\lceil d/2 \rceil})$, i.e. the number of leaves of a max solution tree of depth d. A more severe objection is that counting the number of nodes visited is not a good indication for the running time. First of all, nodes are revisited, but more important, visiting a node does not take a constant amount of time: finding the best node in the OPEN list or the PURGE operator needs more than that.

These observations have been justified experimentally in [2]. For random trees (where the evaluation function will just draw a random number) they reported that SSS* was 1.8 to 57 times slower than Alpha-Beta.

It is clear that a more efficient data structure for SSS* (and SSS-2) was needed. The paper [2] came with a proposal which purged the PURGE operator from the scene. Because SSS-2 always manipulates one max solution tree, the algorithm needs only room for one such a tree. The idea was to pre-allocate this room, structured as a max solution tree, but with 'empty nodes'. The first call of 'expand' now fills in the blanks. Purging is not needed because, for instance when the subtree below f has to be purged (in the call 'diminish$(c, 35)$', cf. the discussion above) the algorithm overwrites the entries belonging to the subtree under f with new values from the subtree under g. In SSS-2 it is clear when overwriting is allowed, because we go from 'diminish-mode' to 'expand-mode'. With respect to the original SSS* code more care had to be exercised.

This idea proved to be successful, experiments showed that this version of SSS* was competitive with Alpha-Beta, sometimes faster (0.93), sometimes slower (1.38). Moreover, in [14] several optimization tricks have been applied to this idea, leading to relative running times of 40% for unordered random trees to 70% for 60% ordered random trees (i.e. random trees in which the oldest child has 60% chance to be the best one).

However, still SSS* performs best for unordered trees, while game trees tend to be rather well ordered. So the advantage of SSS* is doubtful. Furthermore, we saw already that the ordering in the tree is exploited by algorithms like NegaScout, which makes the algorithm to beat even more efficient. This raises the question whether there is any hope for best first algorithms like SSS*.

6 SSS and Alpha-Beta Reconciled

In this section we will elaborate on the similarities between Alpha-Beta and SSS. This will result in an algorithm based on null window search that is equivalent with SSS-2. We will first concentrate on the procedure expand, so let us consider a call 'expand(n, g)'. If this call fails it will return a value $\geq g$, a lower bound defined by a min solution tree. This behaviour is similar to that of an Alpha-Beta call with $\beta = g$ failing high. If, on the other hand, the expand call succeeds, it will have built a max solution tree with value $< g$ or, assuming integer game values, a value $\leq g - 1$. This again resembles behaviour of Alpha-Beta, now failing low on input parameter $\alpha = g - 1$. Apparently the calls 'expand(n, g)' and 'alphabeta$(n, g - 1, g)$ behave similarly.

We study in more detail the case that n is a MAX node for which the expand call succeeds. In that case the call will generate for all children c of n, from left to right, a subcall 'expand(c, g)' with the same g-parameter. These subcalls all succeed and the call 'expand(n, g)' returns the maximum of the return values of the children. Now let us assume that for the children c expand

and alphabeta behave identically. Then Alpha-Beta will also generate a sub-call for each of its children with the same return value as delivered by expand, and 'alphabeta$(n, g-1, g)$' returns the same value as 'expand(n, g)'. Therefore 'expand(n, g)' and 'alphabeta$(n, g-1, g)$' behave identically. If one analyzes the other possible cases, and one uses induction, one can prove that the calling tree for 'expand(n, g)' is completely mirrored by the one for 'alphabeta$(n, g-1, g)$', the same set of nodes is visited in the same order and the same value is returned.

The next step is to extend this result to the calling tree generated by the procedure diminish. However, this procedure expects a max solution tree in G which will guide its search. Therefore, the null window search should generate and use equivalent information. The main result of this section will be that this can be realized by using TTalphabeta instead of Alpha-Beta, i.e. by using a transposition table. This means that we obtain an algorithm equivalent with SSS-2 if we change, in the main body of SSS-2, cf. Fig. 7, the calls 'expand(n, g)' and 'diminish(n, g)' by calls 'TTalphabeta$(n, g-1, g)$'. This new version is called 'MTD(∞)'.

We will not prove this formally. Instead, we will try to sketch why the flow of control (the calling tree and the return values) will be essentially the same in both versions. We will first show how a call of TTalphabeta builds a structure inside the transposition table which is equivalent with the maximum solution tree a corresponding expand or diminish call would construct in G. In this analysis we will use induction-like arguments, like 'earlier calls (or inner calls) do whatever we expect from them'. These assumptions are used only to highlight the essential ideas, not to lay the base for an inductive proof. Such a formal proof can be given but it must be set up with some more care. Furthermore, our analysis will be based on yet another assumption, namely that no entries in the transposition table will be overwritten, i.e. there will be no collisions.

First of all we recall that TTalphabeta stores on low failure its return value in the f^+-field of the entry in the transposition table of the node involved. So, assuming that 'expand(n, g)' or 'diminish(n, g)' generate the same result value g' as 'TTalphabeta$(n, g-1, g)$' does, we will see in the max solution tree in G $n.g = g'$ and in the transposition table $n.f^+ = g'$. This means that in the transposition table the max solution tree from G can be partially found back by tracing the f^+-fields. As it stands we cannot recover the max solution tree completely because it is not yet clear which child should be chosen in a MIN node.

Two observations are relevant here. The first one is that on high failure, TTalphabeta stores its return value in the f^--field of the node involved. The second observation follows from a reconstruction of the way expand and diminish include a node c, child of a MIN node n, in the max solution tree G, say with value g. This occurs when a call 'expand(c, g')'or 'diminish(c, g')' with $g' > g$ ends in success with result value g. Again, assuming that there is an equivalent call 'TTalphabeta$(c, g'-1, g')$ failing low with result g, we see that in the transposition table we obtain $c.f^+ = g$. Now what about the other children of n? The younger brothers of c have not yet been subjected to an expand or diminish call, so we assume that there has neither been a TTalphabeta call, and therefore they are not in the transposition table. The older brothers c' of c must have been searched earlier by SSS-2 with parameter $g' > g$ and they must have been rejected, i.e. expand or diminish (and we therefore assume TTalphabeta as well) must have ended in a (high) failure with return values $> g'$ and therefore $> g$. This means that for all older brothers c' the transposition table has entries $c'.f^- > g$. Now it is clear how the current max solution tree is encoded in the transposition table. For the children c of a MAX node n which is in the max solution tree we see in the f^+-field the same value as in the g-field of G. The child of a MIN node n is the child with the same f^+-value as n itself, while all older children have f^--value bigger than $n.f^+$.

Notice that the line of reasoning from the beginning of this section showing that expand and alphabeta generate equivalent calling trees can be extended. We observe that expand and TTalphabeta generate equivalent calling trees and also that the max solution tree generated in G is encoded in the transposition table. This means that in the sequel we need only compare the behaviour of diminish and TTalphabeta.

So, suppose that we have a call 'diminish(n, g)' and 'TTalphabeta$(n, g-1, g)$', where G contains a max solution tree defining the upper bound g for n, and where the transposition table encodes this max solution tree. We will now sketch that both procedures generate equivalent subcalls for essentially the same nodes in the same order. We say 'essentially' because TTalphabeta will pay short visits to nodes that will not be visited by diminish. These short visits are needed to determine which node should be the next, say, 'serious' one to be visited, serious nodes being the ones that are also visited by diminish.

Suppose n is a MAX node. By inspecting the code we see that the diminish call will generate subcalls only for children c with $c.g = g$. On the other hand, the code of TTalphabeta specifies that all children c will be visited. However, for all nodes with $n.f^+ < g$, these visits will be short ones, because the test '$n.f^+ \leq \alpha$' in the body of TTalphabeta will be met. So the only serious calls will be for the children with $c.f^+ = g$ and one easily checks that for such a call diminish and TTalphabeta behave in the same way. (Notice that we have $c.f^- = -\infty$ for all non leaf children because all earlier visits to c must have failed low.)

Next, suppose that n is a MIN node. The first node visited by diminish will be its only child c that is in G. TTalphabeta has to do some short visits to the older brothers c' of c first, in order to find out which child is the one in the max solution tree. All visits to these brothers will meet a shortcut in the test '$n.f^- \geq \beta$' in the body of TTalphabeta. So the first serious visit will be to c and the reader is again invited to check that from this point on diminish and TTalphabeta exhibit the same behaviour.

In Section 4 we mentioned that a transposition table can provide useful information to speed up re-searches, like the ones done by NegaScout. The analysis given here has elaborated on this, 'short visits' will enable a re-search to efficiently avoid old useless paths. This very mechanism is exploited to good effect in NegaScout as well.

Now that we have reduced SSS to a series of null window searches we can compare this algorithm with Alpha-Beta in a fair way, i.e. in an equal environment. We briefly state some results, the reader is referred to [11, 12] for more details. Experiments have been performed for tournament-quality real-life game playing programs for three different games.

First of all, the idea that SSS* uses too much memory proved to be untrue. The experiments showed that MTD(∞) becomes better than Alpha-Beta if the size of the transposition table exceeds roughly 2^{17} entries. Assuming that each entry contains 16 bytes, we see that a transposition table of 2 Mbyte is already adequate. The second result is that in general the difference in efficiency of MTD(∞) and Alpha-Beta is relatively small, the trend being that MTD(∞) is a few percents more efficient. This seems to be in contrast with the results from [14], cf. Section 5. Apparently in real life game trees are so well ordered that the reason why SSS would perform better has almost vanished. Thirdly we found that NegaScout is in general better than both MTD(∞) and Alpha-Beta, though the difference is never more than 10

NegaScout improves upon Alpha-Beta by exploiting knowledge gathered from the previous iteration of iterative deepening. For the null window search framework a similar trick is possible. MTD(∞) is parameterized with the value ∞ and therefore one can view the algorithm as generating it first g-value, an upper bound for $f(\text{root})$, from the assumption that this value equals ∞. However the last iteration of iterative deepening has generated an estimate of $f(\text{root})$ that will be much better. It is reasonable to expect that less iterations in the main loop of SSS-2 will be needed if we start from this better estimate. The algorithm which applies this idea is called MTD(f). We have to be a little careful in the formulation of this algorithm. Starting from ∞ we are sure to get an upper bound after all iterations but the last one. This means that we can always re-search using the window $(g-1, g)$, where g is the result returned previously. In the MTD(f)-case we have to check whether after the first expand call 'g :=TTalphabeta$(\text{root}, f-1, f)$;' we obtain a value $g \geq f$ or $g < f$. In the first case we have to approximate the game value from below using calls 'TTalphabeta$(\text{root}, g, g+1)$' while in the second case we can continue using windows $(g-1, g)$.

Experiments like the one discussed above have shown that MTD(f) performs (almost) consistently better than NegaScout. We observed margins in the range 1–15%. This shows that the most efficient general purpose game tree searching program that we presently know of is MTD(f).

Acknowledgements. It must have been clear from this paper that many of the newer results reported here have been generated by, or in close collaboration with Aske Plaat and Jonathan Schaeffer. We heartily acknowledge their work, and the pleasant cooperation we enjoyed over the years.

References

1. G. M. Baudet, *On the branching factor of the alpha-beta pruning algorithm.* Arti-

ficial Intelligence 10 (1978), pp 173-199.
2. Subir Bhattacharya and A. Bagchi, *A faster alternative to SSS* with extension to variable memory*, Information processing letters 47 (1993), 209–214.
3. Toshihide Ibaraki, *Generalization of alpha-beta and SSS* search procedures*, Artificial Intelligence 29 (1986), 73–117.
4. V. Kumar and L.N. Kanal, *A General Branch and Bound Formulation for Understanding and Synthesizing And/Or Tree Search Procedures*, Artificial Intelligence 21 (1983), 179–198.
5. Donald E. Knuth and Ronald W. Moore, *An analysis of alpha-beta pruning*, Artificial Intelligence 6 (1975), no. 4, 293–326.
6. T. A. Marsland, A. Reineveld, J. Schaeffer, *Low Overhead Alternatives to SSS**, Artificial Intelligence 31 (1987) pp. 185-199.
7. 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.
8. Wim Pijls. *Shortest Paths and Game Trees.* PhD Thesis, Erasmus University Rotterdam, The Netherlands, November 1991.
9. Wim Pijls and Arie de Bruin, *Searching informed game trees*, In: Algorithms and Computation, ISAAC 92 (T. Ibaraki, ed), pp. 332–341, LNCS 650.
10. Wim Pijls and Arie de Bruin, *A theory of game trees based on solution trees*, Tech.Rep. EUR-CS-96-xxx, Erasmus University Rotterdam, 1996.
11. Aske Plaat. *Research Re:Search & Research.* PhD Thesis, Erasmus University Rotterdam, The Netherlands, June 1996.
12. Aske Plaat, Jonathan Schaeffer, Wim Pijls and Arie de Bruin, *A Minimax Algorithm Better than SSS**, In: Artificial Intelligence, to appear.
13. Aske Plaat, Jonathan Schaeffer, Wim Pijls and Arie de Bruin, Exploiting graph properties of game trees. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI '96)*, Portland, OR, August 1996. American Association for Artificial Intelligence, AAAI Press.
14. Alexander Reinefeld and Peter Ridinger. Time efficient state space search. *Artificial Intelligence*, 71 (2), pp. 397–408, 1994.
15. Igor Roizen and Judea Pearl, *A minimax algorithm better than alpha-beta? yes and no*, Artificial Intelligence 21 (1983), 199–230.
16. 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.
17. G. Stockman, *A minimax algorithm better than alpha-beta?*, Artificial Intelligence 12 (1979), no. 2, 179–196.

This article was processed using the LaTeX macro package with LLNCS style