Artificial Intelligence 66 (1994) 91–124
Elsevier

ARTINT 1066

Proof-number search

L. Victor Allis, Maarten van der Meulen and
H. Jaap van den Herik

*Department of Computer Science, University of Limburg, P.O. Box 616,
6200 MD Maastricht, Netherlands*

Received October 1991
Revised January 1993

Abstract

Allis, L.V., M. van der Meulen and H.J. van den Herik, Proof-number search, Artificial Intelligence 66 (1994) 91–124.

Proof-number search (pn-search) is designed for finding the game-theoretical value in game trees. It is based on ideas derived from conspiracy-number search and its variants, such as applied cn-search and $\alpha\beta$-cn search. While in cn-search the purpose is to continue searching until it is unlikely that the minimax value of the root will *change*, pn-search aims at *proving* the true value of the root. Therefore, pn-search does not consider interim minimax values.

Pn-search selects the next node to be expanded using two criteria: the potential range of subtree values and the number of nodes which must conspire to prove or disprove that range of potential values. These two criteria enable pn-search to treat efficiently game trees with a non-uniform branching factor.

It is shown that in non-uniform trees pn-search outperforms other types of search, such as α-β iterative-deepening search, even when enhanced with transposition tables, move ordering for the full principal variation, etc. Pn-search has been used to establish the game-theoretical values of Connect-Four, Qubic, and Go-Moku. There pn-search was able to find a forced win for the player to move first. The experiments described here are in the domain of Awari, a game which has not yet been solved. The experiments are repeatable for other games with a non-uniform branching factor.

This article describes the underlying principles of pn-search, presents an appropriate implementation, and provides an analysis of its strengths and weaknesses.

1. Background

The idea of guiding a search process with the help of conspirators originated in the middle of the 1980s (McAllester [27]). Many researchers were prompted to implement this idea after McAllester's publication [28]. Although the idea was well founded and several attempts were made by various researchers [6,

Correspondence to: L.V. Allis, Department of Computer Science, Faculty of Mathematics and Computer Science, Vrije Universiteit, De Boelelaan 1081, 1081 HV Amsterdam, Netherlands. E-mail: victor@cs.vu.nl.

0004-3702/94/$07.00 © 1994—Elsevier Science B.V. All rights reserved
SSDI 0004-3702(93)E0013-C

15, 19, 20, 25, 32, 33, 39], the results were not encouraging. Persistent research has shown that at least in several domains of game playing the ideas of conspiracy-number search, which has matured into proof-number search, are productive. An advantage of conspiracy-number search is that it pays attention to non-uniform trees. With the help of proof-number search we now demonstrate that, in some cases, non-uniform trees are less intractable than assumed so far.

1.1. Conspiracy-number search

McAllester [28] and Schaeffer [32, 33] both focused on search trees to be expanded at a slow pace via their most-promising nodes. Thus, terminal nodes were changed into internal nodes. We briefly explain the idea. Before the expansion of a node J, a heuristic value v_1 has been assigned to J. After the expansion, J's new value v_2 is obtained from the heuristic values of its children. The values v_1 and v_2 may be equal, but they may also differ. In many games, heuristic evaluation functions (with their enhancements such as quiescence search) achieve a rather reliable correlation between the values v_1 and v_2. In most cases, the expansion of a single terminal node therefore does not have a large impact on the values of its ancestors. In practice, we often see that only the value of J's immediate one or two ancestors are affected. To change the value of the root several terminal nodes must change their value. We may define the minimum number of terminal nodes which must change their value in order to change the value of the root as a measure of likeliness for the root to have its value changed by further expansions.

Conspiracy-number search (cn-search) is based on the definition given above. The algorithm selects a terminal node from a minimal set of terminal nodes which must change their value in order to change the value of the root. It expands the node and, traversing backwards, it updates the tree. As soon as the number of terminal nodes which must change their value in order to change the root's value exceeds a given upper bound, the search is terminated. The upper bound is chosen such that it is rendered unlikely that the root value will change by further expansions.

1.2. Some implementations

Schaeffer's straightforward implementation [32, 33] showed that good results were achievable in tactical chess positions, but not in positional positions due to the refined range of possible values. This handicap has extensively been examined by van der Meulen [39]. He concentrated on a different type of tree by attempting to establish the minimax value of a game tree, in which internal nodes were only expanded, never evaluated (disregarding the implicit evaluation by using iterative deepening). Van der Meulen coined his algorithm

applied cn-search, since it searches iteratively to a fixed depth using the ideas of conspiracy-number search, whereas McAllester's original algorithm [28] searched to an unrestricted depth.

Van der Meulen's algorithm was said to be fit for tournament chess-playing programs. Unfortunately, the results were not so good [17]. As a sequel to van der Meulen's ideas Allis et al. [6] intensified the approach of establishing the minimax value of a full game tree. They were *not* interested in *changing* the root's value but in *proving* its value. This resulted in the introduction of $\alpha\beta$ conspiracy-number search which has turned out to be successful when solving the game of Connect-Four [2, 37]. In [6], it is shown that $\alpha\beta$-cn search outperforms unordered α-β search on non-uniform trees, while $\alpha\beta$-cn search performs worse than unordered α-β search on uniform trees.

Elkan's application of cn-search to and/or trees [15] can also be seen as a predecessor of two-valued pn-search. His implementation has proof numbers (although named differently) but no disproof numbers (cf. Section 3). As a consequence, his selection at or-nodes is equivalent to ours, but at and-nodes his selection mechanism is inferior.

1.3. Multi-valued trees

In the domain of Connect-Four, $\alpha\beta$-cn search has been applied on two-valued trees (a win for White, not a win for White) [2, 37]. By experiments in other domains with multi-valued trees, we have come across cases where $\alpha\beta$-cn search now and then lingers in apparently bad variations. Closer investigations showed that it was caused by the attempt to disprove the root value by as wide a margin as possible instead of disproving it as quickly as possible. This issue is discussed more extensively in Section 2.2. In the current article, $\alpha\beta$-cn search will be improved to the extent that it is also applicable to multi-valued trees. The solution is to treat multi-valued trees as if they were two-valued (see Section 2). Apart from solving this problem, the algorithm presented has been given elegance and clarity. Since proving and disproving is the central theme of the algorithm, the name proof-number search (pn-search) has been adopted.

In this contribution, we use the game of Awari as testing ground. Awari is a two-person zero-sum game with perfect information. Its decision complexity is indeterminate and its search complexity is estimated at 10^{12} [3]. Hence, the game is a sincere challenge for the application of new searching methods. From 1989 onwards it has been played annually at the Computer Olympiads [23, 24, 38]. We note that there are three more reasons supporting the choice of Awari. First, a series of endgame databases guarantees for a part of the game a perfect terminal-node evaluation function [8]. Second, the number of moves ranging from 1 to 6 in combination with the rules of the game ascertains the occurrence of trees with non-uniform branching factors. Third, our Awari program Lithidion (containing a combination of α-β search, pn-search and

endgame databases) has won the gold medal at the Awari competition of the second, third, and fourth Computer Olympiads [7, 40]. This ensures that our α-β search implementation is a strong comparison measure for pn-search (see Sections 6 and 7).

1.4. Contents

The course of this article is as follows. Section 2 describes the concept of *most-proving* node and introduces the idea of proof-number search informally. In Section 3, we define *proof numbers* and *disproof numbers*, and provide a full description of the proof-number search algorithm in pseudo-code. In Section 4, two techniques which reduce the size of the tree to be kept in memory during the search are presented, together with two techniques which slightly reduce the execution time. Section 5 describes applications for pn-search, ranging from solving games to tournament play. Section 6 contains a brief description of Awari and a short introduction to two other algorithms: α-β iterative-deepening search with move ordering for the full principal variation, with and without transposition tables. In Section 7 the performances of the three algorithms are compared. A generalization of the test results is given in Section 8. Section 9 contains the conclusions.

2. An introduction to proof-number search

The minimax algorithm introduced by von Neumann [41], and widely disseminated by von Neumann and Morgenstern [42], was the first algorithm to be able to determine the minimax value of game trees. It visits each and every node in the tree. More sophisticated algorithms, such as α-β search [21] and SSS* [12, 36] achieve the same result, while pruning many nodes which can never influence the final outcome. Whereas α-β search is a depth-first search algorithm, SSS* traverses the tree in a best-first manner. In two-valued trees SSS* and α-β search visit the same set of nodes, while in multi-valued trees SSS* will never visit a node pruned by α-β search and often prunes more [36].

In this section we face the main question of any best-first search algorithm: which node to expand next? Although Russell and Wefald [31] have shown that there exists a definite answer, it is not easy to compute. Therefore, each best-first algorithm has a central theme which guides the selection. For SSS*, upper bounds on subtrees are of paramount importance, while node counts play the main role in all variants of cn-search. In B*, [10] evaluation functions produce reliable upper and lower bounds, which are used in turn to select the next node to be expanded. This idea has been elaborated upon by Palay [29] who provided a set of rules for making decisions throughout the B* algorithm. In cn-search the number of conspirators which may change the minimax value

is the decisive criterion for the selection of a node. Since pn-search aims at proving the true value of the root it does not consider interim minimax values. Pn-search uses two criteria for the selection of a node: (i) the potential range of subtree values, and (ii) the number of nodes which must conspire to *prove* or *disprove* the range of potential values. For two-valued trees, the use of node counts as a selection criterion leads to the search algorithm of Section 2.1. In multi-valued trees, however, some dilemmas occur, as described in Section 2.2. Anticipating the conclusions from that section, we mention that $\alpha\beta$-cn search approaches these dilemmas suboptimal. In pn-search we have resolved this problem by performing several two-valued searches instead of one multi-valued search. It leads to better results. The risk of lingering in bad variations, while simple solutions may be at hand, has been eliminated.

2.1. The most-proving node

In this section we discuss the selection criteria which efficiently guide the search to the final goal of proving the root's true minimax value in a two-valued game tree. Throughout the article we assume that all game trees have a max node as the root. Moreover, if two or more children of a node cannot be distinguished by the search process, the leftmost child will be evaluated first. To achieve our aim, we define the *most-proving* node of a search tree as the node which potentially contributes most to the establishment of the minimax value of the root with the least possible effort. The terminal nodes of all trees examined in this section have values 0 or 1.

We start showing how node counts are related to the concept of most-proving node. Then at each max node we focus at a child through which the value 1 is expected to be achievable with the least possible effort. Finally, we repeat the latter procedure by selecting for each min node a child through which the value 0 is expected to be achievable with the least possible effort.

Throughout the analysis, we assume that all temporary terminal nodes of the tree have the same expected solution time. Furthermore, we assume that the solution times of all nodes are independent. Finally, we assume that in each node the probability of proof outcome 0 is equal to the probability of proof outcome 1. All three assumptions can be relaxed when domain-dependent information is available, possibly leading to different initializations of the proof and disproof numbers, thereby improving the efficiency. Related analyses can be found in [16, 31, 34]. In this article we assume that no domain-dependent information is available.

In Fig. 1, both nodes b and c may still reach the value 1. SSS* will therefore continue the search within the leftmost subtree, resulting in the evaluation of node e. However, in our opinion, it is worthwhile first to calculate the probabilities of pruning one or more nodes. If the nodes e and f both evaluate to value 1, node h can be pruned, while both e and f can be pruned, if h

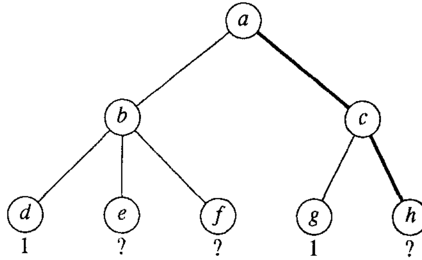

Fig. 1. A 2-ply tree in which *h* is the most-proving node.

evaluates to 1. In other words, either two nodes must conspire to prune one, or the evaluation of one node may result into pruning two. For the tree of Fig. 1, node *h* is therefore the *most-proving* node to be selected by the pn-search algorithm. In Fig. 1 this is indicated by the bold path. At this point, we would like to remark that node *h* might turn out to be the wrong choice. If node *h* has value 0, and the nodes *e* and *f* both have value 1, pn-search does not prune any node, whereas SSS* prunes *h*. If, however, the values of the nodes *e*, *f*, and *h* have the same probability distribution, then on average pn-search will search less nodes than SSS*. This observation is the key issue necessary to conclude that pn-search outperforms SSS*, although occasionally SSS* may do better. It also indicates that node counts are a useful criterion for selecting the most-proving node as defined above.

In the two-valued tree of Fig. 2, all terminal nodes have an as yet unknown value. Starting at root *a*, we must select the most-proving node. There are two possible outcomes for the root. First, let us assume that the root will have the value 0. This can only be the case if both nodes *b* and *c* are proven to have the value 0. To complete such a proof, nodes in both trees must be evaluated. There is no preference for the order of visiting. Second, let us assume that the root will have the value 1. This can be shown by proving just one of its children to have value 1. Selecting the child which proves this value with the least

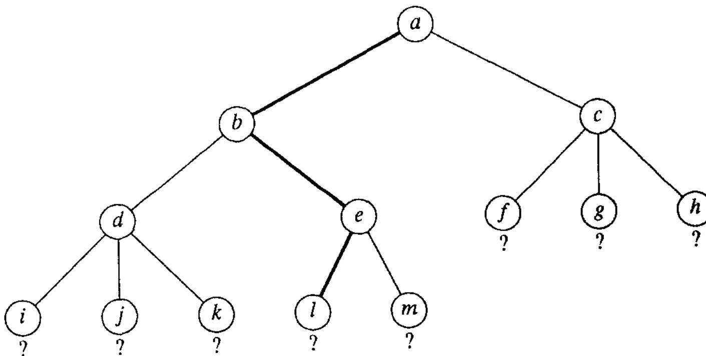

Fig. 2. A 3-ply tree in which *l* is the most-proving node.

possible effort may reduce the total number of evaluations considerably. In general, at max nodes the only selection criterion is the amount of effort needed to prove value 1. In the tree of Fig. 2, only two nodes need to conspire to prove 1 as value of *b* (viz. one of *i*, *j*, and *k*, and one of *l* and *m*), while three nodes must conspire to prove 1 as value of *c* (viz. *f*, *g*, and *h*). Thus, the most-proving node lies within subtree *b*.

Node *b* is a min node. Proving value 1 therefore requires a proof of value 1 for both *d* and *e*. This does not indicate any preference. Moreover, we know that proving value 0 requires only a proof in one of them. The selection of a child should therefore be based solely on the amount of effort required to prove 0 as value. In subtree *d*, it takes three nodes to prove the value 0 (*i*, *j*, and *k*), while in subtree *e* only two nodes need to evaluate to 0 (*l* and *m*). Therefore, the most-proving node lies within subtree *e*. In general, at min nodes the only selection criterion is the amount of effort needed to prove the value 0.

At node *e*, no distinction can be made between nodes *l* and *m*, resulting in the selection of the leftmost child *l* (the bold lines in Fig. 2).

We conclude that selecting the most-proving node in a two-valued tree can be performed in a straightforward manner using node counts.

2.2. Dilemmas in multi-valued trees

As announced at the beginning of this section, the selection of the most-proving node in multi-valued trees may lead to dilemmas. Figure 3 presents an example tree in which such a dilemma exists.

We assume all terminal nodes in Fig. 3 to have integer values between 0 and 3, both inclusive. Node *b* has a value between 0 and 3, node *c* between 0 and 2 for node *c* cannot obtain the value 3. Thus, independent of the value of node *g*, one or more evaluations must take place within subtree *b*, either to prove or disprove 3 as root value. A choice must be made between the evaluation of

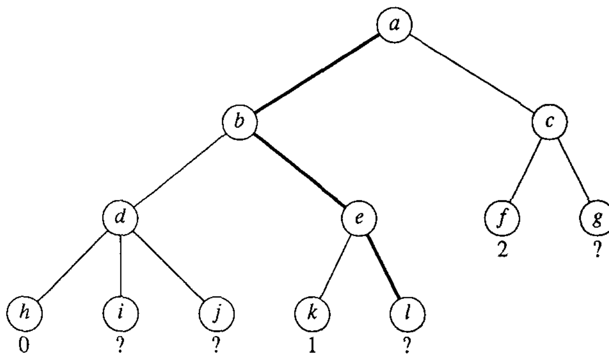

Fig. 3. A 3-ply tree with a dilemma.

node i within subtree d, and node l within subtree e. To prove 3 as value of node b, both subtrees d and e must attain the value 3. At least l and i, and maybe j must then be evaluated. On this basis no preference can be given to either i or l. To disprove 3 as value, a difference exists between subtrees d and e, which becomes apparent when considering the possible values of node g.

First, let us assume that the value of node g equals 0, resulting in a value of 0 for subtree c. It will then be essential to evaluate at least node i (and possibly j) to see whether more than 0 can be obtained within subtree b. If nodes i and j do not yield more than 0, evaluating l becomes useless. In that case, node i should be preferred to l as the next node to evaluate.

Second, suppose that the value of node g equals 2, resulting in the same value for subtree c. In that case evaluation of node l to 2 or less is sufficient to disprove b as best child of a, while in subtree d both i and j must evaluate to 2 or less to achieve the same feat. This time l should be chosen as next node to evaluate.

Hence, depending on the as yet unknown value of g, different choices are to be made. The seemingly obvious alternative of first evaluating g and then deciding whether to choose i or l, is not optimal. In more complicated cases, g will actually be a subtree consisting of many nodes. This means that evaluation has to be read as expansion, and then determining g's value may take a long time, whereas the whole subtree c can be pruned if either i or j, and l have a value of 2 or more.

The advantage of first evaluating i and then j is that there is a probability that the upper bound of subtree b will drop as far as the value 0, while first evaluating l can only achieve a decrease in upper bound to value 1. The advantage of first evaluating l is that disproving the upper bound may be achieved after evaluating only one node, instead of two. In general, the choice is between selecting a node which may disprove the current upper bound of its parent by as wide a margin as possible, and selecting a node which may as quickly as possible disprove the current upper bound.

Let us suppose that, as has been done in $\alpha\beta$-cn search [6], we choose to disprove the upper bound of the root by as wide a margin as possible, and therefore prefer subtree d instead of subtree e. If in the nodes i and j actually two subtrees originate consisting of many nodes, then the expansions (and evaluations) within subtree d will continue either until an upper bound within the subtree is established of less than or equal to 2 (in which case subtree c becomes a better choice), or until the lower bound of subtree d is raised to at least 1. In the unfortunate, but quite possible, event that subtree d grows to a large tree before one of these conditions is met, we may end up spending all our search efforts within it, without achieving any result. This is contrary to the idea of conspiracy-number search, which dictates that subtrees should only be further expanded if the same result cannot be obtained in another part of the tree with less effort. Since the evaluation of both g and l to the value 1 suffices to prove that a has a value of 1, the idea is clearly violated.

Experiments with implementations of $\alpha\beta$-cn search on Awari have shown that the risk of choosing an unfortunate subtree is real. Although most trees are traversed quickly without ill effects, in some cases the algorithm stumbles into such a subtree. The node-count criterion for either proving the upper bound or disproving it with as wide a margin as possible continues to be valid without any solution in the offing. The fact that this actually happened prompted us to the improvement described below.

Let us examine what can happen in the other case, if we decide to prefer disproving the upper bound of the root as quickly as possible. First node l will be visited. If it expands to a large tree in which more than two nodes must conspire to disprove upper bounds greater than 2, preference is automatically shifted to subtree d. Then both trees will grow more or less simultaneously without the risk of missing a simple disproof in one of them. As soon as the upper bound is established to be at most 2, node g becomes the most-proving node. The shortest proof occurs when l is evaluated first and has value 1, and thereafter node g evaluates to the value 1. Subtree d will then be discarded.

The implications of this choice are that during the search in any max node we should try to prove the current upper bound of the root with the least possible effort, and at any min node we should try to disprove the current upper bound as quickly as possible, regardless of the new upper bound which may result of such a disproof. All values less than the given bound are therefore equivalent for the duration of the search, as are all values greater than or equal to the given bound we try to (dis)prove. The multi-valued search has effectively degenerated into a two-valued search.

Consequently, each multi-valued search is to be split up into a number of two-valued searches performed one after the other. In the next subsection, we explain the selection criteria of pn-search considering two-valued trees only.

2.3. *An informal description of pn-search*

Above we have seen that the most-proving node should be selected based on both trying to prove a root value v at every max node and trying to disprove it at every min node. This process attempts to make clear if a value is reachable or not. For two-valued trees, e.g., with the values 0 and 1, we may select the value 1 and try to prove or disprove it. If the algorithm proves it, the value of the tree has been established. A disproof of 1 is equivalent to a proof of the value 0. For multi-valued trees, e.g., with integer values ranging from 0 to 10, matters are more complicated. We could start (dis)proving the value 10, and then decreasing this value every time a proof fails. After at most ten searches, we will have established the exact root value. A better alternative is to make use of binary search, which will yield the root value within four searches. However, stepwise decreasing the search value can be useful, viz. for establishing a bound on the root value. If such a procedure is combined with increasing the search value starting from 1, it may be possible to prove quickly that the

root value lies within the range of 4 to 6, while the exact value (4, 5, or 6) takes a long time to determine. The binary-search method would immediately start with the hard part, and if resources for finishing the search are not available, no information on the root value is gained. In our Awari experiments we have used binary search in all cases.

As a case in point we present the following example. Assume, we may wish to decide whether a given position is good or bad for one of the players. Suppose the game has as possible outcomes the integer values between -48 and $+48$ both inclusive (as has Awari). The game is won by South (the player who moves first) for all positive root values, while North wins if the root value is negative. Root value 0 indicates a draw. If South wants to know whether a given position can be won, he may try to prove or disprove the value 1. During the search, the values -48 to 0 are therefore treated as equivalent, as are the values between 1 and 48. In general, while (dis)proving a value v, all values greater than or equal to v are equivalent, and referred to as v. All values less than v are also equivalent and referred to as $\neg v$.

Given a value v, the algorithm starts from a tree consisting of a single node. At each step the most-proving node is selected and expanded. In order to make the selection of the most-proving node straightforward, at each node J a proof number and a disproof number is stored (these will be defined formally in Section 3). A proof number indicates the minimum number of nodes within the subtree which must conspire in order to prove v as value of J, while a disproof number indicates the minimum number of nodes for disproving v as value of J. As soon as the proof number of the root equals 0, the search is terminated; the disproof number then will equal infinity. Conversely, if the disproof number equals 0, the proof number will equal infinity. In Section 3, we will elaborate on the formal parts of the pn-search algorithm.

3. Proof numbers and proof-number search

Above we have informally introduced (dis)proof numbers and proof-number search. To establish the exact minimax value of the root, we have described a two-valued proof-number search algorithm to be invoked several times for different values. During each run the most-proving node of a tree is repeatedly selected and expanded whereupon the (dis)proof numbers in the tree are updated.

We remark that in the examples of Section 2, trees with some unevaluated terminal nodes were presented. In the formal description of the algorithm, however, we have chosen to inspect all newly created children immediately after the expansion of a node. Those children, which are terminal to the full game tree, are assigned a value. This value can be obtained from an evaluation function which defines the game-theoretical value of the position. An endgame

database can be used to define such an evaluation function. No values are assigned to temporary terminal nodes. Such a node will be expanded if and when it is selected as most-proving node.

In this section we first formally define (dis)proof numbers. Then we give the full pn-search algorithm in pseudo-code.

3.1. Proof numbers and disproof numbers

As stated before, all values greater than or equal to v are equivalent for the purpose of (dis)proving v, and are referred to as v, while all values less than v are referred to as $\neg v$. Below we consider four types of nodes.

First, we look at temporary terminal nodes. These nodes have an unknown value; for proving the value of such a node to be v, only the node itself must evaluate to v. Analogously, the node needs to evaluate to $\neg v$ to disprove v. Therefore the proof number and the disproof number of a temporary terminal node are equal to 1.

Second, we consider terminal nodes with a known value. If a terminal node J has been assigned a known value equivalent to v, then it has already been proven that v is its value. Its proof number will therefore equal 0, while its disproof number equals infinity. If J has a value equivalent to $\neg v$, its proof number equals infinity, while its disproof number is 0.

Third, we concentrate on internal max nodes. We assume that the (dis)proof numbers of its children are all known and available. For a max node J the following holds: to prove value v, it is sufficient to have one child which proves value v. It is expected that proving this value with the least possible effort is proving it for the child with the smallest proof number. In other words, the proof number of a max node is equal to the minimum of the proof numbers of its children. The only way to disprove v for a max node J is to disprove v for all its children. Therefore, the disproof number of J is equal to the sum of the disproof numbers of all its children.

Finally, we look at internal min nodes. Here we reason analogously as for internal max nodes, with minimum and sum exchanged. This means: the proof number of a min node is equal to the sum of the proof numbers of its children and the disproof number of a min node is equal to the minimum of the disproof numbers of its children. The formal definition of (dis)proof numbers is presented in pseudo-code in Fig. 4. We continue with an example of its application.

In Fig. 5 the tree of Fig. 3 is repeated with proof numbers and disproof numbers for the value 3. They are listed within the nodes. First the proof number is shown, then the disproof number. The nodes h, k, and f are terminals with a known value. All three disprove 3 as value, since each of them has a value less than 3. The proof numbers thus equal infinity, while the disproof numbers equal 0. The nodes i, j, l, and g are terminals with as yet

```
if is_a_terminal_node(J) then
    if evaluation_value_known then
        if value(J) ≥ v then
            proof(J) := 0
            disproof(J) := ∞
        else
            proof(J) := ∞
            disproof(J) := 0
        endif
    else
        proof(J) := 1
        disproof(J) := 1
    endif
elseif is_an_internal_node(J) then
    if max_node(J) then
        proof(J) := min_{j∈childs(J)} proof(j)
        disproof(J) := Σ_{j∈childs(J)} disproof(j)
    else
        proof(J) := Σ_{j∈childs(J)} proof(j)
        disproof(J) := min_{j∈childs(J)} disproof(j)
    endif
endif
```

Fig. 4. The definition of (dis)proof numbers.

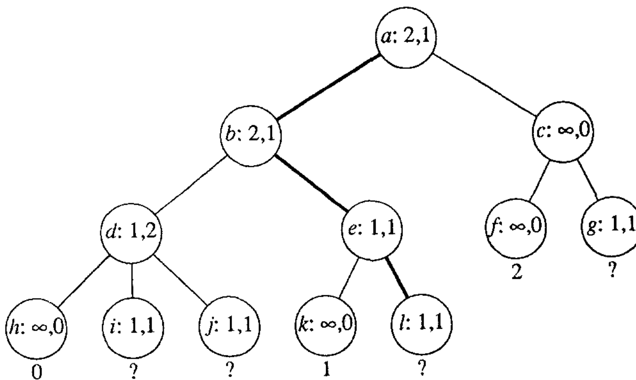

Fig. 5. A tree with (dis)proof numbers for the value 3.

unknown values. Both the proof numbers and the disproof numbers therefore equal 1. The nodes *d* and *e* are internal max nodes (as is node *a*). Hence, proving the value 3 in node *d* or *e* can be done by one of their children, whereas the conspiration of all children must disprove it. The proof numbers are the minima of their children's proof numbers, resulting in 1 for both *d* and *e*, while the disproof numbers are the sum of the disproof numbers of their children, resulting in 2 for *d*, and 1 for *e*. The nodes *b* and *c* are internal min nodes. The proof numbers are obtained by summing the proof numbers of their children, resulting in 2 for node *b* and infinity for node *c*. Their disproof numbers are equal to the minima of the disproof numbers of their children, giving 1 for *b* and 0 for *c*. From the proof and disproof numbers of node *c* it can be seen that it has already been disproven that 3 will be its value. Finally, node *a*, like the nodes *d* and *e*, minimizes the proof numbers and sums the disproof numbers.

3.2. Selecting the most-proving node

In this section we formulate how the most-proving node is selected using (dis)proof numbers. We first demonstrate the selection procedure for the value 3 on the tree of Fig. 5, then a general selection criterion for finding the most-proving node in an arbitrary tree is formulated. Starting at the root node *a* it is checked which subtree is assumed to prove 3 as value of *a* with the least effort. In subtree *b* this only takes an effort of two nodes (the proof number of *b*), while in node *c* it takes an infinite effort as indicated by the proof number of *c*. Therefore, subtree *b* is chosen as the most-proving subtree. At node *b*, the subtree is chosen in which disproving 3 as value of *b* is assumed to happen as quickly as possible (cf. the arguments in Section 2.2).

Comparing the disproof numbers of the nodes *d* and *e*, we see that in subtree *d* disproving takes at least two nodes to be expanded, while in subtree *e* one expansion may suffice. Thus, subtree *e* is selected as most-proving subtree. At node *e*, which is a max node, the proof numbers of its children are checked. The proof number of node *k* (infinity) indicates that it is impossible to prove 3 as value through *k*, while only one expansion might be needed through *l*. Thus node *l* is selected as most-proving node. The selection procedure terminates here: node *l*, a temporary terminal node, will be expanded.

From this example the following observations can be made. At a max node a subtree is selected with a proof number equal to that of its parent, and at a min node a subtree is selected with a disproof number equal to that of its parent. To these observations we add that if two or more children have a (dis)proof number equal to that of their parent the leftmost child is selected. We are now able to formulate the selection criterion for finding the most-proving child in a methodically precise way. At a max node, select the leftmost child with proof number equal to that of its parent, while at a min node select the leftmost child

with a disproof number equal to that of its parent. An advantage of this new formulation is that, on average, only half of the children have to be checked before selecting one. At the cost of a small amount of bookkeeping and by incrementally updating, one even can select the most-proving node immediately (see Section 4.2).

In Fig. 6, we present the algorithm for the selection of the most-proving node in pseudo-code.

3.3. *Updating the (dis)proof numbers*

After the most-proving node of the tree has been selected, it is expanded, and all its newly generated children are immediately inspected. If they are terminal to the full game tree they are evaluated and assigned a proof number and a disproof number, using the definition in Fig. 4. If they are temporary terminal nodes they are assigned a proof number and a disproof number only (cf. Fig. 4 again). The assignments of the proof and disproof numbers may affect the proof and disproof numbers of the most-proving node and of some or all of its ancestors. It is therefore important to update the (dis)proof numbers of the ancestors of the newly generated nodes. Figure 7 contains an algorithm in pseudo-code for this task.

3.4. *Two-valued pn-search*

Using the definitions, functions, and procedures defined in the previous subsections, we present in Fig. 8 the algorithm for two-valued pn-search. We assume that v is set to the higher value. This value does not occur in the while-loop of Fig. 8, since it is only used when determining the (dis)proof numbers of the terminal nodes (cf. the definition in Fig. 4).

```
function select_most_proving_node(J)
begin
    while is_an_internal_node(J) do
        if max_node(J) then
            J := leftmost_child_with_equal_proof_number(J)
        else
            J := leftmost_child_with_equal_disproof_number(J)
        endif
    done
    return J
end
```

Fig. 6. Selection of the most-proving node.

```
procedure update_proof_numbers(J)
begin
    while J ≠ NIL do
        if max_node(J) then
            proof(J) := min_{j∈childs(J)} proof(j)
            disproof(J) := Σ_{j∈childs(J)} disproof(j)
        else
            proof(J) := Σ_{j∈childs(J)} proof(j)
            disproof(J) := min_{j∈childs(J)} disproof(j)
        endif
        J := parent(J)
    done
end
```

Fig. 7. Updating the (dis)proof numbers after expansion.

```
function two_valued_pn_search
begin
    create_root(root)
    while proof(root) ≠ 0 and disproof(root) ≠ 0 do
        most_proving := select_most_proving_node(root)
        expand_node(most_proving)
        update_proof_numbers(most_proving)
    done
    if proof(root) = 0 then
        return v
    else
        return ¬v
    endif
end
```

Fig. 8. Algorithm for proving or disproving v as value of the root.

```
function multi_valued_pn_search
begin
    while lower < upper do
        v := integer_greater_or_equal((lower + upper)/2)
        if two_valued_pn_search = v then
            lower := v
        else
            upper := v - 1
        endif
    done
    return upper
end
```

Fig. 9. Finding the minimax value of the root in a multi-valued domain.

3.5. Multi-valued pn-search

As mentioned in Section 2, there are two different methods to incorporate the two-valued pn-search algorithm into a multi-valued pn-search algorithm. The first method is examining the possible root values linearly, the second one uses a binary-search procedure.

Even though the binary-search method has the advantage of determining the root value in less searches, the linear search still has some useful properties, especially in cases where narrowing the window of possible root values is preferred, e.g., in cases where it is impossible to prove an exact value within a given time limit. The choice between these two algorithms mainly depends on the structure of the domain under consideration and its constraints. In Fig. 9, we have presented an algorithm in pseudo-code for the binary-search method. This completes the implementation of our multi-valued pn-search. In the algorithm it is assumed that all possible terminal-node values are integers between *lower* and *upper*.

4. Reducing memory usage and execution time

So far we have described a new search algorithm which may be able to find the game-theoretical value in game trees. Now the time is ripe to investigate whether the algorithm has some disadvantages and, if so, whether these could be lessened.

A well-known disadvantage of all variants of cn-search is their requirement to keep the entire search tree in memory. As a result, only relatively small trees can be treated [28]. A second disadvantage is that cn-search variants

execute somewhat slower than, for instance, α-β search. This is due to the repeated traversals from the root to a terminal node and back again, as well as to the bookkeeping of the conspiracy numbers [32, 33]. In this section we consider ways in which these disadvantages can be reduced. Moreover, in the sections hereafter we show that in tournament programs for games, such as Awari, the advantages of pn-search already outweigh the disadvantages mentioned here.

4.1. Reducing memory usage

A search tree must be kept in memory since any node already generated, but not yet expanded, may be selected as most-proving node at some future time; therefore the node must remain available. Of course, it is possible to delete subtrees and to regenerate them when needed, as for instance is done in IDA* [22]. However, this technique can have a severe impact on the execution time. Therefore, we will not consider any techniques removing nodes at one time and regenerating them later. Many researchers have faced analogous problems and have made suggestions for a solution (e.g., Ibaraki [18], Marsland et al. [26], Bhattacharyya and Bagchi [11], Chakrabarti et al. [13]). Using the characteristics of pn-search we present below two techniques which reduce the size of a generated search tree: the *DeleteSolvedSubtrees* technique and the *DeleteLeastProving* technique.

The *DeleteSolvedSubtrees* technique involves removing all nodes in which the value v has been proven or disproven, i.e., all nodes with either proof number or disproof number equal to 0. During the update of the (dis)proof numbers of the ancestors of newly created nodes, all subtrees with a (dis)proof number equal to 0 can be removed.

In Section 7, when comparing pn-search and two variants of α-β iterative-deepening search, we have listed the total number of nodes visited and the maximum number of nodes to be stored in memory (see Table 1). From the results, it can be concluded that in searches which succeed in determining the value of the root, the size of the memory needed is between 2 and 4 times less than the size of the full tree. It shows that this simple technique is quite effective. In searches which fail to determine the root value, less subtrees exist in which v has been proven or disproven. The reduction technique is therefore less successful in those cases.

The *DeleteLeastProving* technique should only be applied as a last resort when all memory has been used. Of course, one could decide to terminate the search but trying to postpone the end is also possible, e.g., by removing parts of the tree least likely needed in a continued search. We call these parts least-proving parts. To recognize them the concept of a least-proving node is to be defined. The least-proving node of a tree is the node to be selected last as most-proving node when using the node-count criterion (cf. Section 3.2). The

algorithm for selecting the least-proving node is presented in Fig. 10.

The *DeleteLeastProving* technique has been realized as follows. As soon as the search process has run out of memory, one or more least-proving nodes are selected. At these nodes, both the proof number and disproof number are set to infinity, indicating that through these nodes nothing will be proved. Using the *DeleteSolvedSubtrees* technique, those nodes will be removed. Then the search continues until one of the following three cases occur:

(1) The root value becomes proven or disproven. Then the search is terminated successfully.
(2) Both the proof number and disproof number of the root equal infinity. Then the search is terminated unsuccessfully; it indicates that too many nodes have been deleted making a proof in the remaining tree impossible.
(3) The search process runs out of memory again, another set of least-proving nodes is selected and removed, and the search continues.

4.2. Reducing execution time

Pn-search's main extra cost in execution time with respect to, e.g., α-β search is due to updating node counts and the selection of the most-proving node. In domains where move generation and position evaluation is fast, the overhead of pn-search per node may be substantial (up to 100% overhead), while in domains with slow move generation and/or position evaluation, the overhead is negligible. We remark, however, that the expected reduction in number of nodes grown by pn-search is a factor far larger than 2 (cf. Section 6). The overhead per node is therefore of minor importance.

```
function select_least_proving_node(J)
begin
    while is_an_internal_node(J) do
        if max_node(J) then
            J := rightmost_child_with_maximum_proof_number(J)
        else
            J := rightmost_child_with_maximum_disproof_number(J)
        endif
    done
    return J
end
```

Fig. 10. Selection of the least-proving node.

Nevertheless, three small improvements can be made to the algorithms presented in Section 3. They may save some time. First, after the expansion of the most-proving node, normally not the node counts of all ancestors are affected. Let us suppose that J_k is the first affected ancestor on the path from root J_0 to the most-proving node J_n. This implies that we may stop updating node counts, as soon as we have found that node J_{k-1} is not affected. Second, as all node counts outside subtree J_k are unchanged, J_k will lie on the path to the next most-proving node. The selection algorithm may therefore start at J_k instead of at the root. Third, while updating the proof number and disproof number of a parent, one must always determine the minimum of the proof or disproof numbers of the children. At that time the most-proving node of the subtree may be stored at the parent, so that at the cost of a small amount of bookkeeping one can select the most-proving node of the whole tree instantly.

In large trees, these three improvements may save quite a few node traversals. Although the total savings in execution time will be small, the straightforwardness of these improvements call for implementation.

5. Applications of pn-search

Game-tree search algorithms can be applied to positions in at least three categories: tournament play, post-mortem analysis (and also adjourned games), and game solving. In the course of the development of pn-search, we have applied it to instances of all three categories.

Our Awari program Lithidion has first used pn-search in the 1991 Awari competition of the Third Computer Olympiad [38]. At the tournament, pn-search outperformed state-of-the-art α-β search implementations in determining winning lines of play. A description of the results is presented in Section 7.3.

Post-mortem analyses allow deeper searches than tournament play. For directional search algorithms, such as α-β search, only time constraints limit the search depth during a post-mortem analysis. Pn-search is also constrained by the size of available memory. However, our results show that the combination of today's memory sizes and the reduction of nodes by pn-search with respect to α-β search make pn-search the better choice for post-mortem analysis on Awari (cf. Section 7).

Solving Connect-Four, Qubic, and Go-Moku

The ultimate post-morten analysis of a game considers the initial position. For three games we have performed this feat: Connect-Four [2, 37], Qubic [9], and Go-Moku [4]. Pn-search and one of its predecessor cn-search variants were

the main contributors to the completion of this research. In the first two cases, confirmation of the results obtained exist: Allen [1] showed independently of Allis et al. at almost the same time that White can win at Connect-Four. Patashnik [30] proved more than a decade before Allis and Schoo [9] the win for White in Qubic. Whereas Patashnik as a strong player selected all White's strategic moves himself, in our experiments pn-search did the move selection without any built-in strategic knowledge of the game. The solution of Go-Moku is very recent; a proper description is in preparation.

For Connect-Four and Qubic we have developed a rather slow evaluation function (10 to 100 nodes per second) which either returns the game-theoretical value of a position or indicates that it cannot determine the correct value. The overhead of pn-search per node expansion is negligible compared to the node-evaluation time. Hence, the lesser number of nodes searched by pn-search in comparison with α-β search is a major asset.

To anticipate slightly, we here conclude that pn-search already has proven its worth in several applications of game-tree search. Sections 6 and 7 will support this conclusion.

6. Implementation details and performance measures

Since our experiments have the purpose to determine whether pn-search is to be preferred to sophisticated implementations of α-β search on a practical domain, we must address several issues. Therefore we first describe our test domain, the game of Awari. Then we present the selected test positions. Thereafter we mark out the two α-β search implementations of which the results are compared with those of pn-search. Finally, we explain how we compare the performances of the three algorithms.

6.1. Awari

Awari is a two-person (South and North) zero-sum game with perfect information. It is one instance of a large family of games named Mancala, of which some 1200 variants are known. Another well-known game in this family is Kalah [35]. The Mancala games originate from Africa, and Awari is mainly played in its western countries, such as Nigeria. For the game described here, the names Wari or Awele are also used [14].

Awari is played on a wooden board containing two rows of six pits. Each player controls the row on his side of the board. South's pits (from left to right, as seen by South) are named A through F, while North's pits (from left to right, as seen by North) are named a through f. At the right-hand side of each row, an auxiliary pit is used to contain a player's captured stones. At the start of the game each pit contains four stones, for a total of 48 stones on the board.

At each move, a player selects a non-empty pit X from his row. Starting with X's neighbour's pit, he then sows all stones from X, one at the time, counter-clockwise over the board (of course, omitting the two auxiliary pits). If X contains sufficient stones to go around the board (12 stones or more), pit X is skipped and sowing continues. Thus, after the move, X will always be empty. Finally, captured stones, if any, are removed. Stones are captured if the last stone sown lands in an enemy pit which after landing contains 2 or 3 stones. If such a capture is made and if the preceding pit is an enemy pit that contains 2 or 3 stones (of course, after sowing), then those stones are also captured. This procedure is repeated for preceding pits and ends when either a pit is encountered which contains a number of stones other than 2 or 3, or when the end of the opposing row is reached.

A move is described by the name of the pit, followed by the number of stones sown (the name of the pit by itself defines the move, but such a notation is prone to error). The number of stones captured, if any, is indicated by the amount preceded by a "×". In Fig. 11, an example position is shown with South to move. Legal moves for South are: *A1*, *C4* × *2*, *D19* × *7*, *E4*, and *F2* × *4*.

The goal of Awari is to capture more stones than your opponent. Thus the game ends as soon as one of the players has collected 25 or more stones. However, two other conditions exist which terminate the game. First, if a player is unable to move (i.e., all his pits are empty), the remaining stones are captured by his opponent. Second, if the same position is encountered for the third time, with the same player to move, the remaining stones on the board are evenly divided among the players. In all cases, after the end of the game, the winner is the player who has captured most stones. If both players have captured 24 stones, the game is drawn.

A last atypical rule exists to avoid that players run out of moves early in the game. Whenever possible, a player is forced to choose such a move that his opponent is able to make a reply move. It is, however, not compulsory to look one or several moves ahead to ensure that the opponent will continue to be able to reply. As an example, Fig. 12 shows a position in which South by playing *1. B1* can deliberately create a position in which he is unable to offer North any stones on his next move, since after North's move *1. . . . f1* none of

North

	f	*e*	*d*	*c*	*b*	*a*	
7	0	0	1	3	1	1	5
	1	0	4	19	4	2	
	A	*B*	*C*	*D*	*E*	*F*	

South (to move)

Fig. 11. A position with legal moves *A1*, *C4* × *2*, *D19* × *7*, *E4*, and *F2* × *4*.

North

	f	*e*	*d*	*c*	*b*	*a*	
23	1	0	0	0	0	0	22
	0	1	0	0	1	0	
	A	*B*	*C*	*D*	*E*	*F*	

South (to move)

Fig. 12. *1. B1 f1* wins. After *1. E1? f1*, South must play *2. F1*.

South's possible moves (*2. A1*, *2. C1*, or *2. E1*) results in a position in which North can move. By playing *1. B1* South will capture all three remaining stones after his second move, and hence wins the game. However, should he play *1. E1*, then after *1. ... f1* he is forced to play *2. F1*, leaving the game for the moment undecided.

6.2. Selected Awari positions

In 1990 we constructed an Awari-playing tournament program, named Lithidion, which uses α-β search and endgame databases for all positions of 13 stones and less. The positions in our experiments are taken from the official games played by Lithidion and MARCO at the Second Computer Olympiad (London, August 1990) reported in [24]. MARCO, programmed by Remi Nierat, was the winner of the Awari tournament of the First Computer Olympiad (London, August 1989) as recorded in [23]. Lithidion won the 1990 competition by five to nil due to its superior play in the late middle game and endgame (using its databases); MARCO played a slightly better opening and early middle game.

For many games, notably Chess, it is a habit to perform post-mortem analyses determining the game-theoretical value of critical positions. For Awari, we are interested in determining the game-theoretical value of game positions not in the databases. Immediately after the match, we thus analyzed the fifth game, in which it was shown that MARCO had missed a winning variation in a seemingly hopeless position [7].

For this article, we have investigated all late-middle-game positions of the five tournament games, as far as resources allowed us to perform this task. We used three implemented search algorithms. In all five games we started from a position with a known (database) outcome, then we went backwards until a position was reached which none of the three algorithms could solve using the allotted resources.

6.3. Implementations of α-β search

Since tournament programs for games such as Chess, Othello, and Awari are (almost) all based on variants of α-β search, this search procedure is the natural sparring partner for the new pn-search algorithm.

The first variant of the α-β iterative-deepening search has the following characteristics. At each node, moves are pre-ordered on capture size. The largest captures are evaluated first, since the resultant positions are assumed to hit a database more quickly, leading to a fast determination of the position value for the player to move. Another reason for doing captures first is that they are often good moves anyway. An iterative-deepening search is performed with a depth increase of 1 per iteration. The result of each iteration is both a window of the remaining possible values, and a move ordering for the full principal variant. The search terminates as soon as the value of a position has been determined.

The second variant of the α-β iterative-deepening search has the same characteristics as the previous one, but is extended with a transposition table of a quarter million entries. A transposition table is a hash table containing positions which have been evaluated earlier during the search. Whenever a position is examined it is checked whether the position is stored in the transposition table, and if so, whether the previous evaluations can be used for the current examination. In case of a collision in the transposition table, preference has been given to positions which had been searched more deeply. At each transposition-table entry the full Gödel code of the position represented has been stored to ensure complete reliability of the values extracted from it.

Especially in the middle game, when empty pits and pits containing single stones are common, transposition tables are useful in Awari, as will transpire from the results of our experiments presented in Section 7.

6.4. Comparing the performances

When selecting a search algorithm for an application, the elapsed CPU time is an important criterion. However, many distinct factors, such as the static node-evaluation time and several implementation details, make it difficult to use this criterion for comparison with other results and for generalization to other domains.

We therefore compare the number of nodes visited as do most authors. In this case, however, a careful analysis is needed to determine the fairest way to compare the node counts.

First, let us consider the number of nodes visited by pn-search. For a two-valued search, we merely count the number of nodes created during the search. For a binary multi-valued search, we sum the number of generated nodes of each two-valued search, and thus obtain a total number of nodes visited. We remark that in many cases two or more of the two-valued searches visit a large number of identical nodes. Thus, the number of nodes visited could be reduced by reusing the previous search tree. The disadvantage of such a reuse would be that the tree-reduction techniques described in Section 4 can

no longer be applied. As a result, more nodes should be kept in memory at the same time and the algorithm also becomes somewhat more complex. We have therefore included the extra nodes visited in our node counts.

Second, let us consider the number of nodes visited by the α-β iterative-deepening search. On the one hand, we could sum the number of nodes visited in each iteration. However, this would be unfair to α-β iterative-deepening search, since a smaller number of iterations (e.g., by searching to even-ply depths only) may result in almost the same ordering and thus reduce the number of nodes visited. On the other hand, we could just take the number of nodes visited in the last iteration. But that would be unfair towards pn-search, as the last iteration often starts with a far reduced window of possible root values, thus using more than only the move ordering of the previous iterations. Moreover, the α-β iterative-deepening search with transposition tables obtains many early cut-offs during the last iteration, due to the results of previous iterations stored in the transposition table.

As a measure of comparison, we have chosen to count at iteration i only the nodes at depth i. Then the extra iterations are only an asset to the α-β iterative-deepening search, without costing it anything in terms of the number of visited nodes. We remark that, if, by re-ordering the moves, in a new iteration terminal nodes appear, which are not at the deepest level, they are not counted at all. This slight bias in favour of the α-β iterative-deepening search does not influence the results.

To summarize, for pn-search all nodes grown during the multi-valued search are counted, while for the α-β iterative-deepening searches nodes at depth i are counted only during iteration i.

7. Results on Awari

In this section we compare three tree-search algorithms (pn-search, α-β iterative-deepening search with and without transposition tables) on Awari positions taken from five tournament games. First we present the exact figures of the algorithms' performances in five tables. Then we draw conclusions from the test results, and present a graphical representation indicating the average gain of pn-search with respect to α-β iterative-deepening search. Finally, we provide some results on our tournament program Lithidion.

7.1. Test figures of five tournament games

The results of the three search procedures on the submitted positions of the five tournament games are presented in the Tables 1–5, of which Tables 2–5 are exhibited in Appendix A. For pn-search, each two-valued search was terminated as soon as the part of the tree held in memory exceeded 2,400,000

Table 1
Test results of game 5.

Position (to move)	Number of nodes				
	α-β	α-β with trans.	pn-search (sum)	max pos	max tree
55 (N)	1	1	5	1	1
55 (S)	212	188	419	231	78
54 (N)	245	225	843	239	79
54 (S)	49,671	5,670	1,589	749	198
53 (N)	318,119	12,820	4,271	1,278	523
53 (S)	1,709,963	366,618	5,727	2,503	889
52 (N)	3,475,319	458,497	11,492	2,900	829
52 (S)	15,502,284	2,874,623	87,967	47,162	26,180
51 (N)	51,176,384	6,853,242	90,150	38,215	23,541
51 (S)	37,965,868	2,433,370	224,314	156,348	43,205
50 (N)	144,071,919	3,134,803	768,686	344,395	87,903
50 (S)	161,134,944	5,378,362	613,750	397,154	≈83,966
49 (N)	24,879,754	2,977,981	668,746	382,383	≈105,604
49 (S)	27,440,022	3,508,885	799,694	369,907	≈119,737
48 (N)	75,709,058	4,899,017	898,058	554,404	≈145,802
48 (S)	75,837,856	4,961,444	1,340,785	554,416	≈145,791
47 (N)	75,837,857	4,961,445	902,685	554,417	≈145,792
47 (S)	76,126,790	5,000,194	1,349,656	554,679	≈145,605
46 (N)	76,126,791	5,000,195	910,627	554,680	≈145,606
46 (S)	96,567,363	6,390,241	2,109,880	903,273	≈208,961
45 (N)	38,673,386	9,605,678	4,199,020	1,442,758	≈617,544
45 (S)	39,486,568	10,369,223	4,722,589	1,442,835	≈617,473
44 (N)	50,133,433	14,903,279	4,341,932	1,451,921	≈634,121
44 (S)	61,816,867	11,284,030	4,789,675	1,452,571	≈633,620
43 (N)	104,434,567	23,327,160	4,377,050	1,454,076	≈633,976
43 (S)	118,018,891	28,960,404	5,420,622	1,583,829	≈633,914
42 (N)	361,547,784	62,164,036	9,857,925	3,758,110	≈1,251,625
42 (S)	>500,000,000	>100,000,000	14,703,057	5,539,659	≈2,368,832
41 (N)	>500,000,000	>100,000,000	17,099,650	5,562,074	≈2,368,622

nodes. Using the first tree-reduction technique, trees of up to ten million nodes could be searched for a total of over ten million nodes per multi-valued search. For the α-β iterative-deepening search without transposition tables the search was terminated when more than 500,000,000 nodes were searched, while the α-β iterative-deepening search with transposition tables was terminated after 100,000,000 nodes.

Each table consists of two columns with the heading "Position" and "Number of nodes" respectively; the latter is subdivided into five columns containing experimental results on the three search procedures. Each position entry represents an Awari position, being the position of the game under consideration (i.e., game 1 to 5); the exact position is described by its move number, with the side to move in parentheses. The analysis as shown in the tables proceeds retrogradely. The first position analyzed is the position in which the

transition to database knowledge (perfect knowledge) takes place, i.e., in Table 1: the position 55(N). The analysis is performed up to the point where all three search techniques fail to solve a position. For reproduction purposes of our experiments the game scores of all five games are available on request [5]. The five columns with experimental results on the number of nodes visited are to be read as follows. The column headed "α-β" contains the number of nodes searched by the α-β iterative-deepening search without transposition tables. The column labelled "α-β with trans." contains the number of nodes searched by the α-β iterative-deepening search with transposition tables. The next column, labelled "pn-search (sum)", contains the number of nodes searched by a multi-valued pn-search, being the sum of all individual two-valued pn-searches. The column "max pos" contains the number of visited nodes taken from the largest tree of all two-valued pn-searches. The last column, labelled "max tree", contains the maximum size of the two-valued pn-search tree at any time during the search. Any number preceded by a $\approx$ is an estimate, which can be up to 5% off.

The fifth game (cf. Table 1) has been analyzed most deeply, which can be explained by the course of the game: after 33 moves only 19 stones remained on the board. From the 33rd move 42 half-moves were played without any capture (cf. Appendix B), during which South created a strong positional advantage. A majority of these 19-stone positions were successfully analyzed by all three algorithms. The remarkable increase in node counts in the "α-β" column going from move 49 of North to move 50 of South can be explained as follows. At move 49, North could choose between a move which would end the game relatively quickly and a move which would lengthen the game, both with equal outcome. North chose the latter option and thus amplified the α-β search tree. The reduction-of-nodes factor of pn-search with respect to α-β iterative-deepening search without transposition tables ranges from 8 to more than 560, for positions where more than 1,000 nodes are involved. The same factor with respect to α-β iterative-deepening search with transposition tables ranges from more than 2 to more than 75.

7.2. Interpretation of the test results

In all five tables it is observed that in searches terminating within a few hundred nodes pn-search does not perform as well as α-β search. The explanation is as follows. If most moves lead directly to database positions, only a small number of nodes must be searched. All three algorithms then face the same small search tree. However, pn-search must search this tree several times. (For instance, for the 19-stone position of Table 1, 39 possible values exist, resulting in six steps in the binary search.) A detailed inspection of the separate two-valued pn-searches showed that in four cases only, a two-valued pn-search resulted in a larger tree than the trees searched by α-β search. In each of these four cases, the difference was less than 50 nodes.

In all positions but one where pn-search visits more than a thousand nodes, pn-search outperforms both variants of α-β iterative-deepening search. Moreover, the Tables 1–5 clearly show that α-β iterative-deepening search without transposition tables is outperformed by one or two orders of magnitude in positions where pn-search only searches approximately one million positions. When extrapolating, we remark that this factor tends to increase with the size of the search tree. Without any overstatement we safely claim that α-β iterative-deepening search without transposition tables is no match for pn-search. And although transposition tables greatly improve the performance of α-β iterative-deepening search on Awari positions, as can be seen from the Tables 1–5, pn-search still claims to do much better, having tree sizes often differing an order of magnitude.

In Fig. 13, we have generalized the figures of the Tables 1–5. For the purpose of comparison we have taken the α-β iterative-deepening search without transposition tables as the standard search algorithm. This choice enables us to show the impact of the use of transposition tables as well as the effectiveness of pn-search. We have transferred all tree sizes to manageable classes. Therefore, we have chosen class n to contain all tree sizes ranging from 2^{3n} to $2^{3(n+1)}$. Thus, the class boundaries are 2^0, 2^3, 2^6, etc. Next we have distributed all positions (from the five tables) over the various classes. Each position is placed in a class according to the tree size of α-β iterative-deepening search without transposition tables (cf. column "α-β"). Hence, within each class we find positions for which the sizes of the trees searched by α-β iterative-deepening search without transposition tables are of equal magnitude.

For each of the entries in a class, we determine the size of the search tree searched by pn-search (cf. column "pn-search (sum)"). These tree sizes are averaged. For each class of positions we thus obtain the average of the sizes of

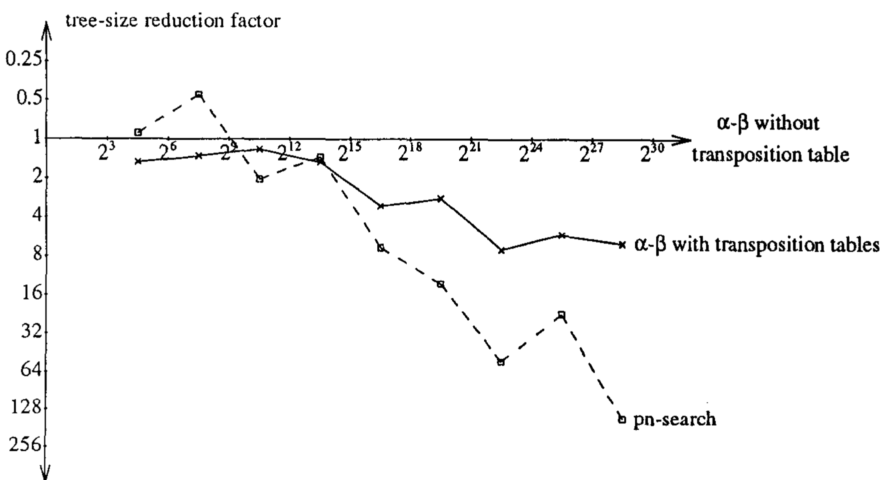

Fig. 13. Tree-size comparison of α-β (with and without transposition tables) and pn-search.

trees searched by pn-search. The same procedure has been applied to the corresponding set of α-β iterative-deepening searches. The $\log_2$ of the results of these calculations have been presented in Fig. 13. The figure shows that α-β iterative-deepening search with transposition tables clearly outperforms the standard α-β iterative-deepening search without transposition tables. Moreover, it indicates that pn-search outperforms both. Furthermore, it may be remarked that the pn-search performance seems to intensify when the tree size increases.

An example may elucidate the results obtained. The small square at coordinates approximately (2^{22}, 64) indicates that Awari positions solved by α-β search in 2^{22} ($\approx$4,200,000 positions) were solved by pn-search with a reduction factor in tree size of 64, i.e., approximately 65,000 positions were needed.

We conclude our interpretation of test results with the statement that for at least one domain—Awari—pn-search is distinctly superior to sophisticated implementations of α-β search.

7.3. Pn-search in Lithidion

At the Third Computer Olympiad played a few months after the experiments of the previous sections, Lithidion played against a strong newcomer, MyProgram, written by Erik van Riet Paap [38]. MyProgram used a 16-stone endgame database, transposition tables, singular extensions and a fast α-β search algorithm (35,000 nodes/sec on a 486-33 MHz).

For the tournament, Lithidion had been equipped with a 17-stone endgame database. The transposition table was not used, as it slowed down the search too much, and still only 20,000 nodes/sec were calculated on a SPARC-station 2. The main new feature of Lithidion, however, was its application of pn-search. After every move generated by the α-β search, a two-valued pn-search was performed up to a maximum tree size of 150,000 nodes to investigate whether the move was a game-theoretical loss. If so, another move was generated by the α-β search, and the procedure was repeated until a non-losing move was found, or all moves were checked.

Moreover, in the opponent's thinking time, all his moves were checked to see whether a winning line for Lithidion could be found, again up to a tree size of 150,000 nodes.

The tournament was won with the smallest possible difference by Lithidion (3 won, 1 drawn, 2 lost). In two games, Lithidion found a deep winning line using pn-search several moves before the opponent's α-β search found the same result. In game 5, a winning line of 28-ply was found using a tree of only 263,000 nodes, while the maximum tree size at any time during the search remained within the 150,000 nodes limit (cf. Section 4.1, *DeleteSolvedSubtrees*).

Using Lithidion's new tournament version, the fifth game of the match

against MARCO (cf. Section 7.1) was replayed. It turned out that at move 15, MARCO played a losing move, which Lithidion found at tournament speed in 283,000 nodes (40-ply), again staying within the 150,000 nodes limit.

Comparing Lithidion's average search depth at tournament speed (15–20 ply) with the 15–20 ply plus extensions search depth of MyProgram (approx. 2,000,000 nodes per move), we are convinced that the early wins found by Lithidion in two of the three won games were a major contribution to its tournament victory. It shows that pn-search is a major asset to Awari-playing programs, not only for post-mortem analysis, but also for tournament play.

8. Generalizing the test results

In our preliminary analysis (cf. Sections 1 and 2) we have stressed that pn-search may perform well if the search tree is sufficiently non-uniform. Clearly, Awari game trees meet this requirement. Moreover, we have seen that pn-search outperforms α-β iterative-deepening search convincingly, even when the α-β iterative-deepening search is enhanced with transposition tables (cf. Section 7). From these results we believe that we may generalize to other games with sufficiently non-uniform search trees, such as Othello and Checkers. Below we support this generalization briefly.

Tournament programs for Othello search at the end of the game rather deeply to determine the game-theoretical value of the position. The search trees built have small and varying branching factors (from 1 to around 10). Pn-search should therefore be able to determine the game-theoretical value of a position earlier in the game than α-β iterative-deepening search.

In Checkers (8×8), the average branching factor is rather small (1.2 for capture moves, about 8 for non-captures), although it varies less than in Awari or Othello. Still, forced moves create extra non-uniformity in the tree. We believe that Checkers (and Draughts (10×10)) therefore is a domain on which pn-search may be implemented with success.

In Chess, the average branching factor (36) is larger and varies less than in all games discussed so far. Even though pn-search's predecessors have mainly been applied to Chess, we, maybe surprisingly, speculate that in this domain pn-search is not likely to be a good alternative to α-β search.

In summary, disregarding Connect-Four, Qubic, and Go-Moku, we conjecture that pn-search may be implemented successfully in tournament programs for Awari, Othello, Checkers, and Draughts.

Although emphasis has been placed on solving positions rather than on taking the right decision *per se*, we would like to argue that the heuristic decision making has been moved from the obvious comparison of two (or more) move evaluations towards an implicit search-strategy decision making which leads to a desired position. This means a trade-off between knowledge

and search in the direction of search. Hence, the more powerful the machine, the better the search results will be. As a direct consequence of the algorithmic ideas on pn-search, solutions to other interesting AI problems, such as Elkan's [15] application to theorem proving will be within our horizon.

9. Conclusions

It has become clear that pn-search adequately treats multi-valued non-uniform trees. Moreover, the current article has shown that pn-search is able to find the theoretical value in difficult positions, where other search techniques fail. Pn-search aims at *proving* the true value of the root of a minimax tree whereas, for instance, cn-search has as its goal to establish it unlikely that the root's value will *change* (cf. Section 3).

The main idea of pn-search is that in any max node it tries to prove the current upper bound of the root with the least possible effort, and at any min node to disprove the current upper bound as quickly as possible, regardless of the new upper bound which may result of such a disproof. The power of pn-search's dealing with multi-valued non-uniform trees resides in its treatment: the search takes place as a series of proofs in two-valued trees.

In pn-search, we have approached the well-known disadvantage of all variants of cn-search, viz. the requirement to keep the entire search tree in memory, by two reduction techniques, *DeleteSolvedSubtrees* and *DeleteLeastProving* (cf. Section 4.1). These were shown to be successful (cf. Section 7.3).

In three applications, viz. tournament play, post-mortem analysis, and game solving, pn-search has proven to be a competent best-first algorithm. From our experiments it is clear that in non-uniform trees pn-search outperforms α-β iterative-deepening search without and with transposition tables (cf. Section 7.2). It also outperforms sophisticated implementations of α-β search variants in the Awari endgame under tournament conditions (cf. Section 7.3). The game-solving ability has been proved by cracking the games Connect-Four, Qubic and Go-Moku (cf. Section 5).

We expect that pn-search soon will become a major contributing factor to game-playing programs for many different games and that it will show its merits in the field of theorem proving.

Appendix A

In Tables 2–5 we have tabulated the results of the post-mortem analysis of the first four games played between MARCO and Lithidion (cf. Section 7.2).

Table 2
Test results of game 1.

Position (to move)	Number of nodes				
	α-β	α-β with trans.	pn-search (sum)	max pos	max tree
21 (N)	1	1	5	1	1
21 (S)	3,667,963	73,386	96,988	38,163	19,375
20 (N)	2,187,888	127,172	144,858	40,856	21,422
20 (S)	6,368,724	540,573	168,858	65,561	29,313
19 (N)	28,001,675	1,766,418	182,516	66,170	29,893

Table 3
Test results of game 2.

Position (to move)	Number of nodes				
	α-β	α-β with trans.	pn-search (sum)	max pos	max tree
23 (N)	1	1	4	1	1
23 (S)	85	53	165	63	20
22 (N)	94	67	150	72	24
22 (S)	5,928	4,625	2,339	1,183	549
21 (N)	266,958,048	69,253,174	>7,383,208	>6,729,910	>2,400,000
21 (S)	>500,000,000	>100,000,000	5,178,283	2,032,629	≈979,683
20 (N)	323,846,021	32,023,476	3,197,485	2,138,422	≈1,207,005

Table 4
Test results of game 3.

Position (to move)	Number of nodes				
	α-β	α-β with trans.	pn-search (sum)	max pos	max tree
34 (N)	1	1	5	1	1
34 (S)	1,882	932	897	394	145
33 (N)	2,496	1,475	528	429	146
33 (S)	12,137	7,345	11,271	5,278	1,625
32 (N)	689,282	304,407	67,103	31,282	19,560
32 (S)	26,654	17,137	17,607	8,569	2,958
31 (N)	57,482	27,295	13,759	9,173	2,959
31 (S)	57,784	27,542	16,667	9,194	2,960
30 (N)	3,172,034	1,040,212	79,772	26,713	11,685
30 (S)	3,803,572	963,044	103,727	48,820	18,176
29 (N)	178,529,355	26,281,753	860,716	438,528	≈194,660
29 (S)	182,125,887	25,794,398	713,597	438,785	≈194,444
28 (N)	>500,000,000	>100,000,000	6,128,261	3,965,303	≈1,467,315
28 (S)	>500,000,000	>100,000,000	6,532,084	3,982,276	≈1,460,310
27 (N)	>500,000,000	97,395,265	153,139	72,213	22,082
27 (S)	>500,000,000	97,395,266	84,438	72,214	22,083
26 (N)	>500,000,000	>100,000,000	3,665,820	3,037,312	≈1,438,987

Table 5
Test results of game 4.

Position (to move)	Number of nodes				
	α-β	α-β with trans.	pn-search (sum)	max pos	max tree
25 (S)	1	1	4	1	1
24 (N)	15	15	25	10	8
24 (S)	4,169	3,783	1,034	336	299
23 (N)	10,556	5,777	1,492	411	268
23 (S)	12,544	7,169	16,503	14,951	13,663
22 (N)	112,031	52,938	21,751	16,922	13,664
22 (S)	273,576	144,032	2,342	721	380
21 (N)	778,702	286,704	153,929	105,342	≈40,646
21 (S)	1,182,772	437,521	108,942	105,474	≈40,640
20 (N)	4,097,062	720,591	194,358	121,433	≈62,995
20 (S)	4,112,651	727,356	128,264	121,976	≈62,861
19 (N)	32,213,813	8,785,109	636,140	394,360	≈127,175
19 (S)	>500,000,000	>100,000,000	1,984,524	535,948	≈260,170

Appendix B

Below we have reproduced the game score of the fifth game of the match between MARCO and Lithidion.

Round 5: South: MARCO. North: Lithidion.

1. D4 b5 2. B4 e5 × 2 3. C6 b1 4. D1 f6 5. C1 c7 × 3 6. C1 d6 × 27. B4 a6 8. A10 a1 9. D2 f3 10. C4 a1 11. A1 e4 12. A1 f1 13. B5 d2 14. A1 e1 15. C2 f2 × 2 16. D3 × 2 c2 17. A1 e1 18. B1 b4 19. F14 × 7 f3 × 7 20. D1 e2 21. A1 f1 22. A1 d3 23. B2 f1 × 2 24. D1 e1 25. C1 f1 26. E19 a2 27. B1 f2 28. D2 e2 29. E1 d2 30. B1 b3 31. A5 d1 32. B1 f2 33. E1 c4 × 2 34. D1 f1 35. A1 d1 36. E1 e5 37. A1 f1 38. C6 c1 39. D2 a1 40. B4 b2 41. A1 d2 42. C1 e1 43. B1 c1 44. D2 d1 45. C1 f2 46. B1 e1 47. A1 f1 48. E4 c1 49. D1 b1 50. C1 d1 51. A1 e1 52. E1 a1 53. D1 b1 54. E1 c2 55. F14 × 6 e2 × 2 56. C1 d2 57. D2 e1 58. B3 f5 59. F1 a1 60. B1 b1 61. E4 × 2 a1 62. A1 b2 63. D2 d1 64. E1 c1 65. C3 e1 66. B1 d1 67. D1 f1 68. A1 e1 69. C1 f1 70. E2 a1 71. B1 b1 72. D1 c1 73. A1 d1 74. C1 e1 75. D1 f1 76. E2 a1 77. B1 b1 78. A1 c1 79. B1 d1 80. C2 e1 81. D1 f1 82. E2 a1 83. A1 b1 84. B1 c1 85. C1 d1 86. D1 e1 87. F7 b1 88. A1 f2 × 2 89. A1 e1 90. B1 d1 91. E1 a1 92. C1 c2 93. D1 b1 94. F1 e2 95. E1 a1 96. F1 d1 97. A1 f2 × 2.

North has won by 26 stones to 17.

Acknowledgements

We would like to thank the anonymous referees for their comments and suggestions. The investigations were (partly) supported by the Foundation for Computer Science Research in the Netherlands (SION) with financial support from the Netherlands Organization for Scientific Research (NWO). Moreover, we would like to express our gratitude to the Faculty of Mathematics and Computer Science of the Vrije Universiteit in Amsterdam for giving us access to their computer resources for computing the tables listed in Section 7 and Appendix A. Special words of thanks go to Matty Huntjens, who volunteered his time and competence towards running our software at the Vrije Universiteit.

References

[1] J.D. Allen, A note on the computer solution of Connect-Four, in: D.N.L. Levy and D.F. Beal, eds., *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad* (Ellis Horwood, Chichester, England, 1989) 134–135.

[2] L.V. Allis, A knowledge-based approach of Connect-Four, Report No. IR-163, M.Sc. Thesis, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam (1988).

[3] L.V. Allis, H.J. van den Herik and I.S. Herschberg, Which games will survive? in: D.N.L. Levy and D.F. Beal, eds., *Heuristic Programming in Artificial Intelligence: The Second Computer Olympiad* (Ellis Horwood, Chichester, England, 1991) 232–243.

[4] L.V. Allis, H.J. van den Herik and M.P.H. Huntjens, Go-Moku solved by new search techniques (in preparation).

[5] L.V. Allis, M. van der Meulen and H.J. van den Herik, Lithidion: an Awari-playing program, Report CS90-05, University of Limburg, Maastricht, Netherlands (1990).

[6] L.V. Allis, M. van der Meulen and H.J. van den Herik, $\alpha\beta$ conspiracy-number search, in: D.F. Beal, ed., *Advances in Computer Chess* **6** (Ellis Horwood, Chichester, England, 1991) 73–95.

[7] L.V. Allis, M. van der Meulen and H.J. van den Herik, Omniscience in Lithidion, in: D.N.L. Levy and D.F. Beal, eds., *Heuristic Programming in Artificial Intelligence: The Second Computer Olympiad* (Ellis Horwood, Chichester, England, 1991) 27–32.

[8] L.V. Allis, M. van der Meulen and H.J. van den Herik, Databases in Awari, in: D.N.L. Levy and D.F. Beal, eds., *Heuristic Programming in Artificial Intelligence: The Second Computer Olympiad* (Ellis Horwood, Chichester, England, 1991) 73–86.

[9] L.V. Allis and P.N.A. Schoo, Qubic solved again, in: H.J. van den Herik and L.V. Allis, eds., *Heuristic Programming in Artificial Intelligence: The Third Computer Olympiad* (Ellis Horwood, Chichester, England, 1992).

[10] H. Berliner, The B* tree search algorithm: a best-first proof procedure, *Artif. Intell.* **12** (1979) 23–40.

[11] S. Bhattacharyya and A. Bagchi, Making the best use of available memory when searching game trees, in: *Proceedings AAAI-86*, Philadelphia, PA (1986) 163–167.

[12] M.S. Campbell and T.A. Marsland, A comparison of minimax tree search algorithms, *Artif. Intell.* **20** (4) (1983) 347–367.

[13] P.P. Chakrabarti, S. Ghose, A. Acharya and S.C. de Sarkar, Heuristic search in restricted memory, *Artif. Intell.* **41** (1989) 197–221.

[14] A. Deledicq and A. Popova, *Wari et Solo: Le Jeu de Calcul African* (CEDIC, Paris, 1977).

[15] C. Elkan, Conspiracy numbers and caching for searching and/or trees and theorem-proving, in: *Proceedings IJCAI-89*, Detroit, MI (1989) 341–346.
[16] O. Etzioni, Embedding decision-analytic control in a learning architecture, *Artif. Intell.* **49** (1991) 129–159.
[17] D. Hartmann and P. Kouwenhoven, The 9th Dutch Computer-Chess Championship, *ICCA J.* **12** (4) (1989) 249–251.
[18] T. Ibaraki, Depth_m search in branch and bound algorithms, *Int. J. Comput. Inf. Sci.* **7** (1978) 315–343.
[19] N. Klingbeil, Search strategies for conspiracy numbers, M.Sc. Thesis, University of Alberta, Edmonton, Alta. (1989).
[20] N. Klingbeil and J. Schaeffer, Search strategies for conspiracy numbers, in: *Proceedings Seventh Biennial Conference of the Canadian Society for Computational Studies of Intelligence*, Edmonton, Alta. (1988) 133–139.
[21] D.E. Knuth and R.W. Moore, An analysis of alpha-beta pruning, *Artif. Intell.* **6** (4) (1975) 293–326.
[22] R.E. Korf, Depth-first iterative-deepening: an optimal admissible tree search, *Artif. Intell.* **27** (1985) 97–109.
[23] D.N.L. Levy and D.F. Beal, eds., *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad* (Ellis Horwood, Chichester, England, 1989).
[24] D.N.L. Levy and D.F. Beal, eds., *Heuristic Programming in Artificial Intelligence: The Second Computer Olympiad* (Ellis Horwood Limited, Chichester, England, 1991).
[25] L. Lister, M.Sc. Thesis (unpublished) on conspiracy-number search, University of Alberta, Edmonton, Alta. (1989).
[26] T.A. Marsland, A. Reinefeld and J. Schaeffer, Low overhead alternatives to SSS*, *Artif. Intell.* **27** (1985) 185–199.
[27] D.A. McAllester, A new procedure for growing mini-max trees, Tech. Report, Artificial Intelligence Laboratory, MIT, Cambridge, MA (1985).
[28] D.A. McAllester, Conspiracy numbers for min-max search, *Artif. Intell.* **35** (1988) 287–310.
[29] A.J. Palay, The B* tree search algorithm—new results, *Artif. Intell.* **19** (1982) 145–163.
[30] O. Patashnik, Qubic: $4 \times 4 \times 4$ Tic-Tac-Toe, *Math. Mag.* **53** (1980) 202–216.
[31] S.J. Russell and E.H. Wefald, Principles of metareasoning, *Artif. Intell.* **49** (1991) 361–395.
[32] J. Schaeffer, Conspiracy numbers, in: D.F. Beal, ed., *Advances in Computer Chess* **5** (North-Holland, Amsterdam, 1989) 199–217.
[33] J. Schaeffer, Conspiracy numbers, *Artif. Intell.* **43** (1) (1990) 67–84.
[34] H.A. Simon and J.B. Kadane, Optimal problem-solving search: all-or-none solutions, *Artif. Intell.* **6** (1975) 235–247.
[35] J.R. Slagle and J. Dixon, Experiments with some programs that search game trees, *J. ACM* **16** (2) (1969) 189–207.
[36] G. Stockman, A minimax algorithm better than alpha–beta?, *Artif. Intell.* **12** (1979) 179–196.
[37] J.W.H.M. Uiterwijk, H.J. van den Herik and L.V. Allis, A knowledge-based approach to Connect Four: the game is over white to move wins, in: D.N.L. Levy and D.F. Beal, eds., *Heuristic Programming in Artificial Intelligence: The First Computer Olympiad* (Ellis Horwood, Chichester, England, 1989) 113–133.
[38] H.J. van den Herik and L.V. Allis, eds., *Heuristic Programming in Artificial Intelligence: The Third Computer Olympiad* (Ellis Horwood, Chichester, England, 1992).
[39] M. van der Meulen, Conspiracy-number search, *ICCA J.* **13** (1) (1990) 3–14.
[40] M. van der Meulen and E. van Riet Paap, Pn-search decides Awari match, in: H.J. van den Herik and L.V. Allis, eds., *Heuristic Programming in Artificial Intelligence: The Third Computer Olympiad* (Ellis Horwood, Chichester, England, 1992) 19–24.
[41] J. von Neumann, Zur Theorie der Gesellschaftsspiele, *Math. Ann.* **100** (1928) 295–320; reprinted in: A.H. Taub, ed., *John von Neumann Collected Works* **VI** (Pergamon Press, Oxford, 1963) 1–26.
[42] J. von Neumann and O. Morgenstern, *Theory of Games and Economic Behavior* (Princeton University Press, Princeton, NJ, 1st ed., 1944; 2nd ed., 1947).