An Effective Two-Level Proof-Number Search Algorithm

Mark H.M. Winands * Jos W.H.M. Uiterwijk
H. Jaap van den Herik

Department of Computer Science, Institute for Knowledge and Agent Technology, Universiteit Maastricht, P.O. Box 616, 6200 MD Maastricht, The Netherlands

Abstract

The paper presents a new proof-number (PN) search algorithm, called PDS-PN. It is a two-level search (like PN^2), which performs at the first level a depth-first Proof-number and Disproof-number Search (PDS), and at the second level a best-first PN search. Hence, PDS-PN selectively exploits the power of both PN^2 and PDS. Experiments in the domain of Lines of Action are performed. They show that within an acceptable time frame PDS-PN is more effective for really hard endgame positions than $\alpha\beta$ and any other PN-search algorithm.

Key words: proof-number search, PDS, two-level search, Lines of Action

1 Introduction

Most modern game-playing computer programs successfully use $\alpha\beta$ search [12] with enhancements for online game-playing [11]. However, the enhanced $\alpha\beta$ search is sometimes not sufficient to play well in the endgame. In some games, such as chess, this problem is solved by the use of endgame databases [18]. Due to memory constraints this is only feasible for endgames with a relatively small state-space complexity, although nowadays the size may be considerable. An alternative approach is the use of a specialised binary (win or non-win)

* Corresponding author Tel.: (+31) (0)43-38 83898; fax: (+31) (0)43-38 84897
Email addresses: `m.winands@cs.unimaas.nl` (Mark H.M. Winands), `uiterwijk@cs.unimaas.nl` (Jos W.H.M. Uiterwijk), `herik@cs.unimaas.nl` (H. Jaap van den Herik).

search method, such as proof-number (PN) search [3]. The latter method is inspired by the conspiracy-number algorithm [14, 15, 22]. In many domains PN search outperforms $\alpha\beta$ search in proving the game-theoretic value of endgame positions. The PN-search idea is a heuristic, which prefers expanding shallow subtrees over wide ones. PN search or a variant thereof has been successfully applied to the endgame of Awari [3], chess [7], checkers [23] and Shogi [24]. Since PN search is a best-first search, it is has to store the whole search tree in memory. When the memory is full, the search has to end prematurely. To overcome this problem PN^2 was proposed in [2], as an algorithm to reduce memory requirements in PN search. It is elaborated upon in [6]. Its implementation and testing for chess positions is extensively described in [8]. PN^2 performs two levels of PN search, one at the root and one at the leaves of the first level. As in the B* algorithm [5], a search process is started at the leaves to obtain a more accurate evaluation. Although it uses far less memory than PN search, it is still a best-first search algorithm with the disadvantage that the search can end prematurely because of memory exhaustion. Recently, the idea behind the MTD(f) algorithm [19] is successfully applied in PN variants: try to construct a depth-first algorithm behaving as its corresponding best-first algorithm. In 1995, Seo formulated a depth-first iterative-deepening version of PN search, later called PN* [24]. The advantage of this variant is that there is no need to store the whole tree in memory. The disadvantage is that PN* is slower than PN [21]. Other depth-first variants are PDS [16] and df-pn [17]. Although their generation of nodes is even slower than PN*'s, they are building smaller search trees. Hence, they are in general slightly more efficient than PN*.

In this paper we provide details on a new PN-search algorithm, called PDS-PN. It is a two-level algorithm combining a first-level PDS with a second-level PN search. The algorithm is tested on endgame positions in the domain of Lines of Action (LOA), with emphasis on really hard problems.

The remainder of this paper is organised as follows. In section 2 we explain the working of PDS-PN by elaborating on PDS and the idea of two-level search algorithms. Then, in section 3, the results of experiments with PDS-PN in the domain of Lines of Action are given. Finally, in section 4 we present our conclusions and propose topics for further research.

2 PDS-PN

In this section we give a description of PDS-PN search, which is a two-level search with PDS at the first level and PN at the second level. In subsection 2.1 we motivate why we developed the method. In subsection 2.2 we describe the first-level PDS, and in subsection 2.3 we provide background information on the second-level technique.

2.1 Motivation

We were motivated to develop the PDS-PN algorithm by the clear advantage that PDS is traversing a depth-first tree instead of a best-first tree. Hence, PDS is not restricted by the available working memory. As against this, PN has the advantage of being fast compared to PDS.

The PDS-PN algorithm is designed to combine the two advantages. At the first level, the search is a depth-first search, which implies that PDS-PN is not restricted by memory. At the second level the focus is on fast PN. It is a complex balance, but we expect that PDS-PN would be faster than PDS and PDS-PN would not be hampered by memory restrictions. Since the expectation on the effectiveness of PDS-PN is difficult to prove we have to rely on experiments (see section 3). In the next two subsections we start describing PDS-PN. The pseudo code is given in the appendix.

2.2 First-level: Proof-number and Disproof-number Search

PDS-PN is a two-level search like PN^2. At the first level a PDS search is performed, denoted pn_1. For the expansion of a pn_1 leaf node, not stored in the transposition table, a PN search is started, denoted pn_2.

Proof-number and Disproof-number Search (PDS) [16] is a straightforward extension of PN*. Instead of using only proof numbers such as in PN*, PDS uses disproof numbers too. PDS exploits a method called *multiple-iterative deepening*. Instead of iterating only in the root such as in ordinary iterative deepening, PDS iterates in *all* interior nodes. The advantage of using the multiple-iterative-deepening method is that in most cases it accomplishes to select the most-proving node (see below), not only in the root, but also in the interior nodes of the search tree. To keep iterative deepening effective, the method is enhanced by storing the expanded nodes in a TwoBig transposition table [9]. PDS uses two thresholds in searching, one for the proof numbers and one for the disproof numbers. A *proof number* of a node represents the minimum number of leaf nodes which have to be proved in order to prove the node. Analogously, a *disproof number* of a node represents the minimum number of leaves which have to be disproved in order to disprove the node. Because the goal of the tree is to prove a forced win, winning nodes are regarded as proved. Lost or drawn nodes are regarded as disproved. Therefore, proved nodes have proof number 0 and disproof number ∞, since no leaves have to be proved anymore to prove the win, and no number of leaves to be expanded suffices to disprove the node. Analogously, disproved nodes have proof number ∞ and disproof number 0. The proof number of an internal AND node is equal to the sum of its childrens' proof numbers, since to prove an AND node all the children have to be proved. The disproof number of an AND node is equal to the minimum of its childrens' disproof numbers, since to disprove an AND node only one child has to be disproved. Analogously, the disproof number of an internal OR node is equal to the sum of its childrens' disproof numbers. Its proof number is equal to the minimum of its childrens' proof numbers. To select the next node to expand (the *most-proving* node) the following holds: in an OR node the child with the lowest proof number is selected as successor, and in an AND node the child with the lowest disproof number is selected as successor.

PDS uses two thresholds for a node, one as a limit for proof numbers and one for disproof numbers. Once the thresholds are assigned to a node, the subtree rooted in that node is stopped to be searched if both the proof number and disproof number are larger than or equal to the thresholds *or* if the node is proved or disproved. The thresholds are set in the following way. Initially, the proof-number threshold *pnt* and disproof-number threshold *dnt* of a node are equal to the node's proof number *pn* and disproof number *dn*. If it seems more likely that the node can be proved than disproved (called *proof-like*), the proof-number threshold is increased. If it seems more likely that the node can be disproved than proved (called *disproof-like*), the disproof-number threshold is increased. In passing we note that it is easier to prove a tree in an OR node, and to disprove a tree in an AND node. Below we repeat Nagai's [16] heuristic to determine proof-like and disproof-like.

In an interior OR node n with parent p (direct ancestor) the solution of n is proof-like, if the following condition holds:

$$pnt_p > pn_p \;\; AND \;\; (pn_n \leq dn_n \;\; OR \;\; dnt_p \leq dn_p) \tag{1}$$

otherwise, the solution of n is disproof-like.

In an interior AND node n with parent p (direct ancestor) the solution of n is disproof-like, if the following condition holds:

$$dnt_p > dn_p \;\; AND \;\; (dn_n \leq pn_n \;\; OR \;\; pnt_p \leq pn_p) \tag{2}$$

otherwise, the solution of n is proof-like.

When PDS does not prove or disprove the root given the thresholds, it increases the proof-number threshold if its proof number is smaller than or equal to its disproof number, otherwise it increases the disproof-number threshold. Finally, we remark that only expanded nodes are evaluated. This is called *delayed evaluation* (cf. [2]). The expanded nodes are stored in a transposition table. The proof and disproof number of a node are set to unity when not found in the transposition table.

PDS is a depth-first search algorithm but behaves like a best-first search algorithm. By using transposition tables PDS suffers from the graph-history-interaction problem (cf. [10]). However, in the current PDS algorithm we ignore this problem, since we believe that it is less relevant for the game of Lines of Action (see subsection 3.1) than for chess.

A detailed description

A detailed step-by-step example of the working of PDS is given in figure 1. A square denotes an OR node, and a circle denotes an AND node. The numbers at the upper side of a node denote the proof-number threshold (left) and disproof-number threshold (right). The numbers at the lower side of a node denote the proof number (left) and disproof number (right).

In the first iteration (top of figure 1), threshold values of the root A are set to unity. A is expanded, and nodes B and C are generated. The proof number of A becomes 1 and the disproof number becomes 2. Because both numbers are larger than or equal to the threshold values the search stops.

In the second iteration (middle of figure 1), the proof-number threshold is incremented to 2, because the proof number of A (i.e., 1) is the smaller one of both A's proof number and disproof number (i.e., 2). We again expand A and re-generate B and C. The proof number of A is below its proof-number threshold and we continue searching. Now we have to select the child with minimum proof number. Because B and C have the same proof number, the left-most node B is selected. Initially, we set the proof-number and disproof-number threshold of B to its proof and disproof number (both 1). Because B is an AND node we have to look whether the solution of B is disproof-like by checking condition 2. The disproof-number threshold of A is not larger than its disproof number (both are 2), therefore the solution of B is not disproof-like but proof-like. Thus, the proof-number threshold of B has to be incremented to 2. Next, node B is expanded and the nodes D, E, F and G are generated. The search in node B is stopped because its proof number (i.e., 4) and disproof number (i.e., 1) are larger than or equal to the thresholds (i.e., 2 and 1, respectively). Node B is stored in the transposition table with proof number 4 and disproof number 1. Then the search backtracks to A. There we have to check whether we still can continue searching A. Since the proof number of A is smaller than its threshold, we continue and subsequently we select C, because this node has now the minimum proof number. The thresholds are set in the same way as in node B. Node C has two children H and I. The search at node C is stopped because its proof number (i.e., 2) and disproof number

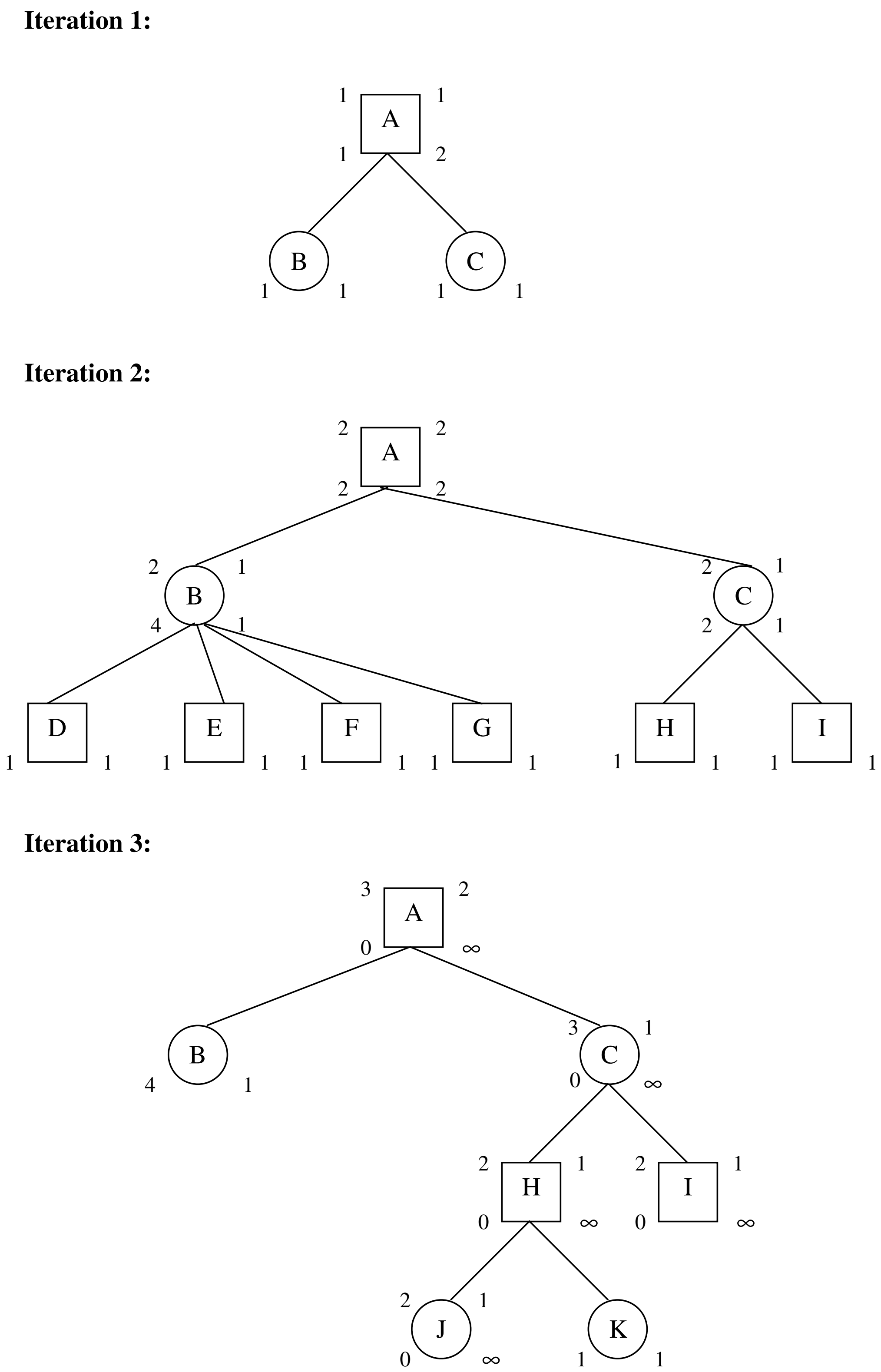

Fig. 1. An illustration of PDS.

(i.e., 1) are not below the thresholds. C is stored in the transposition table with proof number 2 and disproof number 1. The search backtracks to A and is stopped because its proof number (i.e., 2) and disproof number (i.e., 2) are larger than or equal to the thresholds. We would like to remark that at this moment B and C are stored because they were expanded.

In the third iteration (bottom of figure 1) the proof-number threshold of A is incremented to 3. Nodes B and C are again generated, but this time we can find their proof and disproof numbers in the transposition table. The node with smallest proof number is selected (C with proof number 2). Initially, we set the proof-number threshold and disproof-number threshold of C to its proof and disproof number (i.e., 2 and 1, respectively). Because C is an AND node we have to look whether the solution is disproof-like by checking condition 2. The disproof-number threshold of A is not larger than its disproof number (both are 2), therefore the solution is not disproof-like but proof-like. Thus, the proof-number threshold of C has to be incremented to 3. C has now proof-number threshold 3 and disproof-number threshold 1. Nodes H and I are generated again by expanding C. This time the proof number of C (i.e., 2) is below the proof-number threshold (i.e., 3) and the search continues. The node with minimum disproof number is selected (i.e., H). Initially, we set the proof-number threshold and disproof-number threshold of H to its proof and disproof number (i.e., both 1). Because H is an OR node we have to look whether the solution is proof-like by checking condition 1. The proof-number threshold of C (i.e., 3) is larger than its proof number (i.e., 2), therefore the solution is proof-like. Hence, the search expands node H with proof-number threshold 2 and disproof-number threshold 1. Nodes J and K are generated. Because the proof number of H (i.e., 1) is below its threshold (i.e., 2), the node with minimum proof number is selected. Because J is an AND node we have to look whether the solution of J is disproof-like by checking condition 2. The disproof-number threshold of H (i.e., 1) is not larger than its disproof number (i.e., 2), therefore the solution of J is not disproof-like but proof-like. J is expanded with proof-number threshold 2 and disproof number threshold 1. Since node J is a terminal win position its proof number is set to 0 and its disproof number set to ∞. The search backtracks to H. At node H the proof number becomes 0 and the disproof number ∞, which means the node is proved. The search backtracks to node C. The search continues because the proof number of C (i.e., 1) is not larger than or equal to the proof-number threshold (i.e., 3). We select now node I because it has the minimum disproof number. The thresholds of node I are set to 2 and 1, as was done in H. The node I is a terminal win position; therefore its proof number is set to 0 and its disproof number to ∞. At this moment the proof number of C is 0 and the disproof number ∞, which means that the node is proved. The search backtracks to A. The proof number of A becomes 0, which means that the node is proved. The search stops at node A and the tree is proved.

At the leaves of the first-level search tree, the second-level search is called. The characteristics of the pn_2 search are described in subsection 2.3.

2.3 Second level: PN Search

The PN search of the second-level, denoted pn_2 search, is bounded by the number of nodes that may be stored in memory. The number is a fraction of the size of the pn_1-search tree, for which we take the current number of nodes stored in the transposition table of the PDS search. Preferably, this fraction should start small, and grow larger as the size of the first-level search tree increases. A standard model for this growth is the logistic-growth model [4]. The fraction *f(x)* is therefore given by the logistic growth function, x being the size of the first-level search:

$$f(x) = \frac{1}{1 + e^{\frac{a-x}{b}}} \tag{3}$$

with parameters a and b, both strictly positive. The parameter a determines the transition point of the function: as soon as the size of the first-level search tree reaches a, the second-level search equals half the size of the first-level search. Parameter b determines the S-shape of the function: the larger b, the more stretched the S-shape is. The number of nodes y in a pn_2-search tree is restricted by the minimum of this fraction function and the number of nodes which can still be stored. The formula to compute y is:

$$y = min(x \times f(x), N - x) \tag{4}$$

with N the maximum number of nodes to be stored in memory.

The pn_2 search is stopped when the number of nodes stored in memory exceeds y or the subtree is (dis)proved. For details on PN search we refer to [3]. After completion of the pn_2-search tree, only the root of the pn_2-search tree is stored in the transposition table of the PDS search. We would like to remark that for pn_2-search trees *immediate evaluation* (cf. [2]) is used. This two-level search is schematically sketched in figure 2.

In the second-level search proved or disproved subtrees are deleted. If we do not delete proved or disproved subtrees in the pn_2 search, the number of nodes searched becomes the same as y. When we include deletions the second-level search can continue on average considerably longer. Preliminary results have shown that deleting proved or disproved subtrees in the pn_2 search causes a significant reduction in the number of nodes investigated [25].

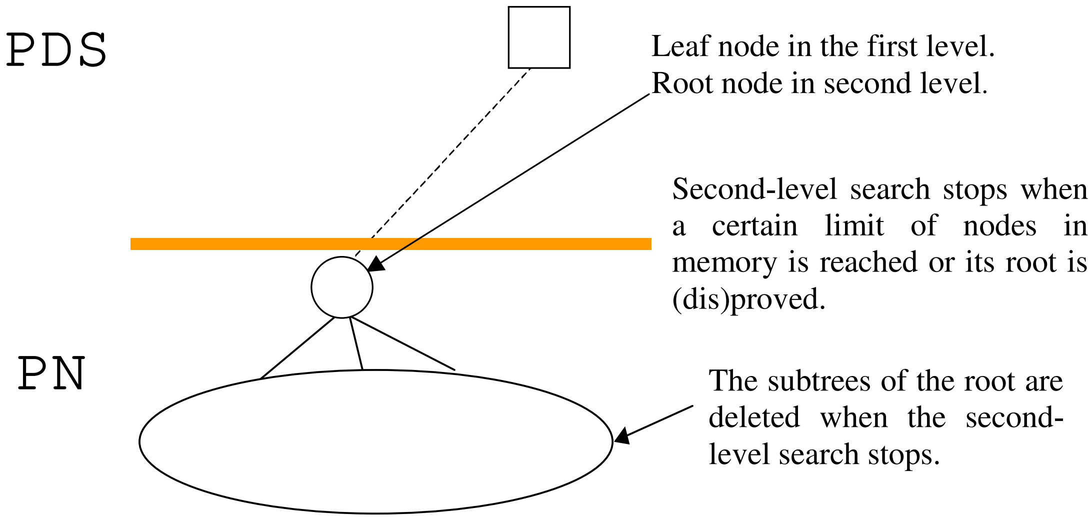

Fig. 2. Schematical sketch of PDS-PN.

3 Experiments

In this section we compare $\alpha\beta$, PN^2, PDS and PDS-PN search with each other. All experiments have been performed in the framework of the tournament program MIA (Maastricht In Action).[1] It has been written in Java and runs on every well-known operating system. For $\alpha\beta$ MIA performs a depth-first iterative-deepening search using a TwoDeep transposition table [9], neural-network move ordering [13] and killer moves [1]. For PDS and PDS-PN MIA uses a TwoBig transposition table. In subsection 3.1 we briefly describe the game of Lines of Action (LOA). In subsection 3.2 we compare PDS-PN with $\alpha\beta$, PN^2 and PDS on a set of 488 LOA positions. A second comparison of PDS-PN with PN^2 and PDS is made in subsection 3.3. In subsection 3.4 we compare PDS-PN with PN^2 on a set of really hard LOA problems.

3.1 Lines of Action

Lines of Action (LOA) is a two-person zero-sum chess-like connection game with perfect information. The details of this game are described in [20]. An interesting property of the game is that it is not suitable for building endgame databases. Although reasonable effort has been undertaken to construct adequate evaluation functions for LOA [26], experiments have shown that these are not very good predictors in the case of forced wins [25]. Therefore, LOA seems an appropriate test domain for PN-search algorithms.

[1] MIA can be played at the website: http://www.cs.unimaas.nl/m.winands/loa/.

In the experiments with PN^2, PDS and PDS-PN all nodes evaluated during the search are counted; for the $\alpha\beta$ depth-first iterative-deepening searches nodes at depth i are counted only during iteration i. We adopted this method from [2]. It makes a general comparison possible. The maximum number of nodes searched is 50,000,000. The limit corresponds roughly to tournament conditions. The maximum number of nodes stored in memory is 1,000,000. The parameters (a,b) of the growth function used in PN^2 are set at (1800K, 240K) according to the suggestions in [8]. Our experiments revealed that the best parameter configuration for the growth function of PDS-PN is (450K, 300K). The smaller value of a corresponds with the smaller pn_1 trees resulting from the use of PDS-PN instead of PN^2. The fact that PDS is much slower than PN is an important factor too.

$\alpha\beta$, PN^2, PDS and PDS-PN are tested on a set of 488 forced-win LOA positions. [2] The results are given in table 1. In the first column the four algorithms are mentioned. In the second column we see that 382 positions are solved by $\alpha\beta$, 470 positions by PN^2, 473 positions by PDS and 467 positions by PDS-PN. In the third and fourth column the number of nodes and the time consumed are given for the subset of 371 positions, which all four algorithms are able to solve. This set contains no position that only could be solved by $\alpha\beta$ search. A look at the third column shows that PDS search builds the smallest search trees and $\alpha\beta$ by far the largest. PN^2, PDS and PDS-PN solve significantly more positions than $\alpha\beta$. This suggests that PN-search algorithms are better endgame solvers than $\alpha\beta$. PN^2 and PDS-PN investigate more nodes than PDS, but both are still faster in CPU time than PDS for this subset. Due to the limit of 50,000,000 nodes and the somewhat lower search efficiency, PDS-PN solves three positions fewer than PN^2 and six fewer than PDS.

Table 1
Comparing the search algorithms on 488 test positions with a limit of 50,000,000 nodes.

Algorithm	# of positions solved (out of 488)	371 positions	
		Total # of nodes	Total time (ms.)
$\alpha\beta$	382	2,645,022,391	33,878,642
PN^2	470	505,109,692	3,642,511
PDS	473	239,896,147	16,960,325
PDS-PN	467	924,924,336	5,860,908

To investigate whether the memory restrictions are an actual obstacle we

[2] The test set is found at http://www.cs.unimaas.nl/m.winands/loa/tscg2002a.zip.

increase the limit of nodes searched to 500,000,000 nodes. Now PN^2 solves 479 positions and PDS-PN becomes the best solver with a performance of 483 positions. The detailed results are given in table 2.

Table 2
Comparing PN^2 and PDS-PN on 488 test positions with a limit of 500,000,000 nodes.

Algorithm	# of positions solved (out of 488)	479 positions	
		Total # of nodes	Total time (ms.)
PN^2	479	2,261,482,395	13,295,688
PDS-PN	483	4,362,282,235	23,398,899

The performances of PDS-PN in table 2 are more effective than those of PN^2, viz. 483 to 479. However, we should thoughtfully take into account the condition for the total number of nodes searched and the time spent. Therefore we continue our research in the direction of nodes searched and time spent with the 50,000,000 nodes limit. A reason for this decision is that the experimental time constraints are necessary for the PDS experiments.

3.3 A Second Comparison

For a better insight into the relation between PN^2, PDS and PDS-PN we did a second comparison. In table 3 we provide the results of PN^2, PDS and PDS-PN on a new subset of 457 positions of the principal test set, viz. all positions the three algorithms could solve under the 50,000,000 nodes limit condition. Now, PN^2 searches 2.6 times more nodes than PDS. The reason for the difference of performance is that for hard problems the pn_2-search tree becomes as large as the pn_1-search tree. Therefore, the pn_2-search tree is causing more overhead. However, if we look at the CPU time we see that PN^2 is almost four times faster than PDS. PDS has a relatively large overhead of time because it performs multiple-iterative deepening at all nodes. PDS-PN searches 3.7 times more nodes than PDS but is still three times faster than PDS in CPU time. This is because PDS-PN is focussing more on the fast PN at the second level than on PDS at the first level. PDS-PN searches more nodes than PDS since the pn_2-search tree is repeatedly rebuilt and removed. The overhead is even bigger than PN^2's overhead because the children of the root of the pn_2-search tree are not stored (i.e., this is done to focus more on the fast PN search). It explains why PDS-PN searches 1.4 times more nodes than PN^2. Hence, our provisional conclusions are that on this set of 457 positions and under the 50,000,000 nodes condition: (1) PN^2 outperforms PDS-PN, and (2) PDS-PN is a faster solver than PDS and therefore more effective than PDS.

Table 3
Comparing PN^2, PDS and PDS-PN on 457 test positions (all solved) with a limit of 50,000,000 nodes.

Algorithm	Total # of nodes	Total time (ms.)
PN^2	1,275,155,583	9,357,663
PDS	498,540,408	36,802,350
PDS-PN	1,845,371,831	11,952,086

3.4 Hard Problems

Since the impact of the 50,000,000 nodes condition somewhat obscured our provisional conclusions above and since we felt that PDS-PN had its own merits in comparison with PN^2 we performed a new experiment with really hard LOA problems. In this experiment PN^2 and PDS-PN are tested on a different set of 286 really hard LOA positions. [3] The conditions are the same as in the previous experiments except that the maximum number of nodes searched is set at 500,000,000. The PDS algorithm is not included because it takes too much time given the current node limit. In table 4 we see that PN^2 solves 265 positions and PDS-PN 276. We would like to remark that PN^2 solves 10 positions, which PDS-PN does not solve. The ratio in nodes and time between PN^2 and PDS-PN for the positions solved by both (255) is roughly similar to the previous experiments. The reason why PN^2 solves fewer positions than PDS-PN is its being restricted in working memory. We are in a delicate position since new experiments with much more working memory are now on the list to be performed. However, we assume that the nature of PN^2 with respect to using so much memory cannot be overcome. Hence we conclude that within an acceptable time frame PDS-PN is a more effective endgame solver than PN^2 for really hard problems.

Table 4
Comparing PN^2 and PDS-PN on 286 really hard test positions with a limit of 500,000,000 nodes.

Algorithm	# of positions solved (out of 286)	255 positions	
		Total # of nodes	Total time (ms.)
PN^2	265	10,061,461,685	57,343,198
PDS-PN	276	16,685,733,992	84,303,478

[3] The test set can be found at http://www.cs.unimaas.nl/m.winands/loa/tscg2002b.zip.

4 Conclusions and Future Research

Below we offer three conclusions and one suggestion for future research. Our first conclusion is that PN^2, PDS and PDS-PN are able to solve significantly more LOA endgame problems than $\alpha\beta$. However, we remark that PN^2 is restricted by working memory, and that PDS is four times slower than PN^2 and three times slower than PDS-PN (see table 3), because of multiple-iterative deepening.

Our second conclusion is that the PDS-PN algorithm is almost as fast as PN^2 when the parameters for its growth function are chosen properly. Our third conclusion states that (1) PDS-PN solves more really hard positions than PN^2 within an acceptable time frame and (2) PDS-PN is more effective than PN^2 because it does not run out of memory for really hard problems. Hence, in summary we conclude that PDS-PN is a more effective endgame solver for a set of really hard problems than PDS and PN^2.

Finally, we believe that an adequate challenge is testing PDS-PN in other domains with difficult endgames. Recently, for Shogi (Japanese chess) some of the hard problems including solutions over a few hundred ply are solved by PN* [24] and PDS [21]. It would be interesting to test PDS-PN on these problems.

Acknowledgements

The authors would like to thank the members of the Maastricht Search & Games Group and the referees for their valuable comments.

References

[1] S.G. Akl and M.M. Newborn. The principal continuation and the killer heuristic. In *1977 ACM Annual Conference Proceedings*, pages 466–473. ACM, Seattle, 1977.

[2] L.V. Allis. *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. Thesis, University of Limburg, Maastricht, The Netherlands, 1994.

[3] L.V. Allis, M. van der Meulen, and H.J. van den Herik. Proof-number search. *Artificial Intelligence*, 66(1):91–123, 1994.

[4] D.D. Berkey. *Calculus*. Saunders College Publishing, New York, NY, USA, 1988.

[5] H.J. Berliner. The B*-tree search algorithm: a best-first proof procedure. *Artificial Intelligence*, 12(1):23–40, 1979.

[6] D.M. Breuker. *Memory versus Search in Games.* Ph.D. Thesis, Universiteit Maastricht, Maastricht, The Netherlands, 1998.

[7] D.M. Breuker, L.V. Allis, and H.J. van den Herik. How to mate: Applying proof-number search. In H.J. van den Herik, I.S. Herschberg, and J.W.H.M. Uiterwijk, editors, *Advances in Computer Chess 7*, pages 251–272. University of Limburg, Maastricht, The Netherlands, 1994.

[8] D.M. Breuker, J.W.H.M. Uiterwijk, and H.J. van den Herik. The PN^2-search algorithm. In H.J. van den Herik and B. Monien, editors, *Advances in Computer Games 9*, pages 115–132. IKAT, Universiteit Maastricht, Maastricht, The Netherlands, 2001.

[9] D.M. Breuker, J.W.H.M. Uiterwijk, and H.J. van den Herik. Replacement schemes and two-level tables. *ICCA Journal*, 19(3):175–180, 1996.

[10] D.M. Breuker, H.J. van den Herik, J.W.H.M. Uiterwijk, and L.V. Allis. A solution to the GHI problem for best-first search. *Theoretical Computer Science*, 252(1-2):121–149, 2001.

[11] M. Campbell, A.J. Hoane Jr., and F. h. Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

[12] D.E. Knuth and R.W. Moore. An analysis of alpha-beta pruning. *Artificial Intelligence*, 6(4):293–326, 1975.

[13] L. Kocsis, J.W.H.M. Uiterwijk, and H.J. van den Herik. Move ordering using neural networks. In L. Montosori, J. Váncza, and M. Ali, editors, *Engineering of Intelligent Systems, Lecture Notes in Artificial Intelligence, Vol. 2070*, pages 45–50. Springer-Verlag, Berlin Heidelberg, 2001.

[14] U. Lorenz and V. Rottmann. Parallel controlled conspiracy-number search. In H.J. van den Herik and J.W.H.M. Uiterwijk, editors, *Advances in Computer Chess 8*, pages 129–152. Universiteit Maastricht, Maastricht, The Netherlands, 1997.

[15] D.A. McAllester. Conspiracy numbers for min-max search. *Artificial Intelligence*, 35(1):278–310, 1988.

[16] A. Nagai. A new AND/OR tree search algorithm using proof number and disproof number. In *Proceedings of Complex Games Lab Workshop*, pages 40–45. ETL, Tsukuba, Japan, 1998.

[17] A. Nagai and H. Imai. Application of df-pn+ to Othello endgames. In *Proceedings of Game Programming Workshop in Japan '99*, pages 16–23. Hakone, Japan, 1999.

[18] E.V. Nalimov, G.M^{c}C. Haworth, and E.A. Heinz. Space-efficient indexing of chess endgame tables. *ICGA Journal*, 23(3):148–162, 2000.

[19] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin. Best-first fixed-depth minimax algorithms. *Artificial Intelligence*, 87(2):255–293, 1996.

[20] S. Sackson. *A Gamut of Games.* Random House, New York, NY, USA, 1969.

[21] M. Sakuta and H. Iida. The performance of PN*, PDS and PN search on 6×6 Othello and Tsume-Shogi. In H.J. van den Herik and B. Monien, editors, *Advances in Computer Games 9*, pages 203–222. Universiteit Maastricht, Maastricht, The Netherlands, 2001.

[22] J. Schaeffer. Conspiracy numbers. *Artificial Intelligence*, 43(1):67–84, 1990.

[23] J. Schaeffer and R. Lake. Solving the game of checkers. In R. J. Nowakowski, editor, *Games of No Chance*, pages 119–133. Cambridge University Press, Cam-

bridge, UK, 1996.

[24] M. Seo, H. Iida, and J.W.H.M. Uiterwijk. The PN*-search algorithm: Application to Tsume-Shogi. *Artificial Intelligence*, 129(1-2):253–277, 2001.

[25] M.H.M. Winands and J.W.H.M. Uiterwijk. PN, PN^2 and PN* in Lines of Action. In J.W.H.M. Uiterwijk, editor, *The CMG Sixth Computer Olympiad Computer-Games Workshop Proceedings*. Technical Reports in Computer Science CS 01-04, Universiteit Maastricht, Maastricht, The Netherlands, 2001.

[26] M.H.M. Winands, J.W.H.M. Uiterwijk, and H.J. van den Herik. The quad heuristic in Lines of Action. *ICGA Journal*, 24(1):3–15, 2001.

Appendix

Below the pseudo code of PDS-PN is given. For ease of comparison we use similar pseudocode as given in [16] for the PDS algorithm. The proof number in an OR node and the disproof number in an AND node are equivalent. Analogously, the disproof number in an OR node and the proof number in an AND node are equivalent. As they are dual to each other, an algorithm similar to negamax in the context of minimax searching can be constructed. This algorithm is called NegaPDSPN. In the following, `proofSum(n)` is a function that computes the sum of the proof numbers of all the children. The function `disproofMin(n)` computes the minimum of all the children. The procedures `putInTT()` and `lookUpTT()` store and retrieve information to and from the transposition table. `isTerminal(n)` checks whether a node is a win, a loss or a draw. The function `generateChildren(n)` generates the children of the node. By default, the proof number and disproof number of a node are set to unity. The procedure `findChildrenInTT(n)` checks whether the children are already stored in the transposition table. If a hit occurs for a child, its proof number and disproof number are set to the values found in the transposition table. The procedure `PN()` is just the plain PN search. The algorithm is described in [2] and [6]. The function `computeMaxNodes()` computes the number of nodes which may be stored for the PN search, according to equation 4.

```
//iterative deepening at root r
procedure NegaPDSPN(r){

  r.proof = 1;
  r.disproof = 1;

  while(true){
    MID(r);
    // terminate when the root is proved or disproved
```

```
    if(r.proof = 0 || r.disproof = 0)
      break;

    if(r.proof <= r.disproof)
      r.proof++;
    else
      r.disproof++;
  }
}

//explore node n
procedure MID(n){

  //Look up in the transposition table
  lookUpTT(n,&proof,&disproof)
  if(proof = 0 || disproof = 0
  || (proof >= n.proof && disproof >= n.disproof)){
    n.proof = proof; n.disproof = disproof;
    return;
  }

  //Terminal node
  if(isTerminal(n)){
    if((n.value = true && n.type = AND_NODE)
    ||(n.value = false && n.type = OR_NODE)){
      n.proof = INFINITY; n.disproof = 0;
    }
    else{
      n.proof = 0; n.disproof = INFINITY;
    }
    putInTT(n);
    return;
  }

  generateChildren(n);
  //avoid cycles
  putInTT(n);

  //Multiple iterative deepening
  while(true){
    //Check whether the children are already stored in the TT.
    //If a hit occurs for a child, give its proof number and
    //disproof number the values found in the TT.
    findChildrenInTT(n);
```

```
    //Terminate searching when both proof and disproof number
    //exceed their thresholds
    if(proofSum(n) = 0 || disproofMin(n) = 0 || (n.proof <=
    disproofMin(n) && n.disproof <= proofSum(n))){
      n.proof = disproofMin(n);
      n.disproof = proofSum(n);
      putInTT(n);
      return;
    }

    proof = max(proof,disproofMin(n));
    n_child = selectChild(n,proof);

    if(n.disproof > proofSum(n) && (proof_child <= disproof_child
      || n.proof <= disproofMin(n)))
      n_child.proof++;
    else
      n_child.disproof++;

    //This is the PDS-PN part
    /////////////////////////////////
    if(!lookUpTT(n_child)){
      //Call PN search with a second argument the maximum number
      //of nodes in memory
      PN(n_child,computeMaxNodes());
      putInTT(n_child);
    }
    else
    /////////////////////////////////
      MID(n_child);
  }
}

//Select among children
selectChild(n,proof){

  min_proof = INFINITY;
  min_disproof  = INFINITY;
  for(each child n_child){
    disproof_child = n_child.disproof;
    if(disproof_child != 0)
      disproof_child = max(disproof_child,proof);

    //Select the child with the lowest disproof_child (if there are
    //plural children among them select the child with the lowest
```

```
    //n_child.proof)
    if(disproof_child < min_disproof || (disproof_child = min_disproof
    && n_child.proof < min_proof)){
      n_best = n_child;
      min_proof = n_child.proof;
      min_disproof = disproof_child;
    }
  }
  return n_best;
}
```