PDS-PN: A New Proof-Number Search Algorithm

Application to Lines of Action

Mark H.M. Winands, Jos W.H.M. Uiterwijk, and 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
{m.winands, uiterwijk, herik}@cs.unimaas.nl

Abstract. The paper introduces a new proof-number (PN) search algorithm, called PDS-PN. It is a two-level search, 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. First, we thoroughly investigate four established algorithms in the domain of LOA endgame positions: PN, PN^2, PDS and $\alpha\beta$ search. It turns out that PN^2 and PDS are best in solving hard problems when measured by the number of solutions and the solution time. However, each of those two has a practical disadvantage: PN^2 is restricted by the working memory, and PDS is relatively slow in searching. Then we formulate our new algorithm by selectively using the power of each one, viz. the two-level nature and the depth-first traversal respectively. Experiments reveal that PDS-PN is competitive with PDS in terms of speed and with PN^2 since it is not restricted in working memory.

1 Introduction

Most modern game-playing computer programs successfully use $\alpha\beta$ search with enhancements for online game-playing [10]. However, the enriched $\alpha\beta$ 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 [15]. 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) search method, such as proof-number (PN) search [3]. In some domains PN search outperforms $\alpha\beta$ search in proving the game-theoretic value of endgame positions. PN search or a variant thereof has been applied successfully to the endgame of Awari [3], chess [6], checkers [18] and Shogi [19]. In this paper we investigate several PN algorithms in the domain of Lines of Action (LOA). It turns out that the algorithms are restricted by working memory *or* by searching speed. To remove both restrictions we introduce a new PN algorithm, called PDS-PN.

The remainder of this paper is organised as follows. Section 2 explains the rules of LOA. Section 3 describes PN, PN^2, PDS; and examines their solution power and solution time, in relation to that of $\alpha\beta$. In section 4 we explain the working of PDS-PN. Subsequently, the results of the experiments with PDS-PN are given in section 5. Finally, in section 6 we present our conclusions and propose topics for further research.

2 Lines of Action

Lines of Action (LOA) [16] is a two-person zero-sum chess-like connection game with perfect information. It is played on an 8×8 board by two sides, Black and White. Each side has twelve pieces at its disposal. The black pieces are placed in two rows along the top and bottom of the board (see figure 1a), while the white pieces are placed in two files at the left and right edge of the board. The players alternately move a piece, starting with Black. A move takes place in a straight line, exactly as many squares as there are pieces of either colour anywhere along the line of movement (see figure 1b). A player may jump over its own pieces. A player may not jump over the opponent's pieces, but can capture them by landing on them. The goal of a player is to be the first to create a configuration on the board in which all own pieces are connected in one unit (see figure 1c). In the case of simultaneous connection, the game is drawn. The connections within the unit may be either orthogonal or diagonal. If a player cannot move, this player has to pass. If a position with the same player to move occurs for the third time, the game is drawn.

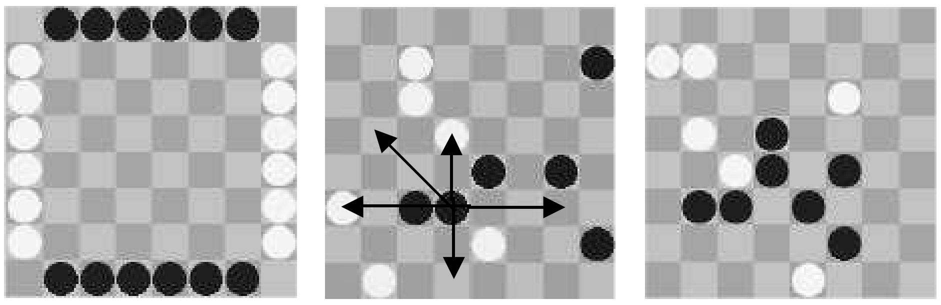

Fig. 1. (a) The initial position of LOA. (b) An example of possible moves in a LOA game. (c) A terminal LOA position.

An interesting property of the game is that most terminal positions still have more than ten pieces remaining on the board [20], which makes the game not suitable for endgame databases. Although reasonable effort has been undertaken to construct adequate evaluation functions for LOA [22], experiments still suggest that these are not very good predictions in the case of forced wins. Therefore, LOA seems an appropriate test domain for PN search algorithms.

3 Three Proof-Number Search Algorithms

In this section we give a short description of PN search, PN^2 search and PDS. We end with a comparison between PN, PN^2, PDS and $\alpha\beta$.

3.1 Proof-Number Search

Proof-number (PN) search is a best-first search algorithm especially suited for finding the game-theoretical value in game trees [2]. Its aim is to prove the true value of the root of a tree. A tree can have three values: *true*, *false* or *unknown*. In the case of a forced win, the tree is *proved* and its value is true. In the case of a forced loss or draw, the tree is *disproved* and its value is false. Otherwise the value of the tree is unknown. In contrast to other best-first algorithms PN search does not need a domain-dependent heuristic evaluation function to determine the most-promising node to be expanded next [3]. In PN search this node is usually called most-proving node. PN search selects the most-proving node using two criteria: (1) the shape of the search tree (the number of children of every internal node) and (2) the values of the leaves. These two criteria enable PN search to treat game trees with a non-uniform branching factor efficiently.

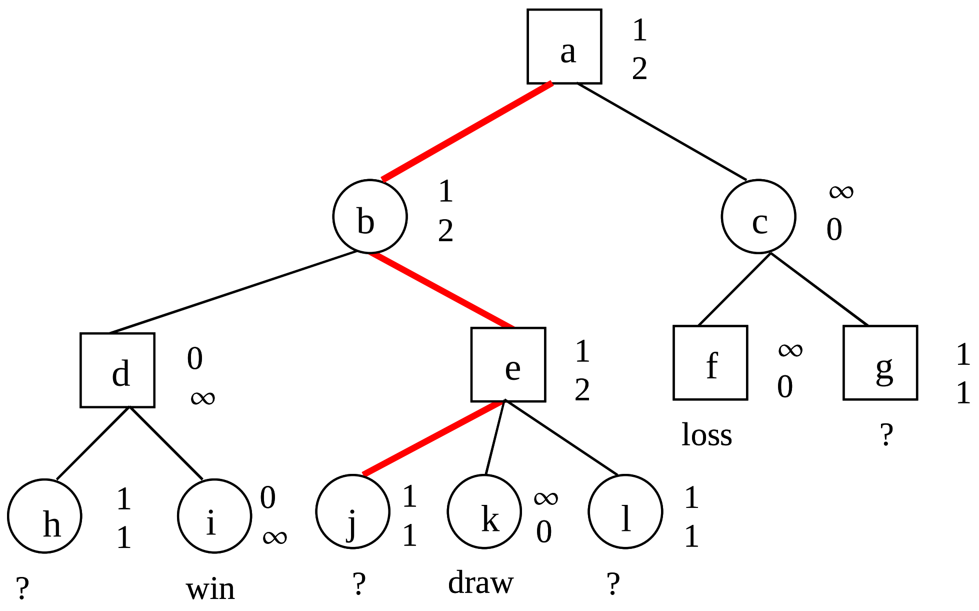

Fig. 2. An AND/OR tree with proof and disproof numbers.

Below we explain PN search on the basis of the AND/OR tree depicted in figure 2, in which a square denotes an OR node, and a circle denotes an AND node. The numbers to the right of a node denote the proof number (upper) and disproof number (lower). A *proof number* represents the minimum number of leaf nodes which have to be proved in order to prove the node. Analogously, a *disproof number* 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. Therefore, they have proof number 0 and disproof number ∞ (e.g., node i). Lost or drawn nodes are regarded as disproved (e.g., nodes f and k). They have proof number ∞ and disproof number 0. Unknown leaf nodes have a proof and disproof number of unity (e.g., nodes g, h, j and l). 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. The disproof number of an internal OR node is equal to the sum of its childrens' disproof numbers, since to disprove an OR node all the children have to be disproved. Its proof number is equal to the minimum of its childrens' proof numbers. The procedure of selecting the most-proving node to expand is the following. We start at the root. Then, at each OR node the child with the lowest proof number is selected as successor, and at each AND node the child with the lowest disproof number is selected as successor. Finally, when a leaf node is reached, it is expanded and its children are evaluated. This is called *immediate evaluation.* The selection of the most-proving node (j) in figure 2 is given by the bold path.

In the naive implementation, proof and disproof numbers are each initialised to unity in the unknown leaves. In other implementations, the proof number and disproof number are set to 1 and n for an OR node (and the reverse for an AND node), where n is the number of legal moves. In LOA this initialisation leads to a speed-up by a factor of 6 in time [21].

A disadvantage of PN search is that the whole search tree has to be stored in memory. When the memory is full, the search process has to be terminated prematurely. A partial solution is to delete proved or disproved subtrees [2]. In the next subsections we discuss two variants of PN search that handle the memory problem more adequately.

3.2 PN^2 Search

PN^2 is first described in [2], as an algorithm to reduce memory requirements in PN search. It is elaborated upon in [5]. Its implementation and testing for chess positions is extensively described in [7]. PN^2 consists of two levels of PN search. The first level consists of a PN search (pn_1), which calls a PN search at the second level (pn_2) for an evaluation of the most-proving node of the pn_1-search tree. This pn_2 search is bound by a maximum number of nodes that can be stored in memory. The number is a fraction of the size of the pn_1-search tree. The fraction *f(x)* is given by the logistic growth function [4], x being the size of the first-level search:

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

with parameters a and b, both strictly positive. The number of nodes y in a pn_2-search tree is restricted to the minimum of this fraction function or the number of nodes which can still be stored. The formula to compute y is:

$$y = min(x \times f(x), N - x) \tag{2}$$

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. After completion of the pn_2 search, the children of the root of the pn_2-search tree are preserved, but subtrees are removed from memory. The children of the most-proving node (the root of the pn_2-search tree) are not immediately evaluated by a second-level search, only when they are selected as most-proving node. This is called *delayed evaluation*. We would like to remark that for pn_2-search trees immediate evaluation is used.

As we have seen in subsection 3.1, proved or disproved subtrees can be deleted. If we do not delete proved or disproved subtrees in the pn_2 search the number of nodes searched is the same as y, otherwise we can continue the search 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 [21].

3.3 Proof-number and Disproof-number Search

In 1995, Seo formulated a depth-first iterative-deepening version of PN search, later called PN* [19]. Nagai [12, 13] proposed a depth-first search algorithm, called Proof-number and Disproof-number Search (PDS), which is a straight extension of PN*. Instead of using only proof numbers such as in PN*, PDS uses disproof numbers too. PDS uses a method called *multiple-iterative deepening*. Instead of iterating only in the root node such as in the ordinary iterative deepening, it iterates in *all* nodes. To keep iterative deepening effective, the method is enhanced by storing the expanded nodes in a TwoBig transposition table [8]. PDS uses two thresholds in searching, one for the proof numbers and one for the disproof numbers. Once the thresholds are assigned to a node, the subtree rooted at that node is continued to be searched as long as either the proof or disproof number is below the assigned thresholds. Each OR (AND) node assigns the thresholds to its children with minimum proof (disproof) number. If the threshold of the (dis)proof number is incremented in the next iteration, the search continues mainly using the (dis)proof number to find a (dis)proof. If the proof number is smaller than the disproof number, it means that it seems to have a proof solution and the threshold of the proof number is incremented. Otherwise, it seems to be a disproof solution and the threshold of the disproof number is incremented. When PDS does not (dis)prove the root given the thresholds, it increases one of the threshold values and continues searching. Finally, we remark that we check whether nodes are terminal when they are expanded. This is called delayed evaluation. 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. It is asymptotically equivalent to PN search regarding the selection of the most-proving node. In passing, we would like to remark that PDS by using transposition tables suffers from the graph-history-interaction problem (cf. [9]). Especially the GHI evaluation problem can occur in LOA too. For instance, draws can be agreed upon due to the three-fold-repetition rule. Thus, dependent on its history a node can be a draw or can have a different value. In the current PDS algorithm this problem is ignored.

3.4 Comparison

In this subsection we compare PN, PN^2, PDS and $\alpha\beta$ search with each other. All experiments have been performed in the framework of the tournament program MIA (Maastricht In Action)[1]. The program has been written in Java and can easily be ported to all platforms supporting Java. MIA performs an $\alpha\beta$ depth-first iterative-deepening search, and uses a TwoDeep transposition table [8], neural-network move ordering [11] and killer moves [1].

For the $\alpha\beta$ depth-first iterative-deepening searches nodes at depth i are counted only during the first iteration that the level is reached. This is how the comparison is done in [2]. For PN, PN^2 and PDS search all nodes evaluated for the termination condition during the search are counted. For PDS this node count is equal to the number of expanded nodes (function calls of the recursive PDS algorithm), for PN and PN^2 this node count is equal to the number of nodes generated. 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 [7].

PN, PN^2, PDS and $\alpha\beta$ are tested on a set of 488 forced-win LOA positions[2]. In the second column of table 1 we see that 470 positions were solved by the PN^2 search, 473 positions by PDS, only 356 positions by PN, and 383 positions by $\alpha\beta$. In the third and fourth column the number of nodes and the time consumed are given for the subset of 314 positions, which all four algorithms could solve. If we have a look at the third column, we see that PN search builds the smallest search trees and $\alpha\beta$ by far the largest. PDS and PN^2 build larger trees than PN but can solve significantly more positions. This suggests that both algorithms are better suited for harder problems. PN^2 investigates 1.2 times more nodes than PDS, but PN^2 is six times faster than PDS for this subset.

[1] MIA can be played at the website: http://www.cs.unimaas.nl/m.winands/loa/.
[2] The test set can be found at http://www.cs.unimaas.nl/m.winands/loa/tscg2002a.zip.

Table 1. Comparing the search algorithms on 488 test positions.

Algorithm	# of positions solved (out of 488)	314 positions	
		Total nodes	Total time (ms.)
$\alpha\beta$	383	1,711,578,143	22,172,320
PN	356	89,863,783	830,367
PDS	473	118,316,534	6,937,581
PN^2	470	139,254,823	1,117,707

For a better insight into the relation between PN^2 and PDS we did another comparison. In table 2 we compare PN^2 and PDS on the subset of 463 positions, which both algorithms could solve. Now, PN^2 searches 2.6 times more nodes than PDS. The reason for the decrease 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 have a look at the CPU time we see that PN^2 is still three times faster than PDS. The reason is that PDS has a relatively large time overhead because of the delayed evaluation (see subsection 3.3). Consequently, the number of nodes generated is higher than the number of nodes expanded. In our experiments, we observed that PDS generated nodes 7 to 8 times slower than PN. Such a figure for the overhead is in agreement with experiments performed in Othello and Tsume-Shogi [17]. We remark that difference between our LOA results and Nagai's [13] Othello results are mainly caused by domain-dependent heuristics used for the initialisation of the proof and disproof numbers.

Table 2. Comparing PDS and PN^2 on 463 test positions.

Algorithm	Total nodes	Total time (ms.)
PDS	562,436,874	34,379,131
PN^2	1,462,026,073	11,387,661

From the experiments we draw three conclusions. First, PN-search algorithms clearly outperform $\alpha\beta$ in solving endgame positions in LOA. Second, the memory problems make the plain PN search a weaker solver for the harder problems. Third, PDS and PN^2 are able to solve significantly more problems than PN and $\alpha\beta$. Finally, we note that PN^2 is restricted by its working memory, and that PDS is considerably slower than PN^2.

4 PDS-PN

In the previous section we have seen that an advantage of PN^2 over PDS is that it is faster. The advantage of PDS over PN^2 is that its tree is constructed as a depth-first tree, which is not restricted by the available working memory. To combine the advantages of both algorithms we propose an algorithm, called PDS-PN, which does not suffer from memory problems and has potentially the speed of PN^2. PDS-PN is a two-level search as is PN^2. At the first level a PDS search is performed. When a node has to be expanded, which is not stored in the transposition table, a PN search is started instead of the recursive call of the PDS algorithm. The pn_2 search is stopped as soon as (1) the subtree is (dis)proved or (2) the number of the stored nodes exceeds the number obtained by formula 2, where x equals the number of non-empty positions in the transposition table. After completion of the pn_2-search tree, only the root of the pn_2-search tree is stored in the transposition table. The PDS-PN algorithm has two advantages. First, the pn_1-search is a depth-first search, which implies that PDS-PN is not restricted by memory. Second, in PDS-PN the pn_1-search tree is growing slower in size than in PN^2. It implies that the focus is on fast PN. Hence, PDS-PN should in principle be faster than PDS. The pseudo code of PDS-PN is given in the appendix.

5 Experiments

In this section we test PDS-PN with different parameters a and b for the growth function. Next, we evaluate the algorithms PDS-PN and PN^2 in solving problems under restricted memory conditions. Finally, we compare PDS-PN with optimised parameters against PN^2. We remark that in PDS-PN at the first-level the nodes are counted as in PDS and at the second-level as in PN.

5.1 Parameter Tuning

In the following series of experiments we measured the solving ability with different parameters a and b. Parameter a takes values of 150K, 450K, 750K, 1050K and 1350K, and for each value of a parameter b takes values of 60K, 120K, 180K, 240K, 300K and 360K. The results are given in table 3. For each a holds that the number of solved positions grows with increasing b, when the parameter b is still small. If b is sufficiently large, increasing it will not enlarge the number of solved positions. In the process of parameter tuning we found that PDS-PN solves the most positions with (450K, 300K). However, the difference with parameters configurations (150K, 180K), (150K, 240K), (150K, 300K), (150K, 360K), (450K, 360K) and (1350K, 300K) is not significant. On the basis of these results we deemed that it is not necessary to perform experiments with a larger a.

5.2 Memory Results

From the experiments in subsection 3.4 it is clear that PN^2 will not be able to solve really hard problems since it will run out of working memory. To support this statement experimentally, we tested the solving ability of PN^2 and PDS with restricted working memory. In these experiments we started with a memory

Table 3. Number of solved positions for different a and b.

a	b	# of solved positions	a	b	# of solved positions
150,000	60,000	460	750,000	240,000	463
150,000	120,000	458	750,000	300,000	460
150,000	180,000	466	750,000	360,000	461
150,000	240,000	466	1,050,000	60,000	421
150,000	300,000	465	1,050,000	120,000	448
150,000	360,000	466	1,050,000	180,000	451
450,000	60,000	445	1,050,000	240,000	459
450,000	120,000	463	1,050,000	300,000	459
450,000	180,000	460	1,050,000	360,000	460
450,000	240,000	461	1,350,000	60,000	421
450,000	300,000	467	1,350,000	120,000	433
450,000	360,000	464	1,350,000	180,000	447
750,000	60,000	432	1,350,000	240,000	454
750,000	120,000	449	1,350,000	300,000	465
750,000	180,000	461	1,350,000	360,000	459

capacity sufficient to store 1,000,000 nodes, subsequently we divided the memory capacity by two at each next step. The parameters a and b were also divided by two. The relation between memory and number of solved positions for both algorithms is given in figure 3. We see that the solving performance rapidly decreases for PN^2. The performance of PDS-PN remains stable for a long time. Only when PDS-PN is restricted to fewer than 10,000 nodes, it begins to solve fewer positions. This experiment suggests that PDS-PN is preferable above PN^2 for the really hard problems, because it is not suffering from memory constraints.

5.3 Comparison with PN^2

In this subsection we compare PDS-PN (450K, 300K) with PN^2. Table 4 shows that PDS-PN was able to solve 467 positions and PN^2 470. The overlap of both sets yields a subset of 461 positions. In the third and fourth column we see that for this subset PDS-PN searches 1.4 times more nodes than PN^2. Simple calculation shows that PDS-PN is generating nodes with the same speed as PN^2. Because PDS is three times slower than PN^2, we may conclude that PDS-PN outperforms PDS in speed.

Table 4. Comparing PDS-PN and PN^2 on 488 test positions.

Algorithm	# of positions solved (out of 488)	461 positions	
		Total nodes	Total time (ms.)
PDS-PN	467	1,879,690,850	15,887,380
PN^2	470	1,302,157,677	11,339,920

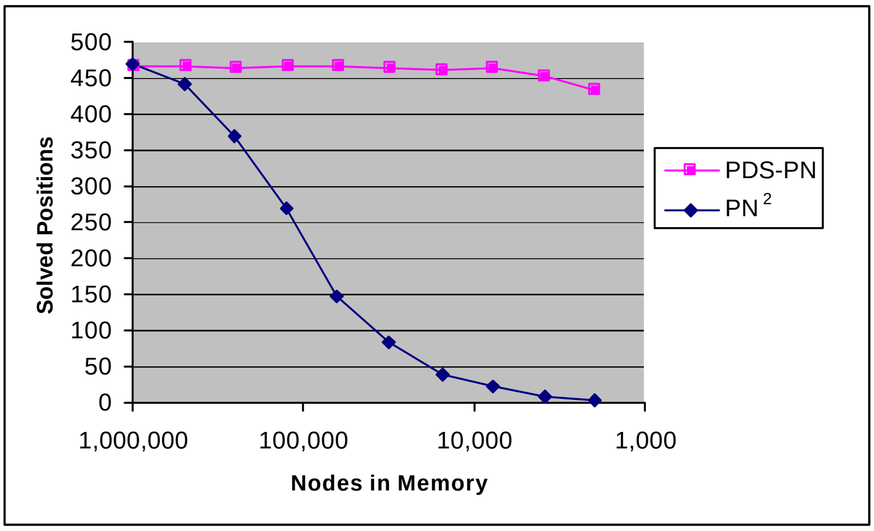

Fig. 3. Results with restricted memory.

In section 5.2 it is suggested that PDS-PN outperforms PN^2 on really hard problems. For support of this statement, PDS-PN and PN^2 are tested on a different set of 286 really hard LOA positions[3]. The conditions were the same as in previous experiments except that maximum number of nodes searched is set at 500,000,000. In table 5 we see that PDS-PN solves 276 positions and PN^2 265. We therefore conclude that, for harder problems, PDS-PN is a better endgame solver than PN^2.

Table 5. Comparing PDS-PN and PN^2 on 286 really hard test positions.

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

6 Conclusions and Future Research

Below we offer four conclusions and one suggestion for future research. First, we have seen that PN-search algorithms outperform $\alpha\beta$ in solving endgame positions in LOA. Second, the memory problems make the plain PN search a weaker solver

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

for the harder problems. Third, PDS and PN^2 are able to solve significantly more problems than PN and $\alpha\beta$. However, we remark that PN^2 is still restricted by working memory, and that PDS is three times slower than PN^2 (table 2) because of the delayed evaluation. Fourth, the PDS-PN algorithm is almost as fast as PN^2 when the parameters for its growth function are chosen properly. PDS-PN performs quite well under harsh memory conditions. Hence, we conclude that PDS-PN is an appropriate endgame solver, especially for hard problems and for environments with very limited memory such as hand-held computer platforms.

We believe that an adequate challenge is testing the PDS-PN in other games, e.g., the game of Tsume-Shogi since that game is notoriously known for its difficult endgames. Recently, some of the hard problems including solutions over a few hundred ply are solved by PN* [19] and PDS [14]. 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 for their useful remarks.

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. D.M. Breuker. *Memory versus Search in Games.* Ph.D. Thesis, Universiteit Maastricht, Maastricht, The Netherlands, 1998.
6. 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.
7. 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.
8. 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.
9. 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.
10. M. Campbell, A.J. Hoane Jr., and F. h. Hsu. Deep Blue. *Artificial Intelligence*, 134(1-2):57–83, 2002.

11. 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, 2001.
12. 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.
13. A. Nagai. *A New Depth-First-Search Algorithm for AND/OR Trees.* M.Sc. Thesis, The University of Tokyo, Tokyo, Japan, 1999.
14. A. Nagai. *Df-pn Algorithm for Searching AND/OR Trees and Its Applications.* Ph.D. Thesis, The University of Tokyo, Tokyo, Japan, 2002.
15. 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.
16. S. Sackson. *A Gamut of Games.* Random House, New York, NY, USA, 1969.
17. 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.
18. 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, Cambridge, UK, 1996.
19. 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.
20. M.H.M. Winands. *Analysis and Implementation of Lines of Action.* M.Sc. Thesis, Universiteit Maastricht, Maastricht, The Netherlands, 2000.
21. 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.
22. 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 [12] for the PDS algorithm. The proof number at an OR node and the disproof number at an AND node are equivalent. Analogously, the disproof number at an OR node and the proof number at an AND node are similar. 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 NegaPDS. 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 of 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 [5]. The function `computeMaxNodes()` computes the number of nodes which may be stored for the PN search, according to equation 2.

```
//iterative deepening at root r
procedure NegaPDS(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();
  //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, its proof number and disproof number
    //are set to 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)){
      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;
}
```