Improving Depth-First PN-Search: $1 + \varepsilon$ Trick

Jakub Pawlewicz and Łukasz Lew

Institute of Informatics,
Warsaw University, Warsaw, Poland
{pan,lew}@mimuw.edu.pl

Abstract. Various efficient game problem solvers are based on PN-Search. Especially depth-first versions of PN-Search like DF-PN or PDS – contrary to other known techniques – are able to solve really hard problems. However, the performance of DF-PN and PDS decreases drastically when the search space significantly exceeds the available memory. A straightforward enhancement trick to overcome this problem is presented. Experiments on Atari Go and Lines of Action show great practical value of the proposed enhancement.

1 Introduction

In many two-person zero-sum games with perfect information we frequently encounter game positions there often with a non-trivial but forced win for one of the players. To compute such a winning strategy a large tree search is performed. The most profitable algorithms that successfully perform the task of finding a solution are the Proof-number search algorithms [1]. They may search as deep as 20 plies and even more deeply. However, the usage of the basic PN-Search is limited to rather short runs because of the high memory requirements. A straightforward improvement is the PN^2 algorithm which was thoroughly investigated in [3]. Yet, it is still a best-first algorithm and therefore needs memory to work with. So, the length of a single run is still limited. As a further improvement, several depth-first versions of PN-Search have appeared to overcome the memory requirements problem. They were successful in many fields. Seo [7] successfully applied it to many difficult problems in his Tsume-Shogi solver using PN^*. The PDS algorithm – an extension of PN^* was developed by Nagai [5]. Another successful algorithm is DF-PN [6] which is a straightforward transformation of PN-Search to a depth-first algorithm.

Yet even, all these methods lose their effectiveness on very hard problems when the search lasts so long that the number of positions to explore significantly exceeds the available memory. The methods spend most of the time repeatedly re-producing trees stored in the transposition table but overwritten by a search in other branches of a game tree. So, we must admit that this kind of performance leak does not only occur in alpha-beta search.

From these results, there is room for improvement of these methods. For instance, Winands et al. [9] presented a method called PDS-PN used in the LOA program MiA. This variation was created by taking the best of PN^2 and

H.J. van den Herik et al. (Eds.): CG 2006, LNCS 4630, pp. 160–171, 2007.

PDS. Kishimoto and Müller [4] successfully applied DF-PN in their tsume-Go problems solver. They enhanced DF-PN by additional threshold increments.

This paper presents a more general approach of threshold increments, which reduces the number of tree reproductions during the search and results in a quite efficient practical enhancement. The enhancement is applicable both to DF-PN and PDS and possibly to other variants of tree-traversing algorithms.

A deeper understanding of DF-PN is needed to understand the merits of the enhancement. Section 2 precisely describes the transformation of the PN-Search algorithm to a depth-first search algorithm. Section 3 provides a further insight into DF-PN search and shows its weak point along with a remedy. The same section presents also an application of our enhancement to PDS. Section 4 presents results of experiments. The last section concludes.

2 A Depth-First Transformation of PN-Search

Section 2.1 briefly describes PN-Search. Section 2.2 describes DF-PN as a depth-first transformation of PN-Search.

2.1 PN-Search

For detailed description of PN-Search we refer to [1]. We recall only a few selected properties essential for an analysis in the later sections.

The algorithm maintains a tree in which each node represents a game position. With each node we associate two numbers: the *proof-number* (PN) and the *disproof-number* (DN).

The PN(DN) of a node v is the minimum number of leaves in the subtree rooted at v valued *unknown* such that if they change theirs value to true(false) then the value of v would also change to true(false).

In other words, the PN(DN) is a lower bound on the number of leaves to expand in order to prove(disprove) v.

PN and DN for a proved node are set to 0 and $+\infty$ and for a disproved node are set to $+\infty$ and 0. In an unsolved leaf node we set both PN and DN to 1. In an unsolved internal node PN and DN can be calculated recursively as shown in Fig. 1. A square denotes an OR node and a circle denotes an AND node.

The algorithm iteratively selects a leaf and expands it. To minimize the total number of expanded nodes in the search it chooses a leaf to expand in a such way that proving(disproving) it decreases the root's PN(DN) by 1. Such a leaf is called the *most proving node* (MPN).

It can be easily shown that an MPN always exists. This leaf may be found by traversing downwards through the tree and choosing a child only on the basis of the PNs and DNs of the node's children. In a node of type OR(AND) we choose a child with the minimum PN(DN). That is the child with the same PN(DN) as its parent.

After expanding the MPN, PNs and DNs are updated by going back on the path up to the root. The algorithm stops when it determines a game value of

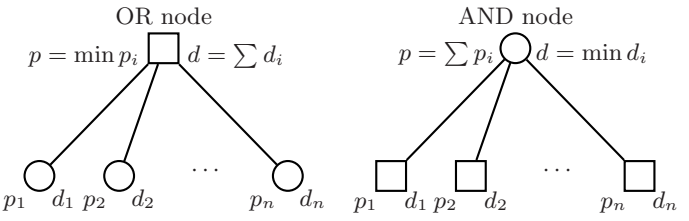

Fig. 1. In an OR node PN is a minimum of children's PNs while DN is a sum of children's DNs. In an AND node PN is a sum of children's PNs while DN is a minimum of children's DNs.

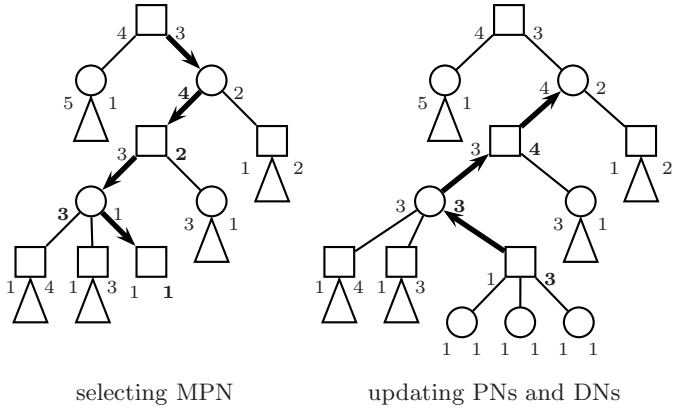

Fig. 2. Selecting MPN and updating PNs and DNs

the root, i.e., if one of the root's numbers will be infinity while the other will drop to zero. An example of selecting an MPN and updating PNs and DNs is shown in Fig. 2.

2.2 DF-PN

As in Fig. 2 we see that updating PNs and DNs can be stopped before we reach the root. It happens for instance when updating a node does change neither PN nor DN. In that case we may begin the search of the next MPN from the last updated node instead of from the root.

In fact we can shorten the way up even more. If the next MPN is in a subtree rooted in the node currently visited we can suspend update of the parent and further ancestors. Observe that we need valid values of PNs and DNs of the ancestors only while traversing downwards and selecting the MPN. Then, in general, we can suspend updating the ancestors as long as the MPN resides in the subtree of the node.

To take advantage of the above observation we introduce PN and DN thresholds. The thresholds are stored only for nodes along the path from the root to the current node. Let p and d be the PN and DN of node v. We want to define the thresholds pt and dt for node v in such a way that if $p < pt$ and $d < dt$ then there exists an MPN in the subtree rooted at v and conversely if there exists an MPN in the subtree rooted at v then $p < pt$ and $d < dt$.

We will now determine the rules for setting the thresholds. For the root we set the thresholds to $+\infty$. Clearly, the condition $p < +\infty$ and $d < +\infty$ holds only if the tree is not solved.

Now we take a closer look on what happens in an internal OR node (Fig. 3). Let p and d be the node's PN and DN respectively. Let pt and dt be

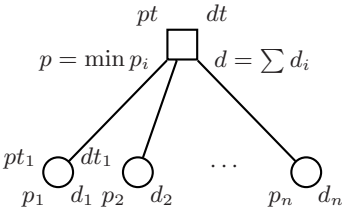

Fig. 3. Visiting the first child in an OR node and setting the thresholds

its thresholds. Assume the node has n children. Assume the i-th child's PN and DN equal p_i and d_i. Without loss of generality we assume $p_1 \leq p_2 \leq \ldots \leq p_n$.

The subtree where the MPN resides is rooted at the child with the minimum PN. Since p_1 is the smallest value, MPN lies in the leftmost subtree, so we are going to visit the first child. We have to set the thresholds pt_1 and dt_1 for this child such that if the PN or DN reaches its threshold (i.e., if $p_1 \geq pt_1$ or $d_1 \geq dt_1$) then MPN must lie outside this child's subtree.

Now, we set constraints for pt_1 to deduce the actual value. When p_1 exceeds p_2, then the second child will have the minimum PN, and MPN will no longer lie in the first child's subtree. Hence $pt_1 \leq p_2 + 1$. When p_1 reaches pt, but does not exceed p_2, then the PN p in the parent will also reach its threshold pt, and by pt threshold definition, MPN will no longer be a descendant of the parent and thus it will not lie in the child's subtree. Hence, the second constraint is $pt_1 \leq pt$. These two constraints give us the formula $pt_1 = \min(pt, p_2 + 1)$.

Consequently, when d_1 increases such that d reaches dt, then again MPN will be outside the current subtree. Now we calculate how d changes when d_1 changes to d_1'. Let d' denote DN of the parent, after DN of the first child has changed from d_1 to d_1'. Then we have $d' = d + (d_1' - d_1)$. We are interested whether $d' \geq dt$. Replacing d' and rewriting the inequality we may obtain the answer. That is: if $d_1' \geq dt - d + d_1$ then $d' \geq dt$. Therefore we have a formula for the DN threshold $dt_1 = dt - d + d_1$.

In summary, we arrive at the following formulas for an OR node for the first child's thresholds:

$$pt_1 = \min(pt, p_2 + 1)\,, \tag{1}$$
$$dt_1 = dt - d + d_1\,. \tag{2}$$

It remains to show that for the above thresholds the inequalities $p_1 < pt_1$ and $d_1 < dt_1$ hold if and only if MPN is in the first child's subtree. We have already seen that if $p_1 \geq pt_1$ or $d_1 \geq dt_1$ then MPN is not in the child's subtree. So assume $p_1 < pt_1$ and $d_1 < dt_1$. Then $p < pt$ and $d < dt$, thus MPN lies in the parent's subtree. Moreover $p_1 < p_2 + 1$. This is the same as $p_1 \leq p_2$. Thus the first child has the smallest PN among all children. Therefore MPN lies in the first child's subtree.

Similarly, we arrive at the formulas for an AND node for the first child's thresholds (assuming $d_1 \leq d_2 \leq \ldots \leq d_n$):

$$pt_1 = pt - p + p_1\,, \tag{3}$$
$$dt_1 = \min(dt, d_2 + 1)\,. \tag{4}$$

A main advantage is that using thresholds we can suspend updates as long as it is possible.

A second important advantage of this approach is the possibility of switching from (1) maintaining a whole search tree to (2) using a transposition table to store positions' (nodes') PN and DN.

If in such implementation the algorithm visits a node, we may try to retrieve its PN and DN from the transposition table searching for an early cut. In case of failure we initialize the node's PN and DN the same way as we do it for a leaf allowing the further search to reevaluate them. The resulting algorithm called DF-PN [6] is a depth-first algorithm, and it can be implemented in a recursive fashion.

The main property of DF-PN is the following. If all nodes can be stored in a transposition table, then the nodes are expanded in the same order as in the standard PN-Search. Of course, in DF-PN, we are not forced to store all nodes, and usually we store only a fraction of them, with a slight loss of efficiency.

3 Enhancement

Section 3.1 shows a usual scenario for which DF-PN has a poor performance. Section 3.2 shows our enhancement (i.e., the $1 + \varepsilon$ trick) applied to DF-PN. Section 3.3 describes an analogous improvement of PDS.

3.1 Weak Point of DF-PN

We remind that in case of a failure in retrieving the children's values from a transposition table, DF-PN initializes the children's PNs and DNs to 1 and re-search their values. Usually in an OR(AND) node with a large subtree all the

children's PNs (DNs) are very similar, because DF-PN searches the child with the smallest PN (DN) and returns as soon as it exceeds the second smallest number.

Let us consider the following typical situation during a run of DF-PN. Assume we are in an OR node with at least two children. Assume further that the threshold is big and search lasts so long that most of the investigated nodes do not fit in the transposition table.

After some time, the searched tree will be so large, that the algorithm will not be able to store most of the searched nodes. Now DF-PN will make a recursive tree search for the first child with a PN threshold fixed to the second child's PN plus one. Eventually, search will return but due to the weak threshold the new PN will be only slightly greater than before the recursion. For an example see Fig. 4.

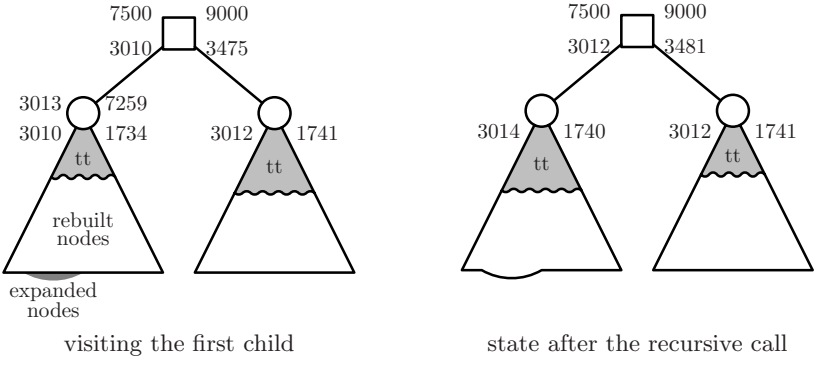

Fig. 4. Example of a single recursive call in an OR node during a run of DF-PN. The numbers are potential values during the search. Gray part of a tree marked as *tt* denotes nodes stored in a transposition table. The left picture shows what happens during the recursive call for the first child. The state after the call is shown in the right picture.

Then the control goes back to the parent level and we call DF-PN for its second child, again setting the PN threshold to its sibling's PN plus one. After expensive reconstruction of the second child's tree, its PN increases insignificantly and we will have to switch again to its sibling. However, most information from the previous search in the first child has been lost due to insufficient memory.

We see that for each successive recursive call, we have to rebuild almost the whole child's tree. Usually the number of recursive calls in the parent node is *linear* to the parent's PN threshold.

3.2 The $1 + \varepsilon$ Trick

The above example shows that when we are in an OR node, setting the PN threshold to the number one larger than p_2 can lead to quite a large number of visits in a single child, causing multiple reconstructions of a tree rooted in that

child. To be more effective we should spend more time in a single node, doing some search in advance.

We have a constraint $pt_1 \leq p_2 + 1$ for the child's PN threshold in an OR node. We can relax that constraint somewhat to a small multiplicity of p_2, for example to $1 + \varepsilon$, where ε is a small real number greater than zero. Thus we change the constraint to $pt_1 \leq \lceil p_2(1+\varepsilon) \rceil$ and the old formula (1) transforms into the new formula (5) for the child's PN threshold in an OR node

$$pt_1 = \min(pt, \lceil p_2(1+\varepsilon) \rceil)\,. \tag{5}$$

So, after each recursive call the child's PN increases by a constant factor rather by a constant addend. More precisely after the call either the parent's DN threshold is reached or the child's PN increases by at least $1 + \varepsilon$ times. Therefore, a single child can be called at most $\log_{1+\varepsilon} pt = O(\log pt)$ times before reaching the parent's PN threshold. As a consequence the described trick has a nice property of reducing the number of recursive calls from linear to *logarithmic* in the parent's PN threshold.

This enhancement not only improves the way a transposition table is used, but also reduces the overhead of multiple replaying the same sequences of moves. Here we may observe that using the presented trick we lose the property of visiting nodes in the same order as in PN-Search.

3.3 Application to PDS

The $1+\varepsilon$ trick is also applicable to PDS. In PDS we use two thresholds pt and dt like in DF-PN. The main difference is their meaning. The algorithm remains in a node until both thresholds are reached or the node is solved. PDS introduces the notion of proof-like and disproof-like nodes. When making a recursive call at a child with PN p_1 and DN d_1, it sets the thresholds $pt_1 = p_1 + 1$ and $dt_1 = d_1$ if the node is proof-like, and $pt_1 = p_1$ and $dt_1 = d_1 + 1$ if the node is disproof-like. PDS uses a straightforward heuristic to decide whether the node is proof-like or disproof-like. We refer the reader to [5] for the details.

The similar weakness as in DF-PN, described in 3.1, harms PDS. Here we apply our technique by setting the thresholds: $pt_1 = \lceil p_1(1+\varepsilon) \rceil$, $dt_1 = d_1$ for a proof-like child, and $pt_1 = p_1$, $dt_1 = \lceil d_1(1+\varepsilon) \rceil$ for a disproof-like child.

4 Experiments

Below we examine the practical efficiency of DF-PN and PDS with and without the presented enhancement. We focus on the solving times for various set-ups. First, Subsection 4.1 describes the background for the performed experiments. Then the results of these experiments are described. In Subsection 4.2 we test the influence of the size of a transposition table. In Subsection 4.3 we compare the algorithms under tournament conditions. In Subsection 4.4 we explore capabilities to solve hard problems.

4.1 Experimental Environment

We choose two games for our experiments. The first one is Atari Go, the capture game of Go. In Subsection 4.2 we take as a starting position a 6×6 board with a crosscut in the centre. The second game is Lines of Action. For more information we refer to Winands' web page [8]. The rules and testing positions, used in Subsection 4.3 and 4.4, were taken from that web page.

Our implementations of four search methods in the Atari Go and LOA games do not have any game-specific enhancements. For a transposition table, `TwoBig` scheme [2] is used.

For the accelerated version of DF-PN with the $1+\varepsilon$ trick, ε was set empirically to 1/4. With larger values of ε, enhanced DF-PN tends to over-explore significantly some nodes. With smaller values, enhanced DF-PN is usually slower.

In PDS, spending more time in one child is a common behavior because the ending condition requires to exceed both thresholds simultaneously. Usage of $1+\varepsilon$ trick in PDS makes this deep exploring behavior even more exhaustive, which often leads to over-exploring. Therefore ε should be much smaller in enhanced version of PDS. We found 1/16 as the best ε value.

All experiments were performed on 3GHz Pentium 4 with 1GB RAM under Linux.

4.2 The Size of a Transposition Table, Tested on Atari Go

We run all four methods with different transposition table sizes. The results are shown in Fig. 5 and exact times for the sizes between 2^{12} and 2^{22} nodes are shown in Table 1.

Table 1. Solving times in seconds of Atari Go 6×6 with a crosscut

	TT size in nodes										
Algorithm	2^{12}	2^{13}	2^{14}	2^{15}	2^{16}	2^{17}	2^{18}	2^{19}	2^{20}	2^{21}	2^{22}
DF-PN with $1 + \varepsilon$ trick	672	236	95	98	55	70	32	77	38	54	68
DF-PN	–	–	3403	279	189	187	104	168	111	113	106
PDS with $1 + \varepsilon$ trick	–	4895	3246	1017	468	440	527	350	400	235	269
PDS	–	–	–	4057	1448	1124	776	494	353	261	195

Obviously enhanced DF-PN is the fastest method and plain DF-PN is the second fastest.

Setting the size greater than 2^{20} does not noticeably affect the times. In that range there is no remarkable difference between plain and enhanced PDS. For sizes smaller than 2^{20}, enhanced PDS becomes faster than plain PDS.

A noticeable drop of performance can be observed when the size is below 2^{16}. Within a 2 hour time limit the following results are te be reported: (1) PDS is unable to solve the problem for transposition table with a size of 2^{14} nodes, (2) DF-PN with a size of 2^{13} nodes and (3) the enhanced PDS with a size of 2^{12} nodes.

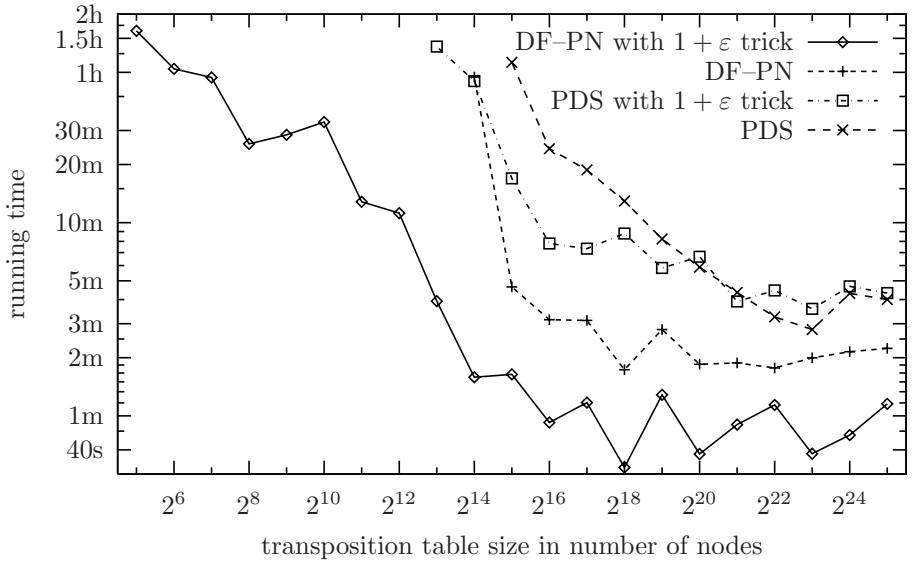

Fig. 5. Solving times of Atari Go 6×6 with a crosscut

The enhanced versions are much better for really small transposition tables. For sizes 2^{14} and smaller, enhanced DF-PN is far better than any other method. Enhanced DF-PN is able to solve the problem almost not using a transposition table at all. With a memory of 256 nodes it needed 1535 seconds and with a memory of 32 nodes it needed 5905 seconds. Of course there is a substantial information stored in the local variables in each recursive call.

4.3 Efficiency Under Tournament Conditions, Tested on a Set of Easy LOA Positions

We have already seen that DF-PN with the $1 + \varepsilon$ trick performs excellently when the search space significantly exceeds the size of a transposition table. In this Subsection we check the practical value of the methods on the set of 488 LOA positions. The purpose of this test is to evaluate the efficiency of solving the positions with tournament time constraints, as it is desired in the best computer programs.

The size of a transposition table is set to 2^{20} nodes and should fit into the memory of most computers. The results are shown in Fig. 6. The figure was created by measuring solving time for each method for every position from the set. Then for every time limit we can easily find the number of solved positions with time not exceeding the limit. Exact numbers of solved positions for selected time limits are shown in Table 2.

Here enhanced DF-PN is clearly the most efficient method, plain DF-PN is the second best and both PDS versions are the least efficient. The difference between enhanced PDS and plain PDS is unnoticeable.

Table 2. Numbers of solved positions from the set `tscg2002a.zip` [8] for selected time limits

	Time limit									
Algorithm	0.5s	1s	2s	5s	10s	20s	30s	1m	2m	5m
DF-PN with $1+\varepsilon$ trick	214	278	343	410	438	457	468	478	482	486
DF-PN	184	246	304	379	419	449	457	471	479	486
PDS with $1+\varepsilon$ trick	100	154	217	299	358	414	430	450	471	481
PDS	102	144	214	300	365	409	425	451	466	480

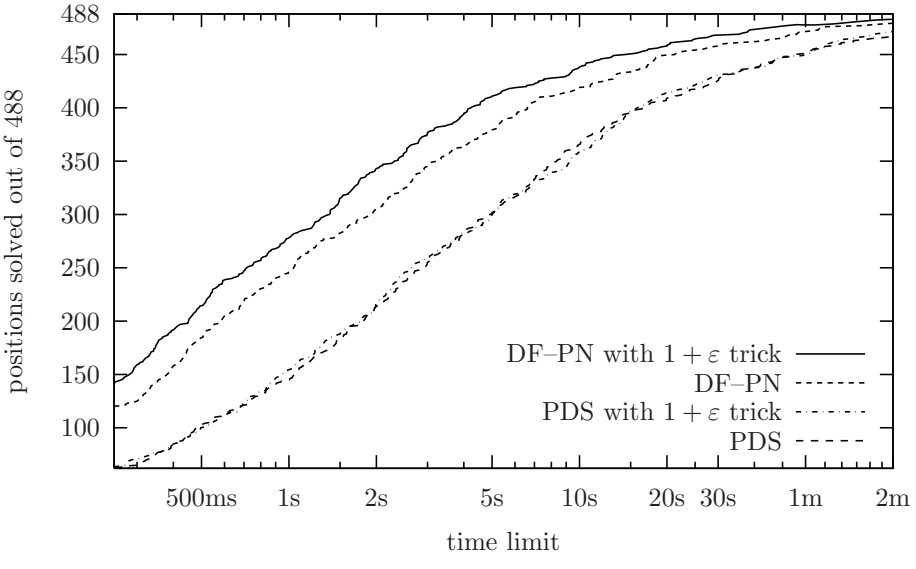

Fig. 6. Numbers of solved easy LOA positions from the set `tscg2002a.zip` [8] for given time limit

4.4 Efficiency of Solving Hard Problems, Tested on a Set of Hard LOA Positions

This experiment is aimed at checking our ability of solving harder problems in reasonable time. The 286 test positions were taken from [8]. Again we define the size of a transposition table to be 2^{20} nodes. The results are shown in Fig. 7 and the exact numbers of solved positions for selected time limits are shown in Table 3.

Again, as in the previous test, the enhanced DF-PN is the most efficient method, plain DF-PN is the second best, and both PDS versions are the least efficient. The difference between enhanced PDS and plain PDS is now noticeable. For each time limit greater than 90 seconds enhanced PDS solves more positions than plain PDS. It shows the advantage of enhanced PDS over plain PDS for harder positions.

To illustrate the speed differences in numbers for each two methods we calculated a geometric mean of the ratios of solving times (see Table 4). The geometric

Table 3. Numbers of solved positions from the set `tscg2002b.zip` [8] for selected time limits.

	Time limit										
Algorithm	5s	10s	20s	30s	1m	2m	3m	5m	10m	20m	30m
DF-PN with $1+\varepsilon$ trick	44	107	158	182	228	258	272	280	285	286	286
DF-PN	25	64	123	154	198	241	258	273	282	284	285
PDS with $1+\varepsilon$ trick	1	18	52	80	138	186	212	228	257	272	278
PDS	3	22	56	83	134	179	206	224	252	265	274

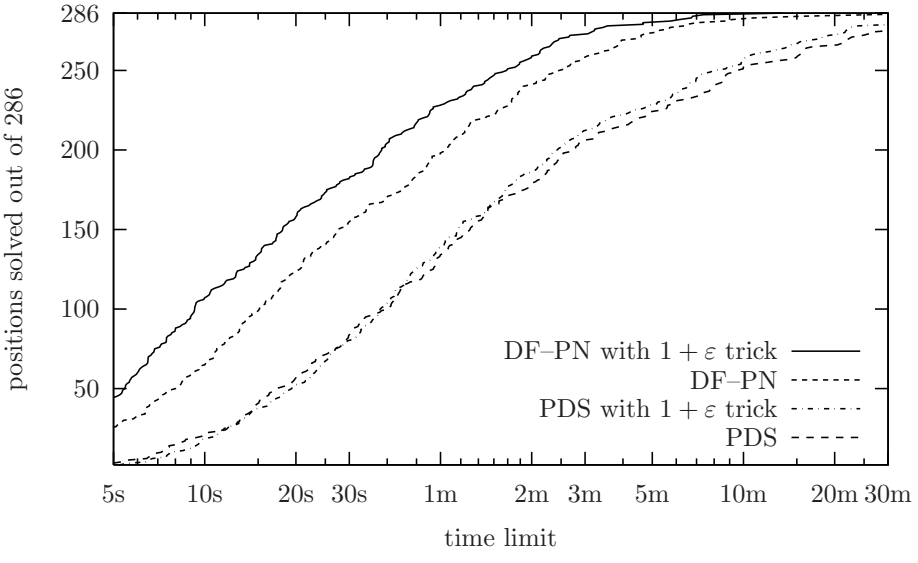

Fig. 7. Numbers of solved hard LOA positions from the hard set `tscg2002b.zip` [8] for given time limit

Table 4. Overall comparison for positions from the set `tscg2002b.zip`. The number r in the row A and the column B says A is r times faster than B in average.

	DF-PN	enhanced DF-PN	PDS	enhanced PDS
DF-PN	1.00	0.63	2.83	2.64
enhanced DF-PN	1.58	1.00	4.46	4.17
PDS	0.35	0.22	1.00	0.93
enhanced PDS	0.38	0.24	1.07	1.00

mean is more appropriate for averaging ratios than the arithmetic mean because of the following property: if A is r_1 times faster than B, B is r_2 times faster than C, and A is r_3 times faster than C then $r_3 = r_1 r_2$.

5 Conclusions and Future Work

The $1+\varepsilon$ trick has been introduced to enhance DF-PN. We have shown that the trick can also be used in PDS. The experiments showed a noticeable speedup when the search space significantly exceeds the size of a transposition table.

The trick is particularly well-suited to DF-PN, since the experiments have shown a large advantage of enhanced DF-PN over the other methods. The AtariGo experiment has shown that this method performs extremely well in low memory conditions. Moreover, DF-PN with the $1+\varepsilon$ trick was the most efficient method in the "real-life" experiment on the LOA positions so it should be valuable in practice. The $1+\varepsilon$ trick is possibly applicable to other threshold-based depth-first versions of PN-Search.

The nice property of the enhancement is that it can be added to existing implementations with little effort. Its value has to be confirmed in other games different from Atari Go and LOA. We notice that in other games and in other depth-first variants of PN-Search, the value of ε can be different, and it should be further investigated.

We hope that the presented enhancement becomes attractive for a brute-force search for solving games. However, there are still some weak points in PN-based methods and more work for improvements is required to make depth-first PN-Search even more useful.

References

1. Allis, L.V., van der Meulen, M., van den Herik, H.J.: Proof-Number Search. Artificial Intelligence 66, 91–124 (1994)
2. Breuker, D.M., Uiterwijk, J.W.H.M., van der Herik, H.J.: Replacement Schemes and Two-Level Tables. ICCA J. 19(3), 175–180 (1996)
3. Breuker, D.M., Uiterwijk, J.W.H.M., van der Herik, H.J.: The PN^2-Search Algorithm. In: van den Herik, H.J., Monien, B. (eds.) 9th Advances in Computer Games (ACG9), pp. 115–132. Department of Computer Science, Universiteit Maastricht, Maastricht, The Netherlands (2001)
4. Kishimoto, A., Müller, M.: Search Versus Knowledge for Solving Life and Death Problems in Go. In: Twentieth National Conference on Artificial Intelligence (AAAI-05), pp. 1374–1379 (2005)
5. Nagai, A.: A New AND/OR Tree Search Algorithm using Proof Number and Disproof Number. In: Proceeding of Complex Games Lab Workshop, pp. 40–45, Tsukuba, ETL (November 1998)
6. Nagai, A.: Df–pn Algorithm for Searching AND/OR Trees and its Applications. Ph.d. thesis, The University of Tokyo, Tokyo, Japan (2002)
7. Seo, M., Iida, H., Uiterwijk, J.W.H.M.: The PN*–Search Algorithm: Application to Tsume-Shogi. Artificial Intelligence 129(1–2), 253–277 (2001)
8. Winands, M.H.M.: Mark's LOA Homepage (2007), `http://www.cs.unimaas.nl/m.winands/loa/`
9. Winands, M.H.M., Uiterwijk, J.W.H.M., van den Herik, H.J.: An Effective Two-Level Proof-Number Search Algorithm. Theoretical Computer Science 313(3), 511–525 (2004)