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

Jakub Pawlewicz, Łukasz Lew
{pan,lew}@mimuw.edu.pl

Institute of Informatics, Warsaw University
Banacha 2, 02-097 Warsaw, Poland

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 dramatically when the search space significantly exceeds available memory. A simple 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 popular two-person zero-sum games there often arises a situation with a non-trivial but forced win for one of the players. In order to compute a winning strategy a large tree search is performed. PN-Search like algorithms are able to solve a problem with a forced win sequence as deep as 20 ply and deeper.

Unfortunately usage of the basic PN-Search is in fact limited to very short runs because of high memory requirements. The simple improvement is PN^2 algorithm which was deeply investigated in [1]. It is still a best-first algorithm and needs some memory to work with, so the length of the single run is still limited. Several depth-first versions of PN-Search have appeared to overcome the memory requirements problem and they were successful in many fields. Seo [2] 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 [3]. Another successful algorithm is DF-PN [4] which is a straightforward transformation of PN-Search to a depth-first algorithm.

However, all those 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 time repeatedly reproducing trees stored in the transposition table but overwritten by a search in other branches of a game tree. This kind of performance leak does not occur in alpha-beta search.

There is still a room to improve these methods. For instance Winands [5] presented a method called PDS-PN used in his LOA program. This variation was created by taking the best of PN^2 and PDS. Kishimoto and Müller [6] 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 very efficient practical enhancement. The enhancement is applicable both to DF-PN and PDS and possibly to other variants of tree-walking algorithms.

A deeper understanding of DF-PN is needed to see the enhancement value. The second section precisely describes a transformation of PN-Search algorithm to the depth-first search algorithm. The third section makes a further insight into DF-PN search and shows its weak point along with a remedy. The same section presents also an application of the trick to PDS. The fourth section presents results of experiments. The last section concludes.

2 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 [7]. We recall only 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.

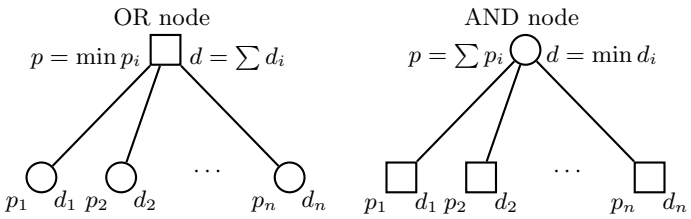

Fig. 1. In an OR node PN is a minimum of children's PNs while DN is a sum of children's DNs. In an AND node PN is a sum of children's PNs while DN is a minimum of children's DNs

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 MPN always exists. This leaf may be found by walking down the tree and choosing child only on the basis of 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 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 MPN and updating PNs and DNs is shown in Fig. 2.

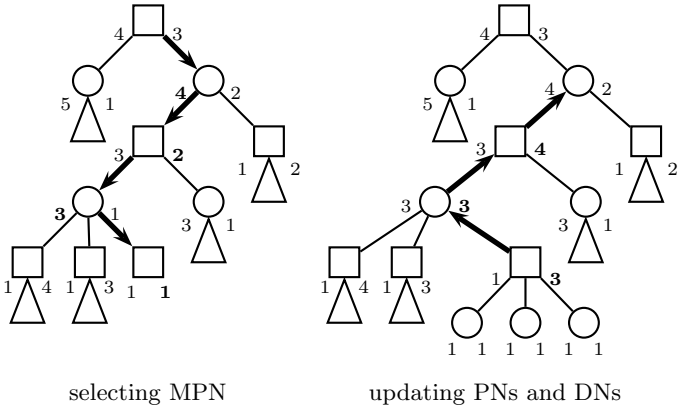

Fig. 2. Selecting MPN and updating PNs and DNs.

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 search of the next MPN from the last updated node instead 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 walking down and selecting MPN. Therefore in general we can suspend updating the ancestors as long as 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 MPN in the subtree rooted at v and conversely if there exists 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.

We now look closer what happens in an internal OR node (Fig. 3). Let p and

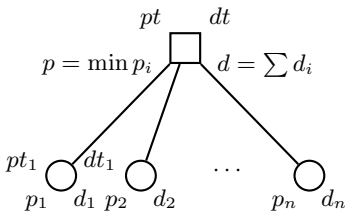

Fig. 3. Visiting the first child in an OR node and setting the thresholds

d be the node's PN and DN respectively. Let pt and dt be 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 MPN lies 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.

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. 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 get the answer. That is: if $d_1' \geq dt - d + d_1$ then $d' \geq dt$. Therefore we have a formula for DN threshold $dt_1 = dt - d + d_1$.

Summarizing, we get the formulas for an OR node for the first child's thresholds:

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

It remains to show that for the above thresholds 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 suppose $p_1 < pt_1$ and $d_1 < dt_1$. Then $p < pt$ and $d < dt$, hence 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 can get the formulas for an AND node for the first child's thresholds (assuming $d_1 \leq d_2 \leq \ldots \leq d_n$):

$$pt_1 = pt - p + p_1,$$
$$dt_1 = \min(dt, d_2 + 1).$$

Using thresholds we can suspend updates as long as it is possible.

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

In such implementation if algorithm visits a node, we can 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 [4] 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 nodes are expanded in the same order as in standard PN-Search. Of course in DF-PN, we are not obligated 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 poor performance. Section 3.2 shows 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 children's values from a transposition table, DF-PN initializes 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 search the child with the smallest PN(DN) and return as soon as it exceeds the second smallest.

Consider the following typical situation during a run of DF-PN. Suppose we are in an OR node with at least two children. Suppose further that the threshold is big and search lasts so long that most of the nodes searched 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 search. See Fig. 4 for example.

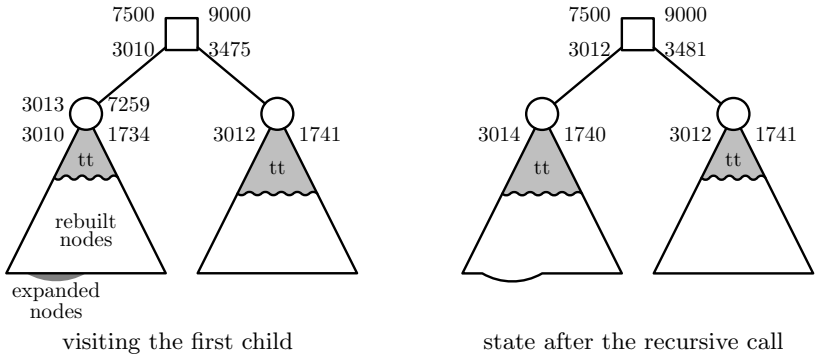

Fig. 4. Example of a single recursive call in an OR node during a run of DF-PN. The numbers are potential values during the search. Gray part of a tree marked as *tt* denotes nodes stored in a transposition table. The left picture shows what happens during the recursive call for the first child. The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

Then control goes back to the parent level and we call DF-PN for the 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 again switch to its sibling. Unfortunately, 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 $1 + \varepsilon$ 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 the very big 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 loosen that constraint a little 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 new formula for the child's PN threshold in an OR node is

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

That way 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 at least $1+\varepsilon$ times. Therefore 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 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. 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 stays 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 simple heuristic to decide whether the node is proof-like or disproof-like. We refer the reader to [3] for the details.

The similar weakness as in DF-PN, described in 3.1, hurts PDS. 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

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

4.1 Experiment Environment

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

Our implementation of four search methods in the Atari Go and LOA games doesn't have any game-specific enhancements. For a transposition table, `TwoBig` scheme [9] is used.

For accelerated version of DF-PN with the $1+\varepsilon$ trick, ε was set empirically to $1/4$. With bigger values of ε, enhanced DF-PN tends to significantly over-explore 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, what can often lead 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.

tt size in nodes	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

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

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

Setting the size greater than 2^{20} doesn't 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 PDS is unable to solve the problem for transposition table with size of 2^{14} nodes, DF-PN with size of 2^{13} nodes and enhanced PDS with size of 2^{12} nodes.

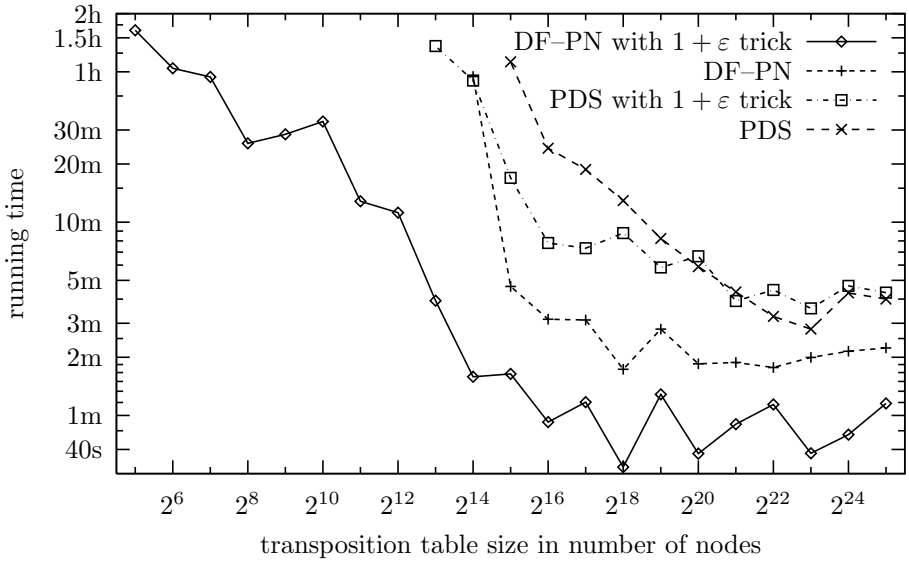

Fig. 5. Solving times of Atari Go 6×6 with a crosscut

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 section we check a practical value of the methods on the set of [8] 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 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.

time limit	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

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

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. 286 test positions were taken from [8]. Again we set the size of a transposition table to 2^{20} nodes. The results are shown in Fig. 7 and exact number of solved positions for selected time limits are shown in Table 3.

time limit	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

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

Again, as in the previous test, 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 advantage of enhanced PDS over plain PDS for harder positions.

To illustrate speed differences in numbers for each two methods we calculated a geometric mean of ratios of solving times (Table 4). The geometric 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 big advantage of enhanced DF-PN over the other methods. The AtariGo

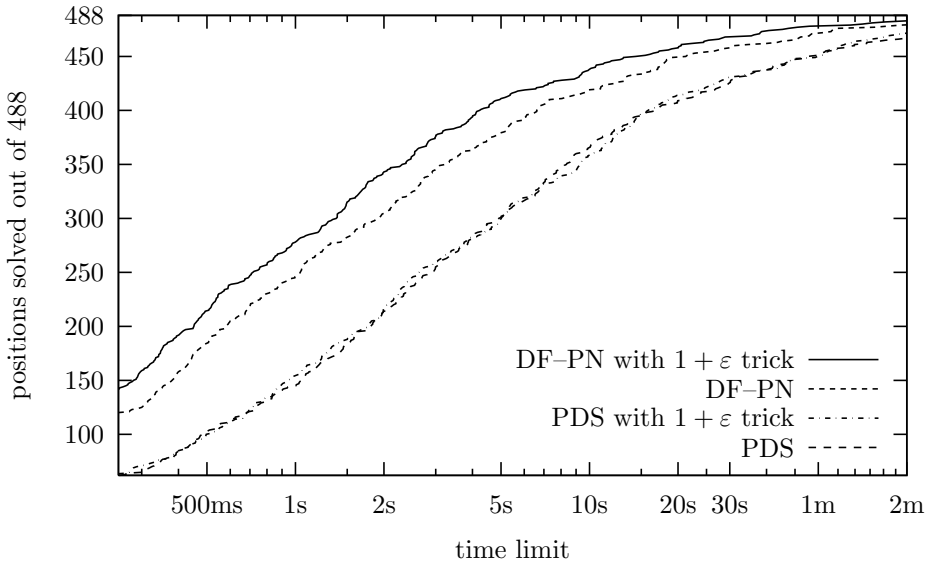

Fig. 6. Numbers of solved easy LOA positions from the set `tscg2002a.zip` [8] for given time limit

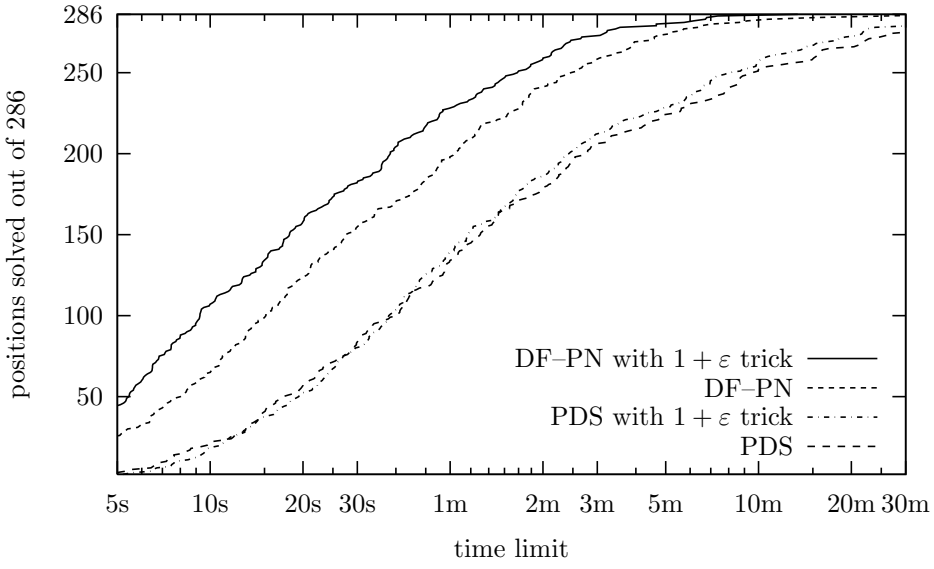

Fig. 7. Numbers of solved hard LOA positions from the hard set `tscg2002b.zip` [8] for given time limit

	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

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.

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 game 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 trick is that it can be added to existing implementations with little effort. Its value has to be confirmed in other games different than 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 technique becomes attractive for a brute-force front search for games solving. 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. Breuker, D., Uiterwijk, J., van der Herik, H.: The PN^2–search algorithm. In van den Herik, H., Monien, B., eds.: Advances in Computer Games. Volume 9., Universiteit Maastricht, Maastricht, The Netherlands (2001) 115–132
2. Seo, M., Iida, H., Uiterwijk, J.: The PN*–search algorithm: Application to tsume-shogi. Artificial Intelligence **129**(1–2) (2001) 253–277
3. Nagai, A.: A new AND/OR tree search algorithm using proof number and disproof number. In: Proceeding of Complex Games Lab Workshop, Tsukuba, ETL (1998) 40–45
4. Nagai, A.: Df–pn Algorithm for Searching AND/OR Trees and its Applications. Ph.d. thesis, The University of Tokyo, Tokyo, Japan (2002)
5. Winands, M., Uiterwijk, J., van den Herik, H.: An effective two-level proof-number search algorithm. Theoretical Computer Science **313**(3) (2004) 511–525
6. Kishimoto, A., Müller, M.: Search versus knowledge for solving life and death problems in go. In: Twentieth National Conference on Artificial Intelligence (AAAI-05). (2005) 1374–1379
7. Allis, L., van der Meulen, M., van den Herik, H.: Proof–number search. Artificial Intelligence **66** (1994) 91–124
8. Winands, M.: (Mark's LOA Homepage) `http://www.cs.unimaas.nl/m.winands/loa/`.
9. Breuker, D., Uiterwijk, J., van der Herik, H.: Replacement schemes and two–level tables. ICCA J. **19**(3) (1996) 175–180