Evaluation-Function Based Proof-Number Search

Mark H.M. Winands and Maarten P.D. Schadd

Games and AI Group, Department of Knowledge Engineering,
Faculty of Humanities and Sciences,
Maastricht University, Maastricht, The Netherlands
{m.winands,maarten.schadd}@maastrichtuniversity.nl

Abstract. This article introduces Evaluation-Function based Proof–Number Search (EF-PN) and its second-level variant EF-PN^2. It is a framework for setting the proof and disproof number of a leaf node with a heuristic evaluation function. Experiments in LOA and Surakarta show that compared to PN and PN^2, which use mobility to initialize the proof and disproof numbers, EF-PN and EF-PN^2 take between 45% to 85% less time for solving positions. Based on these results, we may conclude that EF-PN and EF-PN^2 reduce the search space considerably.

1 Introduction

Most modern game-playing computer programs successfully apply $\alpha\beta$ search with enhancements for online game-playing. 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. Due to memory constraints this is only feasible for endgames with a relatively small state-space complexity. An alternative approach is the use of a specialized binary (win or non-win) search method, such as Proof-Number (PN) search [2]. 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 narrow subtrees over wide ones. PN search or a variant thereof has been successfully applied to the endgames of Awari [1], Chess [3], Shogi [7, 11], Othello [7], LOA [8, 9, 15], Go [5], Checkers [10], and Connect6 [17].

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. Although, adding specific domain-dependent knowledge has been shown to improve the performance of PN search in the past [1, 7, 10, 17]. These evaluation functions have been specially designed for PN search. An alternative is to apply a traditional evaluation function, as used in an $\alpha\beta$ game-playing program. In this article we investigate how we can include such an evaluation function in PN search. We introduce therefore a framework, called *Evaluation-Function based Proof-Number Search* (EF-PN). Since (EF-)PN is a best-first search, it has to store the complete search tree in memory. When the memory is full, the search has to end prematurely. To test the framework for harder problems, we apply it to PN^2 (subsequently called EF-PN^2). In the article, we test EF-PN and EF-PN^2 in (6×6) LOA and Surakarta.

The article is organized as follows. In Sect. 2 we discuss PN and PN^2 search. Next, we propose EF-PN search in Sect. 3. Subsequently, we test EF-PN and its second-level variant EF-PN^2 for the game of LOA and Surakarta in Sect. 4. Finally, Sect. 5 gives conclusions and an outlook on future research.

2 Proof-Number Search

In this section, we give a short description of PN search (Subsect. 2.1) and PN^2 search (Subsect. 2.2).

2.1 PN Search

Proof-Number (PN) search is a best-first search algorithm especially suited for finding the game-theoretic value in game trees [1]. Its aim is to prove the correct 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. As long as the value of the root is unknown, the most-promising node is expanded. In contrast to other best-first algorithms PN search does not need a domain-dependent heuristic evaluation function to determine the most-promising node [2]. In PN search this node is usually called the *most-proving* node. PN search selects the most-proving node using two criteria: (1) the shape of the search tree (the branching factor 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.

Below we explain PN search on the basis of the AND/OR tree depicted in Fig. 1, 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* (*pn*) represents the minimum number of leaf nodes which have to be proved in order to prove the node. Analogously, a *disproof number* (*dpn*) represents the minimum number of leaf nodes that 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. So, they have $pn = 0$ and $dpn = \infty$ (e.g., node i). Lost or drawn nodes are regarded as disproved (e.g., nodes f and k). They have $pn = \infty$ and $dpn = 0$. Unknown leaf nodes have $pn = 1$ and $dpn = 1$ (e.g., nodes g, h, j, and l). The pn of an internal OR node is equal to the minimum of its children's proof numbers, since to prove an OR node it suffices to prove one child. The dpn of an internal OR node is equal to the sum of its children's disproof numbers, since to disprove an OR node all the children have to be disproved. The pn of an internal AND node is equal to the sum of its children's proof numbers, since to prove an AND node all the children have to be proved. The dpn of an AND node is equal to the minimum

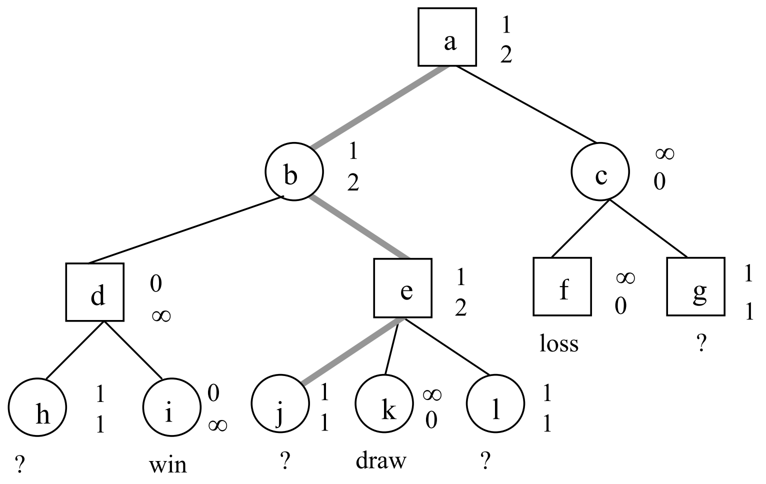

Fig. 1. An AND/OR tree with proof and disproof numbers

of its children's disproof numbers, since to disprove an AND node it suffices to disprove one child. The procedure of selecting the most-proving node to expand next is as follows. The algorithm starts at the root. Then, at each OR node the child with the smallest *pn* is selected as successor, and at each AND node the child with the smallest *dpn* is selected as successor. Finally, when a leaf node is reached, it is expanded (which makes the leaf node an internal node) and the newborn children are evaluated. This is called *immediate evaluation*. The selection of the most-proving node (j) in Fig. 1 is given by the bold path.

2.2 PN^2 Search

A disadvantage of PN search is that the complete 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 [1]. To overcome the memory problem of PN search, depth-first variants such as PN* [11], PDS [6], and df-pn [7] were proposed. They use multiple-iterative deepening to transform the best-first search into a depth-first search. However, due to the overhead of the re-searches, these depth-first variants can only function properly by using a transposition table and garbage collection [6, 7]. An alternative to these depth-first variants is PN^2 [1, 4]. It is a straightforward extension of PN search. PN^2 performs two levels of PN search, one at the root and one at the leaves of the first level. Although PN^2 uses far less memory than PN search [1], it does not fully overcome the memory obstacle. For the test positions used in this paper, PN^2 does not run out of memory.

As mentioned before, PN^2 consists of two levels of PN search. The first level consists of a regular 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.

In our implementation, the number of nodes y in a PN_2-search tree is restricted to the size of the PN_1 tree and the number of nodes which can still be stored in memory. The formula to compute y is:

$$y = min(x, M - x) \ , \tag{1}$$

with x being the size of the first-level search and M the maximum number of nodes to be stored in memory. The PN_2 search is stopped when its 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 their 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; evaluation of such a child node happens only after its selection as most-proving node. This is called *delayed evaluation*. We remark that for PN_2-search trees immediate evaluation is used.

As we have pointed out in Subsect. 2.1, 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. In our implementation, we do delete the subtrees in the PN_2 search. This gives the PN^2 search a speedup of 10% [12].

3 Initializing Proof and Disproof Numbers

In the previous section we saw that *pn* and *dpn* are each initialized to unity in the unknown leaves. However, this approach is rather naive. There are better ways of initializing the *pn* and *dpn*, such that the performance of PN search is increased. In Subsect. 3.1 we discuss initialization techniques proposed in the past. Next, we introduce our evaluation-function based approach in Subsect. 3.2.

3.1 Alternative Initialization Techniques

Several techniques for initializing the *pn* and *dpn* were introduced in the past. One of the most effective techniques, proposed by Allis [1], is taking the *branching factor*, i.e., the *mobility* of the moving player in the leaf into account. The idea is that mobility an important characteristic for most games (e.g., Give-Away-Chess [1], Chess [3], LOA [12]). For an OR node, the *pn* and *dpn* are set to 1 and n (and the reverse for an AND node), where n is the number of legal moves. The advantage of this technique is that it does not need domain knowledge to be effective. In games such as Give-Away-Chess or LOA an improvement of a factor 6 has been reported [1, 12]. However, when computing the number of moves is relatively expensive or mobility is not important for a game, it may not lead to any improvement.

An alternative to mobility is to use a domain-specific heuristic evaluation function. For instance, Allis [1] took material into account to set the variables in Awari. Nagai [7] applied a pattern-based approach in his Othello df-pn program. In Tsume-Go, Kishimoto and Müller [5] initialized the *pn* and *dn* for the defending player by approximating the minimum number of successive defender moves to create two eyes. For the attacking player the *pn* and *dn* were set to the estimated number of moves to create a dead shape. These approaches have generally in common that instead of applying a complicated and expensive positional evaluation function as used by an $\alpha\beta$ program, they apply a simpler (and computationally inexpensive) evaluation function that is more suitable for PN search. These evaluation functions have a smaller number of features than their $\alpha\beta$ counterpart. However, finding the right set of features may be complicated (e.g., as seen in Checkers [10]). Moreover, for some games a large set of features has to be considered, making the evaluation function more difficult to construct (e.g., LOA [12]). An option is then to use a traditional positional evaluation function – possibly available in an existing $\alpha\beta$ program – instead.

3.2 Evaluation-Function Based Proof-Number Search

As discussed before, an option is to apply a traditional positional evaluation function to set *pn* and *dn*. Applying such an evaluation function evaluation raises three issues. (1) They take too much time. The reduction of nodes per second outweighs the benefit of a smaller search tree. (2) They violate the assumption that the *pn* and *dpn* are lower bounds on the effort required to solve the tree [1]. The positive influence of different initializations may at the same time result in negative effects. Allis [1] found that for some games (e.g., Othello) it is necessary to perform a large number of experiments to fine-tune the initialization process. (3) They may ignore the benefit of taking the mobility into account. A general purpose evaluation function may not give enough weight to mobility and therefore be outperformed by simply counting the number of moves.

To tackle these three issues we propose *Evaluation-Function based Proof-Number Search* (EF-PN). Regarding (1), we only evaluate a node when it is expanded. We use its evaluation score to initialize its newborn children. Regarding (2), we squash the evaluation score by using a *step function*. This reduces the risk of overestimating the *pn* and *dpn*. This idea is similar to Nagai's [7], who used a *sigmoid function* to squash the evaluation scores in Othello. During the course of our research, we did not observe much difference between the performance of the two functions. Regarding (3), we multiply the evaluation score with the number of moves n available in the leaf node. For an OR leaf node, this multiplication is used to set its *dpn*, while for an AND leaf node, this multiplication is used to set its *pn*.

The initialization rule for an OR node i is now as follows:

$$pn_i = 1 + a \cdot (1 - step(eval(p))), \tag{2}$$

$$dpn_i = n_i + n_i \cdot b \cdot (1 + step(eval(p))), \tag{3}$$

where a and b are two parameters to be tuned, p the parent node, *eval()* the evaluation function, and *step()* the step function. The evaluation score is positive when the OR player is ahead, and negative when the AND player is ahead. Initialization for an AND node i is done in a similar way:

$$pn_i = n_i + n_i \cdot b \cdot (1 - \mathit{step}(\mathit{eval}(p))), \tag{4}$$

$$dpn_i = 1 + a \cdot (1 + \mathit{step}(\mathit{eval}(p))). \tag{5}$$

Finally, the step function is as follows:

$$\mathit{step}(x) = \begin{cases} -1, & x \leq -t \\ 0, & -t < x < t \\ 1, & x \geq t \end{cases} \tag{6}$$

where t is a parameter that indicates a decisive winning advantage (e.g., a Rook ahead in Chess).

The behavior of the above framework is as follows. In an OR node, when the OR player has a substantial advantage the pn is 1 and the dpn is $(2b + 1) \cdot n$. When the AND player has a substantial advantage the pn is $2a + 1$ and the dpn is n. Otherwise the pn is $a + 1$ and the dpn is $(b + 1) \cdot n$. In an AND node, when the OR player has a substantial advantage the pn is n and its dpn is $2a + 1$. When the AND player has a substantial advantage the pn is $(2b + 1) \cdot n$ and the dpn is 1. Otherwise the pn is $(b + 1) \cdot n$ and the dpn is $a + 1$.

4 Experiments

In this section we evaluate the performance of EF-PN and its second-level variant EF-PN^2. First, we describe the test environment in Subsect. 4.1. Next, the EF-PN parameters a and b are tuned in Subsect. 4.2. Then, in Subsect. 4.3 PN, PN^2, EF-PN and EF-PN^2 are compared with each other in the game of LOA. Subsequently, Subsect. 4.4 matches PN^2 and EF-PN^2 against each other to solve 6×6 LOA. Finally, we verify the results of EF-PN^2 in the game of Surakarta in Subsect. 4.5.

4.1 Test Environment

All experiments were performed on an AMD Opteron 2.2 GHz computer. The algorithms were implemented in Java. Moreover, PN and PN^2 used mobility to initialize their leaf nodes. As test domains we used the games of LOA and Surakarta, which we explain below. The evaluation functions for LOA and Surakarta were taken from the programs MIA [16] and SIA [13], respectively. To speed up computation, opponent-independent evaluation-function features were cached [16].

LOA. Lines of Action (LOA) is a two-person zero-sum 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

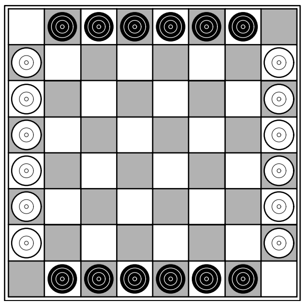
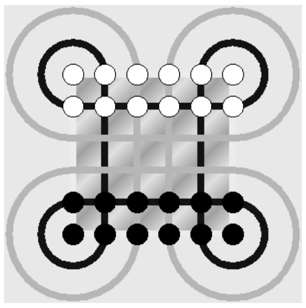

Fig. 2. (a) Initial LOA position. (b) Initial Surakarta position.

along the top and bottom rows of the board, while the white pieces are placed in the left- and right-most files of the board (see Fig. 2a). The players alternately move a piece, starting with Black. A piece moves in a straight line, exactly as many squares as there are pieces of either color anywhere along the line of movement. A player may jump over its own pieces, but not the opponent's, although opponent's pieces are captured by landing on them. The goal of the players is to be the first to create a configuration on the board in which all own pieces are connected in one unit. The connections within the unit may be either orthogonal or diagonal.

Surakarta. The game of Surakarta is an Indonesian board game that traditionally is played by using stones vs. shells, though other easily-distinguished sets of pieces may be used (e.g., Black and White in Fig. 2b). Players take turns moving one of their own pieces. In non-capturing moves, a piece travels - either orthogonally or diagonally - to a neighboring intersection. In a capturing move, a piece travels along a line, *traveling over at least one loop*, until it meets one of the opponent pieces. The captured piece is removed, and the capturing piece takes its place. The first player to capture all opponent's pieces wins.

4.2 Parameter Tuning

In the following series of experiments we tried to find the parameter setting (a and b) of EF-PN, which gave the largest node reduction. A set of 333 LOA positions was used that every parameter setting could solve within the limit of 5,000,000 nodes searched. This set of positions is a subset of 488 LOA endgame positions,[1] which has been used frequently in the past [8, 9, 15]. Parameter a took the values of 0, 5, 10, 15, 20, 50, and 100, whereas parameter b took the values of 0, 5, 10, 20, 50, and 100. The t value was fixed to 500, which indicates a considerable winning advantage in the MIA 4.5 evaluation function. In Fig. 3 the total number of nodes searched for each pair of parameters is given. The default PN result is given for comparison (cf. Def.).

[1] The test set of 488 LOA positions can be found at: http://www.personeel.unimaas.nl/m-winands/loa/tscg2002a.zip.

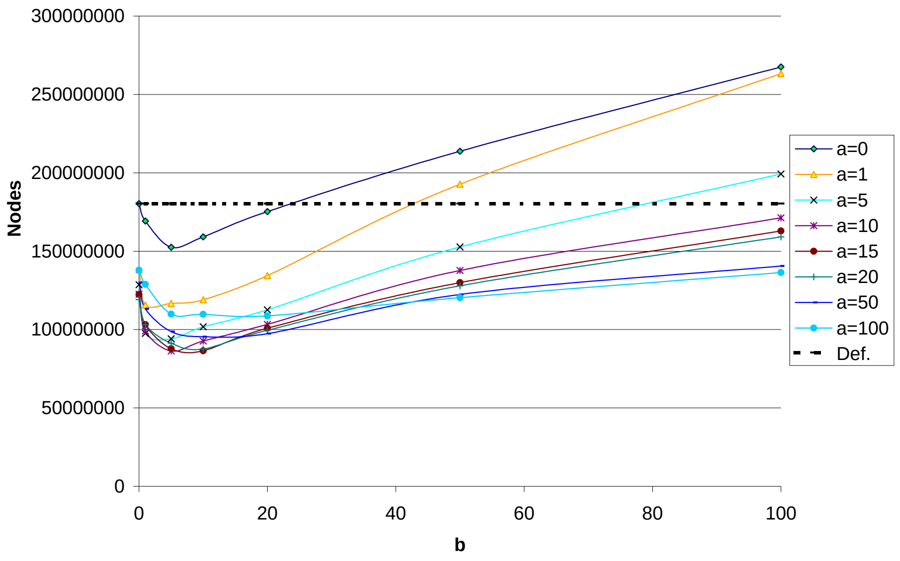

Fig. 3. Tree sizes for different a and b.

In Fig. 3 we see that parameter configurations (5, 5), (10, 5), (10, 10), (15, 5), (15, 20), (20, 5), and (20, 10) lead to a reduction of approximately 50% in nodes searched. Moreover, setting either a or b to 0 gives a reasonable reduction in nodes, but this reduction is considerably smaller than for the optimal parameter configurations. Finally, we remark that if the parameter a is set too small in comparison with b, e.g., (0, 50), (0, 100), (1, 50), (1, 100), or (5, 100), EF-PN performs even worse than the regular PN.

4.3 Experiments in LOA

In this subsection we ran additional experiments in LOA to get a better insight in the performance of EF-PN. We not only compared EF-PN with PN, but also its two level-variant EF-PN^2 with PN^2. For each algorithm, the maximum number of nodes stored in memory was 10,000,000 and the maximum number of nodes searched was 50,000,000. The EF-PN parameters a, b, and t were set to 20, 5, and 500, respectively.

In the first experiment of this subsection we compared EF-PN with PN-Naive (initializing pn and dn with 1), PN, and PN-Eval (applying an evaluation function in the leaf to set pn and dn). For PN-Eval, a step function with $a = 20$ is used in the following way: $pn = 1 + a \cdot (1 - step(eval(n))$ and $dn = 1 + a \cdot (1 + step(eval(n)))$. We used the same set of 488 LOA endgame positions as in Subsect. 4.2. The results are given in Table 1. In the second column we see that 367 positions are solved by PN-Naive, 436 PN-Eval, 461 by PN, and 478 by EF-PN. In the third and fourth column the number of nodes and the time consumed are given for the subset of 367 positions, which the algorithms are able to solve. We observe that the performance of PN-Eval compared to PN and EF-PN is disappointing. PN-Eval only explores 60% fewer nodes and consumes 40% less

Table 1. Comparing the search algorithms on 488 test positions

Algorithm	# of positions solved	367 positions	
	(out of 488)	Total nodes	Total time (ms.)
PN-Naive	367	1,336,925,805	1,195,800
PN-Eval	436	540,954,044	737,503
PN	461	162,286,551	198,092
EF-PN	478	97,880,765	125,254

time than PN-Naive. On the contrary, PN and EF-PN solve more problems, and they explore considerably smaller trees than PN-Naive and PN-Eval. For the best variant, EF-PN, positions were solved thirteen times faster in nodes and almost ten times faster in CPU time than PN-Naive. Moreover, EF-PN used 40% fewer nodes and 35% less time than PN.

In the second experiment we tested the immediate variant of EF-PN, called IEF-PN. Instead of using the parent's evaluation score, this variant uses the evaluation score of the leaf node. In Table 2 we compare EF-PN with IEF-PN on a subset of 478 test positions, which both algorithms were able to solve. The table shows that IEF-PN searches the least number of nodes. However, EF-PN is between 15% to 20% faster in time than IEF-PN. Based on the results of Tables 1 and 2 we may conclude that EF-PN outperforms IEF-PN and PN in LOA.

In the third experiment we tested the performance of EF-PN's second-level variant EF-PN^2. Table 3 gives the results of comparing PN, EF-PN, PN^2, and EF-PN^2 with each other. The table shows that 461 positions are solved by PN, 478 by PN^2, 476 by EF-PN, and 482 by EF-PN^2. The reason that PN and EF-PN solved less than their second-level variants is that they sometimes ran out of memory. In the third and fourth column the number of nodes and the time consumed are given for the subset of 459 positions, which all algorithms are able to solve. The results suggest that the evaluation-function approach is more beneficial for PN than for PN^2. EF-PN solves positions in 55% less time than PN, whereas EF-PN^2 solves positions in 45% less time than PN^2. The difference in performance can be explained by the fact that the parameters were originally tuned for EF-PN. Finally, we remark that the overhead in CPU time of PN^2 and EF-PN^2 compared to PN and EF-PN is 1.5 and 1.9, respectively.

PN^2 and EF-PN^2 are especially designed for harder problems, which PN or EF-PN cannot solve due to memory constraints. Since PN or EF-PN were able to solve most of the problems in the previous experiment, this set was not really appropriate for comparing PN^2 and EF-PN^2 with each other. We therefore performed a fourth experiment with a different set of LOA problems in an attempt to find more insights into the intricacies of these complex algorithms. In the

Table 2. Comparing EF-PN on 478 test positions

Algorithm	Total nodes	Total time (ms.)
IEF-PN	328,748,918	655,512
EF-PN	406,941,802	542,564

Table 3. Comparing (EF-)PN and (EF-)PN^2 on 488 test positions

Algorithm	# of positions solved (out of 488)	459 positions Total nodes	459 positions Total time (ms.)
PN	461	599,272,821	748,219
EF-PN	478	256,151,258	328,008
PN^2	476	1,069,663,432	1,124,973
EF-PN^2	482	546,398,711	636,154

Table 4. Comparing PN^2 and EF-PN^2 on 286 test positions

Algorithm	# of positions solved (out of 286)	282 positions Total nodes	282 positions Total time (ms.)
PN^2	282	15,342,372,938	16,510,118
EF-PN^2	286	7,171,634,916	8,604,965

fourth experiment PN^2 and EF-PN^2 are tested on a set of 286 LOA positions, which were on average harder than the ones in the previous test set.[2] In this context 'harder' means a longer distance to the final position (the solution), i.e., more time is needed. The conditions are the same as in the previous experiments except that the maximum number of nodes searched is set at 500,000,000. In Table 4 we see that PN^2 solves 282 positions, and EF-PN^2 solves all 286 positions. The ratio in nodes and time between PN^2 and EF-PN^2 for the positions solved by both (282) is roughly similar to the previous experiment. Based on these results, we may draw the conclusion that EF-PN^2 is a better solver than PN^2.

4.4 Solving 6×6 LOA

Because the mobility of an opponent is increased when playing a non-forcing move, PN search prefers to investigate the lines that confine the opponent the most. At the moment PN search has to solve a position where the solution requires mostly non-forcing moves, it does not perform well [3]. For PN search every move seems the same then. However, adding heuristic knowledge may guide PN search through these difficult situations. In the next experiment, PN^2 and EF-PN^2 were used to compute the game-theoretic value of 6×6 LOA. Especially in the beginning of this game, there are hardly any forcing moves. We remark that this game was already solved by PN^2 in 2008 [14].

For this experiment, symmetry was taken into account to prevent redundancy. The parameter setting of EF-PN^2 was the same as the previous subsection. The results are given in Table 5. The table reveals that applying EF-PN^2 gives a reduction of 86% in nodes searched and a reduction of 85% in CPU time. Compared to the results in the previous subsection, this improvement is quite drastic and may be explained in the fact that EF-PN^2 is better in tackling situations with non-forcing moves.

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

Table 5. Solving 6 × 6 LOA

Algorithm	Total nodes	Total time (ms.)	Outcome
PN^2	220,375,986,787	350,194,664	Black Wins (b6-b4)
$EF\text{-}PN^2$	31,345,495,339	54,584,203	Black Wins (b6-b4)

4.5 Experiments in Surakarta

To investigate the performance of the framework in another domain than LOA, we compared PN^2 and $EF\text{-}PN^2$ with each other in the game of Surakarta. The maximum number of nodes searched was set to 1,000,000,000. The t value was set to 2,000, which indicates a two-stones winning advantage in SIA. For the remainder, the setup was the same as in the previous experiments. The test set consisted of 30 positions.[3] In Table 6 we see that PN^2 solves 7 positions and $EF\text{-}PN^2$ 13 positions. For the 7 positions both algorithms could solve, $EF\text{-}PN^2$ used almost 85% less CPU time than PN^2. Although the number of 7 positions may appear small, the total size in nodes was comparable to the subset used in Table 3. We may therefore draw the conclusion that $EF\text{-}PN^2$ is a better solver than PN^2 in Surakarta as well. Admittedly, a larger set is needed to determine the exact speedup.

Table 6. Comparing EF-PN and $EF\text{-}PN^2$ in Surakarta

Algorithm	# of positions solved	7 positions	
	(out of 30)	Total nodes	Total time (ms.)
PN^2	7	1,469,130,908	1,481,190
$EF\text{-}PN^2$	13	239,265,507	226,610

5 Conclusion and Future Research

In this article we proposed *Evaluation-Function based Proof-Number Search* (EF-PN) and its second-level variant $EF\text{-}PN^2$. It is a framework for initializing the proof and disproof number of a leaf node with a heuristic evaluation function. This heuristic evaluation function is only applied when the node is expanded. Its subsequent score is used to initialize the newborn children. To reduce the risk of overestimating the proof and disproof numbers, its evaluation score is squashed by using a step function. To take full advantage of mobility, the evaluation score is multiplied with the number of moves for initializing the disproof number in an OR node and the proof number in an AND node. Experiments in LOA and Surakarta show that EF-PN and $EF\text{-}PN^2$ use between 45% to 85% less time for solving than PN and PN^2. We may conclude that EF-PN and its second-level variant $EF\text{-}PN^2$ reduce the amount of time to solve a position considerably.

There are two directions for future research. First, it would be interesting to test EF-PN and $EF\text{-}PN^2$ in other game domains where mobility is important and evaluation functions are good predictors but computationally expensive. A good

[3] The test set can be found at:
www.personeel.unimaas.nl/m-winands/surakarta/CG2010.zip.

candidate would be Chess, because the endgame is suitable for PN search [3], mobility is important [3] and there is an abundance of good evaluations functions. Second, experiments are envisaged to apply EF-PN in df-pn (especially with the $1 + \epsilon$ trick [8]) on the same hard set of LOA positions.

Acknowledgments. We would like to thank the referees for their comments which helped to improve this paper. This work is funded by the Netherlands Organisation for Scientific Research (NWO) for the project TACTICS, grant number 612.000.525.

References

1. L.V. Allis. *Searching for Solutions in Games and Artificial Intelligence.* PhD thesis, Rijksuniversiteit Limburg, Maastricht, The Netherlands, 1994.
2. L.V. Allis, M. van der Meulen, and H.J. van den Herik. Proof-number search. *Artificial Intelligence*, 66(1):91–123, 1994.
3. D.M. Breuker. *Memory versus Search in Games.* PhD thesis, Universiteit Maastricht, Maastricht, The Netherlands, 1998.
4. 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. Maastricht University, Maastricht, The Netherlands, 2001.
5. A. Kishimoto and M. Müller. Search versus knowledge for solving life and death problems in Go. In M.M. Veloso and S. Kambhampati, editors, *AAAI 2005*, pages 1374–1379. AAAI Press / The MIT Press, 2005.
6. A. Nagai. A new depth-first-search algorithm for AND/OR trees. Master's thesis, The University of Tokyo, Tokyo, Japan, 1999.
7. A. Nagai. *Df-pn Algorithm for Searching AND/OR Trees and its Applications.* PhD thesis, The University of Tokyo, Tokyo, Japan, 2002.
8. J. Pawlewicz and Ł. Lew. Improving depth-first pn-search: $1 + \epsilon$ trick. In H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers, editors, *Computers and Games (CG 2006)*, volume 4630 of *Lecture Notes in Computer Science (LNCS)*, pages 160–171. Springer-Verlag, Heidelberg, Germany, 2007.
9. M. Sakuta, T. Hashimoto, J. Nagashima, J.W.H.M. Uiterwijk, and H. Iida. Application of the killer-tree heuristic and the lamba-search method to Lines of Action. *Information Sciences*, 154(3–4):141–155, 2003.
10. J. Schaeffer. Game over: Black to play and draw in checkers. *ICGA Journal*, 30(4):187–197, 2007.
11. 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.
12. M.H.M. Winands. *Informed Search in Complex Games.* PhD thesis, Universiteit Maastricht, Maastricht, The Netherlands, 2004.
13. M.H.M. Winands. SIA wins Surakarta tournament. *ICGA Journal*, 30(3):162, 2007.
14. M.H.M. Winands. 6×6 LOA is Solved. *ICGA Journal*, 31(3):234–238, 2008.
15. M.H.M. Winands, J.W.H.M. Uiterwijk, and H.J. van den Herik. An effective two-level proof-number search algorithm. *Theoretical Computer Science*, 313(3):511–525, 2004.
16. M.H.M. Winands and H.J. van den Herik. MIA: a world champion LOA program. In *The 11th Game Programming Workshop in Japan 2006*, pages 84–91, 2006.
17. C. Xu, Z.M. Ma, J. Tao, and X. Xu. Enhancements of proof number search in Connect6. In *Control and Decision Conference*, pages 4525–4529. IEEE, 2009.