Reducing the Seesaw Effect with Deep Proof-Number Search

Taichi Ishitobi, Aske Plaat, Hiroyuki Iida, and Jaap van den Herik

Japan Advanced Institute of Science and Technology, Japan
Leiden Institute of Advanced Computer Science, the Netherlands
{itaichi,iida}@jaist.ac.jp
{aske.plaat,jaapvandenherik}@gmail.com

Abstract. In this paper, we propose a new search technique based on proof numbers, named Deep Proof-Number Search (DeepPN). Proof-number search has been introduced by Allis et al. in 1994. Proof-number search uses the difficulty of deciding upon a game-theoretical value in each node as a guide for the search: easy nodes are searched first. Proof-number search is well-suited for solving games, and many games have indeed been solved with it, but some challenging problems remain. One of these problems is the so-called "seesaw effect:" proof-number search alternates between selecting one of two branches in the tree. This effect is frequently encountered in games such as Othello and Hex. The see-saw effect reduces the efficiency of proof-number search.

In this paper, DeepPN is introduced. It is a modified version of PN-search. It introduces a procedure to solve the seesaw effect. DeepPN employs two important values associated with each node, viz. the usual proof number and a *deep value*. The deep value of a node is defined as the depth to which each child node has been searched. So, the deep value of a node shows the progress of the search in the depth direction. By mixing the proof numbers and the deep value, DeepPN works with two characteristics, viz. the best-first manner of search (equal to the original proof-number search) and the depth-first manner. By adjusting a parameter (called R in this paper) we can choose between best-first or depth-first behavior. In our experiments, we tried to find a balance between both manners of searching (see formula (3)). As it turned out, best results were obtained at an R value in between the two extremes of best-first search (original proof number search) and depth-first search.

Our experiments showed better results for DeepPN compared to the original PN-search: a point in between best-first and depth-first performed best. For random Othello and Hex positions, DeepPN works almost twice as good as PN-search. From the results, we may conclude that by the application of formula (3), Deep Proof-Number Search outperforms PN-search considerably in Othello and Hex.

Keywords: Proof-Number Search, Seesaw Effect, Deep Proof-Number Search, Othello, Hex

1 Introduction

Proof Number Search (PN-search) was developed by Allis et al. in 1994 [16]. It is one of the most powerful algorithms for solving games and complex endgame positions. PN-search focuses on an AND/OR tree and tries to establish the game-theoretical value in an efficient way, in a greedy least-work-first manner. Each node has a proof number (pn) and disproof number (dn). This idea was inspired by McAllister's concept of conspiracy numbers, the number of children that need to change their value for a node to change its value [10]. A proof number shows the scale of difficulty in proving a node, a disproof number does analogously for disproving a node. PN-search tries to expand a most-proving node, which is the most efficient one for proving (disproving) a node. PN-search is a best-first search in the sense that it follows a least-work-first order.

The success of PN-search prompted researchers to create many derivatives of PN-search, e.g., PN* [17], PDS [6] and df-pn [7]. The history of these algorithms are described by Kishimoto et al. [2]. Using one of these algorithms, many games and puzzles were solved, e.g., Checkers [13] [14], and Tsume-shogi (the mating problem of Japanese chess) [17]. The algorithms used many well-known techniques such as transposition tables. On the one hand researchers sought to benefit from large computer power and memory [15], and on the other hand, researchers worked at some of the problems of PN-search, such as the seesaw effect (see section 2).

In this paper, we propose a new proof-number algorithm called Deep Proof-Number Search (DeepPN). DeepPN tries to solve the seesaw problem with a different approach together with iterative deepening. In the experimental section, we use endgame positions of Othello and the game of Hex as benchmarks, and try to measure the performance of DeepPN.

The remainder of this paper is as follows. We briefly summarize the details of the seesaw effect in section 2. The algorithm of DeepPN and its performance, are discussed in section 3. In section 4, we give our conclusion and suggestions for future work.

2 The Seesaw Effect

The derivatives of PN-search address many issues of the algorithm, and have been used to solve many games. However, there are still some problems with PN-search that remain. Pawlewicz and Lew [12], and Kishimoto et al. [3] [4] showed one such weak point, of df-pn. The weak point has been named the Seesaw-effect by Hashimoto [11]. Below we provide a short sketch.

To explain the seesaw effect, we show an example in figure 1. In figure 1a, a root node has two large subtrees. The size of both subtrees is almost the same. Further assume that the proof number of subtree L is larger than the proof number of subtree R. In this case, PN-search will focus on subtree R, will continue the searching, and will expand the most-proving node. When PN-search expands a most-proving node, the shape of a game-tree changes as shown

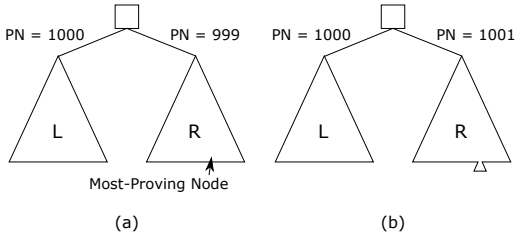


Fig. 1: An example of the Seesaw effect: (a) An example game tree (b) Expanding the most-proving node

in figure 1b. By expanding the most-proving node, the proof number of subtree R becomes larger than the proof number of subtree L. Because of this, the position of the most-proving tree changes from subtree R to subtree L. Similarly, when the search expands the most-proving node in subtree L, the proof number of subtree L changes to a larger value than the proof number of subtree R. Thus, the search switches its focus from subtree L to subtree R. This changing continues to go back and forth and looks like a seesaw. Therefore, it is named the Seesaw effect. The Seesaw effect happens when the two trees are almost equal in size.

If the Seesaw effect occurs, the performance of PN-search and df-pn deteriorates significantly [7]. Df-pn tries to search efficiently by staying around a most-proving node as in figure 1a. However, when the seesaw effect occurs, df-pn should go back to the root node, and switch focus to another subtree and start to find a new most-proving node existing in subtree L. If the seesaw effect occurs frequently, the performance of df-pn becomes close to that of PN-search, because df-pn loses the power of its depth-first behavior.

For PN-search, the algorithm uses the proof numbers to search efficiently in a best-first manner. If the seesaw effect occurs frequently, PN-search will concentrate alternatively on one subtree. PN-search will then expand subtrees L and R equally and it cannot reach the required depth. In games which need to reach a large fixed depth for solving, this effect works strongly against efficiency.

The causes of the seesaw effect are mostly (1) the shape of the game tree and (2) the way of searching. Concerning the shape of game tree, there are two characteristics: (1a) a tendency for the number of child nodes to become equal and (1b) many nodes with equal values exist deep down in a game tree. In (1a), if the number of a child node in each node becomes almost the same, then the seesaw effect may occur easily. For (1b), this is the case in games such as Othello and Hex. In many cases, these games need to search a large fixed number of moves before settling, and it is difficult to assess upon a win, loss, or draw before a certain number of moves has been played. In the game tree of these games, the nodes can establish their value after a certain depth has been searched. Thus, when the seesaw effect occurs and the search cannot reach the required depth, it cannot determine the status of the subtrees. Instead of seesawing between subtrees, the search should stick with one subtree and search more deeply. A game tree that has these characteristics is called a *suitable tree* by

Hashimoto. Games such as Othello, Hex and Go are able to build up a suitable tree easily. For (2), the way of searching, i.e., the best-first manner, causes the seesaw effect. The most-proving node of PN-search and df-pn is determined using proof numbers. Thus, in the figure 1, df-pn has to go back to the root node again and again, and PN-search and df-pn cannot reach a required depth in the subtree.

One solution for the seesaw effect is the "$1 + \epsilon$ trick" proposed by Pawlewicz and Lew [12]. They focused on df-pn and changed the term for calculating the threshold. To paraphrase their explanation, they add a margin determined by ϵ to the thresholds. This margin is calculated by the size of other subtrees, and it is recalculated in each seesaw. By the added margin for the thresholds, df-pn can reach node in a specific branch more deeply than the original algorithm. Hence, the frequency of the seesaw effect is reduced. Consequently, df-pn with the $1 + \epsilon$ trick works better than the original df-pn. However, we expected that this trick has at least three problems. First, the trick breaks a rule about the most-proving node. The original thresholds keep the definition of most-proving node, but $1 + \epsilon$ just adds a margin to the thresholds. Second, if the game tree changes become too large, then also the margin becomes too large, because the margin is calculated by the size of the other subtree. On the one hand, a large margin can reduce the frequency of the seesaw effects, on the other hand, if a subtree to be searched is found not to lead to any result, then the search cannot change the subtree until reaching that margin. Third, the $1 + \epsilon$ trick only reduces the frequency of the seesaw effect and does not completely solve the problem.

3 DeepPN

In this section, we explain a new algorithm based on proof numbers named Deep Proof-Number Search (DeepPN). DeepPN is modeled after the original PN-search, and all nodes have proof numbers and dis-proof numbers. Additionally, for DeepPN, each node is assigned also a so-called *deep value*. The deep values are determined and updated by the terminal node analogously to the proof and disproof numbers. DeepPN has been designed to: (1) combine best-first and depth-first search, and (2) to try and solve the problem of the Seesaw effect. For evaluating the performance of DeepPN, we use endgame positions of Othello and Hex.

3.1 The Basic Idea of DeepPN

In the original PN-search, the most-proving node is defined as follows [16].

Definition For any AND/OR tree T, a most-proving node of T is a frontier node of T, which by obtaining the value true reduces T's proof number by 1, while by obtaining the value false reduces T's disproof number by 1.

This definition implies that the most-proving node sometimes exists in a plural form in a tree, i.e., there are many fully equivalent most-proving nodes.

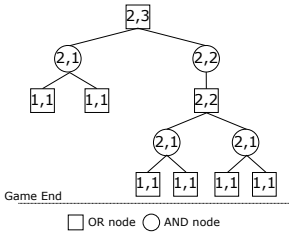


Fig. 2: An example of a suitable tree for an Othello end-game position. This game tree has a uniform depth of 4, and the terminal nodes are reached at game end.

For example, if the child nodes have the same proof or disproof number then both subtrees have each a most-proving node. The situation that the child nodes has the same proof (disproof) number in an OR (AND) node is called a tie-break situation. Now, we have the question about which most-proving node is the best for calculating the game-theoretical value. PN-search chooses the leftmost node with the smallest proof (disproof) number, also in a tie-break situation. In particular, the proof and disproof number do not take other information into account, and therefore PN-search cannot choose a more favorable most-proving node in a tie-break situation.

Determining the best most-proving node in a tie-break situation is a difficult task, because the answer depends on many aspects of the game. However, when focusing on games which build up a suitable tree, we may develop some solutions. In a suitable tree, the "best" most-proving node is indicated by its depth number. Let us look at the example (given in figure 2).

This game tree is based on Othello. The game end is shown by "Game End" in figure 2. All level-two nodes are most-proving nodes, because the proof numbers of child nodes under the root node are the same (i.e., 2). So, we have a tie-break situation. Now, in the next search step, PN-search will focus on the most-proving node that exists in left side as produced by the original PN-search algorithm. However, if the search focuses immediately on the most-proving node of the right side, then the search will be more efficient, because the nodes on the left side do not reach the game end and their value cannot be found yet. In contrast, nodes that exist at the right side reach the game end, and if we try to expand these nodes, then the game value of each node is known. In this example, we follow the idea that a most-proving node in the deepest tree of a suitable game tree, is the best.

To test this idea, we performed a small experiment. We prepared an original PN-search and a modified PN-search. In a tie-break situation, PN-search focuses on a most-proving node that exists in the leftmost node, and the modified PN-search focuses on the *deepest* most-proving node. For checking performance, we prepared 100 Othello endgame positions. The performance of the modified PN-search is better than the results of the original PN-search (about 10% reduction).

These results suggest that the *deepest* most-proving node works advantageously for finding the game-theoretical value.

In addition, the example of figure 2 shows the essence of the seesaw effect. If the game end exists and has a depth of more than 4, then the search for a proof number goes back and forth between the two subtrees. Even if the game end is of depth 4, then the search that focuses on the right subtree will change its focus on the left subtree. But, when modifying PN-search, the small seesaw effect is suppressed. This phenomenon of modifying PN-search suggests a new heuristic. The search depth of nodes can be used for solving the seesaw effect in a suitable game tree. In fact, this is what $1 + \epsilon$ trick [12] in effect tries to accomplish, to stay deep in a suitable game tree. Now, let us try to think of a new technique. For instance, consider the moves that the modified PN-search plays when finding the deepest most proving node. We noticed that these moves combined best-first with depth-first behavior. The modified PN-search works in a best-first manner, and in a tie-break situation, PN-search work depth-first for the most-proving nodes. Depending on how often tie-breaks occur, the algorithm works more frequently best-first than depth-first. The resulting improvement, when measured in number of iterations and nodes leads to a small result. Thus, we will design a new algorithm that can change the ratio of best-first manner and depth-first manner. Its description is an follows. This system is named Deep Proof-Number Search (DeepPN). Here, $n.\phi$ means proof number in OR node and disproof number in AND node. In contrast, $n.\delta$ means proof number in AND node and disproof number in OR node.

1. The proof number and disproof number of node n are now calculated as follows.

$$n.\phi = \begin{cases} n.pn \text{ (n is an OR node)} \\ n.dn \text{ (n is an AND node)} \end{cases}$$

$$n.\delta = \begin{cases} n.dn \text{ (n is an OR node)} \\ n.pn \text{ (n is an AND node)} \end{cases}$$

2. When n is a terminal node
   (a) When n is proved (disproved) and n is an OR (AND) node, i.e., OR wins

$$n.\phi = 0,\ n.\delta = \infty$$

   (b) When n is disproved (proved) and n is an AND (OR) node, i.e., OR does not win

$$n.\phi = \infty,\ n.\delta = 0$$

   (c) When n is unsolved, i.e., its value is unknown

$$n.\phi = 1,\ n.\delta = 1$$

(d) When n is terminal node, then n has deep value

$$n.deep = \frac{1}{n.depth} \tag{1}$$

3. When n is an internal node
   (a) The proof and disproof number are defined as follows

$$n.\phi = \min_{n_c \in \text{children of n}} n_c.\delta$$

$$n.\delta = \sum_{n_c \in \text{children of n}} n_c.\phi$$

   (b) The deep values, DPN(n) and n.deep are defined as follows.

$$n.deep = n_c.deep \text{ where } n_c = \underset{n_i \in \text{unsolved children}}{\arg\min} DPN(n_i) \tag{2}$$

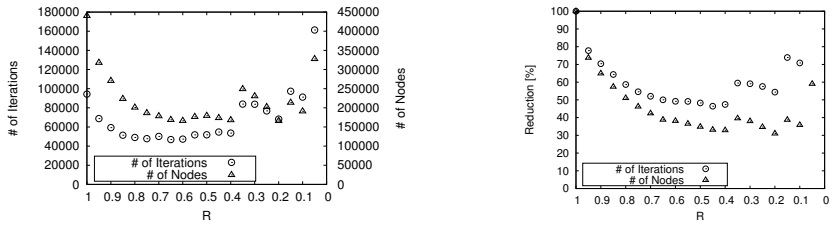
$$DPN(n) = (1 - \frac{1}{n.\delta})R + n.deep(1 - R) \qquad (0.0 \leq R \leq 1.0) \tag{3}$$

The proof and disproof number are the same as in the original PN-search. The improvement is the new term, i.e., the concept of the *deep* value. The deep value in a terminal node is calculated by formula (1). The deep value is designed to decrease inversely with depth. In an internal node, calculating the deep value has only a limited complexity. First, we define a function named DPN (see formula 3). DPN has two features: (a) $n.\delta$ is normalized and designed to become larger according to the growth of $n.\delta$ and (b) a fixed parameter R is chosen. R has a value between 0.0 and 1.0. If R is 1.0 then DeepPN works the same as PN-search, and if R is 0.0 then DeepPN works the same as a primitive depth-first search. Therefore, the normalized δ fulfills the role of best-first guide and the *deep* acts as a depth-first guide. This means that by changing the value of R, the ratio of best-first and depth-first search of DeepPN can be adjusted. Second, in an internal node, the deep value is updated by its child nodes using formula (2). The deep value of node n is decided by a child node n_c which has smallest $DPN(n_c)$. A point to notice is that the updating value is only *deep*, not $DPN(n_c)$. Additionally, when n_c is solved, then the deep value of n_c is ignored in arg min.

In DeepPN, an expanding node in each iteration is chosen as follows.

$$select\_expanding\_node(n) := \underset{n_c \in \text{children of n except solved}}{\arg\min} DPN(n_c) \tag{4}$$

This sequence is repeated until the terminal node is reached. That terminal node is the node that is to be expanded. If $R = 1.0$, then this expanding node is the most-proving node.



(a) The variation (b) The reduction rate

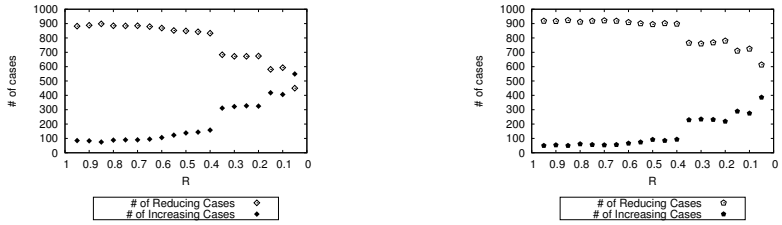
Fig. 3: Othello: The number of # of Iterations and # of Nodes. $R = 1.0$ is PN-search, $R = 0.0$ is depth-first search, and $1.0 > R > 0.0$ is DeepPN. Lower is better.

3.2 Performance with Othello

For measuring the performance of DeepPN, we prepared a solver using the DeepPN algorithm and Othello endgame positions. We configured a primitive DeepPN algorithm for investigating the effect of DeepPN only, without any supportive mechanisms such as transposition tables and ϵ-thresholds. We prepared 1000 Othello endgame positions. They are constructed as follows. The positions are taken from the 8 x 8 board. We play 44 legal moves at random from the begin position. This implies that 48 squares from the 64 are covered. So, the depth of the full tree to the end is 16.

In all our experiments DeepPN is applied to these 1000 endgame positions. Our focus is the behavior of R (see formula (3)). For $R = 1.0$, DeepPN works the same as PN-search and shows the same results. For $R = 0.0$, DeepPN works the same as a primitive depth-first search. When R is between 1.0 to 0.0, then DeepPN behaves as a mix between best-first and depth-first. We changed R from 1.0 to 0.0 by increments of 0.05. We focus on the values of two concepts, viz. the number of iterations and the number of nodes. The number of iterations is given by counting the number of traces of finding the most-proving node from the root node. This value indicates an approximate execution time unaffected by the specifications of a computer. The number of nodes is an indication of the total number of nodes that are expanded by the search. This value is an approximation of the size of memory needed for solving. We show the results in figure 3.

Figure 3a shows the variation of (1) the number of iterations and (2) the number of nodes. Each point is as mean value calculated from the results of 1000 Othello endgame positions. $R = 1.0$ shows the results of PN-search, and this value is the base for comparison. As R goes to 0.8, the number of iterations and nodes decrease almost by half. From $R = 0.8$ to 0.6, the number of iterations stops decreasing, but the number of nodes decreases slowly. From $R = 0.6$ to 0.4, the decrease stops, and the number of iterations starts increasing again slowly. In $R = 0.35$, both numbers increase rapidly. We see that for R of around 0.4,



(a) # of Iterations (b) # of Nodes

Fig. 4: Othello: The changes of Reducing and Increasing Cases for # of Iterations and # of Nodes

the balance between depth-first and best-first behavior appears to be optimal. We surmise that DeepPN is stuck in one subtree and cannot get away since the algorithm is too strongly depth-first. For $R = 0.35$ to 0.2, the number of iterations and nodes is decreasing. Around $R = 0.2$, the balance was broken again, and is decreasing towards 0.1. Finally, DeepPN performs worse when R approaches 0.0 closely. In $R = 0.0$, almost no Othello end game position can be solved, and this value is omitted from figure 3a.

In Figure 3a, the scale of the number of iterations and nodes are different. To ease our understanding, Figure 3b shows the amount of the reduction rate. This reduction rate is normalized by the result of PN-search, i.e., the reduction rate of $R = 1.0$ is 100%. Each point is the mean value of the reduction rate calculated by the results of 1000 Othello endgame positions. The results of 3b show almost the same characteristics as 3a. There is a different point where the number of iterations decreases after $R = 0.8$ and the number of nodes decreases after $R = 0.6$. In Figure 3b, the number of iterations decreased about 50% in $R = 0.4$ and the number of nodes decreased about 35% in $R = 0.4$. Thus, DeepPN reduced the number of iterations ($\approx$ time) to half and the number of nodes ($\approx$ space) to one-third. In $R = 0.05$, the number of iterations increased to over 100%, which is not shown.

Finally, we show two graphs about the changes in *reducing* and *increasing* cases in Othello endgame positions in figure 4. Please note that in figure 4 we showed the number of iterations and number of nodes.

The plots for reducing cases give the number of Othello endgame positions which are solved efficiently compared to PN-search, i.e., the reduction rate is under 100%. In contrast, the plots for increasing cases give the number of Othello endgame positions that have a reduction rate over 100%. The vertical axis shows the number of Othello endgame positions. Figure 4a shows the number of iterations by which the reducing cases decrease slowly from $R = 0.95$ to 0.4. Likewise, for number of nodes the graph decreases slowly from $R = 0.95$ to 0.4. Around $R = 0.4$, the trend is broken, and the number of increasing cases increases rapidly. From $R = 0.35$ to 0.2 and from 0.15 to 0.1, the number of cases

does not change much. This result indicates that the reason of decreasing from $R = 0.35$ to 0.2 is shown in figure 3a and 3b. As the number of cases is not changed, the decreasing number of iterations and nodes of the Othello end game positions are caused by reducing cases. In brief, some Othello end positions can be handled efficiently as R is reduced. But, for some Othello end game positions a changing R causes an increase. Therefore, Othello end game positions can be categorized in relation to R. The first group belongs to $R = 0.95$ to 0.05. This group does not react to changes in R, they do not switch between the reducing case and increasing case. We can see this group clearly from $R = 0.95$ to 0.40. The second group belongs to $R = 0.35$ to 0.2. This group fitted from $R = 0.95$ to 0.4, and they could not keep efficiency work after $R = 0.4$. The third group belongs to $R = 0.15$ to 0.1, and the characteristics of this group are the same as for the second group. In either group, the cases are not efficiently close to $R = 0.0$.

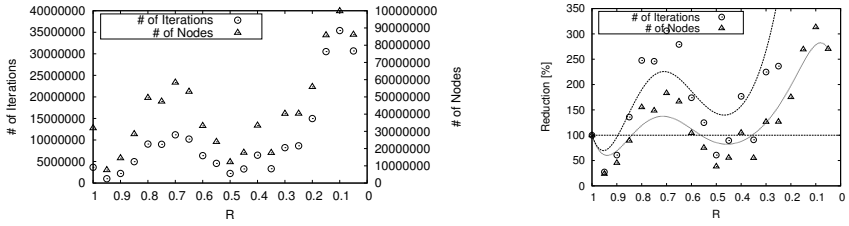
The question remains when DeepPN works most efficiently in the Othello endgame position for 16-ply. The answer depends on the group of Othello endgame positions. However, if we have to choose the best R, then a value of around 0.65 is a good compromise for most cases.

3.3 Performance with Hex

For measuring the performance of DeepPN, we also prepared a solver for Hex. As for the experiments of Othello, we created a primitive DeepPN algorithm for checking the effect of DeepPN only. The Hex program is a simple program that does not have any other mechanisms such as an evaluation function. Our Hex program uses a 4 x 4 board (called Hex(4)), and tries to solve that board using DeepPN. Our focus is on the behavior of R (see formula 3). Concerning the characteristics of R, please see section 3.1 or 3.2. We changed R from 0.0 to 1.0 by 0.5, and tried to solve Hex(4) 10 times in each R. The legal moves of Hex are sorted randomly in every configuration, viz. there is the possibility that each result is different. The results in each R are calculated by the average of the 10 experiments. Next we focused on two concepts: (1) number of iterations and (2) number of nodes. About the characteristics of both values, please see the section 3.2. The experimental data are given in figure 5.

Figure 5a shows the changes in the number of iterations and nodes. We can see that the results of DeepPN decrease (improve) in some positions compared by PN-search. This is not the case for $R = 0.0$, because we cannot solve Hex(4) for this R when we limit ourselves to 500 million nodes. For ease of understanding, we prepare another graph in figure 5b. There we show the reduction rates normalized by the result of PN-search, i.e., the result of PN-search has 100% reduction rate.

Figure 5b shows that the number of iterations and nodes is reduced by a 30% reduction rate between $R = 0.95$ and $R = 0.5$. The result has two downward curves: from $R = 1.0$ to 0.7 and from $R = 0.7$ to $R = 0.0$. The first curve starts from $R = 1.0$ and decreases toward 0.95. After $R = 0.95$, the results start to increase and grow to over 100% after 0.85. The second curve starts from $R = 0.7$



(a) The variation (b) The reduction rate

Fig. 5: Hex: # of Iterations and # of Nodes for Hex(4). $R = 1.0$ is PN-search, $R = 0.0$ is depth-first search, and $1.0 > R > 0.0$ is DeepPN. Lower is better.

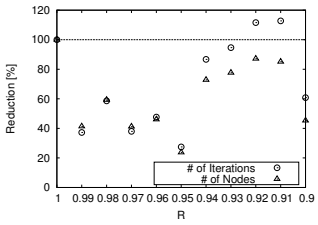


Fig. 6: Hex: The detail of figure 5b. This figure is zoomed $1.0 \leq R \leq 0.9$. The lower is better.

and the results starts to decrease again. At around $R = 0.5$, the results reach about 50%. Finally, the results are increasing again toward $R = 0.0$, like Othello.

For understanding the details of how DeepPN works around $R = 0.95$, we tried to change R by 0.1 between from 1.0 to 0.9. The results are shown in figure 6.

By looking at the results, we can see that DeepPN works almost twice as good as PN-search from $R = 0.99$ to 0.95. Form $R = 0.95$ to 0.90, we have a small curve like figure 5b.

In Hex(4), the optimum value of R is around $R = 0.95$ (and perhaps $R = 0.5$). We can see that depth-first does not work so well for Hex(4) as it does for Othello, although there is an improvement over pure best-first.

3.4 Discussion

DeepPN works efficiently in 16-ply Othello endgame positions, and in Hex(4). It can reduce the number of iterations and nodes almost by half compared to PN-search. It must be noted that the optimum balance of R is different in each game and for each size of game tree. We can see that for both games a certain amount of depth-first behavior is beneficial, but the changes are not the same. The precise relation is a topic of future work.

Both in Othello endgame positions and in Hex(4), we encountered positions that showed increasing (worse) results. We suspect that a reason for this problem may be (1) the holding problem and (2) the length of the shortest correct path. Concerning (1), the depth-first search can remain stuck in one subtree (holding on to the subtree). If this holding subtree cannot find the game-theoretical value, then the number of iterations and nodes become meaningless. When DeepPN employed a strong depth-first manner, then we found many increasing results in Othello endgame positions. Also, in Hex(4), DeepPN cannot work efficiently around $R = 0.0$. Finding an optimal R is a topic of future work.

Concerning (2), the problem is related to (1). In Othello, the shortest correct path is almost the same for each position, because Othello has a fixed number of depth to the end. However, in Hex(4), the shortest winning path may exist before a depth of 16. If we happen to find a balance between depth and best-first, then DeepPN will change the subtree it focuses on time. For example, when $R = 0.95$, then DeepPN quickly finds the shortest path. But after $R = 0.95$, DeepPN misses that path and arrives in regions that are more deeply in the trees. Finding a good value of R in Hex is more difficult than in Othello.

4 Conclusion and Future works

In this work, we proposed a new search algorithm based on proof numbers, named DeepPN. DeepPN has three values (pn, dn, *deep*) a single parameter, R, that allows a choice between depth-first and best-first behavior. DeepPN employs two types of values, viz., proof numbers and deep values which register the depth of nodes. For measuring the performance of DeepPN, we tested DeepPN on solving Othello endgame positions and on the game of Hex. We achieved two indicative results in Othello and Hex. The algorithm owes its success to formula (3) in which best-first and depth-first search are applied in a "balanced" way. The results show that DeepPN works better than PN-search in the games which build up a *suitable* tree.

We have three main topics for future work. First, we have to investigate how to find a good balance for R. In our experiments, the best results are produced by different values of R. Second, DeepPN is too primitive to solve complex problems. Df-pn+ and the use of a transposition table give a good hint for this problem. The idea of deep value may also be applied to other ways of searching. Third, we need to find a reason about the deterioration of search results in Othello endgame positions and in the game of Hex. By investigating this problem, we expect that DeepPN can become a quite useful algorithm for explaining the intricacies.

Previous work on depth-first and best-first minimax algorithms used null-windows around the minimax value to guide the search in a best-first manner [8, 9]. There, a relation between the SSS* and the Alpha-beta algorithm was found. In this work, we focus on best-first behavior guided by the *size* of the minimax tree, as is the essence of proof-number search. It will be interesting to see if both

kinds of best-first behavior can be combined in future work in a new kind of conspiracy number search.

References

1. A. Kishimoto, M. Müller (2008). About the Completeness of Depth-First Proof-Number Search. Computers and Games (CG2008), Lecture Notes in Computer Science Volume 5131, pp 146-156
2. A. Kishimoto, M. Winands, M. Müller, J-T. Saito (2012). Game-Tree Search using Proof Numbers: The First Twenty Years. ICGA Journal, Vol. 35, No. 3
3. A. Kishimoto and M. Muller. Search versus Knowledge for Solving Life and Death Problems in Go,Twentieth National Conference on Artificial Intelligence (AAAI-05), pages 1374-1379, 2005.
4. A. Kishimoto. Correct and Efficient Search Algorithms in the Presence of Repetitions, Ph.D. Thesis, University of Alberta, 2005
5. A. Nagai. Df-pn Algorithm for Searching AND/OR Trees and Its Applications. PhD thesis, Dept. of Information Science, University of Tokyo, Tokyo, 2002
6. A. Nagai.: A new AND/OR tree search algorithm using proof number and disproof number. In:Proceedings of Complex Games Lab Workshop, pp.40-45. ETL, Tsukuba (1998)
7. A. Nagai.: A new depth-first search algorithm for AND/OR trees. M.Sc. Thesis, Department of Information Science, The University of Tokyo, Japan (1999)
8. A. Plaat, J. Schaeffer, W. Pijls, A. de Bruin (1995). Best-first and depth-first minimax search in practice. Proceedings of Computer Science in the Netherlands. 182-193
9. A. Plaat, J. Schaeffer, W. Pijls, A. de Bruin (1994). SSS* = Alpha-beta + TT. Technical Report 94-17, University of Alberta, Edmonton, Canada.
10. D. McAllester, Conspiracy Numbers for Min-Max Search. Artificial Intelligence, Vol. 35, No. 1, pp. 287-310, 1988.
11. J. Hashimoto: A Study on Game-Independent Heuristics in Game-Tree Search. Ph.D. Thesis, School of Information Science, Japan Advanced Institute of Science and Technology (2011)
12. J. Pawlewicz and L. Lew.: Improving depth-first search: $1 + \epsilon$ trick. In Proc. of the 5th International Conference on Computers and Games, volume 4630 of Lecture Notes in Computer Science, pages 160-171. Springer, 2007
13. J. Schaeffer, Y. Björnsson, N. Burch, A. Kishimoto, M. Müller, R. Lake, P. Lu, S. Sutphen: Checkers Is Solved. Science 317(5844), 1518-1522 (2007)
14. J. Schaeffer.: Game Over: Black to Play and Draw in Checkers. ICGA Journal 30(4), 187-197 (2007)
15. K. Hoki, T. Kaneko, A. Kishimoto, T. Ito (2013). Parallel Dovetailing and its Application to Depth-First Proof-Number Search. ICGA Journal, Vol. 36, No. 1
16. L. V. Allis, M. van der Meulen, and H. J. van den Herik. Proof-number search. Artificial Intelligence, 66(1):91-124, 1994.
17. M. Seo, H. Iida, J.W.H.M. Uiterwijk: The PN*-search algorithm: Application to tsume-shogi. Artificial Intelligence 129(4), 253-277 (2001)
18. M. H. M. Winands, Informed Search in Complex Games., Ph.D. thesis, Maastricht University, The Netherlands, 2004
19. T. Ueda, T. Hashimoto, J. Hashimoto, H. Iida (2008). Weak Proof-Number Search. CG 2008