JAIST Repository
https://dspace.jaist.ac.jp/

Title	Deep df-pn and Its Application to Connect6
Author(s)	Song, Zhang; Iida, Hiroyuki; van den Herik, Jaap
Citation	ゲームプログラミングワークショップ2016論文集, 2016: 6-12
Issue Date	2016-10-28
Type	Conference Paper
Text version	publisher
URL	http://hdl.handle.net/10119/14076
Rights	社団法人　情報処理学会, Zhang Song, Hiroyuki Iida, Jaap van den Herik , ゲームプログラミングワークショップ2016論文集, 2016, 6-12. ここに掲載した著作物の利用に関する注意: 本著作物の著作権は（社）情報処理学会に帰属します。本著作物は著作権者である情報処理学会の許可のもとに掲載するものです。ご利用に当たっては「著作権法」ならびに「情報処理学会倫理綱領」に従うことをお願いいたします。 Notice for the use of this material: The copyright of this material is retained by the Information Processing Society of Japan (IPSJ). This material is published on this web site with the agreement of the author (s) and the IPSJ. Please be complied with Copyright Law of Japan and the Code of Ethics of the IPSJ if any users wish to reproduce, make derivative work, distribute or make available to the public any part or whole thereof. All Rights Reserved, Copyright © Information Processing Society of Japan.
Description	The 21st Game Programming Workshop (GPW2016)

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Deep df-pn and Its Application to Connect6

Zhang Song[1,a)] Hiroyuki Iida[1,b)] Jaap van den Herik[2,c)]

Abstract: Depth-first proof-number search (df-pn) is a powerful variant of proof-number search algorithms, widely used for AND/OR tree search or solving games. However, it suffers from the seesaw effect, which can be concluded as frequently going back to the ancestor nodes for selecting the most proving node. It works strongly against the efficiency in some situations. This paper proposes a new proof number algorithm called Deep depth-first proof-number search (Deep df-pn) to reduce the seesaw effect in df-pn. The only difference between Deep df-pn and df-pn is the proof number or disproof number of unsolved nodes. The proof number or disproof number of unsolved nodes is 1 in df-pn, while it is a function of depth with two parameters in Deep df-pn. By adjusting the value of parameters, Deep df-pn changes its behavior from searching broadly to searching deeply. Moreover, this paper proves that Deep df-pn enables to reduce the seesaw effect. Experiments performed in the domain of Connect6 show its effectiveness in searching efficiency.

1. Introduction

Proof-Number Search (PN-search) [1] is one of the most powerful algorithms for solving games and complex endgame positions. PN-search focuses on AND/OR tree and tries to establish the game theoretical value in a best-first manner. Each node in PN-search has a proof number (pn) and disproof number (dn). This idea was inspired by the concept of conspiracy numbers, the number of children that need to change their value to make a node change its value [6]. A proof (disproof) number shows the scale of difficulty in proving (disproving) a node. PN-search expands the most-proving node, which is the most efficient one for proving (disproving) the root.

Although PN-search is an effective AND/OR-tree search algorithm, it still has problems. One is that PN-search uses a lot of memory space because it is a best-first algorithm, and the other is that it is not efficient enough because of frequently updating the proof and disproof number. So Nagai [5] proposed a depth-first algorithm using both proof number and disproof number based on PN-search, which is called depth-first proof-number search (df-pn). The procedure of df-pn can be concluded as selecting the most proving node, updating thresholds of proof number or disprove number in a transposition table, multiple iterative deepening until satisfy the end condition. Nagai proved the equivalence between PN-search and df-pn that df-pn always selects the most proving node as pn-search does in the searching path. But for its depth-first manner and the use of transportation table, df-pn saves more storage and is more efficient than PN-search.

However, both PN-search and df-pn suffer from the seesaw effect which can be concluded as frequently going back to the ancestor nodes for selecting the most proving node, as described in [8], [10], [11]. They showed that the seesaw effect works strongly against the efficiency in some situations. This paper then proposes called Deep depth-first proof-number search algorithm (Deep df-pn) to reduce the seesaw effect in df-pn. The only difference between Deep df-pn and df-pn is the proof number or disproof number of unsolved nodes. In df-pn the proof number or disproof number of unsolved nodes is 1, while in Deep df-pn it is a function of depth with two parameters. By adjusting the value of parameters, Deep df-pn changes its behavior from searching broadly to searching deeply. This paper proves that Deep df-pn can help reduce the seesaw effect. Experiments of solving endgame positions in Connect6 [12] also show its good performance in searching efficiency.

The rest of the paper is as follows. We briefly summarize the details of PN-search and df-pn in Section 2, and introduce the seesaw effect in Section 3. Definitions of Deep df-pn and its characteristics are presented in Section 4. In Section 5, we conduct experiments on Connect6 to show its better performance in reducing seesaw effect. Finally, concluding remarks are given in Section 6.

2. PN-Search and Its Depth-First Variant

In this section, we introduce the original proof-number search (PN-search) and depth-first proof-number search (df-pn), a depth-first variant with advantages on space saving and efficiency.

[1] Graduate School of Information Science, Japan Advanced Institute of Science and Technology
Nomi, Japan
[2] Leiden Institute of Advanced Computer Science
Leiden, The Netherlands
[a)] zhangsong@jaist.ac.jp
[b)] iida@jaist.ac.jp
[c)] jaapvandenherik@gmail.com

© 2016 Information Processing Society of Japan

2.1 PN-Search

Proof-Number Search (PN-search) [1] is a native best-first algorithm, using proof numbers and disproof numbers, always expanding one of the most-proving nodes. This idea was inspired by the concept of conspiracy numbers [6], the number of children that need to change their value to make a node change its value, usually applied in a minimax tree to indicate the stability of the root value.

To apply this concept into AND/OR tree where nodes have a binary value (win or not win), Allis et al. [1] reformed conspiracy numbers as a proof number and a disproof number, showing the scale of difficulty in proving and disproving a node respectively. For all nodes, proof and disproof numbers are stored to indicate which frontier node will be expanded, and updated after expanding. The expanded node is called the most-proving node, which is the most efficient one for proving (disproving) the root.

It is found by exploiting two characteristics of the search tree [7]: (1) its shape (determined by the branching factor of every internal node), and (2) the values of the leaves. Basically, unenhanced pn-search is an uninformed search method that does not require any game-specific knowledge beyond its rules [3]. The detail of proof number, disproof number and most-proving node are given as follows.

Let $n.pn$ and $n.dn$ be the proof number and disproof number of a node n, respectively.

1. When n is a leaf node
   (a) When n is a win of the attacker
   $$n.pn = 0$$
   $$n.dn = \infty$$
   (b) When n is a loss of the attacker
   $$n.pn = \infty$$
   $$n.dn = 0$$
   (c) When the value of n is unknown
   $$n.pn = 1$$
   $$n.dn = 1$$

2 When n is an internal node
   (a) When n is an OR node
   $$n.pn = \underset{n_c \in \text{children of n}}{Min} n_c.pn$$
   $$n.dn = \sum_{n_c \in \text{children of n}} n_c.dn$$
   (b) When n is an AND node
   $$n.pn = \sum_{n_c \in \text{children of n}} n_c.pn$$
   $$n.dn = \underset{n_c \in \text{children of n}}{Min} n_c.dn$$

A most-proving node is a leaf node that is selected by tracing nodes from the root node in the following way.

- For each OR node, trace the child with the minimum proof number.
- For each AND node, trace the child with the minimum disproof number.

Note that Allis et al. [1] defined the most-proving node as the left-most one, if there is arbitrariness.

2.2 Df-pn

Although PN-search is an ideal AND/OR-tree search algorithm, it still has two problems. One is that PN-search uses a lot of memory space because of its best-first manner, and the other is that it is not efficient enough for its frequently updating the proof and disproof number. To solve the first problem, Nagai [5] proposed a depth-first like algorithm using both proof number and disproof number based on PN-search, which is called df-pn (depth-first proof-number search). The procedure of df-pn can be concluded as selecting the most proving node, updating the thresholds of proof number or disprove number in a transposition table, multiple iterative deepening until the end condition is satisfied. It is a depth-first like search but has a same behavior as PN-search. The equivalence between PN-search and df-pn was proved in [5].

In df-pn, proof number and disproof number are renamed as follows.

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

Moreover, each node n has two thresholds: one for the proof number th_{pn} and the other for the disproof number th_{dn}. Similarly, they are renamed th_{pn} and th_{dn} as follows.

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

Df-pn expands the same frontier node as PN-search in a depth-first manner guided by a pair of thresholds (th_{pn},th_{dn}), which indicates whether the most-proving node exists in the current subtree [4]. The procedure is described below [5].

Procedure Df-pn.

For the root node r, assign values for $r.th_\phi$ and $r.th_\delta$ as follows.

$$r.th_\phi = \infty$$
$$r.th_\delta = \infty$$

Step 1. At each node n, the search process continues to search below n until $n.\phi \geq n.th_\phi$ or $n.\delta \geq n.th_\delta$ is satisfied (we call it ending condition).

Step 2. At each node n, select the child n_c with minimum δ and the child n_2 with second minimum δ. (If there is another child with minimum δ, that is n_2.) Search below n_c with assigning

$$n_c.th_\phi = n.th_\delta + n_c.\phi - \sum n_{child}.\phi$$
$$n_c.th_\delta = \min\left(n.th_\phi, n_2.\delta + 1\right).$$

Repeat this process until the ending condition holds.

Step 3. If the ending condition is satisfied, the search process returns to the parent node of n. If n is the root node, then the search is totally over.

© 2016 Information Processing Society of Japan

3. Seesaw Effect and DeepPN

In this section, we introduce the seesaw effect and DeepPN, a variant of proof-number search focusing on reducing the seesaw effect.

3.1 Seesaw Effect

PN-search and df-pn are highly efficient in solving games. However, there are still some problems with them. One of such drawbacks were pointed out [8] [10] [11] and named as *seesaw effect* [9]. It can be concluded as frequently going back to the ancestor nodes for selecting the most proving node.

To explain it precisely, we show, in Fig. 1, an example where the root node has two subtrees. The size of both subtrees is almost the same. Assume that the proof number of subtree L is larger than the proof number of subtree R. In this case, pn-search or df-pn will continue search in subtree R, which means that the most proving node is in subtree R. After pn-search or df-pn expands the most-proving node, the shape of the game tree changes as shown in Fig. 1(b). By expanding the most-proving node, the proof number of subtree R becomes larger than the proof number of subtree L. So pn-search or df-pn changes its searching direction from subtree R to subtree L. Similarly, when the search expands the most-proving node in subtree L, the proof number of subtree L becomes larger than subtree R. Thus, the search changes its focus from subtree L to subtree R. This change keeps going back and forth, which looks like a seesaw. Therefore, it is named as seesaw effect.

The seesaw effect happens when the two trees are almost equal in size. If the seesaw effect occurs frequently, the performance of PN-search and df-pn deteriorates significantly and cannot reach the required searching depth. In games which need to reach a large fixed searching depth, the seesaw effect works strongly against efficiency.

The seesaw effect is mostly caused from two points: the shape of game tree and the way of searching. Concerning the shape of game tree, there are two characteristics: (1) a tendency for the children size becoming equal and (2) many nodes with equal values exist deep down in a game tree. In (1), if the children size of each node becomes almost the same, then the seesaw effect may occur easily. For (2), it is common in games such as Othello and Hex which need to search a large fixed number of moves before settling. It is also common in connect-type games such as Gomoku and Connect6 which have a sudden death in the game tree. Therefore, it is necessary to design a new search algorithm to reduce the seesaw effect in these games.

3.2 DeepPN

To tackle the seesaw effect problem, Ishitobi et al. [2] proposed Deep Proof-Number Search (DeepPN), a variant of PN-search while focusing on reducing the seesaw effect. It employs two important values associated with each node,

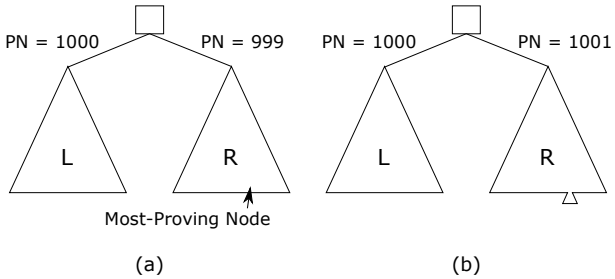

Fig. 1: An example of seesaw effect: (a) An example game tree (b) Expanding the most-proving node

the usual proof number and a deep value. The deep value is defined as the depth of a node which shows the progress of the search in the depth direction. After mixing the proof numbers and the deep value, the DeepPN can change its behaviors from the best-first manner (equal to the original proof-number search) to the depth-first manner by adjusting a parameter R. Compared to the original ON-search, DeepPN shows better results when R comes to a proper value which makes the search between best-first like and depth-first like. The formal definitions of DeepPN are described below.

In DeepPN, the proof number and disproof number of node n are calculated as follows.

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

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

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

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

Definition 1 When n is an internal node

(a) The proof and disproof number are defined as follows

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

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

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

$$n.deep = n_c.deep$$

where

$$n_c = \underset{n_i \in \text{ unsolved children of n}}{arg\ min} DPN\left(n_i\right)$$

and

© 2016 Information Processing Society of Japan

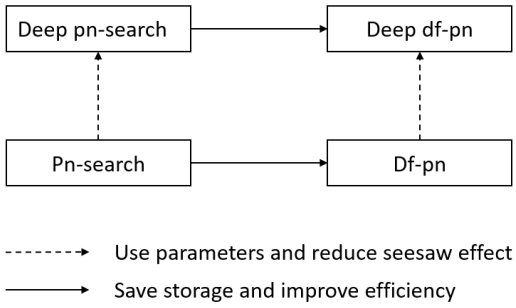

Fig. 2: Relationship between PN-search, df-pn, DeepPN and Deep df-pn

$$DPN(n) = \left(1 - \frac{1}{n.\delta}\right) R + n.deep\,(1 - R)$$

Definition 2 In DeepPN, an expanding node in each iteration is defined as follows.

$$select_expanding_node\,(n) :=$$

$$\underset{n_c \in \text{ unsolved children of n}}{arg\ min} \quad DPN\,(n_c)$$

According to the definitions, the proof and disproof number are the same with the original PN-search. The only difference is the deep value which is designed to decrease inversely with depth. For the terminal nodes, the deep value is the reciprocal of depth. For the internal nodes, the deep value is decided by a child node n_c which has the smallest $DPN(n_c)$. DPN is a function defined with two features: (a) $\left(1 - \frac{1}{n.\delta}\right)$ is normalized and designed to become larger as $n.\delta$ grows. (b) a fixed parameter R is given between 0.0 and 1.0. If R is 1.0, DeepPN works the same with PN-search and the expanding node is the most-proving node. If R is 0.0, DeepPN works the same with a primitive depth-first search. Therefore, by changing the value R, the ratio of best-first and depth-first search of DeepPN can be adjusted.

4. Deep df-pn

The main idea of DeepPN is to improve the original PN-search by using a parameter which mixes the best-first search and depth-first search, which has been verified with better performance on reducing the seesaw effect. However, it still has some drawbacks. Firstly, DeepPN is still a best-first like search in essence. So it surfers from a big cost of storage as pn-search. Secondly, DeepPN spends much time on updating the proof and disproof number, which makes DeepPN not efficient enough. In this section, we propose a new proof-number algorithm based on df-pn to cover the shortage of DeepPN, named as Deep Depth-First Proof-Number Search or Deep df-pn in short. It not only extends the improvements of df-pn on saving storage and efficiency, but also reduces the seesaw effect. Fig. 2 shows the relationship between PN-search, df-pn, DeepPN and Deep df-pn.

Similar to DeepPN, the proof number and disproof number of unsolved nodes in Deep df-pn is a function of depth but with two parameters. By adjusting the values of the two parameters, Deep df-pn changes its behavior from searching broadly to searching deeply. Definitions of Deep df-pn are given below.

Table 1: Behaviors changing by parameters

	$E = 0$	$E > 0$
$D = 0$	Depth-first	Df-pn
$D > 0$	Depth-first	Intermediate

In Deep df-pn, the proof number and disproof number of node n are calculated as follows.

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

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

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

$$n.\phi = DPN\,(n.depth)$$
$$n.\delta = DPN\,(n.depth)$$

(d) When n is an internal node, the proof and disproof number are defined as follows

$$n.\phi = \underset{n_c \in \text{ children of n}}{Min} \quad n_c.\delta$$

$$n.\delta = \underset{n_c \in \text{ children of n}}{\sum} \quad n_c.\phi$$

Definition 3 $DPN\,(x)$ is a function from $\mathbb{N}$ to $\mathbb{N}$, which

$$DPN\,(x) = \begin{cases} E^{D-x} & (D > x \wedge E > 0) \\ 1 & (D \leq x \wedge E > 0) \\ 0 & (E = 0) \end{cases}$$

where E and D are parameters on $\mathbb{N}$.

According to the definitions, the difference between Deep df-pn and df-pn is the proof and disproof number of unsolved nodes. The proof and disproof number of an unsolved node is 1 in df-pn, while it is a function $DPN\,(n.depth)$ in Deep df-pn, which is relevant with parameters E and D. Here, E and D denotes a threshold of branch size and threshold of depth, respectively. The complete algorithm of Deep df-pn is presented in Algorithm 1 and Algorithm 2.

Table 1 shows the behavior of Deep df-pn with different values of E and D. When $E = 0$, Deep df-pn is a depth-first search. When $E > 0$ and $D = 0$, Deep df-pn is the same with df-pn. When $E > 0$ and $D > 0$, Deep df-pn is an intermediate one between depth-first search and df-pn. In other words, if E or D becomes smaller, Deep df-pn tends to search more broadly. While if E or D becomes larger, it tends to search more deeply.

© 2016 Information Processing Society of Japan

Algorithm 1 Deep df-pn (part I)

```
1: // At the root
2: procedure DeepdfpN(r)
3:     r.φ = ∞; r.δ = ∞;
4:     MID(r);
5: end procedure
6:
7:  // Exploring node n
8: procedure MID(n)
9:     // 1. Look up transposition table
10:    LookUpTranspositionTable(n,φ,δ);
11:    if n.φ ≤ φ || n.δ ≤ δ then
12:        n.φ = φ; n.δ = δ;
13:        return ;
14:    end if
15:
16:     // 2. Generation of legal moves
17:    if n is a terminal node then
18:        if (n is an AND node && Eval(n) = true) ||
19:        (n is an OR node && Eval(n) = false) then
20:            n.φ = ∞; n.δ = 0;
21:        else
22:            n.φ = 0; n.δ = ∞;
23:        end if
24:        PutInTranspositonTable(n,n.φ,n.δ);
25:        return ;
26:    end if
27:
28:    GenerateLegalMoves();
29:
30:     // 3. Avoidance of cycle by using transposition table
31:    PutInTranspositonTable(n,φ,δ);
32:
33:     // 4. Multiple Iterative Deepening
34:    while 1 do
35:         // Stop searching if φ or δ is above or equal to
36:         its threshold
37:        if n.φ ≤ ΔMin(n) || n.δ ≤ ΦSum(n) then
38:            n.φ = ΔMin(n); n.δ = ΦSum(n);
39:            PutInTranspositonTable(n,φ,δ);
40:            return ;
41:        end if
42:        n_c = SelectChild(n,φ_c,δ_c,δ_2);
43:        n_φ = n_δ + φ_c − ΦSum(n);
44:        n_δ = min(n.φ, δ_2 + 1);
45:        MID(n_c);
46:    end while
47: end procedure
48:
49:  // Record into the transposition table
50: procedure PutintranspostiontABLE(n,φ,δ)
51:    Table[n].φ = φ; Table[n].δ = δ;
52: end procedure
```

Algorithm 2 Deep df-pn (part II)

```
53:  // Look up the transposition table
54: procedure LookuptranspostiontABLE(n,&φ,&δ)
55:    if n is already recorded then
56:        φ =Table[n].φ; δ =Table[n].δ;
57:    else
58:         // In df-pn φ = 1, δ = 1
59:        if E = 0 then
60:            φ = 0; δ = 0;
61:        else if D ≤ n.depth then
62:            φ = 1; δ = 1;
63:        else
64:            φ = E^{D−n.depth}; δ = E^{D−n.depth};
65:        end if
66:    end if
67: end procedure
68:
69:  // Selection of the child
70: procedure SelectchILD(n,&φ_c,&δ_c,&δ_2)
71:    δ_c = ∞; δ_2 = ∞;
72:    for each child node n_child do
73:        LookUpTranspositionTable(n_child,φ,δ);
74:        if δ < δ_c then
75:            n_best = n_child;
76:            δ_2 = δ_c; φ_c = φ; δ_c = δ;
77:        else if δ < δ_2 then
78:            δ_2 = δ;
79:        end if
80:        if φ = ∞ then
81:            return n_best;
82:        end if
83:    end for
84:    return n_best;
85: end procedure
86:
87:  // Calculate the minimum δ among all the children
88: procedure ΔMin(n)
89:    min = ∞
90:    for each child node n_child do
91:        LookUpTranspositionTable(n_child,φ,δ);
92:        min = min(min, δ);
93:    end for
94: end procedure
95:
96:  // Calculate the summation of φ among all the children
97: procedure ΦSum(n)
98:    sum = 0
99:    for each child node n_child do
100:        LookUpTranspositionTable(n_child,φ,δ);
101:        sum = sum + φ;
102:    end for
103:    return sum;
104: end procedure
```

© 2016 Information Processing Society of Japan

It can be proved that Deep df-pn can help reduce the seesaw effect in df-pn.

Theorem 1 Deep df-pn outperforms df-pn in reducing the seesaw effect.

Proof Assume that node n is a most proving node in a seesaw effect like Fig. 1. Without loss of generality, n is an AND node in subtree L. According to the feature of seesaw effect, after extending n, its proof number becomes larger, which makes the proof number of subtree L larger. Then df-pn changes its focus on subtree R and seesaw effect happens.

From the definitions of Deep df-pn, the proof number of n is given by

$$DPN\left(n.depth\right).$$

After extending n, its proof number is given by

$$\sum_{\text{children of n}} DPN\left(n.depth+1\right) =$$

$$E^{'} \cdot DPN\left(n.depth+1\right).$$

where $E^{'}$ denotes the number of children of n. If $E^{'} \leq E$ and $n.depth+1 < D$, then we have

$$E^{'} \cdot DPN\left(n.depth+1\right) = E^{'} \cdot E^{D-(n.depth+1)}$$

and

$$E^{'} \cdot E^{D-(n.depth+1)} \leq E^{D-depth}.$$

So we obtain the inequation.

$$\sum_{\text{children of n}} DPN\left(n.depth+1\right) \leq DPN\left(n.depth\right).$$

Therefore, Deep df-pn continues focusing on subtree R and seesaw effect dose not happen. As a result, Deep df-pn outperforms df-pn in reducing the seesaw effect.

□

5. Experiments

In this section we evaluate the performance of Deep df-pn. For this purpose, the game of Connect6 is chosen as a benchmark. We first introduce the game of Connect6, then present the experimental design, and experimental results are shown to discuss.

5.1 Connect6

Experiments with Deep df-pn are performed on Connect6 [12] which is a two-player strategy game similar to Gomoku. Black (first player) puts one stone on the board for the first move. Then both players put two stones for each move. The one who gets six or more stones in a row (horizontally, vertically or diagonally) first wins the game. Connect6 has an infinite board, so both state-space and game-tree complexities are infinite too. In order to make it countable, we use a Go board (19×19) for Connect6. Both state-space and game-tree complexities for it are still much higher than those in Gomoku and Renju, in the sense that two stones per move make the branch factor increase by a factor of half of the board size. Based on the standard used in [13], the state-space complexity of Connect6 is 10^{172}, the same as that in Go. If a larger board is used, the complexity is much higher. Threat-based strategy is a common strategy used to play Connect6 [12]. As a result, there exists sudden death in its game tree.

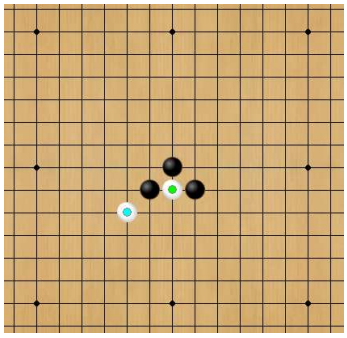

Fig. 3: An example position of Connect6 (White is to move)

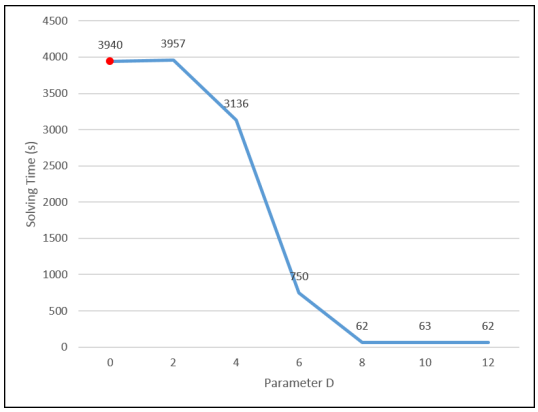

Fig. 4: Deep df-pn and df-pn compared in solving time with various threshold depth D (Df-pn when $D = 0$)

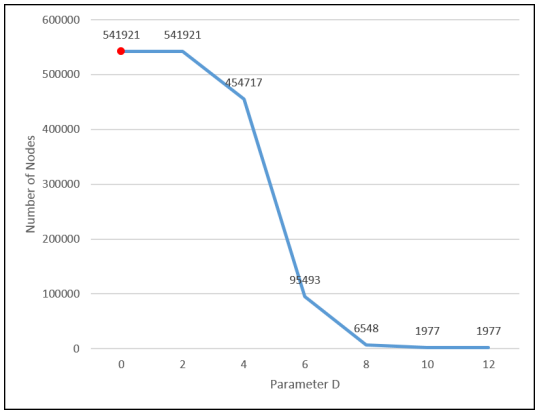

Fig. 5: Deep df-pn and df-pn compared in game-tree size with various threshold depth D (Df-pn when $D = 0$)

5.2 Experimental Design

To solve the endgame positions of Connect6, we use a combination of relevance zones and Deep df-pn. Each time Deep df-pn expands the defender's moves, the relevance zones generator generates a relevance zone to indicate the most pos-

© 2016 Information Processing Society of Japan

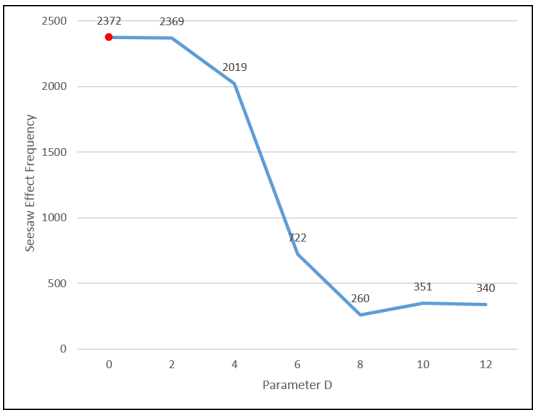

Fig. 6: Deep df-pn and df-pn compared in frequency of seesaw effect with various threshold depth D (Df-pn when $D = 0$)

sible moves which the defender should apply on the board. And each time Deep df-pn expands the attacker's moves, the relevance zones generator only generates the most possible 10 moves to reduce the complexity. For measuring the frequency of seesaw effect, we count the number of seesaw effect, which is initialized by 0 and added by 1 when a node in the game tree is traced again during the search.

5.3 Results and Discussion

In the position of Fig. 3 where Black is to move (which means that White puts two stones out of the region in Fig. 3), if $E = 0$, Deep df-pn is a depth-first search where frequency of seesaw effect is 0. If $E = 3$, three changing curves by parameter D can be obtained as shown in Fig. 4, Fig. 5 and Fig. 6, which implies that the performance of Deep df-pn changes by D for a specific position. If $D = 0$, Deep df-pn is the same with df-pn. If $D > 0$, Deep df-pn takes cost less both in time and frequency of seesaw effect than df-pn. By finding the suitable values of E and D, we can obtain the optimum performance of Deep df-pn for the position.

6. Concluding Remarks

In this paper, we proposed a new proof-number algorithm called Deep Depth-First Proof-Number Search (Deep df-pn) to improve df-pn by reducing the seesaw effect. Deep df-pn is a natural extension of Deep Proof-Number Search (DeepPN) and df-pn. The relation between PN-search, df-pn, DeepPN and Deep df-pn was discussed. The main difference between Deep df-pn and df-pn is the proof number or disproof number of unsolved nodes. In df-pn, the proof number or disproof number of unsolved nodes is 1, while in Deep df-pn it is a function of depth with two parameters. By adjusting the values of parameters, Deep df-pn changes its behavior from searching broadly to searching deeply. Moreover, this paper has shown the theoretical proof that Deep df-pn can help reduce the seesaw effect. Experiments performed in solving endgame positions of Connect6 also show the effectiveness of Deep df-pn with better performance.

In this paper, Connect6 was chosen as a benchmark to evaluate the performance of Deep df-pn. Connect6 is a game with an unbalanced game tree. Further investigations will be made using other type of games with a balanced game tree such as Othello and Hex.

Acknowledgement

This research is funded by a grant from the Japan Society for the Promotion of Science, in the framework of the Grant-in-Aid for Challenging Exploratory Research (grant number 26540189).

References

[1] L V. Allis, M. van der Meulen, H J. van den Herik. "Proof-number search," *Artificial Intelligence*, vol. 66, no. 1, pp.91–124, 1994.

[2] T. Ishitobi, A. Plaat, H. Iida, J. van den Herik. "Reducing the seesaw effect with deep proof-number search," *Advances in Computer Games*, Lecture Notes in Computer Science 9525, Springer International Publishing, pp.185–197, 2015.

[3] A. Kishimoto, M H M. Winands, M. Mller, JT. Saito. "Game-tree search using proof numbers: The first twenty years," *ICGA Journal*, vol. 35, no. 3, pp.131–156, 2012.

[4] T. Kaneko. "Parallel depth first proof number search," *Proceedings of the 24th AAAI Conference on Artificial Intelligence*, pp.95–100, 2010.

[5] A. Nagai. "Df-pn algorithm for searching AND/OR trees and its applications," *PhD Thesis*, Department of Information Science, University of Tokyo, 2002.

[6] D A. McAllester. "Conspiracy numbers for min-max search," *Artificial Intelligence*, vol. 35, no. 3, pp.287–310, 1988.

[7] H J. Van Den Herik, M H M. Winands. "Proof-number search and its variants," *Oppositional Concepts in Computational Intelligence*, Springer Berlin Heidelberg, pp.91–118, 2008.

[8] J. Pawlewicz, L. Lew. "Improving depth-first pn-search: 1+ trick," *International Conference on Computers and Games*, Lecture Notes in Computer Science 4630, Springer Berlin Heidelberg, pp.160–171, 2006.

[9] J. Hashimoto. "A study on game-independent heuristics in game-tree search," *PhD Thesis*, School of Information Science, Japan Advanced Institute of Science and Technology, 2011.

[10] A. Kishimoto, M. Mller. "Search versus knowledge for solving life and death problems in Go," *Proceedings of the 20th AAAI Conference on Artificial Intelligence*, pp.1374–1379, 2005.

[11] A. Kishimoto. "Correct and efficient search algorithms in the presence of repetitions," *PhD Thesis*, University of Alberta, 2005.

[12] I C. Wu, D Y. Huang. "A new family of k-in-a-row games," *Advances in Computer Games*, Lecture Notes in Computer Science 4250, Springer Berlin Heidelberg, pp.180–194, 2005.

[13] H J. Van den Herik, J W H M. Uiterwijk, J. Van Rijswijck. "Games solved: Now and in the future," *Artificial Intelligence*, vol. 134, no. 1, pp.277–311, 2002.

© 2016 Information Processing Society of Japan