Application of df-pn$^+$ to Othello Endgames

Ayumu Nagai[1] and **Hiroshi Imai**[1]
[1]Department of Information Science, University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113-0033, JAPAN
{nagai, imai}@is.s.u-tokyo.ac.jp

Abstract

We recently developed a df-pn algorithm for AND/OR-tree search which behaves the same as Allis' pn-search. In this paper, we propose a new algorithm df-pn$^+$ by extending df-pn. The extension is to use two kinds of information (i.e., **cost** and **h**) which AO* uses. We acquire such information from the pattern based evaluation function constructed for minimax-tree search. The experimental results on Othello endgames show that df-pn$^+$ is very efficient especially for large problems. On average, #nodes visited is reduced by a factor $\frac{1}{6}$ compared with the original df-pn when #disc is 48.

1 Introduction

A minimax tree such that each of the value of the nodes ultimately falls on either of the two values (i.e., **true** or **false**) is called an AND/OR tree. AO* [12] is a representative algorithm for AND/OR-tree search and is intensively studied. Allis developed pn-search [1] which uses both proof and disproof numbers. Since pn-search is an elegant algorithm and is easy to understand, it is in the limelight. Both AO* and pn-search are best-first algorithms.

Since there is a trend to use a depth-first algorithm so as to behave the same as a best-first one (e.g. MT-SSS* [13] is a depth-first algorithm that behaves the same as SSS* [16]), we quite recently developed a depth-first algorithm df-pn [11] using both proof and disproof numbers. We had proved that df-pn behaves the same as pn-search in the meaning of always expanding a most-proving node. Although a depth-first algorithm using both proof and disproof numbers can be constructed by simply extending Seo's algorithm [15] (we can even give a proof that it behaves the same as pn-search), df-pn is a more practical depth-first algorithm. Although PDS [10] is also a depth-first algorithm using both proof and disproof numbers, its basic concept differs from the rest. It does not behave exactly same as pn-search either. (PDS merely behaves asymptotically same as pn-search.)

Nagai had developed an algorithm that is an extension of PDS [10]. The same expansion can be applied to df-pn. We call it df-pn$^+$, and if df-pn$^+$ satisfies admissibility then we call it df-pn*. We will explain these algorithms in Section 2.

By the way, for searching minimax trees, Alpha-Beta is a representative algorithm. Although there are many variations and enhancements for Alpha-Beta, they all uses some kind of an evaluation function. Strong Othello programs use evaluation function constructed automatically from the records of many games. However, we never heard that this kind of evaluation function was used for proof-number search, except for a simple estimation to initialize (dis)proof numbers and studies by Tanaka and etc. [17]

In this paper, we show a method to apply such an evaluation function into a framework of df-pn$^+$. Section 3 explains how we constructed our evaluation function. Section 4 shows our application of this evaluation function to df-pn$^+$. The experimental results are presented in Section 5. Finally, Section 6 provides our conclusion.

2 Search Algorithm

2.1 Df-pn

Proof number at an OR node and disproof number at an AND node are essentially equivalent. Similarly, disproof number at an OR node and proof number at an AND node are essentially equivalent. As they are dual to each other, we can rename the former ϕ and the latter δ.

(Dis)proof number is defined as the least number of tip nodes of the current search tree, which must

be evaluated to be **true** (**false**) in order to ensure that the game-theoretical value of the root is **true** (**false**). The formula for calculating ϕ and δ is as follows.

1. If n is a tip node
   (a) When game-theoretical value is known
      i. If game-theoretical value is **true** (**false**) and n is an OR (AND) node
      $$n.\phi = 0$$
      $$n.\delta = \infty$$
      ii. If game-theoretical value is **true** (**false**) and n is an AND (OR) node
      $$n.\phi = \infty$$
      $$n.\delta = 0$$
   (b) When game-theoretical value is unknown
   $$n.\phi = 1$$
   $$n.\delta = 1$$
2. If n is an internal node
   $$n.\phi = \min_{n_c \in \text{children of } n} n_{\text{child}}.\delta$$
   $$n.\delta = \sum_{n_c \in \text{children of } n} n_{\text{child}}.\phi$$

Df-pn is a depth-first algorithm that behaves the same as pn-search. The characteristic feature of df-pn is that (1) each node has two thresholds: one for ϕ (th_ϕ) and the other for δ (th_δ), and that (2) Multiple Iterative Deepening [15] is used at all nodes. The brief algorithm of df-pn is as follows. (The program list of df-pn is carried on Appendix A.)

1. Assign
   $$r.th_\phi = \infty$$
   $$r.th_\delta = \infty$$
   where r is the root.
2. 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).
3. 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
   $$n_c.th_\phi = n.th_\delta + n_c.\phi - \sum n_{\text{child}}.\phi \quad (1)$$
   $$n_c.th_\delta = \mathbf{min}(n.th_\phi, n_2.\delta + 1). \quad (2)$$
   Repeat this process until the ending condition holds. (Multiple Iterative Deepening)
4. If ending condition holds, the search process returns to the parent node.
5. Continue until search process comes back to the root.

When we have insufficient memory space and if we are at the situation that the information of some nodes may be removed (e.g. by SiblingGC and SmallTreeGC [10]), then the substitution (1) and (2) should be modified into

$$n_c.th_\phi = n_c.\phi + 1$$
$$n_c.th_\delta = \mathbf{min}(n.th_\phi, \mathbf{max}(n_2.\delta, n.\phi) + 1).$$

2.2 Df-pn$^+$

Human experts searches promising moves very deep and narrow, on the other side, the unpromising moves are aggressively pruned. Df-pn searches promising moves deeper. Df-pn$^+$ intends to distinguish promising moves more accurately and to search them much more deeper.

Compared with Elkan's algorithm [8] or Seo's algorithm, AO* uses two kinds of additional information (function **g** and **h**) during the search. Similarly, df-pn$^+$ uses two kinds of information.

1. $\mathbf{cost}_{\text{(dis)proof}}(n, n_{\text{child}})$ is defined as the cost from node n to node n_{child} as a part of a (dis)proof solution. For example, if the position n is advantageous for the first player, then $\mathbf{cost}_{\text{proof}}$ has to become small, since cost is a load against searching deeper.

   The information of **cost** is used in order to enlarge the threshold of (dis)proof number than the original df-pn, whenever if it is the first (second) player's turn and if there is a move whose corresponding evaluation is high. The large threshold of (dis)proof number causes a deep focus on that move, resulting in a deep search.

2. $\mathbf{h}_{\text{(dis)proof}}(n)$ which is a heuristic estimate of the cost to reach any (dis)proof solution from position n. That is, $\mathbf{h}_{\text{(dis)proof}}(n)$ indicates the distance from (dis)proof solutions. For example, if the position n is advantageous for the first player, then $\mathbf{h}_{\text{(dis)proof}}(n)$ has to become small (large), since n is closer to a proof solution.

   The information of $\mathbf{h}_{\text{(dis)proof}}(n)$ is used at leaf nodes. Since $\mathbf{h}_{\text{(dis)proof}}(n)$ corresponds to the

distance from (dis)proof solution, it is used for initializing proof number and disproof number. If the position n is favorable to the first (second) player, (dis)proof number of that position, which is equal to $\mathbf{h}_{(\mathrm{dis})\mathrm{proof}}(n)$, gets small (since we construct it as so). Then the search process goes deep until satisfying the threshold.

Df-pn is equal to df-pn$^+$ with the following coditions.

$$\mathbf{cost} = 0$$
$$\mathbf{h} = 1.$$

If **cost** and **h** is defined, then **g** can be defined. $\mathbf{g}_{(\mathrm{dis})\mathrm{proof}}(n)$ is defined as the cost incurred so far for position n as a part of a (dis)proof solution. For example, if the position n is disadvantageous for the first player, then $\mathbf{g}_{(\mathrm{dis})\mathrm{proof}}(n)$ has to become large (small), since n is closer to a disproof solution. Although **g** is an important function, we do not use **g** explicitly, but **cost** instead.

If df-pn$^+$ satisfies admissibility, then we call it df-pn*, since it is guaranteed to find an optimal (dis)proof solution if it exists. The admissibility is defined as

$$\mathbf{h}_{(\mathrm{dis})\mathrm{proof}}(n) \leq \mathbf{h}^*_{(\mathrm{dis})\mathrm{proof}}(n),$$

where $\mathbf{h}^*_{(\mathrm{dis})\mathrm{proof}}(n)$ is the actual cost to reach a (dis)proof solution from node n. (For details, see [10].) Since **h** used in this paper does not satisfy this inequality, the algorithm used is df-pn$^+$.

Since we construct a negamax algorithm, $\mathbf{h}_\phi$, $\mathbf{h}_\delta$, $\mathbf{cost}_\phi$, and $\mathbf{cost}_\delta$ are defined as follows.

1. For an OR node n

$$\begin{cases} \mathbf{h}_\phi(n) & \equiv & \mathbf{h}_{\mathrm{proof}}(n) \\ \mathbf{h}_\delta(n) & \equiv & \mathbf{h}_{\mathrm{disproof}}(n) \\ \mathbf{cost}_\phi(n) & \equiv & \mathbf{cost}_{\mathrm{proof}}(n) \\ \mathbf{cost}_\delta(n) & \equiv & \mathbf{cost}_{\mathrm{disproof}}(n) \end{cases}$$

2. For an AND node n

$$\begin{cases} \mathbf{h}_\phi(n) & \equiv & \mathbf{h}_{\mathrm{disproof}}(n) \\ \mathbf{h}_\delta(n) & \equiv & \mathbf{h}_{\mathrm{proof}}(n) \\ \mathbf{cost}_\phi(n) & \equiv & \mathbf{cost}_{\mathrm{disproof}}(n) \\ \mathbf{cost}_\delta(n) & \equiv & \mathbf{cost}_{\mathrm{proof}}(n) \end{cases}$$

Then, relation between ϕ,δ and **g**,**h** is as follows.

$$\begin{aligned} n.\phi &= \mathbf{g}_\phi(n) + \mathbf{h}_\phi(n) \\ n.\delta &= \mathbf{g}_\delta(n) + \mathbf{h}_\delta(n) \end{aligned}$$

The formula for calculating ϕ and δ should be modified from the one in Section 2.1 as follows.

1. If n is a tip node

   (b) When game-theoretical value is unknown

$$\begin{aligned} n.\phi &= \mathbf{h}_\phi(n) \\ n.\delta &= \mathbf{h}_\delta(n) \end{aligned}$$

2. If n is an internal node

$$n.\phi = \min_{n_c \in \text{children of } n} (n_{\text{child}}.\delta + \mathbf{cost}_\phi(n, n_{\text{child}}))$$
$$n.\delta = \sum_{n_c \in \text{children of } n} (n_{\text{child}}.\phi + \mathbf{cost}_\delta(n, n_{\text{child}}))$$

The following modification of the 3rd item of df-pn in Section 2.1 leads to the brief algorithm of df-pn$^+$. (The program list of df-pn$^+$ is carried on Appendix B.)

3. At each node n, select the child n_c with minimum $(n_c.\delta + \mathbf{cost}_\phi(n, n_c))$ and the child n_2 with second minimum $(n_2.\delta + \mathbf{cost}_\phi(n, n_2))$. Search below n_c with assigning

$$\begin{aligned} n_c.th_\phi &= n.th_\delta + n_c.\phi \\ &\quad - \textstyle\sum(n_{\text{child}}.\phi + \mathbf{cost}_\delta(n, n_{\text{child}})) \qquad (3) \\ n_c.th_\delta &= \mathbf{min}(n.th_\phi, n_2.\delta + \mathbf{cost}_\phi(n, n_2) + 1) \\ &\quad - \mathbf{cost}_\phi(n, n_c). \qquad (4) \end{aligned}$$

When we have insufficient memory space, then the substitution (3) and (4) should be modified into

$$\begin{aligned} n_c.th_\phi &= n_c.\phi + 1 \\ n_c.th_\delta &= \mathbf{min}(\mathbf{max}(n_2.\delta + \mathbf{cost}_\phi(n, n_2), n.\phi) + 1, \\ &\quad n.th_\phi) - \mathbf{cost}_\phi(n, n_c). \end{aligned}$$

3 Evaluation Function

Strong Othello programs (e.g. LOGISTELLO, HANNIBAL, BRUTUS, ZEBRA, and KEYANO) today use pattern-based evaluation function constructed automatically from the records of tens or hundreds of thousands of actual games. It can be said that this is a kind of learning. For details, see [5] [6] [7] [4].

We constructed an evaluation function for minimax-tree search from 60,000 games played between KITTY and LOGISTELLO (we call it recordA) and 390,000 games played at IOS (we call it recordB). RecordA is available from `ftp://external.nj.nec.com/pub/igord/othello/misc/`. Although the original data contains 100,000 games, we selected 60,000 games by reducing some very

similar games. RecordB is available from `ftp://external.nj.nec.com/pub/igord/othello/ios/`. Although there are 470,000 games in the original data, we selected the games played from the initial position through the end of the game successfully.

The patterns we used are all the configuration of every

- rows and columns (totally 4 types)
- diagonals of length 4 to 8 (totally 5 types)
- 3x3 corner
- 2x4 corner
- parity

with every 52 stages (13 to 64 discs). Each of the configuration has a corresponding value which is taken into account, when evaluating a position, if the configuration appears in the position. Pattern based evaluation function intends to predict the final disc difference of the both players.

There are some characteristic features in our construction.

- We used conjugate gradient method with scaling for the preconditioning instead of steepest descent method.
- Rare configurations makes the evaluation function unreliable. We used recordA mainly and recordB for supplementing positions which include rare configurations. (It means that we used all the positions in recordA corresponding to the 52 stages, but not all in recordB.) Still, there are some rare configurations. For such configurations at the edge that include three or more empty squares in a row, we divided them into two configurations. For the rare configurations at the corner (3x3 and 2x4 corner), we reduced it into smaller configurations. See Figure 1.

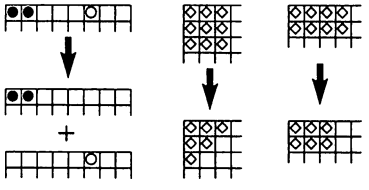

Figure 1: Division of rare configuration (edge) and reduction of rare patterns (3x3 and 2x4 corner)

- We smoothed each configuration values into quartic function as a regression function by the method of least squares. At that time, we used the frequency of the configuration appeared at each stage as a weight of the corresponding value.

4 Applying to Df-pn$^+$

Although the idea corresponding to **h** is already widely used as an initializing function for (dis)proof numbers [2] [14] [3] [9], they requires domain-specific knowledge. (So it is categorized into one of the enhancements.) Tanaka uses a certain function without any parameter [17]. However, we intended to decide **h** automatically from the evaluation function which is mentioned in Section 3 without any domain-specific knowledge.

We define **cost** as

$$\begin{aligned} \mathbf{cost}_\phi(n) &= A \\ \mathbf{cost}_\delta(n) &= 0. \end{aligned}$$

where A is the constant. We tried to decide **cost** from the evaluation function and with the combination of history heuristics. But they resulted in failure. We were unable to think of any effective usage for Othello. Instead, fixing **cost** to a constant was sufficiently effective.

As with $\mathbf{h}(n)$, we defined it as the following sigmoid function.

$$\mathbf{h}_\phi(n) = \frac{B_\phi}{1 + e^{-\mathrm{Eval}(n)/C_\phi}} + 1 \quad (5)$$

$$\mathbf{h}_\delta(n) = \frac{B_\delta}{1 + e^{-\mathrm{Eval}(n)/C_\delta}} + 1 \quad (6)$$

where B and C are the constants. ($C_\phi < 0$, C_δ, B_ϕ, $B_\delta > 0$)

Now we will explain how we decided the values of B and C. First of all, for each of the stages from 47 to 62 discs, we solved 1000 or 2000 problems by the original df-pn. The problems are randomly selected from recordA. When the problem is proved, then by looking at the solution tree, we record the value of the threshold of ϕ at the root (i.e., $r.th_\phi$) which is sufficient to solve the problem, along with the evaluation of that problem (Eval(r)). Similarly, if the problem is disproved, we record the sufficient value of $r.th_\delta$ and Eval(r). The values of $r.th_\phi(r.th_\delta)$ and Eval(r) is used for deciding the values of B_ϕ and C_ϕ (B_δ and C_δ).

From all these data, we decided B and C by the method of least squares, regarding sigmoid function as a regression function. It is done in this way. The purpose is to minimize

$$E = \sum_i \left(\frac{B}{1 + e_i} + 1 - y_i\right)^2$$

where

$$\begin{aligned} e_i &= 1+e^{-\frac{x_i}{C}}, \\ x_i &= \mathrm{Eval}(r), \\ y_i &= r.th. \end{aligned}$$

Since

$$\frac{\partial E}{\partial B}=0 \Leftrightarrow \sum_i\left(\frac{1-y_i}{e_i}+\frac{B}{{e_i}^2}\right)=0$$

$$\Leftrightarrow B=-\sum_i \frac{1-y_i}{e_i}/\sum_i \frac{1}{{e_i}^2} \quad (7)$$

$$\frac{\partial E}{\partial C}=0 \Leftrightarrow \sum_i(x_i(e_i-1)\left(\frac{1-y_i}{{e_i}^2}+\frac{B}{{e_i}^3}\right))=0 \quad (8)$$

Equation (7) is substituted into equation (8) to cancel out B. Since it is difficult to solve equation (8), we decided to solve it by binary search.

Optionally, we devised two ideas. The one is to double the data and the other is to smooth the constants.

- Naively, the data acquired from (dis)proved problems are used only for deciding B_ϕ and C_ϕ (B_δ and C_δ) by regression. However, since B_ϕ is nearly equal to B_δ and C_ϕ is nearly equal to $-C_\delta$, we assume the following two equations.

  $$\begin{aligned} B_\phi &= B_\delta \\ C_\phi &= -C_\delta \end{aligned}$$

  Then each pair of (Eval(r), $r.th_\phi$) acquired from proved problems is modified into ($-$Eval(r), $r.th_\phi$) and is joined with the data acquired from disproved problems. This is done before the regression to sigmoid function. By using all these data, B_δ and C_δ can be obtained.

- Regression to sigmoid function is done for each of the stages from 47 to 62 discs. (That is, B and C depends on #disc.) We thought of smoothing each of the values of B and C into exponential function (Fe^{-Dx}) as an regression function. The procedure is similar to the regression to the sigmoid function. We get simultaneously partial differential equation, substitute one equation to the other, and use binary search.

5 Experimental Results

All the problems used in this experiment is made up from the record of Othello games played at PrincetonII, the computer Othello tournament on IOS (Internet Othello Server) held at October 24 and 25, 1998. Remark that they are different from the games (recordA and recordB) used for constructing evaluation function and parameters of the sigmiod function. We used 100 problems with 16 vacant squares and calculated the average of them. For each of the 100 problems, we solved the problem by df-pn$^+$ which we are focusing on and df-pn as the standard. Then we calculated the ratio of the performance of df-pn$^+$ (from the viewpoint of #nodes visited and memory usage) with each of the problems, and finally acquired the average of them. Therefore, the performance of df-pn is always fixed to 1 in this paper. CPU time is expected to be in proportion to #nodes visited. Memory usage is actually #cells used for transposition table when SiblingGC is in use.

In Section 4, we mentioned two optional ideas (doubling and smoothing) during the regression to sigmoid function. Therefore, there are four possible variants for **h**. Figure 2 shows the ratio of #nodes visited with each of the variants and the case (shown as "only cost") which uses only **cost** and not **h**. **cost**$_\phi$ is fixed to -1. Figure 2 indicates the importance of using **h** especially with large problems, since it is effective to use **h** when #disc is few. Although it is not so clear in Figure 2, Figure 3 which shows the ratio of memory usage with the five cases indicates the necessity of doubling and/or smoothing instead of naive regression.

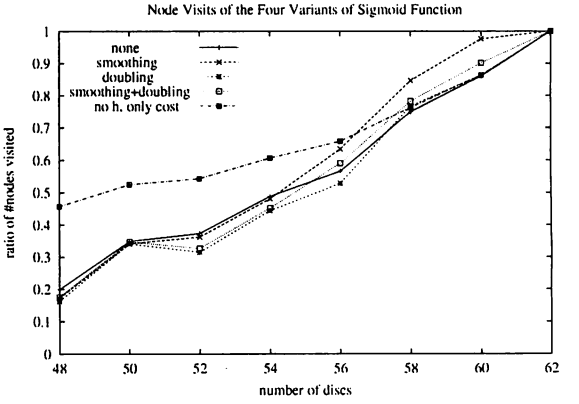

Figure 2: Effect of Function **h** from the viewpoint of #nodes visited

Figure 4 shows the ratio of #nodes visited with the five cases. #disc is fixed to 48. When **cost** is equal to 0, it is equivalent to the case without using **cost**. Although it is only slightly effective to

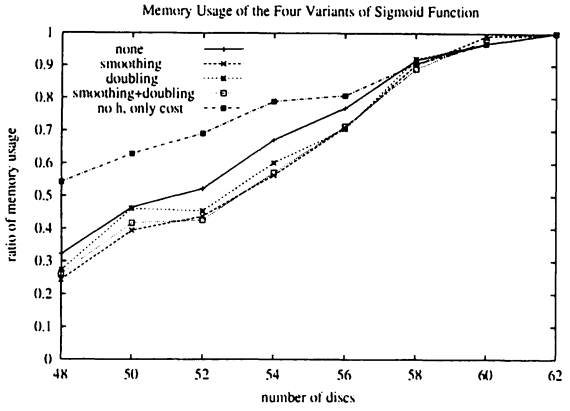

Figure 3: Effect of Function **h** from the viewpoint of memory usage

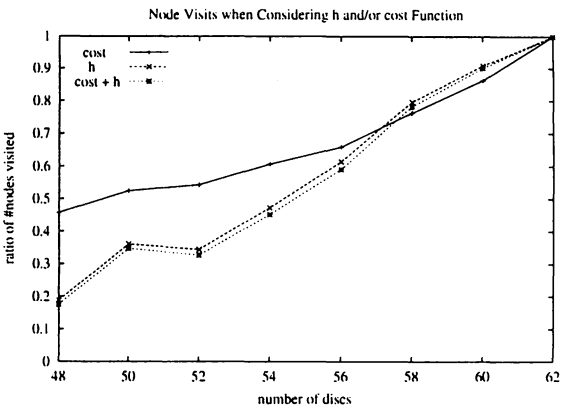

Figure 5: Effect of Both Function **cost** and **h** from the viewpoint of #nodes visited

use **cost** when **h** is used, the case using only **cost** shows that it is effective to use **cost**. It is hard to see from the figure, but the performance of the cases using **h** is getting extremely slightly better when **cost** is fixed to a lower constant. **cost** must be decided carefully when using only **cost**.

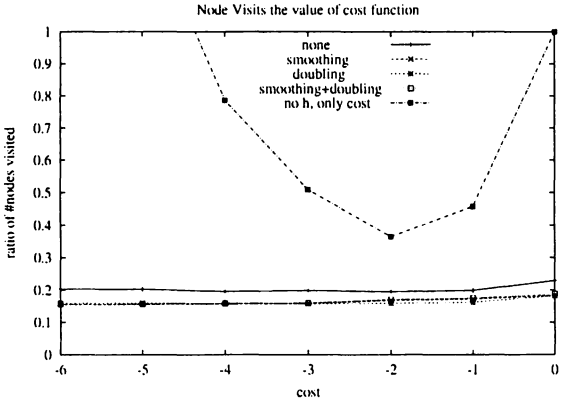

Figure 4: Effect of Function **cost** from the viewpoint of #nodes visited

Figure 5 shows the ratio of #nodes visited with the cases using **h** and/or **cost**. When both of them are not used, df-pn$^+$ is equal to df-pn which shows the performance of 1. Both doubling and smoothing are used during the regression to sigmoid function. Although the case using both **cost** and **h** is slightly better than the case using only **h**, we can easily see that it is the most effective to use both of them when problem size is large enough.

Figure 2 shows that df-pn$^+$ is very effective to solve large problems when **h** is a sigmoid function with both doubling and smoothing in use and when **cost** is fixed to -1. On average, compared with df-pn, #nodes visited is less than half when #disc is 54, less than $\frac{1}{3}$ when #disc is 52, and nearly $\frac{1}{6}$ when #disc is 48. We can expect that the efficiency will become still better when the problem size is larger. Therefore, CPU time is expected to reduce in proportion to #nodes visited.

Othello has a unique feature that most of the terminal nodes locates at a certain depth and that the branching factor gets smaller near the terminal nodes. As a result, there is a tendency for search algorithm to become effective when it search deep to some degree whenever a move decision is made. Df-pn$^+$ is effective since **cost** and **h** was decided to satisfy this feature.

6 Conclusion

A new search algorithm df-pn$^+$ is proposed by extending df-pn. For function **h**, we adopted sigmoid function as regression function and decided the parameter by the method of least squares. For function **h**, we used a constant (-1 is sufficiently effective). Df-pn$^+$ constructed in this way is very efficient especially when the problem size is large. Experimental results on Othello endgames show that #nodes visited is reduced by a factor $\frac{1}{6}$ on average compared with the original df-pn when #disc is 48.

7 Acknowledgements

I would like to thank Eiji Tsuchida for having a fruitful discussion with him and providing some important information for me.

References

[1] Louis V. Allis, Maarten van der Meulen, and H. Jaap van den Herik. *Proof-Number Search.* Report CS 91-01, University of Limburg, Maastricht, Netherlands, 1991. Also available at *Artificial Intelligence*, Vol.66, pp. 91–124, 1994.

[2] Louis V. Allis. *Searching for Solutions in Games and Artificial Intelligence.* Ph.D. Thesis, Department of Computer Science, University of Limburg, Netherlands, 1994.

[3] Dennis M. Breuker, Louis V. Allis, and H. Jaap van den Herik. How to Mate: Applying Proof-Number Search. *Advances in Computer Chess*, Vol. 7, pp. 251–272, 1994.

[4] Mark Brockington. KEYANO Unplugged – The Construction of an Othello Program. Technical Report 97-05, Department of Computing Science, University of Alberta. Available at `http://www.cs.ualberta.ca/~games/keyano/`.

[5] Michael Buro. An Evaluation Function for Othello Based on Statistics. NECI Technical Report #31, 1997.

[6] Michael Buro. Experiments with Multi-ProbCut and a New High-Quality Evaluation Function for Othello. *Workshop on game-tree search*, NECI, August 1997. Also available as NECI Technical Report #96, 1997 at `http://www.neci.nj.nec.com/homepages/mic/publications.html`.

[7] Michael Buro. From Simple Features to Sophisticated Evaluation Functions. *The First International Conference on Computers and Games (CG'98)*, Tsukuba, Japan. To be published in a forthcoming LNCS, Springer-Verlag.

[8] Charles Elkan. Conspiracy Numbers and Caching for Searching And/Or Trees and Theorem-Proving. *Proceedings IJCAI-89*, pp. 341–346, 1989.

[9] Jacco Gnodde. *Aïda, New Search Techniques Applied to Othello.* M.Sc. Thesis, University of Leiden, Netherlands, 1993.

[10] Ayumu Nagai. *A new Depth-First-Search Algorithm for AND/OR Trees.* M.Sc. Thesis, Department of Information Science, University of Tokyo, Japan, 1999.

[11] Ayumu Nagai. Proof for the Equivalence Between Some Best-First Algorithms and Depth-First Algorithms for AND/OR Trees. *KOREA-JAPAN Joint Workshop on Algorithms and Computation*, pp. 163–170, 1999.

[12] Nils J. Nilson. *Principles of Artificial Intelligence.* Tioga Publishing Company, Palo Alto, CA, 1980.

[13] Aske Plaat, Jonathan Schaeffer, Wim Pijls, and Arie de Bruin. *A new Paradigm for Minimax Search.* Technical Report TR 94-18, Department of Computing Science, University of Alberta, Canada, 1994.

[14] Martin Schijf. *Proof-Number Search and Transpositions.* M.Sc. Thesis, University of Leiden, Netherlands, 1993.

[15] Masahiro Seo. *The C* Algorithm for AND/OR Tree Search and its Application to a Tsume-Shogi Program.* M.Sc. Thesis, Department of Information Science, University of Tokyo, Japan, 1995.

[16] George C. Stockman. A Minimax Algorithm Better than Alpha-Beta? *Artificial Intelligence*, Vol. 12, pp. 179–196, 1979.

[17] Seiichi Tanaka, Hiroyuki Iida, and Yoshiyuki Kotani. An Approach to Tsume-Shogi: Applying Proof-Number Search with Estimation Function of Mating. (In Japanese) *Game Programming Workshop in Japan '95*, pp. 138–147, 1995.

A Program List of Df-pn

The program list of df-pn is carried below. As ϕ and δ at each node are dual to each other, an algorithm corresponding to negamax algorithm in the context of minimax-tree search can be constructed.

```
1   // Iterative deepening at the root
2   procedure Nega-df-pn-search(r) {
3     r.φ = ∞;   r.δ = ∞;
4     MID(r);
5   }
6   // Explore node n
7   procedure MID(n) {
8     // 1. Look up the transposition table
9     LookUpTransTable(n, φ, δ);
10    if (n.φ ≤ φ || n.δ ≤ δ) {
11      n.φ = φ;   n.δ = δ;
12      return;
13    }
14    // 2. Generate all the legal moves
15    if (n is a terminal node) {
16      if ((n is an AND node &&
17           Eval(n) == true) ||
18          (n is an OR node &&
19           Eval(n) == false)) {
20        n.φ = ∞;   n.δ = 0;
21      }else {n.φ = 0;   n.δ = ∞; }
22      PutInTransTable(n, n.φ, n.δ);
23      return;
24    }
25    GenerateLegalMoves();
26    // 3. Avoid cycles by using transp. table
```

```
27      PutInTransTable(n, n.φ, n.δ);
28      // 4. Multiple iterative deepening
29      while (1) {
30          // Terminate if either φ or δ
31          // is at least its threshold
32          if (n.φ ≤ ΔMin(n) || n.δ ≤ ΦSum(n)) {
33              n.φ = ΔMin(n);    n.δ = ΦSum(n);
34              PutInTransTable(n, n.φ, n.δ);
35              return;
36          }
37          δ_c = φ;
38          n_c = SelectChild(n, φ_c, δ_c, δ_2);
39          n_c.φ = n.δ + φ_c − ΦSum(n);
40          n_c.δ = min(n.φ, δ_2+1);
41          MID(n_c);
42      }
43  }
44  // Selection among the children
45  procedure SelectChild(n, &φ_c, &δ_c, &δ_2) {
46      δ_bound = δ_c;
47      δ_c = ∞;    δ_2 = ∞;
48      for (each child n_child) {
49          LookUpTransTable(n_child, φ, δ);
50          if (φ ≠ ∞) δ = max(δ, δ_bound);
51          if (δ < δ_c) {
52              n_best = n_child;
53              δ_2 = δ_c;    φ_c = φ;    δ_c = δ;
54          }else if (δ < δ_2) δ_2 = δ;
55          if (φ == ∞) return n_best;
56      }
57      return n_best;
58  }
59  // Look up trans. table for the entry of n
60  procedure LookUpTransTable(n, &φ, &δ) {
61      if (n is recorded) {
62          φ = Table[n].φ;    δ = Table[n].δ;
63      }else {φ = 1;    δ = 1; }
64  }
65  // Record into transposition table
66  procedure PutInTransTable(n, φ, δ) {
67      Table[n].φ = φ;    Table[n].δ = δ;
68  }
69  // Calculate minimum δ among n's children
70  procedure ΔMin(n) {
71      min = ∞;
72      for (each child n_child) {
73          LookUpTransTable(n_child, φ, δ);
74          min = min(min, δ);
75      }
76      return min;
77  }
78  // Calculate the sum of φ of n's children
79  procedure ΦSum(n) {
80      sum = 0;
81      for (each child n_child) {
82          LookUpTransTable(n_child, φ, δ);
83          sum = sum + φ;
84      }
85      return sum;
86  }
```

B Program List of Df-pn$^+$

Df-pn$^+$ is acqired by modifying the program in Appendix A as follows.

```
44  // Selection among the children
45  procedure SelectChild(n, &φ_c, &δ_c, &δ_2) {
46      δ_bound = δ_c;
47      min = ∞;    min_2 = ∞;
48      for (each child n_child) {
49          LookUpTransTable(n_child, φ, δ);
50          if (φ ≠ ∞)
                δ= max(δ, δ_bound − cost_φ(n, n_child));
51          if (δ + cost_φ(n, n_child) < min) {
52              n_best = n_child;
                min_2 = min;
                min = δ + cost_φ(n, n_child);
53              δ_2 = δ_c;    φ_c = φ;    δ_c = δ;
54          }else if (δ + cost_φ(n, n_child) < min_2) {
                δ_2 = δ;    min_2 = δ + cost_φ(n, n_child);
            }
55          if (φ == ∞) return n_best;
56      }
57      return n_best;
58  }
59  // Look up trans. table for the entry of n
60  procedure LookUpTransTable(n, &φ, &δ) {
        ...
63      }else {φ = h_φ(n);    δ = h_δ(n); }
64  }
69  // Calculate minimum δ among n's children
70  procedure ΔMin(n) {
            ...
74          min = min(min, δ + cost_φ(n, n_child));
            ...
77  }
78  // Calculate the sum of φ of n's children
79  procedure ΦSum(n) {
            ...
83          sum = sum + φ + cost_δ(n, n_child);
            ...
86  }
```