PAPER

Proof for the Equivalence between Some Best-First Algorithms and Depth-First Algorithms for AND/OR Trees

Ayumu NAGAI[†], *Nonmember and* **Hiroshi IMAI**[††], *Regular Member*

SUMMARY When we want to know if it is a win or a loss at a given position of a game (e.g. chess endgame), the process to figure out this problem corresponds to searching an AND/OR tree. AND/OR-tree search is a method for getting a proof solution (win) or a disproof solution (loss) for such a problem. AO* is well-known as a representative algorithm for searching a proof solution in an AND/OR tree. AO* uses only the idea of proof number. Besides, Allis developed pn-search which uses the idea of proof number and disproof number. Both of them are best-first algorithms. There was no efficient depth-first algorithm using (dis)proof number, until Seo developed his originative algorithm which uses only proof number. Besides, Nagai recently developed PDS which is a depth-first algorithm using both proof number and disproof number. In this paper, we give a proof for the equivalence between AO* which is a best-first algorithm and Seo's depth-first algorithm in the meaning of expanding a certain kind of node. Furthermore, we give a proof for the equivalence between pn-search which is a best-first algorithm and df-pn which is a depth-first algorithm we propose in this paper.

key words: *AND/OR tree, AO*, pn-search, df-pn*

1. Introduction

Alpha-Beta has been the most popular algorithm for searching minimax trees, even though there are more efficient algorithms (e.g., SSS* [21]) from a viewpoint of the number of nodes visited. There are two reasons for this. First, since Alpha-Beta has an advantage that they are adaptable, it is easy to improve its efficiency [16] by various enhancements, such as history heuristics [15], transposition table [20], iterative deepening [20], etc. Second, as Alpha-Beta is a depth-first algorithm, it uses less memory space and its actual search speed is often faster. To be specific, it can run under memory space of $O(d)$, where d is the search depth. However, best-first algorithms, such as SSS*, must preserve in memory all the explored positions and the whole information that each of the position has. It requires large memory space of $O(w^{\frac{d}{2}})$ [14], where w is the branching factor, or the average number of moves each position has. To put it simply, the naive best-first algorithm SSS* may search efficiently than the naive depth-first algorithm Alpha-Beta with no enhancements from a viewpoint of the number of nodes visited when we have sufficient memory space. However, Alpha-Beta with some enhancements is easier to handle and may search faster than SSS* in actual execution time, since 1) there are very few enhancements for SSS* and since 2) actually it is common that we only have insufficient memory space. Moreover, SSS* is an algorithm for searching fixed-depth tree. Therefore, most of the top level programs have adopted depth-first algorithms, such as Deep Thought (predecessor of Deep Blue) [5] which is a top level chess machine, Logistello [3] which is a top level Othello program, and Chinook [17] which is a top level Checkers program. Similarly, also with AND/OR-tree search, depth-first algorithms are surely more desirable than best-first algorithms, since depth-first algorithms are so adaptive that many enhancements are useful. As mentioned later, df-pn, the algorithm we propose in this paper, is a depth-first algorithm.

There is a trend to use a depth-first algorithm so as to behave the same as a best-first algorithm. For example, IDA* is a depth-first algorithm for single-agent game trees which behaves the same as the best-first algorithm, or A*. Another example is MT-SSS* (MT-DUAL*) [13]. MT-SSS* (MT-DUAL*) is a depth-first algorithm for minimax trees which behaves the same as the best-first algorithm, or SSS* (DUAL*). Based on these studies, Plaat developed MTD(f) [13] for minimax trees. Although Scout algorithm [12] is still a most widely used algorithm, MTD(f) is a significant algorithm not only from the viewpoint of practical use but also from the viewpoint of its basic principle. That is, a depth-first algorithm, which is constructed so as to behave the same as a corresponding best-first algorithm, searches as efficient as the best-first algorithm, and at the same time, is so adaptive that many enhancements are useful. As mentioned later, df-pn, the algorithm for AND/OR trees we propose in this paper, is a depth-first algorithm that behaves the same as pn-search which is a best-first algorithm.

Each node of a minimax tree may be evaluated to any integer (or even real number) [10]. A minimax tree is a model of a two-player zero-sum game with per-

Manuscript received April 16, 2001.
Manuscript revised September 21, 2001.

[†]The author is now with NASDA. This work was done while the author was with the Department of Information Science, The University of Tokyo, Tokyo, 113–0033 Japan.
[††]The author is with the Department of Computer Science, The University of Tokyo, Tokyo, 113–0033 Japan.

Table 1 Relation of the four algorithms.

	information using	
	only proof number	proof number and disproof number
best-first	AO* [11]	pn-search [2]
depth-first	Seo's algorithm [18]	**df-pn**

fect information [1], i.e. Chess and Othello. A minimax tree such that each of the node value ultimately falls on either of the two values (i.e., **true** and **false**, normally standing for win and lose, respectively) is called an AND/OR tree. An AND/OR tree is a model of an endgame of a two-player zero-sum game with perfect information. AO* [11] is a representative algorithm for AND/OR-tree search and is intensively studied. In this paper, we let AO* indicate a limited version, proposed by Elkan [4], of the original AO*. Allis developed a new algorithm (pn-search [2]) which uses both proof number and disproof number. Pn-search may be said that it is a refined version of AO*. Since pn-search is an elegant algorithm and is easy to understand, it attracts a great deal of attention. Both AO* and pn-search are best-first algorithms. There was a predominant preconception that there are no efficient depth-first algorithm for AND/OR trees.

However, Seo accomplished a significant breakthrough and developed a new depth-first algorithm [18]. His depth-first algorithm behaves in the same way as AO* in the meaning of always expanding a certain kind of node. This fact was mentioned in several places such as [6], but no rigorous statement and proof has been made as far as we know (see Table 1). This is accomplished by the idea of Multiple Iterative Deepening, which is a series of iterative deepening performed not only at the root but also at some internal nodes. Moreover, df-pn, which is a depth-first algorithm using both proof and disproof numbers we will propose in this paper, behaves the same as pn-search, which is a best-first algorithm, in the meaning of always expanding a certain kind of node (see Table 1). This is accomplished by the idea of Multiple Iterative Deepening at all nodes. Our purpose is to give rigorous proofs that these two pairs of search algorithms behave in the same way under the assumption that sufficient memory is available (Sect. 2 and Sect. 3). This type of equivalence is very important because of the adaptability and extendibility of depth-first algorithms. That is, if a best-first algorithm and a depth-first algorithm behave in the same way, it is certainly better to adopt the depth-first algorithm, because many enhancements can be used efficiently, as mentioned in the first two paragraphs of this introduction. Finally we will have a conclusion (Sect. 4).

2. Algorithms Using only Proof Number

The purpose of this section is to 1) explain AO* and Seo's algorithm and 2) give a proof that the two algorithms behave in the same way.

An OR node is a position with the first player's turn and an AND node is a position with the second player's turn. A leaf node is a node at the tip (or an unexpanded node) of the current search tree and an internal node is a node which is not a leaf node (or an already expanded node). A terminal node is a leaf node unable to expand.

2.1 Original AO*

AO* is a best-first algorithm using the following two kinds of information at each node.

$\mathbf{g}(n)$: cost incurred so far from the root to node n
$\mathbf{h}(n)$: estimation of the cost from n to any solution

Definition 1: $\mathbf{g}$, $\mathbf{h}$ are defined recursively in the following way.

- When n is a leaf node

$$\mathbf{g}(n) = 0$$
$$\mathbf{h}(n) = \begin{cases} 0 & (n \in \text{ solution}) \\ \text{estimation} & (n \notin \text{ solution}) \end{cases}$$

  That is, $\mathbf{h}$ may be some kind of heuristic evaluation function.
- When n is an internal *OR* node

$$\mathbf{g}(n) = \mathbf{cost}(n, n_c) + \mathbf{g}(n_c)$$
$$\mathbf{h}(n) = \mathbf{h}(n_c)$$

  where

$$n_c \in \{\, x \mid x \in \text{children of } n \wedge \forall n_{\text{child}} \in \text{children of } n, \mathbf{g}(x) + \mathbf{cost}(n, x) + \mathbf{h}(x) \leq \mathbf{g}(n_{\text{child}}) + \mathbf{cost}(n, n_{\text{child}}) + \mathbf{h}(n_{\text{child}}) \}$$

  To put it simply, n_c is a child with minimum $\mathbf{g} + \mathbf{cost} + \mathbf{h}$. $\mathbf{cost}(m, n)$ is the cost from m to n. Actually, **cost** may be some kind of heuristic evaluation function.
- When n is a internal *AND* node

$$\mathbf{g}(n) = \sum_{n_c \in \text{children of } n} \Big(\mathbf{cost}(n, n_c) + \mathbf{g}(n_c) \Big)$$
$$\mathbf{h}(n) = \sum_{n_c \in \text{children of } n} \mathbf{h}(n_c)$$

2.2 AO* in This Paper (Elkan's Algorithm) [4]

AO* is originally an AND/OR graph-search algorithm which uses $\mathbf{g}$ and $\mathbf{h}$ (or **cost** and $\mathbf{h}$ from the viewpoint of evaluation function it uses). Strictly speaking, they are different from proof number defined by Definition 3. However, if **cost** and $\mathbf{h}$ are defined in the following way,

then AO* will become an algorithm incorporating proof number. To put it concretely, by defining **cost** and **h** in this way, no evaluation function is needed.

Definition 2: **cost** and **h**

$$\forall n,\ \mathbf{cost}(n, n_c) = 0$$

$$\forall n \in \text{leaf node},\ \mathbf{h}(n) = \begin{cases} 0 & (n \in \text{ solution}) \\ \infty & (n \text{ is unsolvable}) \\ 1 & (n \text{ otherwise}) \end{cases}$$

This limited version of the original AO* is the same as the algorithm which Elkan proposed in [4]. In this paper, we let AO* indicate this limited version.

Before showing that this is an algorithm using proof number, we define proof number.

Definition 3: $\mathbf{pn}(n)$: Proof number of a node n

1. When n is a leaf node
   a. When the value is **true** (n is a solution)
      $$\mathbf{pn}(n) = 0$$
   b. When the value is **false** (n is not a solution)
      $$\mathbf{pn}(n) = \infty$$
   c. When the value is unknown yet
      $$\mathbf{pn}(n) = 1$$
2. When n is an internal node
   a. When n is an OR node
      $$\mathbf{pn}(n) = \min_{n_c \in \text{children of } n} \mathbf{pn}(n_c)$$
   b. When n is an AND node
      $$\mathbf{pn}(n) = \sum_{n_c \in \text{children of } n} \mathbf{pn}(n_c)$$

Intuitively speaking, proof number indicates the least number of leaf nodes, whose winning solution (for the first player) must be found in order to find the winning solution for the root position. If proof number of a node is large, the effort to find winning solution becomes large. When proof number is 0, the value is **true**, meaning that winning solution is already found. When proof number is ∞, the value is **false**, meaning that there is no winning solution, that is, the first player loses the game.

Now it is easily shown that AO* with **cost** and **h** defined by Definition 2 is the algorithm using proof number, since $\mathbf{g} = 0$ and $\mathbf{h} = \mathbf{pn}$.

We define a most-proving node in the context of AO*. The smaller proof number of a node is, the larger the possibility of a winning solution to be found within less effort becomes. Therefore, the first player is interested in the child node with small proof number. Most-proving node in the context of AO* is defined as follows.

Definition 4: Most-proving node in the context of AO* is a leaf node selected by tracing from the root in the following way.

- For each OR node, trace the child with minimum proof number.
- For each AND node, trace any child with non-zero proof number.

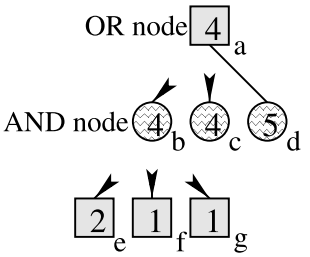

Fig. 1 Selection of most-proving node as with AO* (numbers are the proof numbers of corresponding nodes). c, e, f, g are most-proving nodes in the context of AO*.

As there is arbitrariness at each AND node, and as there is arbitrariness at each OR node when there are more than one child with minimum proof number, most-proving node is not necessarily uniquely determined. (See Fig. 1.)

After all, AO* is a best-first algorithm using proof number. Its search procedure is as following:

Procedure AO*:

step 1) Select a most-proving node by following Definition 4.
step 2) Expand the most-proving node. Calculate proof number of all the generated children by following Definition 3.
step 3) Update proof numbers of all the nodes on the path from the most-proving node up to the root.

Repeat this until the proof number of the root becomes 0 or ∞.

Theorem 1 (Characteristic of AO*): When expanding a leaf node in AO*, the leaf node is always one of most-proving nodes defined by Definition 4.

Proof. Procedure AO* shows that it is obvious that AO* always expands a most-proving node defined by Definition 4. □

The concept of AO* is that most-proving node defined by Definition 4 is always expanded because AO* regards a most-proving node as the nearest node to the solution. AO* is a best-first algorithm because of its manner in selecting the most-proving node.

2.3 Seo's Algorithm [18]

Seo's algorithm is a depth-first algorithm using proof number. Each node n has a threshold $\mathbf{th}_{\mathrm{pn}}(n)$.

Procedure Seo's Algorithm:

Assign $\mathbf{th}_{\mathrm{pn}}(r) = 2$ where r is the root, since ini-

tially $\mathbf{pn}(r)$ is 1.

step 1) At each node n, the search process continues to search below n until $\mathbf{pn}(n)$ defined by Definition 3 satisfies $\mathbf{pn}(n) = 0$ or $\mathbf{pn}(n) \geq \mathbf{th}_{\mathrm{pn}}(n)$ (we call it ending condition).

step 2) When n is an OR node, all the children n_{child} is searched sequentially with assigning its threshold $\mathbf{th}_{\mathrm{pn}}(n_{\mathrm{child}}) = \mathbf{th}_{\mathrm{pn}}(n)$ if ending condition at n_{child} is not satisfied (precisely, if $\mathbf{pn}(n_{\mathrm{child}}) \neq 0$ and $\mathbf{pn}(n_{\mathrm{child}}) < \mathbf{th}_{\mathrm{pn}}(n_{\mathrm{child}}) = \mathbf{th}_{\mathrm{pn}}(n)$).

step 3) When n is an AND node, select any child with non-zero proof number and search below it with assigning $\mathbf{th}_{\mathrm{pn}}(n_{\mathrm{child}}) = \mathbf{pn}(n_{\mathrm{child}}) + 1$. This process is iterated (iterative deepening) until the ending condition of n is satisfied.

This kind of iteration is called **Multiple Iterative Deepening**, since iterative deepening is performed not only at the root but also at all AND nodes. The search process goes deeper until satisfying the ending condition. That is, it is a depth-first algorithm iterating on thresholds, instead of depths.

step 4) If the ending condition is satisfied, the search process returns to the parent node of n. If n is the root, then assign $\mathbf{th}_{\mathrm{pn}}(r) = \mathbf{pn}(r) + 1$.

Iterate this whole process until $\mathbf{pn}(r) = 0$ or $\mathbf{pn}(r) = \infty$ with assigning $\mathbf{th}_{\mathrm{pn}}(r) = \mathbf{pn}(r) + 1$ each time.

Now we explain the search process of Seo's algorithm and Multiple Iterative Deepening by using the example shown in Fig. 2. Assume the situation that the search process is at the root a after some iterations. Assume still more that proof number of the root $\mathbf{pn}(a)$ is 9 and its threshold $\mathbf{th}_{\mathrm{pn}}(a)$ is assigned to 10. Then, the search process goes under b which has the minimum proof number with assigning $\mathbf{th}_{\mathrm{pn}}(b) = 10$. At b, assume that the search process goes under e with assigning $\mathbf{th}_{\mathrm{pn}}(e) = 3$, and that e was solved causing $\mathbf{pn}(e)$ to became 0. Assume that f was selected for the forthcoming search process. If we do not use Multiple Iterative Deepening, we can assign $\mathbf{th}_{\mathrm{pn}}(f) = 6$, because $\mathbf{th}_{\mathrm{pn}}(b) = 10$ and $\mathbf{pn}(g) = 4$. However, by using Multiple Iterative Deepening, we look at $\mathbf{pn}(f)$ being 3 and assign $\mathbf{th}_{\mathrm{pn}}(f) = 4$. If f still can not be solved, and if $\mathbf{pn}(f) \geq 6$ after searching f, then the search process returns to a, since the ending condition of b satisfies. Otherwise, if $\mathbf{pn}(f) = 4$ (5), then f is searched iteratively by assigning $\mathbf{th}_{\mathrm{pn}}(f) = 5$ (6). In this way, at an internal OR node n (not only the root), assignment of its threshold is $\mathbf{th}_{\mathrm{pn}}(n) = \mathbf{pn}(n) + 1$, and the nodes in the range of that threshold is searched. The reason why $\mathbf{th}_{\mathrm{pn}}(f) = 6$ is not assigned from the beginning is as follows. If $\mathbf{th}_{\mathrm{pn}}(f) = 6$ is assigned in the case where actually f can be solved with assigning $\mathbf{th}_{\mathrm{pn}}(f) = 4$, the solution is certainly found, but with many additional nodes expanded. In order to minimize the number of nodes expanded, the threshold is assigned $\mathbf{th}_{\mathrm{pn}}(n) = \mathbf{pn}(n) + 1$.

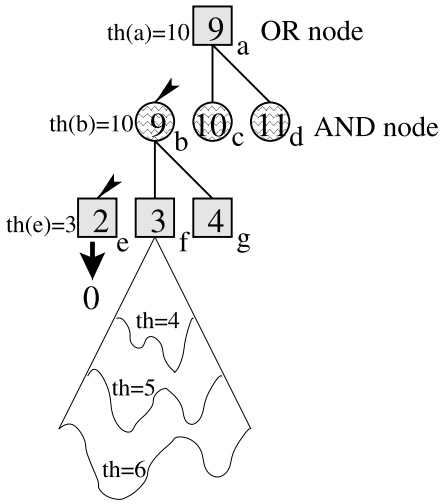

Fig. 2 Seo's algorithm uses multiple iterative deepening (the numbers are the proof numbers).

By adopting Multiple Iterative Deepening, the following lemma holds. (Sufficient memory space is assumed.)

Lemma 2: When the search process goes from a parent node to its child n, the relation between $\mathbf{pn}(n)$ and $\mathbf{th}_{\mathrm{pn}}(n)$ is as follows.

$$\mathbf{pn}(n) + 1 = \mathbf{th}_{\mathrm{pn}}(n)$$

Proof. We show it by proof by contradiction. Assume that

$$\mathbf{pn}(n_c) + 1 \neq \mathbf{th}_{\mathrm{pn}}(n_c) \tag{1}$$

is satisfied when search process goes from a parent n to its child n_c. Step 3) of Procedure Seo's algorithm shows that n is not an AND node, so n_c is an AND node. Expression (1) shows that either of the following two expression holds.

$$\mathbf{pn}(n_c) + 1 > \mathbf{th}_{\mathrm{pn}}(n_c) \tag{2}$$

$$\mathbf{pn}(n_c) + 1 < \mathbf{th}_{\mathrm{pn}}(n_c) \tag{3}$$

When Expression (2) holds, the search process does not go toward n_c, since the ending condition at n_c is already satisfied. There is no such case. Hence Expression (3) holds. As n is an OR node, $\mathbf{pn}(n)$ is equal to the minimum among proof numbers of its children. Then the following expression holds.

$$\mathbf{pn}(n) \leq \mathbf{pn}(n_c) \tag{4}$$

Step 2) of Procedure Seo's algorithm shows

$$\mathbf{th}_{\mathrm{pn}}(n) = \mathbf{th}_{\mathrm{pn}}(n_c). \tag{5}$$

Expressions (3),(4),(5) lead to

$$\mathbf{pn}(n)+1 \leq \mathbf{pn}(n_c)+1 < \mathbf{th}_{\mathrm{pn}}(n_c) = \mathbf{th}_{\mathrm{pn}}(n). \tag{6}$$

However, n is an OR node. Step 3) of Procedure Seo's algorithm shows that

$$\mathbf{pn}(n) + 1 = \mathbf{th}_{\mathrm{pn}}(n). \tag{7}$$

Expressions (6) and (7) conflicts each other. Therefore, Lemma 2 is proved by proof by contradiction. □

Lemma 3: When the search process goes from a parent OR node to its child AND node, the child node is the child with minimum proof number among its sibling nodes.

Proof. Because of Lemma 2 and step 2) of Procedure of Seo's algorithm,

$$\mathbf{pn}(n_c) + 1 = \mathbf{th}_{\mathrm{pn}}(n_c) = \mathbf{th}_{\mathrm{pn}}(n) \tag{8}$$

when the search process goes from the parent OR node n to its child AND node n_c. Assume that n_i ($\neq n_c$) is any child of n. If n_i satisfies its ending condition,

$$\mathbf{pn}(n_i) \geq (\mathbf{th}_{\mathrm{pn}}(n_i)) = \mathbf{th}_{\mathrm{pn}}(n). \tag{9}$$

(If $\mathbf{pn}(n_i) = 0$, then ending condition of n is satisfied, so n_c is not searched.) If n_i does not satisfy its ending condition, Lemma 2 and step 2) of Seo's algorithm lead to

$$\mathbf{pn}(n_i) + 1 = (\mathbf{th}_{\mathrm{pn}}(n_i)) = \mathbf{th}_{\mathrm{pn}}(n). \tag{10}$$

In either case of Expressions (9) or (10), Expression (8) leads to

$$\begin{aligned}\mathbf{pn}(n_i) &\geq (\mathbf{th}_{\mathrm{pn}}(n_i) - 1) = \mathbf{th}_{\mathrm{pn}}(n) - 1 \\ &= \mathbf{th}_{\mathrm{pn}}(n_c) - 1 = \mathbf{pn}(n_c).\end{aligned}$$

It means that proof number of n_c is the minimum among all its sibling node, although n_c may not be the only node. □

Theorem 4 (Seo's algorithm): When expanding a leaf node along Seo's algorithm, the leaf node is always one of most-proving nodes defined by Definition 4.

Proof. In order to show a leaf node selected by Seo's algorithm satisfies Definition 4 (definition of most-proving node), all the nodes on the path from the root to the leaf node must be checked. Definition 4 essentially does not claim anything at each AND node. We only need to check the selection of a child n_c at each OR node n. Assume that $\mathbf{pn}(n_c)$ is the proof number of n_c at the moment when the search process went to n_c. Assume that $\mathbf{pn}_{\mathrm{current}}(n_c)$ is the current proof number of n_c. Because of Lemma 3,

$$\mathbf{pn}(n_c) \leq \mathbf{pn}(n_i) \tag{11}$$

where n_i is any sibling node of n_c. While searching below n_c, since n_c does not satisfy its ending condition, proof number of n_c may be smaller than the proof number of n_c at the moment search process went to n_c. Therefore,

$$\mathbf{pn}_{\mathrm{current}}(n_c) \leq \mathbf{pn}(n) = \mathbf{th}_{\mathrm{pn}}(n) - 1. \tag{12}$$

Expressions (11) and (12) lead to

$$\mathbf{pn}_{\mathrm{current}}(n_c) \leq \mathbf{pn}(n_i).$$

It means that n_c is the child with minimum proof number among its sibling nodes, satisfying Definition 4. Therefore, the leaf node selected by Seo's algorithm is one of most-proving nodes defined by Definition 4. □

Note that Lemma 2, Lemma 3, and Theorem 4 hold if Multiple Iterative Deepening is in use. When Multiple Iterative Deepening is not in use, to put it concretely, it means that at Fig. 2, $\mathbf{th}_{\mathrm{pn}}(f) = 6$ is assigned by skipping 4 and 5. Then Lemma 2 does not hold. As Lemma 3 and Theorem 4 requires Lemma 2, they do not hold either. Then the equivalence between AO* and Seo's algorithm (Theorem 5), which is the purpose of this section, will be broken.

2.4 Equivalence between AO* and Seo's Algorithm

Theorem 5 (Equivalence between AO* and Seo's algorithm): Seo's algorithm behaves the same as AO* in the meaning that Seo's algorithm always expands most-proving node defined by Definition 4.

Proof. Because of Theorems 1 and 4, both AO* and Seo's algorithm always expands one of most-proving nodes defined by Definition 6. □

The feature of AO* is summarized in Theorem 1. That is, AO* always expands a most-proving node defined by Definition 4. Theorem 4 asserts that Seo's algorithm also always expands a most-proving node. In this meaning, Seo's algorithm behaves the same as AO*.

3. Algorithms Using Both Proof Number and Disproof Number

Allis defined disproof number in the following way [2].

Definition 5: $\mathbf{dn}(n)$: Disproof number of a node n

1. When n is a leaf node
   a. When the value is **true** (n is a solution)
      $$\mathbf{dn}(n) = \infty$$
   b. When the value is **false** (n is not a solution)
      $$\mathbf{dn}(n) = 0$$
   c. When the value is unknown yet
      $$\mathbf{dn}(n) = 1$$
2. When n is an internal node
   a. When n is an OR node
      $$\mathbf{dn}(n) = \sum_{n_c \in \text{children of } n} \mathbf{dn}(n_c)$$

b. When n is an AND node

$$\mathbf{dn}(n) = \min_{n_c \in \text{children of } n} \mathbf{dn}(n_c)$$

Intuitively speaking, disproof number indicates the least number of leaf nodes, whose losing solution (for the first player) must be found in order to find the losing solution for the root position. If disproof number of a node is large, the effort to find losing solution becomes large. When disproof number is 0, the value is **false**, meaning that losing solution is already found. (In that case, proof number is ∞.) When disproof number is ∞, the value is **false**, meaning that there is no losing solution, that is, we win the game. (In that case, proof number is 0.)

As it is obvious by comparing Definition 3 and 5, a proof number and a disproof number are dual to each other.

The purpose of this section is to explain pn-search and df-pn and give a proof that the two algorithms behave in the same way.

3.1 Pn-search (Allis' Algorithm)

We already mentioned that the first player is interested in the child node with small proof number. Similarly, the smaller disproof number of a node is, the larger the possibility of a losing solution to be found within less effort becomes. Therefore, the second player is interested in the child node with small disproof number. Most-proving node in the context of pn-search is defined as follows.

Definition 6: Most-proving node in the context of pn-search [2] is a leaf node selected by tracing from the root in the following way.

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

As there is arbitrariness at each OR (AND) node when there is more than one child with minimum (dis)proof number, most-proving node is not necessarily uniquely determined. (See Fig. 3.)

The difference between most-proving node for AO* (Definition 4) and most-proving node for pn-search (Definition 6) is the handling at AND nodes. As it is obvious by Definitions 4 and 6, a most-proving node in the context of pn-search is also a most-proving node in the context of AO*. (The converse does not hold.) Therefore, by using disproof number, we can narrow significant nodes properly.

Pn-search is a naive best-first algorithm using both proof number and disproof number.

Procedure pn-search:

step 1) Select a most-proving node by following Definition 6.

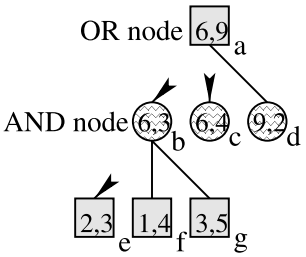

Fig. 3 Selection of most-proving node as with pn-search (the numbers at left side are the proof numbers, and the numbers at right side are the disproof numbers). c, e are most-proving nodes in the context of pn-search.

step 2) Expand the most-proving node. Calculate (dis)proof number of all the generated children by following Definitions 3 and 5.

step 3) Update (dis)proof numbers of all the nodes on the path from the most-proving node up to the root.

Repeat this until either proof number or disproof number of the root becomes 0.

Theorem 6 (Characteristic of pn-search): When expanding a leaf node along pn-search, the leaf node is always one of most-proving nodes defined by Definition 6.

Proof. Procedure pn-search shows that it is obvious that pn-search always expands a most-proving node defined by Definition 6. □

The concept of pn-search is that most-proving node defined by Definition 6 is always expanded because pn-search regards a most-proving node as the nearest node to the solution. Pn-search is a best-first algorithm because of its manner in selecting the most-proving node.

3.2 Df-pn

Here we consider depth-first algorithms using both proof number and disproof number. Although a depth-first algorithm using both proof number and disproof number can be constructed simply by extending Seo's algorithm (we can even give a proof of its equivalence to pn-search), in this paper we propose a more practical depth-first algorithm which we call df-pn. Df-pn is an efficient algorithm because it uses the information of second minimum (dis)proof number as well as the minimum one. Seo's algorithm can even be improved by this idea [19]. Even though PDS [8] is also a depth-first algorithm using both proof number and disproof number, its basic concept differs from pn-search and df-pn. It is not equivalent to pn-search either. (PDS is merely asymptotically equivalent.)

Therefore, we explain what df-pn is like, and give a proof of its equivalence to pn-search in the meaning of expanding a most-proving node. As with df-pn,

each node n has two thresholds: one for proof number $\mathbf{th}_{\mathrm{pn}}(n)$ and the other for disproof number $\mathbf{th}_{\mathrm{dn}}(n)$.

Procedure df-pn: Assign

$$\mathbf{th}_{\mathrm{pn}}(r) = \infty$$
$$\mathbf{th}_{\mathrm{dn}}(r) = \infty$$

where r is the root.

step 1) At each node n, the search process continues to search below n until $\mathbf{pn}(n) \geq \mathbf{th}_{\mathrm{pn}}(n)$ or $\mathbf{dn}(n) \geq \mathbf{th}_{\mathrm{dn}}(n)$ is satisfied (we call it ending condition).

step 2) When n is an OR node, select the child n_c with minimum proof number and the child n_2 with second minimum proof number. (If there is a child other than n_c with minimum proof number, that is n_2.) Search below n_c with assigning

$$\mathbf{th}_{\mathrm{pn}}(n_c) = \mathbf{min}(\mathbf{th}_{\mathrm{pn}}(n), \mathbf{pn}(n_2) + 1) \quad (13)$$
$$\mathbf{th}_{\mathrm{dn}}(n_c) = \mathbf{th}_{\mathrm{dn}}(n) + \mathbf{dn}(n_c) - \sum \mathbf{dn}(n_{\mathrm{child}}). \quad (14)$$

Repeat this handling until the ending condition holds. (Multiple Iterative Deepening.)

step 3) When n is an AND node, select the child n_c with minimum disproof number and the child n_2 with second minimum disproof number. (If there is a child other than n_c with minimum disproof number, that is n_2.) Search below n_c with assigning

$$\mathbf{th}_{\mathrm{pn}}(n_c) = \mathbf{th}_{\mathrm{pn}}(n) + \mathbf{pn}(n_c) - \sum \mathbf{pn}(n_{\mathrm{child}}) \quad (15)$$
$$\mathbf{th}_{\mathrm{dn}}(n_c) = \mathbf{min}(\mathbf{th}_{\mathrm{dn}}(n), \mathbf{dn}(n_2) + 1). \quad (16)$$

Repeat this handling until the ending condition holds. (Multiple Iterative Deepening.)

step 4) If the ending condition is satisfied, the search process returns to the parent node of n. If n is the root, then search is over.

Lemma 7: When the search process goes from a parent node t to its child t_{child}, each constituent node n on the path from the root to t_{child} is the node with minimum proof (disproof) number among its sibling nodes if n is an AND (OR) node.

Proof. Assume that n_c is a node on the path from the root to t_{child}. Assume still more that n_c is an OR node and that n_2 is a sibling node of n_c with second minimum disproof number at the moment when the search process went to n_c. Because of Expression (16),

$$\mathbf{th}_{\mathrm{dn}}(n_c) \leq \mathbf{dn}(n_2) + 1.$$

Hence,

$$\mathbf{dn}_{\mathrm{current}}(n_c) < \mathbf{th}_{\mathrm{dn}}(n_c) \leq \mathbf{dn}(n_2) + 1$$

where $\mathbf{dn}_{\mathrm{current}}(n_c)$ is the current disproof number of n. Therefore,

$$\mathbf{dn}_{\mathrm{current}}(n_c) \leq \mathbf{dn}(n_2) \leq \mathbf{dn}(n_i)$$

where n_i is any sibling node of n_c ($n_i \neq n_c$). Therefore, when n_c is an OR node, n_c is the child with minimum disproof number.

Similarly, when n_c is an AND node, Expression (13) leads to

$$\mathbf{pn}_{\mathrm{current}}(n_c) \leq \mathbf{pn}(n_2) \leq \mathbf{pn}(n_i).$$

Therefore, when n_c is an AND node, n_c is the child with minimum proof number. □

Theorem 8 (df-pn): When expanding a leaf node along df-pn, the leaf node is always one of most-proving nodes defined by Definition 6.

Proof. Because of Lemma 7, the leaf node to expand always matches the conditions of most-proving node defined by Definition 6. □

Although Expressions (14) and (15) are not used in the proof, they are necessary in order to assure that the search process returns to node n right after the ending condition of n held at its child n_c or any descendant of n_c.

3.3 Equivalence between Pn-search and Df-pn

Theorem 9 (Equivalence between pn-search and df-pn): Df-pn behaves the same as pn-search in the meaning that df-pn always expands a most-proving node defined by Definition 6.

Proof. Because of Theorems 6 and 8, both pn-search and df-pn always expands one of most-proving nodes defined by Definition 6. □

The feature of pn-search is summarized in Theorem 6. That is, pn-search always expands a most-proving node defined by Definition 6. Theorem 8 asserts that df-pn also always expands a most-proving node. In this meaning, df-pn behaves the same as pn-search.

4. Conclusions

There are two algorithms using only proof number, meaning that they are old-fashioned, for searching AND/OR trees. That is, AO* which is a best-first algorithm and Seo's depth-first algorithm. We gave a proof that AO* and Seo's algorithm behave in the same way in the meaning of they always expands a most-proving node defined by Definition 4.

There are two algorithms using both proof number and disproof number on equal weights for searching AND/OR trees. That is, pn-search which is a best-first algorithm and df-pn which is a depth-first algorithm proposed in this paper. We also gave a proof that pn-search and df-pn behave in the same way in the meaning of they always expands a most-proving node defined by Definition 6.

Moreover, we asserted that depth-first algorithms for AND/OR trees are very important similarly with minimax-tree search. Depth-first algorithms are superior in the case where we have memory space constraint and they have advantage of adaptability and extendibility by various enhancements. Indeed, we implemented a program to solved Tsume-shogi (Japanese Chess Problem), by using df-pn. This program became the most powerful Tsume-shogi program [9]. This program uses SmallTreeGC and SmallTreeReplacement [8] in order to remove some of the entries from the transposition table, when it gets full.

References

[1] L.V. Allis, Searching for Solutions in Games and Artificial Intelligence, Ph.D. Thesis, Department of Computer Science, University of Limburg, Netherlands, 1994.
[2] L.V. Allis, M. van der Meulen, and H.J. van den Herik, "Proof-number search," Technical Report CS 91-01, University of Limburg, Maastricht, Netherlands, 1991. Also available as Artificial Intelligence, vol.66, pp.91–124, 1994.
[3] M. Buro, "Improving heuristic mini-max search by supervised learning," Technical Report 106, NECI, 2000.
[4] C. Elkan, "Conspiracy numbers and caching for searching And/Or trees and theorem-proving," Proc. the International Joint Conference on Artificial Intelligence (IJCAI-89), pp.341–346, 1989.
[5] F.H. Hsu, T. Anantharaman, M. Campbell, and A. Nowatzyk, "A grandmaster chess machine," Scientific American, vol.263, no.4, pp.18–24, 1990.
[6] T. Itoh, Y. Kawano, M. Seo, and K. Noshita, "Recent progress in solving Tsume-Shogi by computers," JSAI, vol.10, no.6, pp.21–27, 1995.
[7] D.E. Knuth and R.W. Moore, "An analysis of alpha-beta pruning," Artificial Intelligence, vol.6, pp.293–326, 1975.
[8] A. Nagai, A New Depth-First-Search Algorithm for AND/OR Trees, Master's Thesis, Department of Information Science, University of Tokyo, Japan, 1999.
[9] A. Nagai and H. Imai, "Application of df-pn Algorithm to a Program to Solve Tsume-Shogi Problems" IPSJ SIG Notes SIGAL-75-2, 2000.
[10] N.J. Nilson, Problem Solving Methods in Artificial Intelligence, McGraw-Hill, New York, NY, 1972.
[11] N.J. Nilson, Principles of Artificial Intelligence, Tioga Publishing, Palo Alto, CA, 1980.
[12] J. Pearl, "Asymptotical properties of minimax trees and game searching procedures," Artificial Intelligence, vol.14, pp.113–138, 1980.
[13] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin, "A new paradigm for minimax search," Technical Report TR 94-18, Department of Computing Science, University of Alberta, Canada, pp.1–23, 1994.
[14] A. Plaat, J. Schaeffer, W. Pijls, and A. de Bruin, "Best-first fixed-depth minimax algorithms," Artificial Intelligence, vol.87, pp.255–293, 1996.
[15] J. Schaeffer, "The history heuristics," ICCA Journal, vol.6, no.3, pp.16–19, 1983.
[16] J. Schaeffer, "The history heuristic and alpha-beta search enhancements in practice," IEEE Trans. Pattern Anal. & Mach. Intell., vol.11, no.1, pp.1203–1212, 1989.
[17] J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron, "A world championship caliber checkers program," Artificial Intelligence, vol.53, pp.273–290, 1992.
[18] M. Seo, The C* Algorithm for AND/OR Tree Search and its Application to a Tsume-Shogi Program, Master's Thesis, Department of Information Science, University of Tokyo, Japan, 1995.
[19] M. Seo, personal communication by e-mail, 2000.
[20] D. Slate and L. Atkin, "Chess skill in man and machine," in Chess 4.5—the Northwestern University chess program, pp.82–118, Springer-Verlag, 1977.
[21] G.C. Stockman, "A minimax algorithm better than alpha-beta?" Artificial Intelligence, vol.12, pp.179–196, 1979.

Appendix: Program List of Df-pn

The program list of df-pn is carried below. 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, an algorithm of negamax view [7] can be constructed by naming the former ϕ and the latter δ.

```
1   // Iterative deepening at the root
2   procedure Nega-df-pn(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     Look Up Transposition Table(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 && Eval(n)=true) ||
17          (n is an OR node && Eval(n)=false)) {
18        n.φ = ∞;   n.δ = 0;
19      }else {n.φ = 0;   n.δ = ∞; }
20      Put In Transposition Table(n, n.φ, n.δ);
21      return;
22    }
23    GenerateLegalMoves();
24    // 3. Avoid cycles by using transposition table
25    Put In Transposition Table(n, n.φ, n.δ);
26    // 4. Multiple Iterative Deepening
27    while (1) {
28      // Terminate if either φ or δ is
29      // at least its threshold
30      if (n.φ ≤ ΔMin(n) || n.δ ≤ ΦSum(n)) {
31        n.φ = ΔMin(n);   n.δ = ΦSum(n);
32        Put In Transposition Table(n, n.φ, n.δ);
33        return;
34      }
35      δc = φ;
36      nc = SelectChild(n, φc, δc, δ2);
37      nc.φ = n.δ + φc − ΦSum(n);
38      nc.δ = min(n.φ, δ2 + 1);
39      MID(nc);
40    }
41  }
42  // Selection among the children
43  procedure SelectChild(n, &φc, &δc, &δ2) {
44    δbound = δc;
45    δc = ∞;   δ2 = ∞;
```

```
46      for (each child n_child) {
47          Look Up Transposition Table(n_child, φ, δ);
48          if (φ ≠ ∞) δ = max(δ, δ_bound);
49          if (δ < δ_c) {
50              n_best = n_child;
51              δ_2 = δ_c;    φ_c = φ;    δ_c = δ;
52          }else if (δ < δ_2) δ_2 = δ;
53          if (φ = ∞) return n_best;
54      }
55      return n_best;
56  }
57   // Look up transposition table for the entry of n
58  procedure Look Up Transposition Table(n, &φ, &δ){
59      if (n is recorded) {
60          φ = Table[n].φ;    δ = Table[n].δ;
61      }else {φ = 1;    δ = 1; }
62  }
63   // Record into transposition table
64  procedure Put In Transposition Table(n, φ, δ) {
65      Table[n].φ = φ;    Table[n].δ = δ;
66  }
67   // Calculate the minimum of δ of n's children
68  procedure ΔMin(n) {
69      min = ∞;
70      for (each child n_child) {
71          Look Up Transposition Table(n_child, φ, δ);
72          min = min(min, δ);
73      }
74      return min;
75  }
76   // Calculate the sum of φ of n's children
77  procedure ΦSum(n) {
78      sum = 0;
79      for (each child n_child) {
80          Look Up Transposition Table(n_child, φ, δ);
81          sum = sum + φ;
82      }
83      return sum;
84  }
```

Ayumu Nagai received the B.S., M.S., and Ph.D. degrees in information science from University of Tokyo, in 1997, 1999, and 2002, respectively. He is currently working at NASDA. His research interests include search algorithm and artificial intelligence.

Hiroshi Imai obtained B.Eng. in Mathematical Engineering, and M.Eng. and D.Eng. in Information Engineering, University of Tokyo in 1981, 1983 and 1986, respectively. He is now a professor at Department of Computer Science, University of Tokyo. His research interests include algorithms, quantum computing, computational geometry, and optimization. He is a member of IPSJ, OR Soc. Japan, JSIAM, ACM, and IEEE.