A new AND/OR Tree Search Algorithm Using Proof Number and Disproof Number

Ayumu Nagai
Department of Information Science,
University of Tokyo

Abstract

The proof number and the disproof number are significant ideas used to search an AND/OR tree. This paper presents a new depth-first algorithm, which behaves nearly best-first. The basic idea is to find (dis)proof solutions selectively when it seems better to (dis)prove the root by looking at both the proof number and the disproof number. The experimental results on random trees show this algorithm is very useful under memory space constraint and especially in the case when the solution is unknown whether it is proof or disproof.

1 Introduction

In searching an AND/OR tree or an AND/OR graph, AO*[Nilson, 1980] is intensively studied as an algorithm of searching optimum proof solution. Some new algorithms for searching AND/OR trees using the ideas of the proof number or the disproof number were recently proposed.

The origin of these ideas are the idea of the conspiracy numbers which was invented in the context of minimax tree (multi-valued tree) searching [McAllester, 1988]. The large conspiracy numbers show that the minimax value is stable. The proof and disproof number are the ideas obtained by applying the idea of the conspiracy numbers to an AND/OR tree (two-valued tree)[Allis, 1994]. In the early study only the idea of the proof number was taken into account, however in the recent study both the ideas of the proof and disproof number became to taken into account.

The best-first algorithm has a defect in spending a large quantity of memory space without any countermeasures. On the other hand, the algorithm using only the proof number also has a defect that it relatively takes long time in solving a problem with a disproof solution. In order to overcome these defects, we suggest a new algorithm taking a mixture of a good point of best-first algorithm using both the proof and disproof number[Allis, 1994] and a good point of depth-first algorithm using only the proof number[Seo, 1995].

Section 2 explains some terms associated with AND/OR tree search especially the proof number and the disproof number. Section 3 classifies the algorithms for searching an AND/OR tree. Section 4 describes our new search algorithm. Experimental results appear in Section 5. Section 6 describes our conclusions.

2 Proof number and disproof number

An *AND/OR tree* is composed of nodes each of which is whether an *AND node* or an *OR node*. The children of OR nodes must be AND nodes, and vice versa. The evaluation of each node results in one of the three values: **true**, **false** or **unknown**. A *terminal node* is a node with the evaluation of **true** or **false** who cannot be expanded any more. An *internal node* is a node who can be expanded. A *frontier node* is a node at the tip of the current search tree with the evaluation of **unknown** or which has not yet been evaluated.

An AND/OR tree is solved if its root obtains the value of **true** or **false** by minimax propagation under the assumption of the ordering **false** $<$ **unknown** $<$ **true**.

A solved AND/OR tree with value **true** at its root is called *proved*, while a solved tree with value **false** at its root is called *disproved*. A *solution tree* is composed by the nodes which are necessary to verify that the value of its root is **true** or **false**. As to the *proved* solution tree, at least one of the children of each OR node belongs to the solution tree, and all the children of each AND node belong to the solution tree. To the contrary, as to the *disproved* solution tree, all the children of each OR node belong to the solution tree, and at least one of the children of each AND node belongs to the solution tree. The aim of searching an AND/OR tree is to figure out whether the tree ultimately has proof or disproof solution, and to obtain a solution tree.

The indicator that is showing the difficulty to be (dis)proved is *(dis)proof number*. (Dis)proof number is defined as the least number of frontier nodes of the current search tree, which must be evaluated to **true**(**false**) in order to ensure that the game-theoretical value of the root is **true**(**false**).

The concrete method to calculate the proof and disproof number is as follows. (In the following, *n.pn* and

$n.dn$ stands for the proof number and the disproof number at node n respectively)

1. For a terminal node n
   (a) When game-theoretical value is already known
      i. When game-theoretically **proof**
$$\begin{aligned} n.pn &= 0 \\ n.dn &= \infty \end{aligned}$$
      ii. When game-theoretically **disproof**
$$\begin{aligned} n.pn &= \infty \\ n.dn &= 0 \end{aligned}$$
   (b) When game-theoretical value is not known yet
$$\begin{aligned} n.pn &= 1 \\ n.dn &= 1 \end{aligned}$$
2. For an internal node n
   (a) For an OR node
$$\begin{aligned} n.pn &= \operatorname*{Min}_{n_i \in \text{children of } n} n_i.pn \\ n.dn &= \sum_{n_i \in \text{children of } n} n_i.dn \end{aligned}$$
   (b) For an AND node
$$\begin{aligned} n.pn &= \sum_{n_i \in \text{children of } n} n_i.pn \\ n.dn &= \operatorname*{Min}_{n_i \in \text{children of } n} n_i.dn \end{aligned}$$

3 Classification of AND/OR tree search algorithms

Proof-number search for AND/OR trees can be done by only looking at the proof number or by looking at both the proof number and the disproof number. Therefore, it is possible to classify search algorithms from the viewpoint of what kind of criteria is used for the evaluation. Roughly speaking, proof-number search using both the proof and disproof number is the same as conspiracy-number search with three possible values for the conspiracy numbers(three-valued conspiracy-number search), while proof-number search without using the disproof number is the same as two-valued conspiracy-number search. Because both the proof and disproof number appears when conspiracy numbers are three-valued, while either the proof number or the disproof number only appears when conspiracy numbers are two-valued. It is also possible to classify search algorithms whether it is best-first search or depth-first search. As a result, AND/OR tree search algorithms can be classified into four regions (see Table 1).

Elkan's depth-first algorithm is put into brackets in Table 1. This is because this algorithm relatively doesn't have a good performance. Due to limitation of space, we can't show the experimental results. In this paper, let Elkan's algorithm indicate Elkan's best-first algorithm.

	criteria of evaluation used	
	only proof number	proof number and disproof number
best-first	Elkan[1989] (AO*[Nilson, 1980])	Allis[1994]
depth-first	Seo[1995] (Elkan[1989])	**PDS** [this paper]

Table 1: Classification of AND/OR tree search algorithms

Best-first algorithm AO* is also put into brackets in Table 1. This is because AO* uses information of the cost from the root and the heuristic estimate of the cost to a goal(terminal node with trivial proof solution) at each node. Moreover, AO* is a search algorithm for searching a *graph*.

3.1 Allis's Algorithm

Best-first search algorithm selects the frontier node, expands it, and update the information of the expansion toward the root. Allis's algorithm arrives to the frontier node by selecting the child of minimum proof(disproof) number at each OR(AND) node.

In our implementation, a small improvement is added when updating the proof and disproof numbers. As Allis has mentioned [Allis, 1994], updating process can be terminated when the new proof number is equal to the old proof number, and at the same time, the new disproof number is equal to the old disproof number. Our improvement is to terminate the updating process when the new proof number is not more than the old proof number, and at the same time, the new disproof number is not more than the old disproof number. The similar improvement is done to Elkan's algorithm.

3.2 Elkan's Algorithm

The selection algorithm of the frontier node differs from Allis's algorithm. Elkan's algorithm arrives to the frontier node by selecting the child of minimum proof number at each OR node, while selecting any child (except for already proved child) at each AND node.

In our implementation, Elkan's algorithm is modified to select minimum proof number at each AND node, too. The reason for this will be mentioned during the explanation of Seo's algorithm.

Elkan's algorithm correspond to AO* with the following assumptions as long as it is searching AND/OR *tree*, under the definition of h(n) is the heuristic estimate of the cost of going from node n to a goal, and the definition of cost(m, n) is the cost of going from node m to node n.

$$\begin{aligned} \forall n,\ \text{cost}(n_{parent}, n) &= 0 \\ \forall n,\ \text{h}(n) &= \begin{cases} 0 & n \in \text{goal} \\ 1 & n \notin \text{goal} \end{cases} \end{aligned}$$

3.3 Seo's Algorithm

Seo's algorithm uses the proof number as a threshold of iterative deepening. Therefore, this algorithm behaves nearly best-first manner (in the concrete, Elkan's algorithm). The relation between Elkan's algorithm and Seo's algorithm is similar to the relation between A* and IDA* [Korf, 1985].

However the ordinary iterative deepening iterates only at the root, it is possible to iterate at any node. This method is called *multiple iterative deepening* [Seo, 1995; 1998]. Seo's algorithm performs multiple iterative deepening at each *AND node* and *the root*. Basically, once the threshold is given to the node, the subtree rooted at that node is continued to be searched while the proof number of that node is below the given threshold. (Seo mentions in his thesis [Seo, 1995] that at an AND node, even after the proof number exceeds the threshold, it doesn't stop searching the subtree rooted at the AND node for a while, in order to minimize the effort of node reexpansions. However, because it is not clear when to stop searching the subtree, and because it seems some kind of gambling, in our implementation, it stops searching right after the proof number exceeds the threshold.)

Each OR node assigns the given threshold to all its child, while each AND node assigns the threshold to its child iteratively starting from 2. Practically, because of the multiple iterative deepening, each OR node selects the child with nearly minimum proof number. However, it is not clear what kind of selection each AND node is making. Seo mentions in his thesis [Seo, 1995; 1998], at each AND node that it is efficient to select the child which have never been expanded at the previous iterations. This is also performed in order to minimize the effort of node reexpansions. In our implementation, each AND node selects a child with minimum proof number, expecting that it is not expanded at the previous iterations. Because the proof number of the node which have been searched deeper tends to become larger.

Seo also mentions of his replacement scheme of the transposition table. In our experimental results, the table size is so large that no replacement scheme is necessary. Besides, Seo uses some other improvements, but most of them is specialized to *tsume-shogi*, the Japanese chess problem. As we have experimental results on random trees, these improvements are of no use.

3.4 The Necessity of a new Algorithm

In general, the best-first algorithms preserve the whole search tree in transposition table. Therefore without any countermeasures, they unavoidably run out of memory quickly and must terminate the search. On the other hand, in general, the search algorithms using only proof number actively search for proof solutions, but not for disproof solutions. Disproof solutions are seriously delayed to be solved or even can not be solved in practice. Therefore, the depth-first algorithm using both the proof and disproof number is very significant. A new search algorithm PDS(Proof-number and Disproof-number Search) is exactly such an algorithm.

4 PDS(Proof-number and Disproof-number Search) algorithm

We suggest a new depth-first algorithm using both the proof and disproof number. We call this new algorithm PDS. As mentioned above, Seo's algorithm performs multiple iterative deepening at each *AND node* and *the root*. PDS performs multiple iterative deepening at *all nodes*. Therefore, PDS behaves nearly best-first manner (in the concrete, Allis's algorithm). Once the thresholds are given to the node, the subtree rooted at that node is continued to be searched while either the proof or disproof number of that node is below the given thresholds. Each OR(AND) node assigns the thresholds to its child with minimum proof(disproof) number. If the threshold of (dis)proof number is incremented in the next iteration, the search continues mainly using (dis)proof number for selectively finding (dis)proof solution. If the proof number is smaller(larger) than the disproof number, it means that it seems to have a proof(disproof) solution. Therefore, PDS looks at the proof and disproof number in the transposition table, increments the threshold of the proof(disproof) number in case when the proof number is smaller(larger) than the disproof number, and continues searching. In this way, by using the information of both the proof and disproof number, the course of the search can be controlled to some degree.

PDS can be simply modified into Seo's algorithm by changing the strategy of incrementing the thresholds of the iteration. Moreover, all improvements mentioned by Seo can also be applied to PDS.

The proof number at an OR node and the disproof number at an AND node are essentially equivalent. Similarly, the disproof number at an OR node and the proof number at an AND node are essentially equivalent. As they are dual to each other, an algorithm equivalent to negamax algorithm in the context of minimax tree searching can be constructed by naming the former ϕ and the latter δ. We call this algorithm NegaPDS. The concrete Memory-Enhanced NegaPDS algorithm is as follows. (In the following, ΦSum() is a function to calculate the sum of ϕ of all the children. ΔMin() is a function to calculate the minimum of all the children. PutInTT() and LookUpTT() is a function to record and refer the transposition table respectively.)

```
// Iterative deepening at the root $r$
procedure NegaPDS($r$) {
  $r.\phi = 1$;    $r.\delta = 1$;
  while ($r.\phi \neq 0$ && $r.\delta \neq 0$) {
    MID($r$);
    // If the solution ultimately seems to be
    // (dis)proof, the search is continued mainly
    // using (dis)proof number in the next iteration
    if ($r.\phi \leq r.\delta$) $r.\phi$++;
    else $r.\delta$++;
  }
}
```

```
// Expansion of the node $n$
procedure MID($n$) {
  // 1. Refer the transposition table and
  //    terminate searching when it is unnecessary
  LookUpTT($n$, &$\phi$, &$\delta$);
  if ($\phi = 0$ || $\delta = 0$ || ($\phi \geq n.\phi$ && $\delta \geq n.\delta$)) {
    $n.\phi = \phi$;   $n.\delta = \delta$;
    return;
  }
  // 2. Terminate searching if $n$ is a terminal node,
  //    otherwise generate all the legal moves
  if ($n$ is a terminal node) {
    if (($n.value$ = true && $n$ is an AND node) ||
       ($n.value$ = false && $n$ is an OR node)) {
      $n.\phi = \infty$;   $n.\delta = 0$;
    } else { $n.\phi = 0$;   $n.\delta = \infty$; }
    PutInTT($n$, $n.\phi$, $n.\delta$);
    return;
  }
  generate all the legal moves;
  // 3. Avoidance of cycles by using transposition table
  PutInTT($n$, $n.\phi$, $n.\delta$);
  // 4. Multiple iterative deepening
  while (1) {
    // Terminate searching if both proof number and
    // disproof number is over or equal to their thresholds
    if ($\Phi$Sum($n$) = 0 || $\Delta$Min($n$) = 0 ||
       ($n.\delta \leq \Phi$Sum($n$) && $n.\phi \leq \Delta$Min($n$))) {
      $n.\phi = \Delta$Min($n$);   $n.\delta = \Phi$Sum($n$);
      PutInTT($n$, $n.\phi$, $n.\delta$);
      return;
    }
    $\phi = \max(\phi, \Delta$Min($n$));
    $n_{child}$ = SelectChild($n$, $\phi$);
    LookUpTT($n_{child}$, &$\phi_{child}$, &$\delta_{child}$);
    // If the solution ultimately seems to be
    // (dis)proof, the search is continued mainly
    // using (dis)proof number in the next iteration
    if ($n.\delta > \Phi$Sum($n$) &&
       ($\phi_{child} \leq \delta_{child}$ || $n.\phi \leq \Delta$Min($n$))) {
      $n_{child}.\phi = \phi_{child} + 1$;   $n_{child}.\delta = \delta_{child}$;
    } else {
      $n_{child}.\phi = \phi_{child}$;   $n_{child}.\delta = \delta_{child} + 1$;
    }
    MID($n_{child}$);
  }
}

// Selection among the children
procedure SelectChild($n$, $\phi$) {
  for (each child $n_{child}$) {
    $\delta_{child} = n_{child}.\delta$;
    if ($\delta_{child} \neq 0$) $\delta_{child} = \max(\delta_{child}, \phi)$;
  return the child with minimum $\delta_{child}$;
  (If there are plural children with minimum $\delta_{child}$,
  return the child with minimum $n_{child}.\phi$ among them)
}
```

A search algorithm for an AND/OR tree can also be applied to a minimax tree by the method Allis [Allis, 1994] and Schijf [Schijf, 1993] have mentioned. The basic idea is to take notice of the fact that the process of determining whether the game-theoretical value of the minimax tree is over v or not is two-valued. If **true** is defined as the minimax value that is over v, and if **false** is defined as the minimax value below or equal to v, the minimax tree is equivalent to an AND/OR tree. Then AND/OR tree search algorithm is available. In this way, by applying AND/OR tree search algorithm like binary search, the game-theoretical value can be identified.

Another extension is possible to PDS. As mentioned above, AO* uses the information of the cost from the root to each node, and the heuristic estimate of the cost from each node to a goal. PDS can be extended by using the information similar to these. The cost from the root to each node as a part of the (dis)proof solution, and the heuristic estimate of the cost from each node to a (dis)proof goal(terminal node with trivial (dis)proof solution). When this kind of extension is made, PDS terminates in the optimal solution either the problem has a proof solution or a disproof solution [Nagai, 1999].

5 Performance measures on random trees

We used random trees for experiments. In general, when playing two-player games, the one who is more advantageous than the other has a tendency to have relatively more moves and has wider range of selection than the other. Our random tree has the feature reflecting this fact.

avg stands for the average branching factor. N(μ, σ) stands for normal distribution with the average of μ and the variance of σ^2 (In this paper, let μ equal 0). P(λ) stands for Poisson distribution with the average and the variance of λ. The basic idea is that each node n has an internal value $n.v$ and if $n.v$ becomes over 1(under 0), $n.value$ is set to **true**(**false**), otherwise $n.value$ is set to **unknown**. If $n.value$ is either **true** or **false**, n is made to be a terminal node. $n.v$ which is invisible from the search routine shows the degree of advantage and depends on $n_{parent}.v$. $n.bf$ stands for the branching factor, or the number of children. If $n.value$ is equal to **unknown**, $n.bf$ is set to $\mathrm{P}(\lambda_{bf} - 1) + 1$ which is over or equal to 1, while λ_{bf} depends on $n.v$ in order to reflect the degree of advantage. For example, if $n.v$ is equal to 0.5, λ_{bf} is set to *avg*. If $n.v$ is over 0.5, λ_{bf} is set to the value over *avg* at each OR node and is set to the value under *avg* at each AND node. If $n.value$ is either **true** or **false**, $n.bf$ is set to 0 because n is a terminal node.

The large *avg* makes the random tree tend to become wider. The small σ makes the random tree tend to become deeper. The value $root.v$ close to 1(0) makes the probability that the random tree has a proof(disproof) solution tend to become larger. The outline for making a random tree is as follows.

$$n.v = n_{parent}.v + \mathrm{N}(\mu, \sigma)$$

1. For a terminal node n
   (a) $n.v \geq 1$ $\quad n.value = \textbf{true}$, $\; n.bf = 0$
   (b) $n.v \leq 0$ $\quad n.value = \textbf{false}$, $\; n.bf = 0$
2. For an internal node n
   (a) At an OR node
   $$n.value = \textbf{unknown}$$
   $$n.bf = \mathrm{P}(\lambda_{bf} - 1) + 1$$
   $$\text{while } \lambda_{bf} = 2(avg - \lambda_{bfmin}) \times n.v + \lambda_{bfmin}$$
   (b) At an AND node
   $$n.value = \textbf{unknown}$$
   $$n.bf = \mathrm{P}(\lambda_{bf} - 1) + 1$$
   $$\text{while } \lambda_{bf} = -2(avg - \lambda_{bfmin}) \times n.v + 2avg - \lambda_{bfmin}$$

We have implemented four algorithms for the experiments, Elkan's algorithm(as limited version of AO*), Seo's algorithm, Allis's algorithm, and PDS.

As for time requirement, we use the ratio of the number of node generations to that needed by Allis's algorithm. This is because if we use the average of the actual numbers of node generations itself, the average will no longer be an average of all the problems, since the problems which need much effort to solve have more influence on it. Therefore, we use the number of node generations needed by Allis's algorithm as a standard. At this time, if the same node is generated more than two times, the number of node generation in this experiment contains them in duplication. As for memory space requirement, we used the ratio of the number of recorded positions in the transposition table to that needed by Allis's algorithm. Note that because results of Allis's algorithm is always taken as a standard, its performance is always 1.0 in this experimental results.

In general, the number of terminal nodes is most commonly used as a performance measurement. It seems to be proper when node-evaluating time is the main factor of the elapsed CPU time. However when searching an AND/OR tree, node-evaluating time is usually short, e.g., mate at chess. Therefore, we concluded that the number of node generations and the number of recorded positions in the transposition table are more proper criteria.

In our implementation, if an OR(AND) node ultimately become proved(disproved), all subtrees rooted at its children except the only proved(disproved) child are eliminated. In other words, after each node being solved, the partial solution tree is left in the transposition table. All the other descendants of the solved node are eliminated. Furthermore, in order to see the experimental results in the situation under memory space constraint, we implemented an easygoing method to restrict the usage of the transposition table. At each node n, all the children n_{child} are eliminated under some condition just before the search process goes up to n_{parent}. In other words, at each node n, if its proof number or disproof number becomes not less than its threshold, the search process terminates searching under that node and goes up to its parent n_{parent}. At that time, all the children n_{child} are eliminated from the transposition table if they didn't satisfy the surviving condition. The node which does not satisfy the surviving condition can only be in the transposition table temporarily. The strict surviving condition makes the usage of the transposition table be smaller. The defect of this easygoing restriction is that it can not be applied to Elkan's algorithm and Allis's algorithm. As with PDS, we use the surviving condition of $\phi \geq t \wedge \delta \geq t$. As with Seo's algorithm, we use the surviving condition of $pn \geq t$. Note that t is a kind of threshold which is completely different from the threshold of the multiple iterative deepening we mentioned in Section 4.

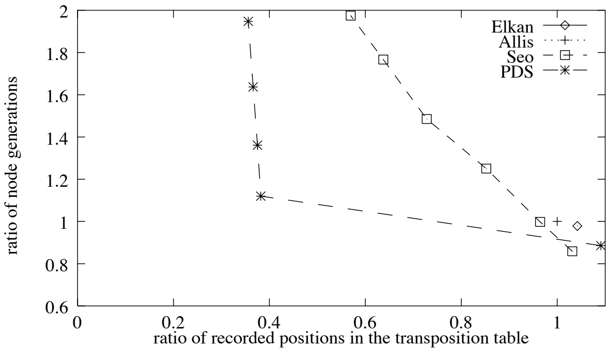

Figure 1: table-recorded positions and node expansions with proof solutions

Figure 1 shows the relation between the ratio of the number of recorded positions in the transposition table and the ratio of the number of node generations in case of *proof solutions* with average branching factor of 8, 12, 16, totally 60 problems. The surviving conditions of Seo's algorithm are no elimination, $pn \geq 3$, $pn \geq 4$, $pn \geq 5$, $pn \geq 6$, and $pn \geq 7$. The surviving conditions of PDS are no elimination, $\phi \geq 2 \wedge \delta \geq 2$, $\phi \geq 3 \wedge \delta \geq 3$, $\phi \geq 4 \wedge \delta \geq 4$, and $\phi \geq 5 \wedge \delta \geq 5$.

Figure 2 shows the same relation in case of *disproof solutions* with the average branching factor of 8, 12, 16, totally 60 problems. As there were many problems which Seo's algorithm and Elkan's algorithm cannot solve in practice, they are omitted in Figure 2. The surviving conditions of PDS are same as Figure 1.

Note that PDS can solve both proof and disproof problems even under relatively restricted memory condition.

Because of multiple iterative deepening, Seo's algorithm and PDS have relatively large overhead. Therefore, practically, they require more CPU time. As PDS performs multiple iterative deepening at *all* nodes, the overhead of PDS seems to be larger than Seo's algorithm.

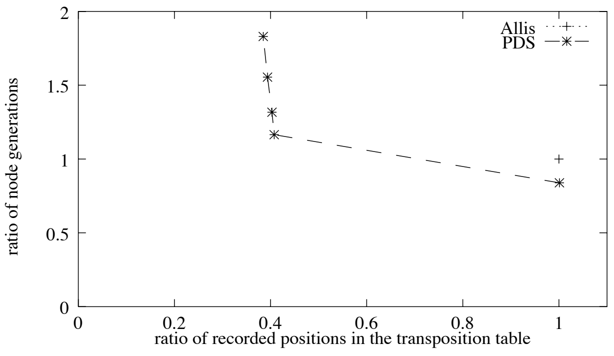

Figure 2: table-recorded positions and node expansions with disproof solutions

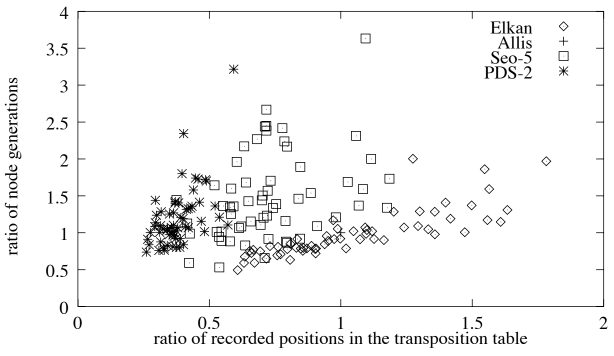

Figure 3: table-recorded positions and node expansions with proof solutions

Figure 3 plots the relation between the ratio of the number of recorded positions in the transposition table and the ratio of the number of node generations in case of *proof solutions*, each problem with average branching factor of 8, 12, 16, totally 60 problems. Surviving condition of Seo's algorithm and PDS is $pn \geq 5$ and $\phi \geq 2 \wedge \delta \geq 2$ respectively.

Figure 4 plots the same relation in case of *disproof solutions*, each problem with average branching factor of 8, 12, 16, totally 60 problems. Surviving condition of PDS is $\phi \geq 2 \wedge \delta \geq 2$.

Figure 3 and Figure 4 show that PDS requires relatively less memory space than the other three algorithms.

6 Conclusion

In this paper we have suggested a new depth-first multiple iterative deepening algorithm PDS for searching an AND/OR tree. It nearly behaves in best-first manner by using both proof number and disproof number as thresholds of multiple iterative deepening. Both proof-number search using only proof number and best-first search algorithm have some defects. The aim of our new algorithm is to overcome these defects. The experimental results on random trees have shown that PDS is very useful under memory space constraint.

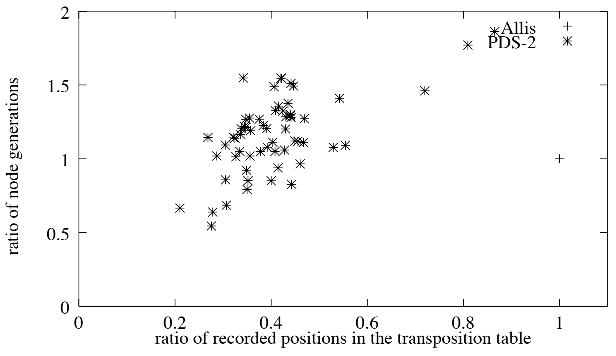

Figure 4: table-recorded positions and node expansions with disproof solutions

Even though PDS is an algorithm for searching an AND/OR tree, we have shown how to apply PDS to minimax tree searching.

References

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

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

[Korf, 1985] Richard E. Korf. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27:97–109, 1985.

[McAllester, 1988] David A. McAllester. Conspiracy Numbers for Min-Max Search. *Artificial Intelligence*, 35:287–310, 1988.

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

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

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

[Seo, 1995] 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.

[Seo, 1998] Masahiro Seo. Solving Tsume-shogi Using Conspiracy Numbers (in Japanese). in Hitoshi Matsubara, editor, *Advances in Computer Shogi 2.* Kyoritsu Press, 1998.