ELSEVIER

Artificial Intelligence 129 (2001) 253–277

Artificial Intelligence

www.elsevier.com/locate/artint

The PN*-search algorithm: Application to tsume-shogi

Masahiro Seo[a], Hiroyuki Iida[b], Jos W.H.M. Uiterwijk[c,*]

[a] *AVC Company, Matsushita Electric Industrial, Co., Ltd., Osaka, Japan*
[b] *Department of Computer Science, Shizuoka University, Hamamatsu, Japan*
[c] *Institute for Knowledge and Agent Technology (IKAT), Department of Computer Science, Universiteit Maastricht, Maastricht, The Netherlands*

Received 29 December 1999; received in revised form 17 March 2001

Abstract

This paper proposes a new search algorithm, denoted PN*, for AND/OR tree search. The algorithm is based on proof-number (PN) search, a best-first search algorithm, proposed by Allis et al. [Artificial Intelligence 66 (1) (1994) 91–124], and on Korf's RBFS algorithm [Artificial Intelligence 62 (1) (1993) 41–78]. PN* combines several existing ideas. It transforms a best-first PN-search algorithm into an iterative-deepening depth-first approach. Moreover, it is enhanced by methods such as recursive iterative deepening, dynamic evaluation, efficient successor ordering, and pruning by dependency relations. The resulting algorithm turns out to be highly efficient as witnessed by the experimental results.

The PN* algorithm is implemented in a tsume-shogi (Japanese-chess mating-problem) program, and evaluated by testing it on 295 notoriously difficult tsume-shogi problems (one problem has a depth of search of over 1500 plies). The experimental results are compared with those of other programs. The PN* program shows by far the best results, solving all problems but one. Needless to say, it outperforms the best human tsume-shogi problem solvers by far. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: PN* search; Recursive iterative deepening; Depth-first search; Proof-number search; Shogi mating problems

* Corresponding author.
E-mail addresses: seo@biu.avcg.mei.co.jp (M. Seo), iida@cs.inf.shizuoka.ac.jp (H. Iida), uiterwijk@cs.unimaas.nl (J.W.H.M. Uiterwijk).

1. Introduction

In many cases, the process of a given problem can be represented by an AND/OR graph. The corresponding search processes have been extensively studied in AI (see, e.g., [7–9, 26,28,32–34,44]). The basic best-first-search algorithm for AND/OR graphs is AO* [33]. Later, some variants of AO* were developed [7,24,25,27,34]. AO* expands leaf nodes in the potential solution graph of minimum cost. The cost of non-terminal leaf nodes is evaluated using a heuristic function mainly based on domain knowledge. The efficiency of AO* depends on the quality of the heuristic function. Korf already described a best-first algorithm needing only linear space like a depth-first algorithm [22]. In [35,36] it has been shown that best-first search such as SSS* can be reformulated as a depth-first search algorithm using α-β pruning and transposition tables (prohibiting repeated search of common subtrees). In the case of using well-defined goals we can abandon the heuristic function, and instead apply AO* without any domain knowledge. In most cases, the well-defined goals are too far away from the starting positions to be reached by any search process. Only a very efficiently guided search may perform such a challenging task successfully. For this challenging purpose we have developed an algorithm, denoted PN*, based on AO* using the concept of proof numbers. So, PN* is closely related to proof-number search [2,3], which goes back to the idea of conspiracy numbers [30,40]. In passing we mention that there are many applications of AND/OR graph (or tree) search, viz. single-agent problems, automated theorem proving, games, puzzles, etc.

The idea of finding a solution in an AND/OR graph is closely related to the concept of a *solution tree*. This is a subtree proving a win for the player to move (MAX) against every defense by the opponent (MIN). It has exactly one continuation (leading to a solution) at every max node and contains all continuations at min nodes.

In Section 2, we give the rules of tsume-shogi and introduce the best tsume-shogi programs to date. Section 3 describes proof-number search. In Section 4 we present our PN* algorithm. Section 5 gives some details concerning dependencies between positions in tsume-shogi. The implementation of the PN* algorithm is schematically listed in Section 6. Experimental results and comparison with other programs are presented in Section 7. Finally, the conclusions of the article are stated in Section 8.

2. Tsume-shogi

Tsume-shogi problems are mating problems which can be seen as a subdomain of shogi (Japanese chess). A noticeable difference between shogi and western chess is that in shogi a player can reuse (*drop*) pieces which have been captured from the opponent. The domain of tsume-shogi problems is popular in Japan, and is for several reasons (given below) a good application for AND/OR tree search.

2.1. The rules of tsume-shogi

A tsume-shogi problem is a shogi composition with typically only one solution. The answer of a tsume-shogi problem is usually presented as a solution sequence of moves, i.e., a sequence of moves from the given position to a mate position. In tsume-shogi Black is the attacking side which moves first. The defending side (White) is assumed to make the best defense possible, i.e., to delay mate as long as possible. The answer to a problem thus is the longest path of all paths from the root to the terminal AND nodes in a solution tree for the position. The solution sequence of moves has to satisfy three conditions [29], viz.

(1) Every move by Black must be a check (note that in this respect tsume-shogi differs from a mating problem in standard shogi).
(2) Black must mate in the shortest sequence.
(3) White must select a move which makes the mating sequence as long as possible, but may not play so-called *useless moves* ("Muda-ai"); a useless move is defined as a drop or interposing move by a piece, which can be captured by the attacker, not affecting the mating sequence.

So the solution sequence corresponds to the sequence of best moves by both sides. Two characteristics of tsume-shogi are essentially different from those of (western) chess mating problems [13,29].

(4) Every piece not otherwise accounted for, except Black's King, is in White's hand. White can drop these pieces on the board on his turn, as long as it is not a useless move.
(5) The length of the solution sequence is not given in advance.

Moreover, tsume-shogi problems should be *complete*, meaning that they are subject to the following three additional conditions [13,29].

(6) A problem that has no solution sequence (i.e., the King is never mated) is disqualified (called "no-mate" or "fu-tsume").
(7) There is only one solution sequence of moves in the sense that
    (a) if Black selects other moves, he will fail to mate. [1]
    (b) if White selects other moves, Black can mate easier (i.e., with a shorter move sequence or by leaving some pieces in hand unused).
(8) According to the solution sequence, Black has no pieces in hand in the final (mate) position.

The solution sequence of a complete tsume-shogi problem is unique, but even a complete problem usually has several solution trees. The longest move sequence shared by such solution trees is the solution sequence. If there exists no solution sequence shared by all the solution trees, then the problem has more than one solution sequence, and is called "yo-tsume", regarded as an incomplete problem. The creator of a tsume-shogi problem is assumed to have investigated all possibilities of "yo-tsume" to confirm the completeness of the problem.

Below we provide four reasons why tsume-shogi is a suitable domain for AND/OR tree search.

(1) Tsume-shogi has an enormous search space, except for some trivial problems, so simple brute-force search is not adequate because of the combinatorial explosion.
(2) A tsume-shogi problem has always a solution, and the solution is unique if the problem is complete. If the solving program finds a solution (represented as a solution tree), then execution can be terminated.

[1] This also is an essential difference from western-chess mating problems.

(3) Many tsume-shogi problems have been composed since the Edo era in Japan and can thus be used for testing purposes. There are various types of problems, including those very hard to solve and those having a very long solution sequence.
(4) Many high-performance tsume-shogi-solving programs have been developed, so the performance of a program can be measured by solving well-known tsume-shogi problems and comparing the performance with other programs.

2.2. Tsume-shogi programs

Tsume-shogi-solving programs have been developed alongside of complete shogi programs, since tsume-shogi is a part of shogi endgames. Until 1990, all tsume-shogi programs were unable to solve shogi-mating problems with more than thirteen steps, because the programs used iterative deepening depth-first search with the inherent combinatorial explosion for non-trivial problems.

In 1992, Noshita developed a program named T2 [31]. T2 uses a transposition table to detect identical positions, and applies various tsume-shogi-specific techniques. T2 can solve almost every problem with at most 25 steps. However, T2 cannot solve problems with more than 30 steps. Then Ito developed the tsume-shogi program named ITO [15]. ITO uses a kind of best-first search, in which the heuristic function is based on the number of legal moves of the white King to escape from checks. ITO solves relatively long problems (with more than 100 steps). Since ITO is based on best-first search, it stores all the generated nodes in memory, so it cannot solve difficult problems with large solution trees, since it then runs out of memory. Subsequently, Kawano [19] developed a tsume-shogi solver using best-first search based on the idea to avoid duplicated search for similar positions. In the mid 1990s, Nakayama et al. [31] developed a tsume-shogi solver based on a parallel algorithm. Their results were encouraging.

The average branching factor of tsume-shogi has not been calculated yet, as far as we know. The numerical results of this paper show that the average branching factor of tsume-shogi is about 5. This number is much smaller than that of shogi itself, but the problems with solutions of more than 17 steps cannot be solved by simple brute-force methods like depth-first minimax enhanced with α-β only [20]. Also, selective searches have thus far not been able to solve these problems in most cases correctly.

Among the existing tsume-shogi programs, those using best-first search cannot solve difficult problems due to memory constraints [15]. Also tsume-shogi programs using depth-first search like T2 are unable to solve problems with longer solution lengths [31]. We developed a program that can solve almost all tsume-shogi problems published today, including those with very long solution sequences.

3. Proof-number search

In order to solve a tsume-shogi problem, at least all nodes in a solution tree must be examined. The length of the solution sequence usually is more than twenty and there are cases in which more than several hundred moves are required. For example, the length of the longest solution sequence of problems solved by our PN* program is 1525 steps.

So as a first estimate a search tree containing $5^{1525/2} \simeq 10^{548}$ nodes exists. It is therefore impossible for simple brute-force algorithms to solve (difficult) tsume-shogi problems in real time (i.e., within a few hours).

To solve tsume-shogi problems effectively, we turned our attention to the concept of proof-number (PN) search as proposed by Allis et al. [2,3]. PN search goes back to the idea of conspiracy numbers introduced by McAllester [30]. Using proof numbers it is possible to identify the *potential solution tree* (denoted by "pst") for which the probability of becoming a solution tree is the highest among all pst's. A potential solution tree is defined as a search tree with one continuation at every max node and all continuations at min nodes, of which the leaf nodes are either won nodes for the max player or unexpanded nodes. It may become a solution tree by expanding unsolved leaf nodes and proving that they all lead to a win for the max player.

An implementation of conspiracy numbers in computer chess was described by Schaeffer [40]. He also reported some experiments. An implementation of proof-number search in computer chess was described by Breuker et al. [4]. A major drawback in searching minimax trees using conspiracy/proof numbers is the memory requirement. Both search methods are typical best-first searches, and require storing all the nodes generated previously together with their conspiracy/proof numbers. So difficult positions cannot be solved because of memory constraints. Even memory-favourable alternatives like PN^2 search [5,6] suffer from this drawback.

3.1. Proof numbers for AND/OR tree search

Proof numbers for minimax tree search can be applied to any algorithm for searching AND/OR trees [2]. The proof number for AND/OR tree search is the minimum number of unsolved leaf nodes that need to be solved in order to solve the root. It can be said that the larger the proof number of the root, the more difficult it is to solve the root. We call the set of unsolved leaf nodes which account for the proof number the *proof set*. The proof set corresponds to the set of all leaf nodes of the pst corresponding with the proof number of the root. The search proceeds by continuously expanding the nodes from the proof set. A node is denoted *terminal* if the corresponding position has no legal moves.

The proof number of an unsolved pst and that of an unsolved node are defined as follows.

Definition 1. In an AND/OR tree, the proof number of an unsolved pst T is equal to the number of unsolved leaf nodes in T.

Definition 2. In an AND/OR tree, the proof number of an unsolved node n is equal to the minimum proof number of the pst's rooted at n.

In the case of a tsume-shogi problem, an OR node corresponds with a position in which the attacker is to move, where any move that solves the problem denotes a solution. The proof number then is the minimum proof number of its children (i.e., the one potentially easiest to solve). If the attacker has no more moves in the position, the problem is unsolvable from that position and the proof number is set to ∞. Likewise, an AND node corresponds with a position with the defender to move. To solve the problem for the

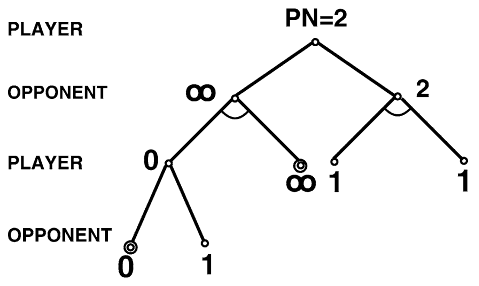

Fig. 1. An example of proof numbers in an AND/OR tree. Double-edged circles denote terminal nodes.

attacker all the defender's children must be proven to lead to the desired result, hence its proof number is the sum of the children's proof numbers. If the defender has no more legal moves (is mated), the goal is reached and the proof number is set to 0. For an easy-to-grasp example, see Fig. 1.

More formally, let $PN(n)$ denote the proof number of a node n in an AND/OR tree. It is calculated as follows.

(1) If n is a leaf node, then
  (a) if n is a terminal AND node, then n is solved, and
$$PN(n) = 0,$$
  (b) else if n is a terminal OR node, then n is unsolvable, and
$$PN(n) = \infty,$$
  (c) else n is an unsolved leaf node, and
$$PN(n) = 1;$$
(2) else if n is an OR node whose successor nodes are n_i, $1 \leqslant i \leqslant k$, then
$$PN(n) = \min_{1 \leqslant i \leqslant k} PN(n_i);$$
(3) else if n is an AND node whose successor nodes are n_i, $1 \leqslant i \leqslant k$, then
$$PN(n) = \sum_{1 \leqslant i \leqslant k} PN(n_i).$$

A large disadvantage of PN search is that, as a genuine best-first algorithm, it uses a large amount of memory, since the complete search tree has to be kept in memory.

4. The PN* algorithm for AND/OR tree search

To tackle the memory disadvantage of PN search we propose the PN* algorithm for AND/OR tree search, which is a depth-first alternative for proof-number search. The idea is derived from Korf's RBFS algorithm [22], which was formulated in the framework of single-agent search. Plaat et al. [35] developed a depth-first alternative for SSS*. PN* is a search algorithm applicable to general AND/OR trees.

4.1. Overview of the PN* *algorithm*

PN* is a depth-first iterative-deepening search algorithm, so the overhead of re-expansions of nodes already expanded at previous iterations is of some concern. The overhead of re-expansions can be reduced by storing the results for nodes previously generated and expanded in a transposition table. We enhance the effectiveness of PN* by methods like *recursive iterative deepening* and *dynamic evaluation*.

By recursive iterative deepening is meant that iterative-deepening search is not only been done at the root but also at all interior OR nodes. If recursive iterative deepening is used, it may seem at first sight that the number of re-expanded nodes becomes larger. However, by re-ordering the successor OR nodes at each iteration when an AND node is expanded, the ratio of the re-expansions is effectively reduced.

When an AND node is expanded and its successor OR nodes are generated, dynamic evaluation sets the proof number of the unsolved successor leaf nodes. Their default value is 1, but if a successor OR node has been expanded at previous iterations, and the information that it could not be solved within proof number N is stored in the transposition table, the value N is assigned. This is called dynamic evaluation.

In some particular positions, *dependency relations* among successor positions allow the pruning of large parts of the search space. The effectiveness of such pruning depends on a proper ordering of moves.

By implementing recursive iterative deepening, dynamic evaluation, and efficient use of dependency relations, the performance of our tsume-shogi program has considerably improved. The structure of the PN* algorithm is described in more detail in Sections 4.2–4.4 and 5 below. In Section 4.5 we address the topics of completeness and complexity of our algorithm. Its implementation is presented in Section 6.

The following domain knowledge is explicitly incorporated.

(1) The goal is to find a solution tree, and the search process is terminated as soon as a solution tree is found. At least one solution tree always exists in the search space. Note that termination of the search process as soon as a solution has been found makes it impossible to discover all multiple solutions (when problems are defined as “yo-tsume”, i.e., not complete), but makes the search process much more efficient. Since the main goal of the solver is finding the one intended solution as quick as possible, and not checking the problems for completeness, we take the disadvantage for granted.

(2) The successor nodes of an OR node (attacker) is of type AND (defender), and the successor nodes of an AND node (defender) is of type OR (attacker), if any.

(3) An AND node having no successors is a solved (won) node; an OR node having no successors is an unsolvable (lost) node; other leaf nodes never expanded before are unsolved nodes.

4.2. Depth-first iterative deepening using proof numbers

Depth-first search needs only linear space, but takes a great deal of time unless the successor nodes are well ordered. If the depth of the solution is not known in advance, we must estimate it before starting a depth-first search. If the estimate is lower than the depth of an optimal solution, the algorithm terminates without finding any solution. If the estimate is much larger than the optimal solution's depth and the ordering of successor nodes is bad, it takes much more time than needed and it may happen that a suboptimal solution is found instead of the optimal one. A good depth bound is hard to determine, as well as a good ordering of successor nodes needed for the required efficiency.

Iterative-deepening search does not suffer from this drawback of depth-first search [10, 21]. It performs a depth-first search to depth 1 at the first iteration, and if a solution is not found at the first iteration, then searches to depth 2 at the next iteration, and so on. All nodes generated at the previous iterations are discarded. Korf [21] has proven that depth-first iterative deepening leads to an optimal solution. Of course, it can be enhanced by transposition tables, killer heuristic, history heuristic, and so on [37,39].

Proof-number search is a best-first-search algorithm. It expands selectively a leaf node in the pst with the smallest proof number. It needs large amounts of memory to store all nodes generated so far. Since the number of nodes generated during the search is often very large, it is impossible to retain all the generated nodes for problems having large search spaces.

Our search algorithm uses a depth-first iterative-deepening method, using a proof-number threshold, which is incremented at each iteration. Best-first search using proof numbers can then be transformed into a depth-first iterative-deepening method by making the selection of the pst from those having a proof number less than or equal to the proof-number threshold and the selection of the node to be expanded in the selected pst in depth-first order. This is schematically presented in Fig. 2.

At the start of the Nth iteration, all the pst's at the root have been generated and proven to be unsolvable within a proof number $N-1$. Then the proof-number threshold N is given to the root (an OR node), which is expanded, and some pst's at depth 1 with proof numbers 1 are generated. The proof number 1 does not exceed the threshold N, so the leftmost successor AND node is expanded, and counts the number of successor OR nodes $(= N_1)$. For example, in Fig. 2, $N_1 = 3$. Then the proof number of the leftmost pst becomes N_1. If N_1 exceeds the threshold N, then the expansion of the leftmost pst is abandoned, and the search process returns to the parent OR node (the root in this case), and expands the next pst rooted at the OR node. Else if N_1 does not exceed the threshold N, then the leftmost pst is expanded by giving the proof-number threshold $N - N_1 + 1$ to the leftmost successor OR node. This threshold stems from the fact that in order to solve an AND node within at most N expansions, a child node should be solved within at most N minus the number of siblings, since each sibling will require at least 1 expansion (the sibling itself).

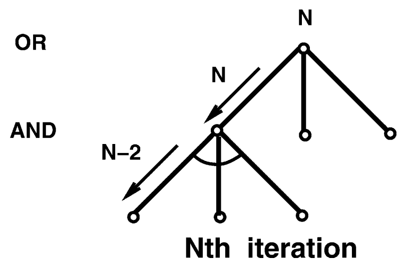

Fig. 2. An example of iterative deepening using proof-number thresholds.

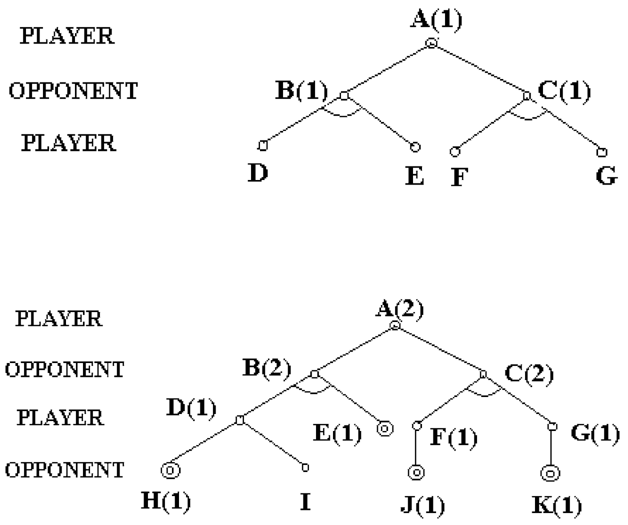

Fig. 3. A detailed example. Double-edged circles denote terminal nodes. Top: first iteration; bottom: second iteration.

A detailed step-by-step example is given in Fig. 3. The figure between parentheses at each node shows the proof-number threshold given to the node. In the first iteration (top in Fig. 3), root A receives proof-number threshold 1. A is expanded, and nodes B and C are generated. The search continues at B (left first) with threshold 1, and by expanding B, nodes D and E are generated. This exceeds the given threshold of 1. The proof number of node B becomes 2. Next, the search continues at node C with threshold 1, and generates nodes F and G by expanding C, which also exceeds the threshold of 1. The proof numbers of nodes C and A become 2.

In the second iteration (bottom in Fig. 3), the proof-number threshold of root A is incremented to 2. Node B is given threshold 2, and the proof number of node E is 1, so node D gets threshold 1. Node H is a terminal (mate) position (no child nodes), so the search below node D is terminated without investigating node I, and the proof number of node D becomes 0. Next the search goes to node E with threshold 1. (If standard iterative deepening were used, threshold 2 would be given to node E, but PN*, using recursive iterative deepening, starts the search of node E with threshold 1. Thus an unexpanded OR node is always given threshold 1.) The proof number of node E turns out to be infinity (no child nodes), and the proof number of node B becomes infinity, exceeding the given threshold 2, so the search backtracks to node A and proceeds at node C with threshold 2. Node C has two child nodes (F and G), so threshold 1 is given to node F. Since node J, the only successor of node F, is a terminal (mate) position, the proof numbers of J and F become 0 and the search continues at node G with threshold 1. The only child node K also is a mate position with proof number 0, and G, C and A also receive proof number 0 successively, and a solution has been found.

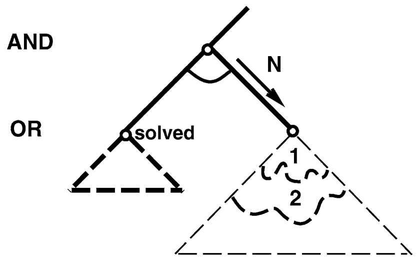

Fig. 4. The recursive-iterative-deepening method.

4.3. Recursive iterative deepening

As indicated above, we use iterative deepening not only at the root but also at all interior OR nodes. This is known as *recursive iterative deepening* (see Fig. 4). The ratio of re-expansions is kept low by optimizing the successor ordering of AND nodes at each iteration. Our experimental results in Section 7 show that the ratio of re-expanded nodes and all expanded nodes is about 20%. By using the recursive-iterative-deepening method, we can always select the optimal pst not only at the root, but also at other OR nodes within the search tree.

4.4. Dynamic evaluation

The depth-first iterative-deepening method discards all the nodes generated in previous iterations except the root, and starts the next iteration by expanding the root. So it needs to use the results of previous iterations effectively to avoid useless re-calculations. The information of a node n expanded at least once is stored in a transposition table after n is solved or proven to be unsolvable within the given proof-number threshold. In the case of collisions (different positions mapping onto the same transposition-table entry), the one investigated deepest is stored (the well-known *deep* replacement scheme).

When a node n turns out to be unsolvable within a given threshold, its proof number is stored in the transposition table. The proof number of an unsolved node n is larger than or equal to the proof-number threshold given to n. It can be larger when an AND node is expanded, since all the successor nodes contribute to the proof number even when a few successor nodes suffice to exceed the given proof-number threshold. This is efficient in order to minimize the effort of node re-expansions.

Next we consider the assignment of proof numbers to leaf nodes which have been expanded at previous iterations. When a leaf node n of type AND is expanded, the proof number of n is the number of the successor OR nodes generated, because the proof number of an unsolved leaf node is initialized as 1. However, when using an iterative-deepening method, some of the successor OR nodes may have been previously expanded and turned out to be unsolvable within the then given proof-number threshold. The previous results of expansions can be obtained from the transposition table. For the leaf nodes which have been previously expanded, the proof numbers are assigned as follows. If the leaf node n has been expanded previously and was unsolved within the proof-number threshold N, it is reasonable and efficient for the node n to obtain proof number N. Else if n has been already solved at the previous iteration, its proof number receives the value 0. Else if n turned out to be unsolvable previously, the proof number obtains value ∞. Only if n has never been expanded before, its proof number gets the default value 1. We call this dynamic method of assigning proof numbers *dynamic evaluation*.

The following lemmata are easily obtained from the definition of proof numbers for AND/OR tree search.

Lemma 3. *The number of iterations required to find a solution tree is at least equal to the maximum number of the successor nodes of all AND nodes in the solution tree.*

Lemma 4. *The number of iterations required to find a solution tree is at most equal to the number of the leaf nodes in the solution tree.*

The number of iterations needed to find a solution tree depends on the successor ordering of AND nodes. The optimal ordering at an AND node is the ordering by which the number of nodes generated to solve the AND node is minimized. The optimal ordering of a solution tree is therefore as follows. At OR nodes in the solution tree, it is efficient to order the successor AND nodes in best-first order, which means that the successor node solvable within the smallest proof-number threshold is expanded first, and within the largest is expanded last. This stems from the fact that solving a single successor of an OR node suffices, for which the one with smallest proof number is the best candidate. At the AND nodes of the solution tree, it is efficient to order the successor OR nodes in the worst-first order which means that the successor node solvable within the smallest proof-number threshold is expanded last and the successor node which is unsolvable is expanded first. In this case, since all successor nodes should be proven, if at least one successor disproves the goal, it is best to disprove such a successor as soon as possible, for which the one with highest proof number is the best candidate. Of course it is difficult to predict the proof number needed to solve successor OR nodes.

The transposition table is also used for the successor ordering of AND nodes. It is efficient to expand first the successor OR nodes which have never been expanded at previous iterations. Re-expanding the successor nodes which have already been searched deeper at a previous iteration is inefficient, because the overhead of the node re-expansions is expected to be high. The successor ordering of AND nodes is difficult in general [32], so it remains a task for future research.

4.5. Completeness and complexity

At first sight it seems possible that the PN* algorithm using proof numbers may not terminate. If an AND node has only one unsolved successor node, the proof-number thresholds given to the AND node and its successor unsolved OR node are equal. So the algorithm may continue to search an infinite move sequence with the same proof-number threshold. An artificial way to solve this is to increment additionally the proof numbers by 1 at each AND node [12]. By such a modification, the algorithm is guaranteed to terminate. However, PN* does not require such a modification, since by the domain of tsume-shogi we are guaranteed that this behavior will quickly lead to transpositions, which we detect by using transposition tables, and the node reached by a move sequence containing cycles is determined to be unsolvable. Moreover, the number of possible board positions of a tsume-shogi problem is finite, so a move sequence containing no cycles will never continue forever. We can ensure that entries in the transposition table referring to positions having occurred earlier on the path (i.e., denoting cycles) are never removed from the table. In this way termination of the algorithm is guaranteed.

PN* uses a recursive-iterative-deepening method, so at any node it always finds an optimal partial solution tree in the sense that it is found within the minimum proof-number threshold, on the condition that the ordering of successor nodes once generated stays fixed. Only the nodes on the path from a root to the currently generated node are kept in memory. The space complexity of PN* is therefore $\mathrm{O}(d)$, when d denotes the maximum depth of the search tree. The most-notable drawback of iterative-deepening methods is the overhead of node re-expansions. In the case that in some iteration the number of nodes that have not been expanded in previous iterations is relatively low, the ratio of the number of re-expanded nodes and the total number of expanded nodes becomes high. Then the algorithm may take much longer than other algorithms such as best-first search. For applications which require too much computation time due to the excessive re-expansions of nodes, a modification of depth-first iterative deepening method is presented in [38]. However, for our algorithm applied to tsume-shogi problems the experimental results in Section 7 show that the ratio of re-expansions is only about 20%, adequately small for using iterative deepening.

5. Dependencies among subproblems

The basic assumption of proof-number search is that subproblems are independent of each other, i.e., the probability that an unsolved subproblem becomes solved is regarded to be independent of other subproblems. There is a strong correlation between the proof number of a node n and the probability of n to be solved. So it is reasonable to expand primarily the leaf nodes of the pst with minimal proof number. But such independence does not hold in many applications [26].

In this section, the incorporation of dependencies between nodes in an efficient search is described. Two kinds of dependencies between nodes are considered. One of them is when the so-called dominance relation holds among the nodes (see Section 5.1); the other occurs when similarity relations between nodes hold (see Section 5.2). The transposition table is used not only to prevent redundant re-expansions of nodes when using recursive iterative deepening, but also to detect such dependencies among nodes. Section 5.3 briefly addresses the problem of using search graphs instead of trees.

5.1. Dominance relations

Even if two subproblems P_i and P_j are not equal, a dominance relation may hold [11, 14]. We say that P_i *dominates* P_j, denoted $P_i \, D \, P_j$, when P_i always has the same or a better solution than P_j. If P_j is solved, P_i is also solved, without need for searching it further.

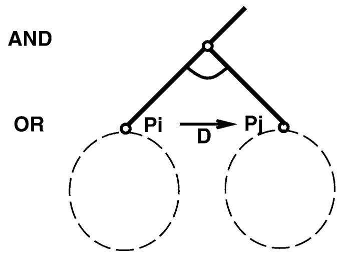

Fig. 5. A dominance relation between two nodes.

Conversely, if P_i is unsolvable, P_j is also unsolvable, so P_j needs not to be expanded further. Therefore, if the dominance relations are checked before expanding a node, the total number of generated nodes may be reduced considerably.

When an AND node is expanded, the dominance relation $P_i D P_j$ may hold between the successor nodes P_i and P_j (see Fig. 5). In such a case, it would be better to expand P_j before expanding P_i. The reason is as follows. If P_j is solved and $P_i D P_j$ holds, P_i can be denoted to be solved without being expanded. Moreover, if P_j is unsolvable, P_i needs not to be expanded because the parent node is of type AND.

In tsume-shogi, the dominance relation between two positions holds only in the following case. Let the set of captured pieces of Black at a position P_i be B_i, and the set of captured pieces of White at P_i be W_i. The dominance relation holds if the board which corresponds with the position P_i is the same as that for P_j, and B_i is a superset of B_j, which also means that W_i is a subset of W_j. In such a case, a dominance relation $P_i D P_j$ holds, and if P_j is solved, P_i can always be solved with the same or shorter move sequence as P_j, so P_i needs not be expanded. In tsume-shogi, no other dominance relations hold.

A typical example of a case where the dominance relation is used to prune nodes efficiently is as follows. The interposed dropping move by the defender on a vacant square not defended by other pieces, against the attacker's check by sliding pieces such as a (promoted) Rook, a (promoted) Bishop or a Lance, is called "Chu-Ai". "Chu-Ai" often becomes a useless move ("Muda-Ai"). Even if "Chu-Ai" moves are neglected and excluded from the possible moves of White, the correct solution can be obtained in many problems. But in some problems, a "Chu-Ai" is not a "Muda-Ai", and it becomes White's best defense. "Chu-Ai" often leads to positions among which many dominance relations hold. The most effective successor ordering at AND nodes in such cases is as follows. All moves except "Chu-Ai" are expanded first, and "Chu-Ai" moves are only considered later. Among "Chu-Ai", they are ordered according to increasing distance from the white King to the dropping square. By using such ordering, many positions can be pruned by using the dominance relation.

5.2. Similarity relations

It often occurs that subproblems do not dominate each other, but that they represent *similar* positions, solved by almost the same move sequence. When such probability is high, the moves leading to a solution for a position would be better to be tried first in solving similar positions. This technique resembles the killer heuristic or history heuristic in computer chess [1,39]. In the case of AND/OR tree search, the killer heuristic is used only at OR nodes.

Our tsume-shogi program uses the killer heuristic against interposing moves (called "Ai-Koma"), and also against non-promoting moves when having the right to promote (called "Narazu"). An interposing piece can be dropped or placed maximally at seven squares between the attacker's piece giving check (when at the first or ninth rank or file) and the opponent King (when at the ninth or first rank or file). So if in such a case the seven positions are solved independently, it would take much longer without the killer heuristic than with it. A human expert would immediately try the same move sequence by the attacker. Similar reasoning holds for moves where White refrains from promoting a piece. For these two cases we have implemented the killer heuristic using the transposition table. If such a position n is solved, then other similar positions are solved by exploring first the best move at n.

5.3. Graphs versus trees

The above cases can be efficiently solved by using a transposition table. But when the solution is a graph instead of a tree (i.e., when transpositions are present), PN* may need more time than would be expected from the difficulty as conceived by human experts to solve the problem. The reason is that when the solution is a cycle and two paths out of an AND node meet again at an OR node n later, PN* counts the proof number of n twice within the same pst, so the number of iterations needed to find the solution tree becomes larger than the solution's true proof number as a graph (see Fig. 6).

Of the five problems from the book "Zuku-Tsumuya-Tsumazaruya" which initially could not be solved (see Section 7), no. 48 and no. 49 are problems of this kind: the solution of each problem is a graph and the proof numbers counted double are considerably large. Especially the structure of the solution graph of no. 49 is very complicated, the problem being appropriately called "A Big Labyrinth".

Although some work on the problem of proof numbers in graphs has been performed (see [42]), the improvement of PN* for this behavior remains to be done as future research.

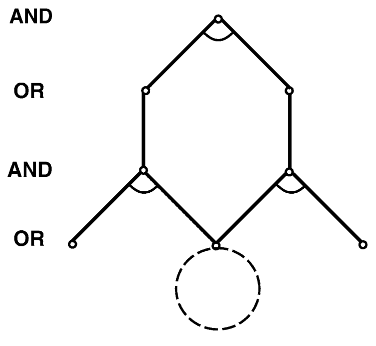

Fig. 6. An example of a potential solution graph.

6. Implementation of the PN* algorithm

Below we present the procedure of the PN* algorithm in pseudocode. The data for an entry to be stored in the transposition table are the board configuration (8 bytes), the proof number (2 bytes), the best move (2 bytes), the solution length (2 bytes), and the size of the solution tree (4 bytes).

The first procedure is the top procedure, guiding the search until the root node has been proven or disproven. The proof-number threshold of the root is initialized to 1 and is augmented during each expansion of the root, until the root is proven or disproven.

```
comment Iterative deepening at the root;
procedure ITERATE:
  comment Initialize the proof-number threshold at the
    root;
  $PN(r) \leftarrow 1$
  while do begin
    EXPAND$(r, PN(r))$
    if $PN(r) = 0$ then
      return proven
    if $PN(r) = \infty$ then
      return disproven
  end
```

The next procedure is the main procedure, which expands nodes, assigns proof numbers, reads and writes the transposition table. It uses recursive iterative deepening and dynamic evaluation. Details concerning the use of dominance relations and move ordering are omitted for the sake of clarity.

```
comment Expand the leaf nodes of the pst rooted at
  node $n$. $PN(n)$ is the proof-number threshold given
  to node $n$, and modified when returned;
procedure EXPAND$(n, PN(n))$:

  comment Look up node $n$ in the transposition table;
  $c \leftarrow$FINDTABLE$(n)$
  comment In case $n$ is already solved or its proof
    number exceeds the given threshold;
  if $c = 0$ or $c > PN(n)$ then begin
    $PN(n) \leftarrow c$
    return
  end
```

Continued on next page

Continued from previous page

```
comment Expand a node $n$ generating all its
  successor nodes;
if GENERATE$(n) = 0$ then begin
  comment Node $n$ has no successors;
  if $n$ is an AND node then begin
    $PN(n) \leftarrow 0$
  end
  if $n$ is an OR node then begin
    $PN(n) \leftarrow \infty$
  end
  PUTINTABLE$(n)$
  return
end

comment To detect cycles, the proof-number threshold
  given to $n$ is stored in the transposition table;
PUTINTABLE$(n)$

if $n$ is an OR node then begin
  comment The recursive call of the successor
    AND nodes;
  for each successor $n_i$ do begin
    EXPAND$(n_i, PN(n))$
    if $PN(n_i) = 0$ then
      break
  end
  comment $PN(n)$ is revised and the information of $n$
    is stored in the transposition table;
  $PN(n) \leftarrow \min_i(PN(n_i))$
  PUTINTABLE$(n)$
  return
end

if $n$ is an AND node then begin
  comment Dynamic evaluation;
  for each successor $n_i$ do begin
    $PN(n_i) \leftarrow$ FINDTABLE$(n_i)$
  end
```

Continued on next page

Continued from previous page

```
    comment When the sum of the proof numbers of the
      successor OR nodes exceeds the proof-number
      threshold of $n$;
    if $\sum_i PN(n_i) > PN(n)$ then begin
      $PN(n) \leftarrow \sum_i PN(n_i)$
      PUTINTABLE($n$)
      return
    end

    comment The recursive call of the successor
      OR nodes;
    for each successor $n_i$ do begin

      comment Recursive iterative deepening;
      $PN(n_i) \leftarrow 1$
      while do begin
        EXPAND($n_i, PN(n_i)$)
        comment In case $n_i$ is solved;
        if $PN(n_i) = 0$ then
          break

        comment In case $n_i$ is unsolved within
          the threshold;
        if $\sum_i PN(n_i) > PN(n)$ then begin
          $PN(n) \leftarrow \sum_i PN(n_i)$
          PUTINTABLE($n$)
          return
        end
      end
    end

    comment All successor OR nodes of node $n$ are
      solved;
    $PN(n) \leftarrow 0$
    PUTINTABLE($n$)
    return
  end
```

The next three procedures are self-explanatory, and are only given for reasons of completeness.

```
comment The procedure for locating the entry of $n$ in
    the transposition table;
procedure FINDTABLE($n$):
    if the entry of $n$ is in the transposition table then
    begin
        return $table(n)$
    end
    return 1
```

```
comment Simplified procedure of saving the information
    of $n$ in the transposition table;
procedure PUTINTABLE($n$):
    $table(n) \leftarrow PN(n)$
```

```
comment Generate all successor nodes;
procedure GENERATE($n$):
    Generate all successor nodes of node $n$
    return the number of successor nodes
```

7. Experimental results and some analysis

7.1. Zoku-Tsumuya-Tsumazaruya

We selected the set of tsume-shogi problems from the book “Zoku-Tsumuya-Tsumazaruya” [18] as a suite of benchmark problems. It contains 203 numbered problems (200 problems, with 1 problem subdivided into 4 subproblems) from the Edo era to the Showa era, created by 41 composers. The shortest problem in it is a 11-step problem and the longest one has a solution of 873 steps. The set contains various types of problems. Generally it is considered a good benchmark to measure the performance of a tsume-shogi-solving program.

Some problems are omitted from the test set, since they are not suited for testing purposes. These comprise three groups. The first group of five problems are the ones without solutions, because the creator of the problem failed to notice the best moves of White, leading to unsolvable positions. These problems are called “fu-tsume”, and are of course regarded as incomplete. The second group of two problems concerns the problems intended to have no solutions. The creator demands that the solver finds the move sequence by White to escape from mate, called “Nogare-Zushiki”. A third problem type omitted is when the problem contains a special piece called “Suizou”, which is used in the game “Chu-Shogi”. There was only one problem of this kind.

After exclusion of the eight unsuitable problems, the test set consisted of 195 tsume-shogi problems. The experimental results of the tsume-shogi-solving programs T2 and ITO for this suite are as follows: T2 solved 70 problems, and ITO solved 135 problems [15,23].

We have tested the performance of our program SEO using the same test set. In all experiments, the length of the solution sequence of each problem is not input to the program in advance. Moreover, the search depth is not artificially limited to the length of the solution sequence. Finally, the number of moves to be expanded at each position is not restricted, like done in some forward-pruning programs.

A complete tsume-shogi problem has only one solution sequence, called “Sakui-Tejun”, intended by the creator. However, in general there are several solution trees even in complete problems. At least one move sequence from the root to a terminal AND node is contained in all the solution trees, and the longest sequence of them is the solution sequence “Sakui-Tejun”. Some solution trees may contain a move sequence longer than “Sakui-Tejun” which would not be generated if the program searched first the moves in the “Sakui-Tejun”. If the program searched one of these solution trees, it answers a solution sequence different from the “Sakui-Tejun”. The solution sequence of this type is called “henka-betsu-tsume”. It means that the program does solve the problem correctly, though not giving the intended solution. This can happen in PN* search, since it is based on iteratively enlarging the proof-number threshold, and not the search depth. In Fig. 7 an example is shown.

For the 195 tsume-shogi problems in the “Zoku-Tsumuya-Tsumazaruya” test suite the time limit was set to 2 hours per problem on a Sun Sparc Station 20, using two transposition

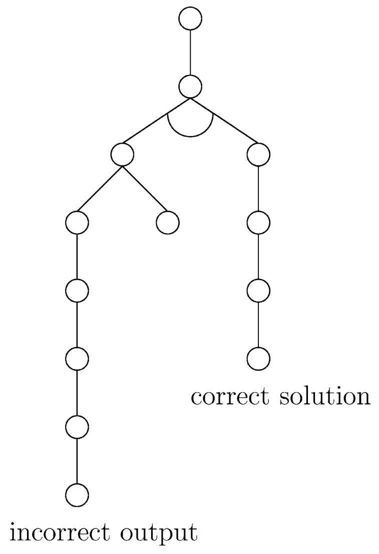

Fig. 7. An example of “henka-betsu-tsume”.

Table 1
The number of solved problems in the "Zoku-Tsumuya-Tsumazaruya" test suite for various programs

Program	# of problems solved	Ratio (%)
ITO	135	69.2
T2	70	35.9
SEO	190	97.4
SEO(New)	194	99.5

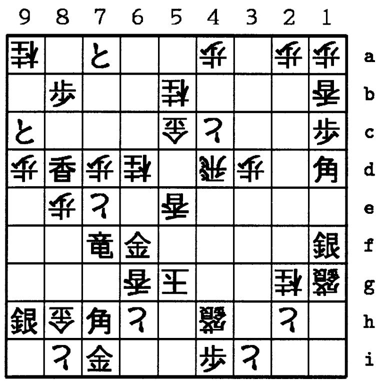

Fig. 8. Problem no. 200 (873 steps) from the "Zoku-Tsumuya-Tsumazaruya" test suite.

tables (one for Black, one for White) of 500k entries each. Since the storage requirement is 16 bytes per entry, a total of 16 MB memory is required for transposition tables. The results are summarized and compared to other results in Table 1. Complete details are given in [43]. The average ratio of re-expansions of the 190 problems solved is calculated as 0.208, with an average branching factor of 5.23.

The five unsolved problems were later subject to a new analysis, this time on a 400 MHz K6-2 machine. The transposition-table size on this machine was set to 7M entries per table (again, one for White, one for Black), for a total requirement of 224 MB memory. With these enlarged transposition tables these problems were easily solved (all within a few minutes) except one. The inclusion of these latest data leads to the last row in Table 1.

The length of the longest solution sequence from this test suite solved by our program is 873 steps for problem no. 200, shown in Fig. 8. It was a new world record solved by computers when we first tried this test suite (1995). The previous record is 611 steps (which is the no. 55 problem in "Zoku-Tsumuya-Tsumazaruya") achieved by the programs KAWANO, ITO, and T3 in 1994 [16].

The only unsolved problem is no. 51 (see Fig. 9). The solution length of it is only 41 steps, which is not so long. The reason that it could not be solved is that the solution tree of the problem is rather wide, since the branching factor in the AND nodes is large. Therefore, the proof numbers in the solution tree are very large, so the algorithm selectively

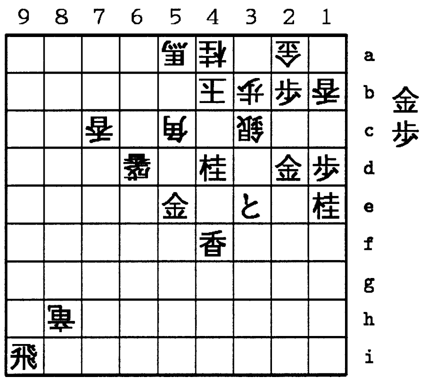

Fig. 9. Problem no. 51 (41 steps) from the “Zoku-Tsumuya-Tsumazaruya” test suite.

Table 2
The number of solved problems in the “Shogi-Zuko” test suite for various programs

Program	# of problems solved	Ratio (%)
ITO	62	62.0
T3	68	68.0
KAWANO	92	92.0
SEO	99	99.0

chooses other pst’s with smaller proof numbers of the roots for expansion instead of the real solution tree.

7.2. Shogi-Zuko

Though “Zoku-Tsumuya-Tsumazaruya” is a good benchmark set containing various types of problems, it has the following two disadvantages.

(1) It has relatively many incomplete problems, i.e., “yo-tsume” or “fu-tsume”.
(2) “Zoku-Tsumuya-Tsumazaruya” is not being used by recent tsume-shogi solving programs, so only the experimental results of other programs obtained several years ago [in 1992 or 1993] are available.

“Shogi-Zuko” is a set of 100 tsume-shogi problems composed by Kanju Ito in the Edo era [17]. The problems contained in it are not only hard to solve, but are also very artistic. “Shogi-Zuko” is sophisticated and the number of incomplete problems is relatively small. There are two “fu-tsume” problems in it, but revised versions of those were created by other persons later, and the solution sequences of these revised problems are the same as the original ones created by Kanju Ito. So we use the revised problems for the two “fu-tsume” problems (no. 73 and no. 93).

The results of our program for the “Shogi-Zuko” test suite are summarized and compared to other results in Table 2. Complete details are given in [43].

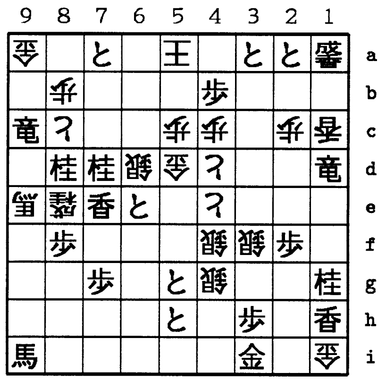

Fig. 10. Microcosmos (1525 steps).

Our program solved 99 problems within 2 hours for each problem on a Sun Sparc Station 20. The only unsolved problem is no. 8, which is the same problem as no. 51 in the “Zoku-Tsumuya-Tsumazaruya” test suite.

7.3. *Microcosmos*

Stimulated by our good results we tried SEO on a well-known problem, called “Microcosmos” (see Fig. 10). This is a tsume-shogi problem with a solution sequence of 1525 (!) steps, the longest known at present.

Though it is known as a very hard problem, SEO solved it using 20 hours and 52 minutes of CPU time on a 166 MHz Pentium. Of course this is again a new world record regarding the length of the solution sequence of tsume-shogi problems solved. The solution required 67 iterations, with a total of 550,939,913 nodes generated, 124,173,118 nodes expanded and 30,501,358 nodes re-expanded.

8. Conclusions

We developed the PN* algorithm, a depth-first iterative-deepening algorithm for AND/OR trees, based on proof numbers. PN* is based on the probability of finding a solution tree and can find the solution tree of many AND/OR trees easily. It is enhanced by such methods as recursive iterative deepening, dynamic evaluation, efficient successor ordering, and pruning by dependency relations. A transposition table is indispensable for these enhancements.

We implemented the PN* algorithm in a tsume-shogi-solving program called SEO, and performed experiments with a suite of 195 tsume-shogi benchmark problems from “Zoku-Tsumuya-Tsumazaruya”. The results shows that 194 problems can be solved within 2 hours per problem on a Sun Sparc Station 20 or a 400 MHz K6-2 computer. The time needed to solve these problems is much shorter than human experts use on average. The results are also far superior to the results of other tsume-shogi programs, which provides support for the usefulness of the PN* algorithm. The problems solved from this test suite included problem no. 200, called “shin-ougi-tsume”, whose solution requires 873 steps. SEO also tried a second test suite, 100 problems in “Shogi-Zuko”, and solved 99 of them. Moreover, the problem called “Microcosmos” with a solution sequence of 1525 steps has been solved by SEO. It is a new record for the length of a solution sequence of a tsume-shogi problem solved by computers.

In conclusion, PN* is an efficient algorithm for problem solving using AND/OR trees. In contrast to PN search it does not have the drawback of typical best-first search algorithms, viz. the need for large amounts of memory.

Since tsume-shogi search can be applied in the endgame of shogi also, it is worthwhile to investigate whether PN* search can be profitably combined with a heuristic searcher in a shogi-playing program. Of course the strict conditions of tsume-shogi can then be relaxed, like the constraint that any move by the attacker should be a check and that no alternate solutions may be present. Also in other games the application of PN* search could be beneficial. This could be both in problem-solving type of applications (endgame positions, problem compositions) and in solving games hitherto unsolvable. For instance, it is an intriguing challenge to investigate whether the use of PN* search could accelerate the solution of checkers, which so far has withstood this attempt [41], but surely is on the list of games expected to be solved within one or at most a few decades.

Acknowledgement

The authors gratefully acknowledge Jaap van den Herik’s comments and suggestions on earlier versions of this paper. Without his stimulating impulse the paper would not have been what it is now. Moreover, the comments of three anonymous referees have been very helpful.

References

[1] S.G. Akl, M.M. Newborn, The principal continuation and the killer heuristic, in: 1977 ACM Annual Conference Proceedings, Seattle, WA, 1977, pp. 466–473.

[2] L.V. Allis, M. van der Meulen, H.J. van den Herik, Proof-number search, Artificial Intelligence 66 (1) (1994) 91–124.

[3] L.V. Allis, Searching for solutions in games and artificial intelligence, Ph.D. Thesis, University of Limburg, Maastricht, The Netherlands, 1994.

[4] D.M. Breuker, L.V. Allis, H.J. van den Herik, How to mate: Applying proof-number search, in: H.J. van den Herik, I.S. Herschberg, J.W.H.M. Uiterwijk (Eds.), Advances in Computer Chess 7, University of Limburg, Maastricht, The Netherlands, 1994, pp. 251–272.

[5] D.M. Breuker, Memory versus search in games, Ph.D. Thesis, Universiteit Maastricht, Maastricht, The Netherlands, 1998.

[6] D.M. Breuker, J.W.H.M. Uiterwijk, H.J. van den Herik, The PN^2-search algorithm, in: H.J. van den Herik, B. Monien (Eds.), Advances in Computer Games 9, Universiteit Maastricht, Maastricht, The Netherlands, 2001, to appear. Also published as Technical Reports in Computer Science CS 99-04, Department of Computer Science, Universiteit Maastricht, Maastricht, The Netherlands, 1999.

[7] P.P. Chakrabarti, S. Ghose, S.C. DeSarkar, Admissibility of AO* when heuristics overestimate, Artificial Intelligence 34 (1) (1987) 97–113.

[8] P.P. Chakrabarti, Algorithms for searching explicit AND/OR graphs and their applications to problem reduction search, Artificial Intelligence 65 (2) (1994) 329–345.

[9] C.L. Chang, J.R. Slagle, An admissible and optimal algorithm for searching AND/OR graphs, Artificial Intelligence 2 (2) (1970) 117–128.
[10] P. Dasgupta, P.P. Chakrabarti, S.C. DeSarkar, Agent searching in a tree and the optimality of iterative deepening, Artificial Intelligence 71 (1) (1994) 195–208.
[11] M.V. Donskoy, Fundamental concepts in search, ICCA Journal 13 (3) (1990) 133–137.
[12] C. Elkan, Conspiracy numbers and caching for searching AND/OR trees and theorem-proving, in: Proc. IJCAI-89, Detroit, MI, 1989, pp. 341–346.
[13] M. Hirose, H. Matsubara, T. Itoh, The composition of tsume-shogi problems, in: H.J. van den Herik, J.W.H.M. Uiterwijk (Eds.), Advances in Computer Chess 8, Universiteit Maastricht, Maastricht, The Netherlands, 1997, pp. 299–318.
[14] T. Ibaraki, The power of dominance relations in branch-and-bound algorithms, J. ACM 24 (2) (1977) 264–279.
[15] T. Ito, K. Noshita, Two fast programs for solving tsume-shogi and their evaluation, J. Information Processing Society of Japan 35 (8) (1994) 1531–1539 (in Japanese).
[16] T. Ito, Y. Kawano, K. Noshita, Challenges for solving tsume-shogi with extremely long solution steps, in: Programming Symposium, Information Processing Society of Japan, 1994, pp. 85–92 (in Japanese).
[17] Y. Kadowaki, Tsumuya-Tsumazaruya, Shogi-Muso, Shogi-Zuko, Heibon-Sha, Tokyo, 1975 (in Japanese).
[18] Y. Kadowaki, Zoku-Tsumuya-Tsumazaruya, Heibon-Sha, Tokyo, 1978 (in Japanese).
[19] Y. Kawano, Using similar positions to search game trees, in: R.J. Nowakowski (Ed.), Games of No Chance, MSRI Publications 29, Cambridge University Press, Cambridge, 1996, pp. 193–202.
[20] D.E. Knuth, R.W. Moore, An analysis of alpha-beta pruning, Artificial Intelligence 6 (4) (1975) 293–326.
[21] R.E. Korf, Depth-first iterative-deepening: An optimal admissible tree search, Artificial Intelligence 27 (1) (1985) 97–109.
[22] R.E. Korf, Linear-space best-first search, Artificial Intelligence 62 (1) (1993) 41–78.
[23] Y. Kotani, Computer Shogi Association Material, Vol. 6, Computer Shogi Association, Electrotechnical Laboratory, Tsukuba, Japan, 1992 (in Japanese).
[24] V. Kumar, L.N. Kanal, A general branch and bound formulation for understanding and synthesizing AND/OR tree search procedures, Artificial Intelligence 21 (1–2) (1983) 179–198.
[25] V. Kumar, L.N. Kanal, Parallel branch-and-bound formulations for AND/OR tree search, IEEE Transactions on Pattern Analysis and Machine Intelligence 6 (6) (1984) 768–778.
[26] G. Levi, F. Sirovich, Generalized AND/OR graphs, Artificial Intelligence 7 (3) (1976) 243–259.
[27] A. Mahanti, A. Bagchi, AND/OR graph heuristic search methods, J. ACM 32 (1) (1985) 28–51.
[28] A. Martelli, U. Montanari, Optimizing decision trees through heuristically guided search, Comm. ACM 21 (1978) 1025–1039.
[29] H. Matsubara, Shogi (Japanese chess) as the AI research target next to chess, ETL Technical Report ETL-TR-93-23, Electrotechnical Laboratory, Tsukuba, Japan, 1993.
[30] D.A. McAllester, Conspiracy numbers for min-max search, Artificial Intelligence 35 (3) (1988) 287–310.
[31] Y. Nakayama, T. Akazawa, K. Noshita, A parallel algorithm for solving hard tsume-shogi problems, ICCA Journal 19 (2) (1996) 94–99.
[32] N.J. Nilsson, Problem-Solving Methods in Artificial Intelligence, McGraw-Hill, New York, 1971.
[33] N.J. Nilsson, Principles of Artificial Intelligence, Morgan Kaufmann, San Mateo, CA, 1980.
[34] J. Pearl, Heuristics: Intelligent Search Strategies for Computer Problem Solving, Addison Wesley, Reading, MA, 1984.
[35] A. Plaat, J. Schaeffer, W. Pijls, A. de Bruin, Best-first fixed-depth minimax algorithms, Artificial Intelligence 87 (1–2) (1996) 255–293.
[36] A. Plaat, Research Re: search & Re-search, Ph.D. Thesis, Erasmus University Rotterdam, Rotterdam, The Netherlands, 1996.
[37] A. Reinefeld, T.A. Marsland, Enhanced iterative-deepening search, IEEE Transactions on Pattern Analysis and Machine Intelligence 16 (7) (1994) 701–710.
[38] U.K. Sarkar, P.P. Chakrabarti, S. Ghose, S.C. DeSarkar, Reducing reexpansions in iterative-deepening search by controlling cutoff bounds, Artificial Intelligence 50 (2) (1991) 207–221.
[39] J. Schaeffer, The history heuristic and alpha-beta search enhancements in practice, IEEE Transactions on Pattern Analysis and Machine Intelligence 11 (11) (1989) 1203–1212.
[40] J. Schaeffer, Conspiracy numbers, Artificial Intelligence 43 (1) (1990) 67–84.

[41] J. Schaeffer, R. Lake, Solving the game of checkers, in: R.J. Nowakowski (Ed.), Games of No Chance, MSRI Publications 29, Cambridge University Press, Cambridge, 1996, pp. 119–133.

[42] M. Schijf, L.V. Allis, J.W.H.M. Uiterwijk, Proof-number search and transpositions, ICCA Journal 17 (2) (1994) 63–74.

[43] M. Seo, H. Iida, J.W.H.M. Uiterwijk, Full details on experimental results described in this paper (2000). URL http//www.cs.inf.shizuoka.ac.jp/~iida/PN$*$-data.html.

[44] J.R. Slagle, J.K. Dixon, Experiments with some programs that search game trees, J. ACM 16 (2) (1969) 189–207.