Solving Kriegspiel-like Problems: Examining Efficient Search Methods

Makoto Sakuta and Hiroyuki Iida

Department of Computer Science, Shizuoka University
3-5-1 Johoku, Hamamatsu, 432-8011 Japan
sakuta@cs.inf.shizuoka.ac.jp, iida@cs.inf.shizuoka.ac.jp

Abstract. We recently proposed a deterministic approach for solving problems with uncertainty, called the Uncertainty Paradigm. Under this paradigm, deterministic solving of such a problem is resolved into a plain AND/OR-tree search. The search under this paradigm is denoted by the Uncertainty Paradigm Search (UPS). As an application, we have chosen the domain of Tsuitate-Tsume-Shogi, which is the mating problem of Kriegspiel-like variant of Shogi. The early implementation of UPS was based on a simple depth-first full-width search with iterative deepening (ID), which was unable to solve several hard problems. This paper explores an efficient search method using UPDS (Uncertainty Paradigm PDS) algorithm, which is a specialized version of PDS (Proof-number and Disproof-number Search) for UPS. UPDS generally performs better than ID or PDS, but fails to solve some easy problems. In addition, several variations of UPDS and ID have also been examined to tackle some hardest problems. All problems in the test set have been solved by a particular variation of UPDS, which shows superiority of the depth-first variants of the proof-number search in UPS.

Keywords: Problem solving, Uncertainty Paradigm, Metaposition, Metamove, UPS, Tsuitate-Tsume-Shogi, PDS, Kriegspiel

1 Introduction

In the domain of computer-game research, most efforts have been devoted to the programming of games with complete information such as chess. There is less literature on computer-game research using imperfect-information games.

We know that developing a program of an incomplete-information game is usually much more difficult than developing the program of a perfect-information game. For searching the game tree with incomplete information, a program has to combine a tree search with information sets and find the most reasonable strategy, according to the conventional approach of a game in extensive form. The information sets are somewhat apart from the tree structure, so the search cannot be resolved into a plain tree search.

However, there are some kinds of problems with uncertainty or some kinds of endgames of the imperfect-information games that can be solvable deterministically. Nevertheless, there are a few studies on the deterministic solving of problems with uncertainty. We have tackled deterministic solving of the problems with uncertainty and succeeded in two domains by introducing the novel paradigm, the Uncertainty Paradigm

[19]. Moreover, we have enhanced the search method based on a technique of a transposition table [17].

In two-agent problems with complete information such as Tsume-Shogi, very efficient search algorithms with high solving ability have been developed. In this paper, we examine one of these algorithms together with our approach, and show the results of applying the search algorithm to the problems of Tsuitate-Tsume-Shogi. In addition, we also examine some variations of search algorithms to tackle hardest problems.

2 Uncertainty Paradigm for Problems with Incomplete Information

2.1 Overview

The Uncertainty Paradigm is a way of thinking that recognizes an uncertain situation as if it is a definite situation. Under this paradigm, an uncertain situation is a hybrid of several certain situations. Here we give a brief sketch of this paradigm. We use the term *solver* to mean the player or agent to solve the problem, and *opponent* as the players or agents to hinder the problem being solved.

A *metaposition* is a hybrid of possible positions that are not distinguishable for the solver. A metaposition corresponds to a super node that represents the positions in an information set of the solver, in a game in extensive form [13]. The count of possible positions of a metaposition can be recognized as an *uncertainty index*.

A *metamove* is simply a move for a metaposition. When a metamove for a metaposition is a definite move, making the metamove corresponds to making the move for each position of the metaposition. When a metamove for a metaposition is a hybrid of several moves, making the metamove corresponds to making all the moves for each position of the metaposition. Consequently, the number of positions of the metaposition increases. This is a *diffusion* of the metaposition. If the solver cannot get any clue in such metamoves, the uncertainty of the metaposition soon results in combinatorial explosion. However, there are *observations* or clues that help the solver. By these observations, a metaposition is split into several metapositions with less uncertainty, i.e., metapositions that have fewer positions. Since the solver has to accept this split in a passive manner, he has to solve all the split metapositions. Therefore, this split is an *AND-split* of a metaposition even in case of the single-agent puzzles.

For all solving problems with uncertainty, the search graph (tree) under the Uncertainty Paradigm is a plain AND/OR graph (tree), which we call the Uncertainty AND/OR graph (tree). In such a graph (tree), a node is a metaposition and an edge is a metamove. In addition, the search under the Uncertainty Paradigm is a plain AND/OR-graph (-tree) search, which we call the *Uncertainty Paradigm Search (UPS)*.

2.2 Related Works

The Uncertainty Paradigm is a new approach for the problems with incomplete information. However, since our approach omits the probabilities and focuses on deterministic solving, the applicable domains are restricted.

Our approach recognizes an information set as a node of a search tree. In the area of computer games with incomplete information, a similar approach is known. The work by Frank and Basin [4] uses "vectors" at nodes to conduct a game tree search. In the same way as our approach, these vectors represent collections of nodes that come from the same information sets. The differences from our approach are that a search is not for an AND/OR tree but for a minimax tree with probability, and that in Bridge the uncertainty is greatest at the start point of a game and decreases as the game proceeds. Consequently, a vector with a fixed length can reasonably used in their methods. In our approach, the uncertainty changes dynamically in searching. That is to say, the uncertainty sometimes increases by the diffusion of a metaposition and sometimes decreases by the split of a metaposition. Though representation of fix-length vectors can be applied by using very large vectors in our problems, it is quite inefficient because the maximum uncertainty in a search is unknown at the beginning of a search.

There is little study in computing chess-like games with incomplete information. In the domain of Kriegspiel [14], Ferguson studied an endgame problem whether the king, bishop and knight can win against the king alone and showed the result that the player with bishop and knight can win with probability one in general (with king initially guarding both bishop and knight) [3]. And Ciancarini *et al.* reported on their works on some simple endgames of Kriegspiel and showed the performance of their program that can play KPK endgames by the knowledge-based approach [2]. However, there had been no study on solving Kriegspiel problems deterministically like ours. In addition, there had been no study about computing Tsuitate-Shogi.

3 Tsuitate-Tsume-Shogi

As the application of the Uncertainty Paradigm to the two-agent problem with uncertainty, we have selected the domain of *Tsuitate-Tsume-Shogi*. We first refer to Tsuitate-Shogi, then explain Tsuitate-Tsume-Shogi.

3.1 Tsuitate-Shogi

Tsuitate-Shogi is a Shogi variant that is one of the best known and most popular among all Shogi variants, as Kriegspiel is in chess. We read 'Tsuitate' (in Japanese) as a screen, therefore we call Tsuitate-Shogi (screen Shogi).

The rules of Shogi are available on the Internet, for example, at:

http://www.jwindow.net/LWT/SHOGI/INTRO/shogi_intro.html.

The most important difference between Shogi and chess is that the players can re-use the captured pieces in Shogi. A captured piece becomes a *piece in hand* for the side that captured it. There are two types of moves in Shogi. One type of move is moving a piece on the board to another square on the board. Another type of move is putting one of the pieces in hand onto a vacant square on the board, which is called *dropping* a piece.

Let us give a short summary of the rules of Tsuitate-Shogi. Note that the rules of Tsuitate-Shogi seem mostly similar to Kriegspiel. Tsuitate-Shogi requires two players, a referee, two Shogi boards with a set of pieces, and a screen. Both players are unable to see the other side because of the screen. The principle of a game is that each player moves normally but he is not told the opponent's moves, which he has to guess through judicious play.

After one plays a move on his own board, the referee approves the move to provide information to both players as required by the rules. Information is given to both players by the referee saying 'Illegal, ... times', 'Black/White has played', 'Check', and 'Mate'. The referee must not tell other information aloud. The referee performs capturing on the board instead of the players.

Each player is allowed to make illegal moves within a certain number of times, typically eight times. If one has made illegal moves more than the allowed number of times, he loses the game on fouls. A game also ends when the king on either side is in mate.

3.2 Rules of Tsuitate-Tsume-Shogi

Tsuitate-Tsume-Shogi (TTS) is a mating problem of Tsuitate-Shogi [7], which is also a variant of Tsume-Shogi, a mating problem of Shogi [6]. Here we give a summary of the rules of TTS. Note that the rules of TTS seem similar to Tsume-Shogi except several points. Since TTS is a two-agent problem, there are two agents: the *attacker* as the *solver* and the *defender* as the *opponent*.

Because the attacker's king often has no meaning in TTS, most TTS problems only have the defender's king but do not have the attacker's king. Nevertheless, there are some problems that have both kings, which we call *problems with double kings*. In TTS, the attacker is unable to see the opponent's pieces and responses except for the initial position of a mating problem. The goal of solving a TTS problem is to mate the opponent's king after the sequence of check moves and their responses. However, the attacker is unable to see the definite information on the defender's responses, while the defender has the perfect information on both sides. The attacker must lead to a checkmate whatever moves the defender may play.

In Tsuitate-Shogi, an illegal move is checked and announced by the referee. Hence, a player can even make an explicit nonsense move as a feint merely in order to confuse the opponent. In TTS, trying an illegal move is also allowed. However, there is no meaning of trying an explicit nonsense move, because it is illegal at all positions of the metaposition and the attacker cannot get any information by trying it. Here we have chosen a strict definition. We have defined a *foul move* as a move that belongs to a subset of possible illegal moves. A foul move is a move that seems to be legal with only the attacker's pieces on the board, but is illegal when all pieces are considered. Thus, the foul moves are:[1]

1. A sliding piece (rook, bishop, lance) jumping over an opponent piece.
2. Dropping a piece onto a square where there is an opponent piece.
3. A pawn-drop move that causes the dropped pawn mate.
4. A move after which the king of the attacker remains in check or comes to be in check, for problems with double kings.

[1] Moreover, there may be a foul move in violation of the rule that forbids a fourfold repetition of a certain position with the successive check moves and their responses. However, since there is at present no problem that needs this type of foul moves, we have not yet implemented it.

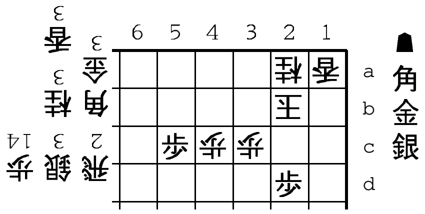

Fig. 1. A sample problem

The attacker is allowed to try the foul moves up to a certain number of times, typically eight times or less. If the attacker has tried a foul move, the count of fouls increases by one and he only knows that it is a foul move. He does not know a type of the foul. The position does not change at all and the attacker has to choose another move that is possibly a check move. The attacker cannot make a foul move if the count of fouls exceeds the allowed limit. Notice that the attacker cannot cancel a move that has been tried once, and it has to be a check move if it has turned out to be a legal move. If the attacker has made a legal move that does not check the king, it turns out that solving is unsuccessful.

Here, we define the concept *steps* of the sequence as the sum of both the number of the check moves of the attacker and the number of the responses of the opponent from the initial (meta)position to the mated (meta)position in a certain solution tree. In TTS as well as in Tsume-Shogi, the steps are equal to the number of plies of a sequence. The longest steps of a sequence among all sequences in the solution tree are defined as the steps of the solution.

3.3 A Sample Problem of TTS and its Solution

Here, let us show a sample TTS problem (Figure 1), which is a definite initial position, and its solution. In this figure, 玉, 飛, 角, 金, 銀, 桂, 香, and 歩 denote a piece of king, rook, bishop, gold, silver, knight, lance, and pawn of the attacker, respectively. Similarly, 王, 飛, 角, 金, 銀, 桂, 香, and 歩 denote a piece of king, rook, bishop, gold, silver, knight, lance, and pawn of the defender, respectively. The pieces beside the board on the right side are the pieces in hand of the attacker, and those on the left side are the pieces in hand of the defender.

The solution is given below. The attacker first makes the silver-dropping move **1. 銀*2c**. There are two moves as its response, **1. ...玉1c** and **1. ...玉3a**. Next, the attacker tries to make the move **2. 角*1c**. If this move is not allowed (foul type 2), the attacker knows that the king is at **1c**. Then the attacker can mate the king by dropping a gold onto **1d**.

If **2. 角*1c** is allowed, the attacker knows that the king is at **3a**. Then the defender has eleven possible moves as the response. Two moves are **2. ...桂x1c** and **2. ...香x1c**. The attacker knows these moves because the bishop at **1c** disappears by capturing and can mate the king by dropping a gold onto **3b**.

There still remain nine possible moves as the second response by the defender. Two moves are **2. …玉4a** and **2. …玉4b**. The other seven moves are the interposing moves by dropping a piece onto **2b**. Next, the attacker tries to make the move **3. 角3a+** (with promotion). If this move is not allowed (foul type 1), the attacker knows that the defender has just made an interposing move. Then the attacker can mate the king by dropping a gold onto **3b**.

If **3. 角3a+** is allowed, the attacker knows that the king is at **4a** or **4b**. Then the defender has two possible moves as the response. One move is **3. …玉x3a**. The attacker knows this move because the promoted bishop at **3a** disappears by capturing and can mate the king by dropping a gold onto **3b**. If no capturing has occurred, the attacker knows that the defender has made another move **3. …玉5a** and can mate the king by dropping a gold onto **5b**.

The longest solution sequence with seven steps is either of the below:

1. 銀*2c (–) 2. 角*1c (–) 3. 角3a+ (3a) 4. 金*3b mate.

1. 銀*2c (–) 2. 角*1c (–) 3. 角3a+ (–) 4. 金*5b mate.

Here, '**(–)**' denotes a response of the defender after which no capturing has occurred, and '(xy)' denotes a response after which the piece at the square xy has been captured by some defender's piece.

3.4 TTS Problem under the Uncertainty Paradigm

Since TTS has uncertainty, the information sets of TTS often consist of many positions. We have dealt with the TTS problems under the Uncertainty Paradigm. In TTS, every metamove at the solver's turn is one of the definite moves and every metamove at the opponent's turn is the uncertain hybrid of moves.

At a certain metaposition where the solver is to move, possible moves are the moves that are either check or foul moves for all positions of the metaposition. One of the possible moves is the metamove to the metaposition. However, it is hopeless to make the metamove that is illegal for all positions of the metaposition, because the solver cannot get any information by making it. No split of the metaposition occurs and the situation is the same as before making that metamove except for increasing the count of fouls. Therefore, the metamove that is a foul move for all positions of the metaposition should be ignored. This is the correct and game-theoretical cut-off.

At a certain metaposition where the opponent is to move, possible moves are all the legal moves of all positions of the metaposition. A metamove is the hybrid of all the possible moves.

The solver can guess the current position and the opponent moves by some clues (observations). The possible observations are:

1. Whether the move of the solver is a check move or a foul move
2. Whether the position is checkmated
3. Whether and where capturing occurs after the opponent move
4. Whether capturing occurs and what kind of piece is captured after the solver's move
5. Whether the countercheck occurs after the opponent move, for problems with double kings

By these observations, a metaposition is split into some metapositions with less uncertainty. Finally, TTS problem is resolved into the search of a generic AND/OR tree under the Uncertainty Paradigm.

4 Implementation of a Problem-Solver

We have implemented the computer program that can solve the TTS problem under the Uncertainty Paradigm.

4.1 Implementation of a Metaposition

Implementation of a metaposition can be categorized into several types. One method is to represent it as itself directly. This method is only applicable to the domains in which a metaposition is very simple or the difference of possible positions of a metaposition is small. However, since a metaposition in TTS problems sometimes has more than hundred positions, we can hardly represent it directly.

Another method is to represent a metaposition as an array of possible positions. Though this method needs much greater memory resources, this is considered the simplest, and probably fastest, method for the domains in which we cannot represent a metaposition directly. Moreover, it is easy to implement and less likely to cause bugs.

A third method is to represent a metaposition as a base position and several sequences of moves from the base position to each possible position. This method can be used for implementing a metaposition in TTS problems. This method reduces the memory requirement drastically compared to the above method. However, computing every position of a metaposition may be time-consuming as the level of the search gets deeper. Moreover, it is difficult to deal with transpositions of metapositions.

Another representation of a metaposition is possible. Partition search [5] is a method to reduce the number of search nodes drastically by introducing the super-nodes that represent several nodes. If we use this method for positions of a metaposition, it seems that the number of positions in every metaposition can be much smaller. Then, it might be better that each metaposition is implemented using more descriptive notation. However, it is not clear that this method is effectively applicable to the domain of chess-like problems.

Consequently, our implementation in TTS represents a metaposition as an array of possible positions.

Compaction of a Metaposition After a metamove has been made for a metaposition, several positions of the metaposition may happen to be the same. It is necessary to check the existence of same positions and remove duplicate positions of a metaposition after every metamove, assuming our implementation of a metaposition. We call this operation the *compaction of a metaposition*. The compaction of a metaposition is indispensable before encoding it.

Encoding of a Metaposition Suppose that every position is encoded by some method, we have to manage to combine all codes of positions into a single code. Generally speaking, this is a problem of selecting an appropriate hashing function. Various methods are possible depending on the applying domain and the encoding of a position. We studied how to encode a metaposition in TTS problems [17]. In TTS, every position is encoded as an exclusive-OR summation of pseudo-random numbers by using the Zobrist method. We examined four encoding methods: simple arithmetic summation, exclusive-OR summation, CRC (cyclic redundancy code), and MD5 for all codes of possible positions. We have found that exclusive-OR summation is liable to cause type-1 errors[2] of codes and is inappropriate, and that no type-1 errors are found in arithmetic summation, CRC and MD5, and that arithmetic summation is slightly faster than the others are. Consequently, our implementation has adopted encoding by arithmetic summation.

5 UPS Exploiting a Transposition Table

At the beginning, our developing program had performed the depth-first full-width iterative deepening, which did not use a transposition table [16]. Therefore, it searched the same positions repeatedly during iteration. Now that we have succeeded to encode a metaposition, we are ready to exploit a transposition table for registration, update and retrieval. In this section, we discuss several methods of UPS exploiting transposition tables.

5.1 ID (Iterative Deepening of depth-first full-width search)

This is a depth-first, full-width iterative deepening using a transposition table, which we abbreviate as ID. Iteration is doubly nested to find a solution of minimum steps and minimum fouls. Outer iteration is for steps and inner iteration is for fouls. This doubly nested iteration guarantees that an obtained solution has minimum steps and fouls.

After deciding the solvable steps and fouls, the re-search is performed to settle an optimal solution sequence. The re-search performs internal iterative deepening at every OR node (at the solver's turn) to find a solution with the shortest steps and fouls. The sequence is recognized as better when:

> The steps of the sequence are fewer than those of another sequence.
> If the above is the same, the fouls of the sequence are fewer than those of another sequence
> If the above is the same, the set of pieces in hand at the mated position is a subset of that of another sequence. (In our implementation, the number of pieces in hand is used for comparison for simplicity instead of evaluating the relation of two sets.)[3]

[2] A type-1 error occurs when more than one metaposition happens to have the same code. This error is fatal because the error is hard to be detected and may lead to an incorrect result.

[3] However, we recognize the problems in which the solver has no piece in hand at the mated position as 'good' problems.

The program selects the metaposition that has the best sequence at every OR node, and the metaposition that has the worst sequence at every AND-split.

5.2 PDS (Proof-number and Disproof-number Search)

AO* [12] is a well-known and representative algorithm of AND/OR-tree search. However, recently, several novel and highly efficient algorithms have been developed and studied. In the domain of Tsume-Shogi, these algorithms have shown the marvelous solving ability and solved most hard problems with over hundred or thousand steps.

Allis developed the proof-number search introducing the concepts of proof number and disproof number [1]. Seo developed the C* algorithm [20] (which was recently renamed to PN* [21] because there had already existed another C* algorithm), which uses a transposition table and performs iterative deepening increasing the threshold of proof number. PN* was inspired by IDA* [8], which is an iterative deepening version of A* increasing the threshold of estimated cost. However, PN* has a new feature that it performs iterative deepening at every OR node. Seo named this the *multiple iterative deepening*. Moreover, Nagai developed the PDS algorithm [9, 10], which performs iterative deepening increasing the thresholds of proof number or disproof number and performs multiple iterative deepening at every AND node as well as at every OR node. Nagai also developed df-pn [11] as a modification of PDS, which is basically a depth-first search but behaves the same as the proof-number search, and df-pn+, which uses the cost estimating function for proving and disproving.

We have compared these algorithms for solving endgames of 6×6 Othello and Tsume-Shogi, both of which are two-agent problems with complete information. Consequently, we have obtained the experimental data showing that the PDS algorithm is superior to others in view of both solving ability and solving speed in Tsume-Shogi [15, 18]. Because UPS is a special kind of AND/OR-tree search, we can apply these algorithms in UPS. We have chosen the PDS algorithm, which seems best among them, in UPS of Tsuitate-Tsume-Shogi.[4]

However, the search tree is not a simple AND/OR tree as in Tsume-Shogi but is a generic AND/OR tree. The node at the solver's turn is basically OR node and the edges that stem from it have a relation of OR-condition. However, because some edges AND-split into plural edges, AND nodes and OR nodes come to intermingle. This problem can be settled by introducing an auxiliary node to separate an AND node and an OR node. See Figure 2. Consequently, AND nodes and OR nodes do not necessarily appear by turns in the obtained AND/OR tree.

Because the search algorithms such as PDS, which behave roughly in a best-first manner, are not full-width searches, they do not suit well finding an optimal solution sequence. We have managed to have the program find an approximately optimal solution sequence using the Algorithm 1 indicated below.

Algorithm 1 (Finding an approximately optimal solution in PDS).

[4] However, a test set (30 problems) we have used in Tsume-Shogi is not large. Moreover, we have not sufficiently examined the df-pn algorithm. The df-pn algorithm should probably be examined in UPS later.

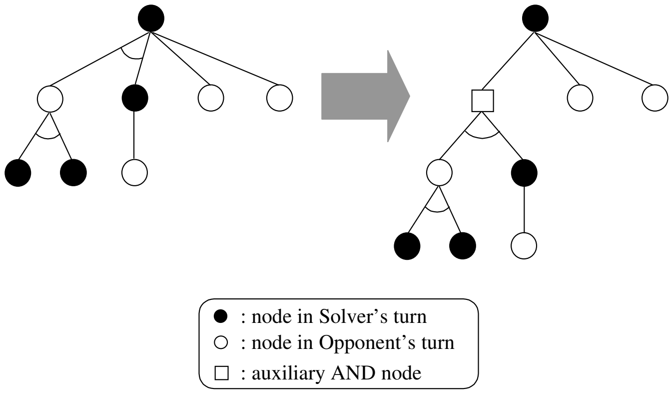

Fig. 2. Transformation of a general AND/OR tree

1. The first search is performed not imposing any restriction on steps and fouls.
   If the result of this search is proved, it is assured that an optimal solution sequence has less or equal steps and fouls.
2. The second search is performed after setting the limit of search steps to the steps obtained minus two.
   If the result of the search is proved, continue the re-search decreasing the limit of search steps.
   If the result of the search is disproved, it is confirmed that no solution does exist with less than a certain steps.
3. Thirdly, the search is performed after setting the limit of allowed fouls to the fouls obtained minus one.
   If the result of the search is proved, continue the re-search decreasing the limit of allowed fouls.
   If the result of the search is disproved, it is turned out that an approximately optimal solution sequence with minimum steps and fouls have been found.

5.3 UPDS (Uncertainty Paradigm PDS)

The searches using the proof number and disproof number have the intention to minimize the cost of search by minimizing the number of nodes during search. This is based on the assumption that the cost of proving or disproving an unexpanded node is averagely equal over the search tree. This assumption is roughly valid in many complete-information two-agent problems including Tsume-Shogi. Therefore, these searches work very well for problems in which the solution tree has a large part of forcing move sequences.

However, in UPS, every node has a property of the uncertainty index (actually, the number of positions). The search cost generally increases as the uncertainty of a node gets larger. This may cause a problem for the variants of the proof-number search in UPS.

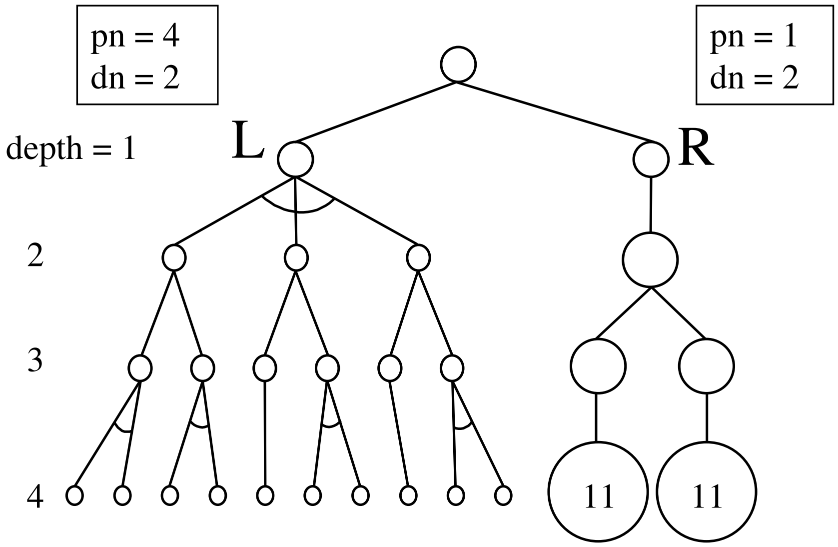

Fig. 3. Schematic example showing a problem of the proof-number search in UPS

First, let us show a schematic example in Figure 3. In this figure, the uncertainty of a node is indicated as the size of a circle. The branches with an arc indicate the AND-split of a metaposition. Suppose that the search has already proceeded partially and the current search node is the root node of this figure after several iterations. The root node has two child nodes, which are denoted by L and R. The node L has more descendant nodes, but the uncertainties of those nodes are small. On the other hand, the node R has fewer descendant nodes, but the uncertainties of those nodes get larger as the depth gets deeper. Suppose that all the nodes at the depth 4 are unexpanded. Then, every proof number and disproof number of those nodes is 1. As a result, the proof number and disproof number of the node L are 4 and 2, while the proof number and disproof number of the node R are 1 and 2. Therefore, the proof numbers indicate that the node R is easier to prove than the node L. This conclusion is generally correct in a normal AND/OR tree. However, is it also correct in an AND/OR tree with uncertainty?

To discuss a little quantitatively, let us compare the expenses of several typical parts of computation for a metaposition of which uncertainty index is N (that is, a metaposition that has N positions) with the expenses of computation for a definite position. (Here we restricted the case that a metaposition is represented as an array of positions.) The list of comparison with the case of a definite position is shown in Table 1. The cost of search seems to be at least $N \log N$ times (or N times, depending on the most critical parts) as much as that of the definite case.

In addition, it is almost impossible to unmake the previous metamove for a certain metaposition because every node has to be represented much dynamically in UPS. Consequently, it is also impossible to update the code of a metaposition linked with

Table 1. Degree of computational expense in a metaposition compared to the cases in a position

	Solver's turn					
	Move generation	Move ordering	Intersection	Making metamove	Encoding	Compaction
position	M	no need	–	–	1	–
metaposition	NM	$NM \log M$	NM	N	N	$N \log N$
	Opponent's turn					
	Move generation	Move ordering	Intersection	Making metamove	Encoding	Compaction
position	M	no need	–	–	1	–
metaposition	NM	no need	–	NM	N	$N \log N$

N : Uncertainty index of a metaposition,
M : Average number of moves at all positions of a metaposition

making/unmaking a metamove. Therefore, the cost of search and the memory requirement become still larger.

Other than the computational expenses, there may exist some relation between the uncertainty index and the easiness of proving/disproving. In TTS, because a metamove at the solver's turn is one element of an intersection of all possible moves for all positions of a metaposition, it is likely to fall into a situation having no metamove as the number of positions of a metaposition gets larger. That is to say, a metaposition that has a large uncertainty index is likely to be disproved. On the other hand, because a metaposition is mated at the opponent's turn only if all positions of the metaposition are mated, a metaposition that has a small uncertainty index is likely to be proved.

Considering all the above discussions, we come up with a search that sets the initial value of proof/disproof number of each unexpanded node to the value according to the formulae listed below, rather than 1 in the normal PDS.

$$\text{proof}_{\text{init}} = N^{\gamma_p}/C_p = (N/D_p)^{\gamma_p}$$

$$\text{disproof}_{\text{init}} = N^{\gamma_d}/C_d = (N/D_d)^{\gamma_d}$$

Here γ_p, γ_d, C_p, C_d, D_p, and D_d are the constant values. γ_p or γ_d indicates a degree trying to suppress the combinatorial explosion of uncertainty. As γ_p or γ_d is set to a larger value, the search is more likely to proceed in the way to select a metaposition with less uncertainty. C_p, C_d, D_p, and D_d are the damping factors of positive values. As they are set to larger values, the effect of γ_p or γ_d is suppressed.

This search can be recognized as a kind of PDS*, which is a variation of PDS that uses the cost estimating functions. However, since it also is a specialized version of PDS for the Uncertainty Paradigm, we have named it Uncertainty Paradigm PDS, which is abbreviated to UPDS. In TTS, it is expected that γ_p and γ_d such that

$$\gamma_p \geq 1, \quad \gamma_d \leq 1$$

are appropriate, considering the computational costs and the relation of easiness of proving/disproving. Nevertheless, if we set γ_p to a much larger value than 1, we can suppress the explosion of uncertainty at every metaposition, while we have to be under the apprehension that the search proceeds to a direction in which the number of nodes to be searched increases more and more. Incidentally, if we set $\gamma_p = \gamma_d = 0$, UPDS behaves exactly the same as PDS.

In our actual implementation of UPDS, we adopt the following formulae in order to set the initial values to the natural numbers within a limited range:

$$\mathrm{proof}_{\mathrm{init}} = \begin{cases} \lceil (N/D)^{\gamma_p} \rceil & \\ \mathrm{IPD}_{\mathrm{max}} & \text{if } \lceil (N/D)^{\gamma_p} \rceil > \mathrm{IPD}_{\mathrm{max}} \end{cases}$$

$$\mathrm{disproof}_{\mathrm{init}} = \begin{cases} \lceil (N/D)^{\gamma_d} \rceil & \\ \mathrm{IPD}_{\mathrm{max}} & \text{if } \lceil (N/D)^{\gamma_d} \rceil > \mathrm{IPD}_{\mathrm{max}} \end{cases}$$

Here $\mathrm{IPD}_{\mathrm{max}}$ is the upper bound of initial values. We set D_p and D_d to the same value that is denoted by D.

Here, let us back to the example in Figure 3. Suppose that all the uncertainty indexes of ten unexpanded nodes in the left subtree are 1, and the uncertainty indexes of two unexpanded nodes in the right subtree are 11. For example, if we set the parameters γ_p, γ_d, D(=D_p=D_d) to $1.5, 0.5, 4$, respectively, the initial proof numbers and disproof numbers on the left side are 1 and 1, while those on the right side are 5 and 2. As a result, the proof number and disproof number of the node L are 4 and 2, while the proof number and disproof number of the node R are 5 and 4. In this case, the proof numbers indicate that the node L is easier to prove than the node R. This is an opposite conclusion from the case of the normal PDS. In this way, the behavior of search can be controlled by changing these parameters.

5.4 Some Variations of UPDS and ID

dpUPDS (UPDS with Dynamic Parameters) At first, we thought that there must be certain parameters most suitable for all problems of TTS in the search using UPDS algorithm. However, after the experiments of solving all problems using various parameters, we have come to realize that the suitable parameters may vary depending on the characteristics of problems. Some problems have a feature that is nearly like that of complete-information problems. For such problems, it seems that $\gamma_p \sim 0, \gamma_d \sim 0$ is appropriate. Other problems need suppressing the uncertainty as much as possible to find a solution. For such problems, it seems that $\gamma_p > 1, \gamma_d \sim 1$ is appropriate. Then, we have added to the program a feature that performs the iterations changing these parameters. We have named it dpUPDS, which means UPDS with dynamic parameters. Unfortunately, since we have not known any guideline to find the suitable parameters, this iteration is something like area bombing and is liable to be time-consuming.

In our actual implementation, D is started from 1, then set to 4, and finally set to 16. (γ_p, γ_d) is set successively to the following values: (1.5, 1), (1.5, 0.5), (1.5, 0), (1.2, 0.9), (1.2, 0.4), (1.2, 0), (0.5, 0.3), (0.5, 0), (0.3, 0), (0, 0). Iteration is continued until the problem is solved.

uidUPDS (UPDS with Iterative Deepening of Uncertainty Index) A metaposition that has a large uncertainty index is likely to be disproved in general, though the computation to such a metaposition is much harder. Therefore, we have hit upon the alternative strategy that makes the program perform the iteration increasing the limit of uncertainty index. In a search, a metaposition that has a larger uncertainty index than the limit value is recognized as disproved and not searched further. We have named it uidUPDS, which means UPDS with iterative deepening of uncertainty index. The uid-UPDS program seems to be a good variation in solving a problem that has a solution with a little uncertainty.

In our actual implementation, the limit of uncertainty index is increased according to the sequence: 16, 24, 32, 40, ..., 128, 512, 4096, 10000. Iteration is continued until the problem is solved.

uidID (ID with Iterative Deepening of Uncertainty Index) The above iteration of uncertainty index can be also applied to ID (iterative deepening of depth-first full-width search). We have named it uidID. We expect that uidID can find an optimal solution with a little uncertainty in a short time. However, since the level of nesting of iteration is two in ID, the level of nesting of iteration becomes three. Therefore, this search may be inefficient for problems with much uncertainty.

In our actual implementation, the limit of uncertainty index is increased according to the sequence: 16, 32, 48, ..., 128, 512, 4096, 10000. Iteration is continued until the problem is solved.

6 Experimental Results and Discussions

Experiments have been done under the following environment:

Gateway2000, G6/GP6 Series (TB298-0109) Pentium II 450MHz,
RAM: 384MB Windows 98.

Since both ID and PDS-variants are basically depth-first searches with transposition tables, the memory requirement is small except for transposition tables. The size of a transposition table is 100MB for ID and 134MB for PDS-variants. In addition, because metapositions change dynamically during the search, every metaposition is stored in the heap memory. Therefore, for problems with large branching factors, long steps and much uncertainty, much heap memory is required.

At first, we have examined PDS and UPDS with several sets of parameters. Both γ_p and γ_d are changed in an increasing order from (0,0), which corresponds to the PDS algorithm, to (1.8,0.6). D is fixed to 4. IPD_{max} is set to a fixed value 50 throughout all experiments.[5] The results of solving time are shown in Table 2. For references, the results using the ID search are also shown. Here, time was measured by seconds.

Solving of a TTS problem has multi-stages. The first stage is only for finding one solution. The first stage finds a solvable steps and fouls of the solution. The solution obtained in the stage is not necessarily optimal. The remaining stages are for finding an optimal solution with fewest steps and fouls. In the experiments using the PDS or UPDS search, we have assigned two hours to each stage. We have recognized the problems that

[5] Using a few set of parameters, we examined the effect of IPD_{max} changing it to 10, 30, 50, 100, 1000, and 10000. The results when IPD_{max} was 1000 or 10000 seemed worse than the other results. However, we could not get the clear difference among the values from 10 to 100. The value 50 was selected somewhat arbitrarily.

have not been solved at the first stage in two hours as unsolved. Even if we give more time, the possibility to solve them is quite small, because the program has almost used up the memory resources or the number of searched nodes gets too large. On the other hand, we have not set a certain time limit for the ID program but continued running at most for 16 days and stopped manually if unsolved.

There are 39 problems in the source [7], which contains some easy problems with seven or nine steps and several hard problems with over twenty steps. Most problems have the solutions with less than four fouls except two problems. One problem (#35) has a solution with 33 steps and 8 fouls and the longest-step problem (#19) has a solution with 43 steps and 4 fouls.

The ID program solved 28 problems but it took much time to solve the relatively long-step problems. The PDS program failed to solve 16 problems. This is worst among the compared methods. Since PDS is a search that aims to minimize the number of search nodes as much as possible, it is likely to fall into the explosion of uncertainty in each metaposition and is unstable. Here, the unstability means that the algorithm happens to solve some hard problems with long steps, but is unable to solve some easier problems with short steps.

As for the parameters of UPDS, among six sets of parameters including PDS, the results of the parameters corresponding to the normal PDS seem to be worst. However, among the remaining five sets, it is not clear to distinguish which set is most suitable. A certain set of parameters is good for some problems, while another set is good for other problems. These results have led us to a new search method dpUPDS described above.

Let us check the speed of solving problems. When we see the results only for solved problems, the average value of solving time of PDS is smallest. (indicated in the 'time ave' row in Table 2) However, the PDS program has not solved several hard problems that the UPDS programs have solved. When we see the results for all problems, the average value of UPDS(1.1,0.4,4) is smallest. (indicated in the 'time all ave' row) Generally speaking, UPDS solves problems fastest among ID, PDS, and UPDS. However, UPDS shows some unstability. For instance, all the UPDS programs could not solve the problem #15 with eleven steps, which was solved in a short time by the ID program.

In view of settling an optimal solution, the iterative deepening that performs the full-width search is superior to the others of course. The Algorithm 1 for PDS or UPDS generally works well. However, for some problems, the search cannot find an optimal solution probably due to the effect of the transpositions.

We further experimented on solving all problems by the programs using uidID, dpUPDS, and uidUPDS. In uidUPDS, γ_p, γ_d, D was set to 0.8, 0.3, 4 respectively, because this set of parameters achieved the slightly superior results in Table 2. Thirty-five problems were solved using the above parameters. For the remaining four problems, the parameters were changed manually with trial and error. The results are listed in Table 3 and the logarithmic plot of solving time against the sum of the steps and fouls of each problem is shown in Figure 4. For references, the results using the ID algorithm are included. All problems were solved at least by one program. Especially, the uidUPDS program solved all problems. Moreover, it generally solved fastest among all programs. However, PDS variants including uidUPDS do not generally guarantee to find an optimal solution. Actually, the UPDS program did not give the optimal solutions

Table 2. Time of solving TTS problems

#	ST	F		ID	PDS	UPDS				
			γ_p	-	0	0.5	0.8	1.1	1.5	1.8
			γ_d	-	0	0.2	0.3	0.4	0.5	0.6
			D	-	-	4	4	4	4	4
1	7	0		2.5	2.1	0.7	1.2	1.3	1.4	1.3
2	13	1		48.1	12.1	7.7	8.0	7.6	7.9	7.7
3	9	2		37.5	7.7	4.8	6.8	5.0	6.6	6.5
4	9	0		1.0	2.8	1.1	1.0	1.0	1.1	1.0
5	11	0		3.4	1.2	1.1	1.1	1.1	1.2	1.2
6	11	0		14.5	3.7	2.2	1.8	3.0	2.9	2.7
7	11	1		16.8	63.3	34.8	35.6	50.0	35.9	162.9
8	13	3		862.6	398.1	274.8	253.1	296.5	214.0	191.0
9	11	1		27.3	×	32.6	16.2	17.7	16.8	16.8
10	13	1		59.4	20.2	4.9	9.7	92.1	11.5	24.9
11	15	1		640.8	×	×	×	×	52.8	76.7
12	19	2		192.2	×	×	6458.6	3595.7	1788.3	×
13	19	3		14124.8	×	×	×	×	×	×
14	19	2		2476.9	703.0	3793.6	2103.7	×	×	×
15	11	3		103.3	×	×	×	×	×	×
16	9	2		15.4	8.0	8.8	8.4	7.4	10.7	1081.2
17	13	1		323.9	53.7	73.3	18.6	20.6	29.1	32.0
18	15	2		1364240	×	×	×	×	×	×
19	43	4		2977.2	21.9	29.8	27.5	23.5	24.6	18.1
20	9	1		2.5	1.5	0.8	0.8	1.0	0.9	0.9
21	11	0		4.4	9.9	1.5	1.7	4.2	8.8	×
22	19	1		208696	×	×	×	×	×	1862.0
23	29	2		×	×	×	×	×	×	×
24	37	1		×	×	×	×	×	×	×
25	39	1		×	×	9.3	12.1	17.7	×	101.8
26	13	2		42.3	16.6	16.3	15.9	15.8	15.8	16.9
27	15	2		×	×	×	×	×	×	×
28	19	1		×	×	×	4106.3	2795.1	1982.2	3161.1
29	17	2		1249.8	×	1139.3	22.1	21.1	20.1	38.6
30	29	1		×	1822.7	595.9	485.0	562.2	7926.2	1145.0
31	15	2		2787.9	564.7	607.6	×	769.7	460.3	605.5
32	39	1		×	106.9	211.8	99.2	88.6	115.1	109.0
33	7	1		2.8	3.0	2.8	2.7	2.8	2.7	2.7
34	21	2		×	833.7	909.5	5825.3	1064.6	1413.9	×
35	33	8		×	1740.7	1101.2	1747.7	3485.8	14415	14416
36	13	2		784.9	×	×	×	×	×	×
37	27	3		×	×	×	×	×	×	×
38	9	1		1.1	0.6	0.8	0.8	0.8	0.8	0.9
39	21	1		×	×	8597.6	8608.7	×	×	×
solved				28	23	27	28	27	27	26
time total				1599739	6398	17465	29880	12952	28566	23084
time ave				57133.5	278.2	646.8	1067.1	479.7	1058.0	887.8
time all total				-	122395	105991	109632	99253	115246	109784
time all ave				-	3138.3	2717.7	2811.1	2544.9	2955.0	2889.0
depth ave				-	29.92	29.40	29.55	29.35	29.40	31.39
uncertainty ave				4.15	11.09	3.96	3.61	3.60	3.16	3.02
nodes ave				2921297	150256	415857	446240	440545	502017	532583

#:problem number, ST,F:steps and fouls of the solution (not necessarily optimal), γ_p, γ_d, D:parameters of UPDS, ×:unsolved,
solved:the number of solved problems, time total(ave):total(average) time for only solved problems,
time all total(ave):total(average) time for all problems, depth ave:average depth of the search,
uncertainty ave:average uncertainty of the search, nodes ave:average number of generated nodes in the search.

Table 3. Results of solving TTS problems by search variations

	ID	uidID	dpUPDS	uidUPDS
solved	28	36	32	39
unsolved	11	3	7	0
total time	1599739	355784	164647	165088
average time	57133.5	9882.9	5145.2	4233.0

in two problems. The uidID program solved 36 problems and mostly gave the optimal solutions. However, in two problems, uidID did not give an optimal solution. This was caused by the fact that the maximum uncertainty of the optimal solution is greater than that of the found solution with longer steps.

The solutions range from 7 steps with no foul to 43 steps and 4 fouls. The maximum value of the maximum number of positions in a metaposition of the solution tree is 66. Here, we show some statistical data of TTS problems based on the results by the ID program that performs a full-width search. The average and the maximum of the number of generated metapositions are 2921297 and 23382373. The average and the maximum of the effective branching factors are 2.20 and 2.89. The average and the maximum of the maximum number of possible positions among the searched metapositions are 987.7 and 10572. The average and the maximum of the average number of possible positions are 4.15 and 10.85.

7 Conclusions and Future Works

We have introduced the Uncertainty Paradigm for deterministic solving of problems with uncertainty. Then the search space of the problem is resolved into a plain AND/OR tree, which has a metaposition as a node and a metamove as an edge. Since solving such a problem is the same as solving an AND/OR tree, we can use the methods that have been developed and studied for years in the domain of artificial intelligence. The efficient search algorithms, which have been developed for the search of an AND/OR tree with complete information, are also applicable to our approach. However, there is a trade-off between reducing the search nodes and suppressing the explosion of uncertainty of each node. Therefore, it was necessary to find the algorithm that could proceed a search keeping the balance whether the search nodes should be minimized or whether the uncertainty should be minimized. We have selected the PDS algorithm and developed a modified algorithm UPDS specialized for UPS. Then, the search has come to be controlled by changing the parameters of UPDS. Moreover, we have developed some variations of these searches. Finally, we have succeeded to solve all problems in the domain of TTS. The problems that is unable to be solved by the full-width search, due to the explosion of uncertainty or due to the explosion of the number of searched nodes, have come to be solved smartly. The disproving ability of PDS, UPDS or its variants is also high, for example, as shown by the fact that the Algorithm 1 works well for not a few problems. On the other hand, the disproving ability of ID or its variants is quite

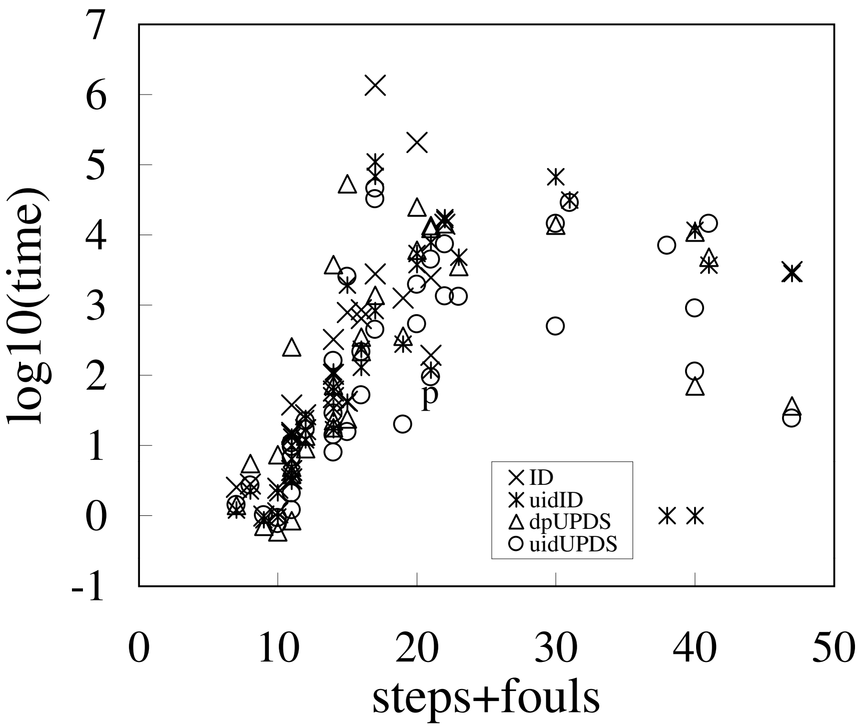

Fig. 4. Logarithmic plot of solving time against the sum of steps and fouls

poor. Therefore, the search algorithms using proof/disproof numbers are very promising in UPS, as well as in the normal AND/OR-tree search.

However, UPDS that we have tested in this paper is not yet sophisticated. We should examine the other formulae giving the initial value of proof/disproof number, or enable to change the parameters at run-time according to the property of the problem. Though the current UPDS is domain-independent, the domain-dependent UPDS might perform better. This is a remaining subject in the future.

We could take the next target of our approach to apply to the endgame problems of Kriegspiel. Our approach is directly applicable to Kriegspiel endgames. However, while the solver's moves are restricted to the check moves in TTS problems, they are not restricted in Kriegspiel endgames. Therefore, the search space might be larger than TTS, though the position of a problem is generally much simpler than that in TTS. In addition, the solutions of Kriegspiel problems often include a randomized mixed strategy of the multistage recursive game [2, 3]. In order to solve this type of problem, we should relax the constraint of deterministic solving (that is to say, semi-deterministic or quasi-deterministic) and make the program exploit the randomized mixed strategy if necessary.

References

1. Louis Victor Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, University of Limburg, Maastricht, The Netherlands, 1994.
2. P. Ciancarini, F. Dalla Libera, and F. Maran. Decision making under uncertainty: A rational approach to Kriegspiel. In H. Jaap van den Herik and Jos W. H. M. Uiterwijk, editors, *Advances in Computer Chess 8*, pages 277–298. Drukkerij Van Spijk B.V., Venlo, The Netherlands, 1997.
3. Thomas S. Ferguson. Mate with bishop and knight in kriegspiel. *Theoretical Computer Science*, 96:389–403, 1992.
4. Ian Frank and David Basin. Search in games with incomplete information: a case study using Bridge card play. *Artificial Intelligence*, 100:87–123, 1998.
5. Matthew L. Ginsberg. Partition Search. In *Proceedings of Thirteenth National Conference on Artificial Intelligence*, pages 228–233, Portland, Oregon, 1996.
6. Reijer Grimbergen. A survey of Tsume-Shogi programs using variable-depth search. In H. Jaap van den Herik and Hiroyuki Iida, editors, *Proceedings of First International Conference on Computers and Games CG'98*, volume 1558 of *Lecture Notes in Computer Science*, pages 300–317, Heidelberg, 1999. Springer.
7. Tetsu Kato. Collection of Kapitein documents No.1. Explanations and problems of Tsuitate-Tsume-Shogi (in Japanese), 1995.
8. Richard E. Korf. Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. *Artificial Intelligence*, 27:97–109, 1985.
9. Ayumu Nagai. A new AND/OR tree search algorithm using proof number and disproof number. In *Proceedings of Complex Games Lab Workshop*, pages 40–45, ETL, Tsukuba, November 1998.
10. Ayumu Nagai. A new depth-first-search algorithm for AND/OR trees. M.Sc. thesis, Department of Information Science, University of Tokyo, Japan, 1999.
11. Ayumu Nagai and Hiroshi Imai. Application of df-pn+ to Othello endgames. In *Proceedings of Game Programming Workshop in Japan '99*, pages 16–23, Hakone, Japan, October 1999.
12. Nils J. Nilsson. *Principles of Artificial Intelligence*. Tioga Publishing Company, Palo Alto, CA, 1980.
13. Guillermo Owen. *Game Theory*. Academic Press, New York, third edition, 1995.
14. D. B. Pritchard. *The Encyclopedia of Chess Variants*. Games & Puzzles Publications, Godalming, UK, 1994.
15. Makoto Sakuta and Hiroyuki Iida. An empirical comparison of innovative AND/OR-tree search algorithms on Tsume-Shogi (in Japanese, with English abstract). *Studies in Information, Shizuoka University*, 5:15–22, 1999.
16. Makoto Sakuta and Hiroyuki Iida. Solving problems with uncertainty: A case study using Tsuitate-Tsume-Shogi. In *Proceedings of Game Programming Workshop in Japan '99*, pages 145–152, Hakone, Japan, October 1999.
17. Makoto Sakuta and Hiroyuki Iida. Solving Kriegspiel-like problems: Exploiting a transposition table. *ICGA Journal*, 23(4):218–229, 2000.
18. Makoto Sakuta and Hiroyuki Iida. The performance of PN*, PDS, and PN Search on 6x6 Othello and Tsume-Shogi. In H. Jaap van den Herik and B. Monien, editors, *Advances in Computer Games 9*, pages 203–222. Universiteit Maastricht, The Netherlands, 2001.
19. Makoto Sakuta, Hiroyuki Iida, and Jin Yoshimura. Solving problems under Uncertainty Paradigm. In *Proceedings of International Conference on Advances in Infrastructure for Electronic Business, Science and Education on the Internet (SSGRR 2000)*, L'Aquila, Italy, August 2000. (in CD-ROM).

20. 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.
21. Masahiro Seo, Hiroyuki Iida, and Jos W.H.M. Uiterwijk. The PN*-search algorithm: Application to tsume-shogi. *Artificial Intelligence*, 129(1-2):253–277, 2001.