Solving Problems under Uncertainty Paradigm: Encoding a Metaposition and Exploiting the Transposition Table

Makoto Sakuta[*], Hiroyuki Iida[†]
Department of Computer Science, Shizuoka University

Abstract

We have recently proposed a novel paradigm for solving problems with uncertainty, called Uncertainty Paradigm. Under this paradigm, solving such a problem deterministically is resolved into the plain AND/OR tree search, which we call Uncertainty Paradigm Search. It uses a metaposition as a node of a game tree instead of a position and a metamove instead of a move. We have shown the applicability and justifiability of this paradigm to both a single-agent problem and an adversary-agent problem. The previous implementation of the search under this paradigm was based on a simple depth-first full-width search with iterative deepening, which did not use a transposition table. In this contribution, we have examined several methods for encoding a metaposition into some value with fewer bits. Then, a transposition table using these encoding methods is incorporated into the search. To confirm the effectiveness of the use of a transposition table, we have chosen the domain of Tsuitate-Tsume-Shogi (mating problems of Kriegspiel-like variant of Shogi). The experiments performed with a test set show us that the efficiency of search with a transposition table is turned out to be higher than the case without it at most by a factor of fourteen. As for encoding methods of a metaposition, we have confirmed that the method that simply sums up each code of position in the metaposition is efficient and allowedly collision-resistant in this domain.

Keywords: Problem solving; Uncertainty Paradigm; Metaposition; Metamove; Uncertainty Paradigm Search; Tsuitate-Tsume-Shogi; Zobrist method; Transposition table

1 Introduction

The game-tree search has been the most important part in developing a computer program for games with perfect information. Each node and edge in a game tree is called a position and a move respectively. Even in programming a game with imperfect information, a node of a game tree has been anyway a position. In addition, information sets[13] arisen from uncertainty have been used with nodes while searching.

We have recently proposed a novel paradigm for solving problems with uncertainty, called *Uncertainty Paradigm*[19]. We have shown the applicability and justifiability of this paradigm to both a single-agent problem (Counterfeit Coin Problem) and an adversary-agent problem (mating problem of Kriegspiel-like Shogi variant). Here, we need to mention our terminology in this paper. We use the term *single-agent problem* as a problem that has only one agent (or player) related to the problem, and the term *adversary-agent problem* as a problem that has two agents (or players) related to the problem, respectively. Moreover, the term *multi-person game* means a game that has more than two players.

[*]Corresponding author. Email: sakuta@cs.inf.shizuoka.ac.jp.
[†]Email: iida@cs.inf.shizuoka.ac.jp.

In the previous study, we used a simple depth-first full-width search with iterative deepening for solving problems under this paradigm. Since this search did not use a transposition table, it was a somewhat inefficient search algorithm. Moreover, without using a transposition table, we could not apply some efficient search algorithms for AND/OR tree such as PN*[20], PDS[10, 11], or df-pn[12], all of which are the depth-first variants of proof-number search[1, 2]. We thus need to exploit a transposition table in the search by encoding a metaposition into some value with fewer bits. In this paper, we examine several methods for encoding a metaposition and apply them for searching with a transposition table.

2 Uncertainty Paradigm for Problems with Incomplete Information

Uncertainty Paradigm is a way of thinking that recognizes the uncertain situation as it really is. Under this paradigm, an uncertain situation is a hybrid of several certain situations. Here we would like to give a brief sketch of this paradigm. We use the term *solver* and *opponent* in the special meaning indicated bellow.

solver : the player or the agent to solve the problem
opponent : the other players or the other agents of the problem

A *metaposition* is a hybrid of possible positions that are not distinguishable for the *solver*. Let Φ be a metaposition, φ_i be a position in the metaposition, the metaposition for deterministic solving is then represented as:

$$\Phi = \sum_i^{n_p} \varphi_i$$

By using the symbol of summation, the localized state φ_i and the delocalized state Φ are represented. The count of the possible positions n_p in the metaposition can be recognized as the *uncertainty index*.

A *metamove* is a move for the metaposition. In the case that a metamove is one of definite moves, a metamove μ is represented as:

$$\mu = m \in \mathcal{M}$$

where m is one of definite moves and $\mathcal{M}$ is a set of possible moves for all positions in the metaposition. On the other hand, in the case that a metamove is a hybrid of moves, a metamove μ is represented as:

$$\mu = \sum_i^{n_p} \sum_j^{n_{m_i}} m_{ij}$$

where n_{m_i} is the number of possible moves for i-th position and m_{ij} is one of moves for i-th position.

Here we represent a position after a certain move and a metaposition after a certain metamove respectively as follows:

$m(\varphi)$: a position from the position φ after the move m
$\mu(\Phi)$: a metaposition from the metaposition Φ after the metamove μ

When a metamove μ for a metaposition Φ is a definite move m, making the metamove corresponds to making the move m for each position in the metaposition.

$$\Phi^{child} = \mu(\Phi) = \sum_i^{n_p} m(\varphi_i) = \sum_i^{n_p^{child}} \varphi_i^{child}$$

When a metamove μ for a metaposition Φ is a hybrid of several moves, making the metamove corresponds to making all the moves for each position in the metaposition. Consequently, the number of positions in the metaposition increases. This is a diffusion of the metaposition.

$$\Phi^{child} = \mu(\Phi) = \sum_{i}^{n_p} \sum_{j}^{n_{m_i}} m_{ij}(\varphi_i) = \sum_{i}^{n_p^{child}} \varphi_i^{child}$$

The count of positions in the metaposition increases to n_p^{child}. If the *solver* cannot get any clue in such metamoves, the uncertainty of the metaposition soon explodes combinatorially. However, there are observations or clues that help the *solver*. By these observations, the metaposition splits into several metapositions with less uncertainty i.e. metapositions that have fewer positions in it.

We have defined the *observable* as an element of which value the *solver* can get as a clue. An observable o always acts on a metaposition (and positions in the metaposition) and returns the corresponding value automatically. Here we represent the value to be returned by an observable o as follows:

$o(\varphi)$: a value returned by acting on the position φ
$o(\Phi)$: a value returned by acting on the metaposition $\Phi = \sum_i^{n_p} \varphi_i$

$$o(\Phi) = \begin{cases} v & \text{if for all } i \ \ (1 \le i \le n_p) \ \ o(\varphi_i) = v \\ \text{uncertain} & \text{otherwise} \end{cases}$$

Let n_o be the number of observables, o_l be a observable, and $(t_{s1}, t_{s2}, \ldots, t_{sn_o})$ be a n_o-tuple of the indexes of observable values that represents s-th split $(1 \le s \le n_s^{child})$. And for each observable o_l, let nv_l be the number of possible values for the observable, and $\{v_{lt} \mid 1 \le t \le nv_l\}$ be a set of possible values for the observable. Then the splitting of a metaposition is represented as follows:

$$\Phi_s^{child} = \sum_{i_s}^{n_{p_s}^{child}} \varphi_{i_s}^{child} \quad \text{such that } \ o_l(\Phi_s^{child}) = v_{lt_{sl}} \qquad (1 \le s \le n_s^{child})$$

So the metaposition Φ^{child} splits into n_s^{child} metapositions $\{\Phi_1^{child}, \ldots, \Phi_{n_s^{child}}^{child}\}$. Since the *solver* should accept this splitting in a passive manner, he/she has to solve all the split metapositions. Therefore, it is an AND-splitting of a metaposition even in case of the single-agent puzzles.

For all solving problems with uncertainty, the search graph(tree) under Uncertainty Paradigm is AND/OR graph(tree), which we have denoted as Uncertainty AND/OR graph(tree). The search graph(tree) under Uncertainty Paradigm is a plain AND/OR graph(tree), as well as the solution graph(tree). The search under Uncertainty Paradigm is a plain AND/OR graph(tree) search, which we call Uncertainty Paradigm Search (UPS).

3 Encoding a Metaposition under Uncertainty Paradigm

Implementation of a metaposition can be categorized into two types: representing a metaposition as itself, or an array of possible positions. As for the case that a metaposition is represented as itself, encoding can be performed same as encoding a position. So no further discussion is needed here.

The problem is how we can encode a metaposition that is represented as an array of positions. First we explain the general method for this problem, and consequently we would focus on that for the specific domain.

3.1 Encoding a Position of a Board Game

Before discussing encoding a metaposition, we would like to describe the encoding method of a position of a board game. Zobrist method is well-known for efficient encoding of a position of general games, especially board games[22].

Suppose there are m distinguishable types of pieces and n locations on the board. There are at most $m \times n$ possibilities for placing a type of piece at a board location, thus every game configuration corresponds to a unique subset of these $m \times n$ possibilities. We take $m \times n$ integers from a random sequence and assign each of them to each placement possibility. Then the code for a board configuration is an exclusive-OR summation of all pieces on the board.

The board code bp of a position is represented as:

$$bp = \sum_{i}^{\oplus} rp[i]$$

where $rp[i]$ is a random integer corresponding to one piece in the board. $\oplus$ indicates a summation by means of an operator of exclusive-OR.

This method has a great advantage that the computation of a position code is incorporated with making (or unmaking) a move for the position and the position code can be incrementally updated by least additional computing. In the domain of computer chess, this method has been successfully applied for years[8].

3.2 General discussion on encoding a metaposition

We assume that each position has its own code that is computed by whatever method used. Moreover, we can assume a metaposition has a certain property that is common for all positions in the metaposition. Considering these, let cc be a code of the common property for all positions in a metaposition, n be the number of positions in the metaposition, and $pc[1], ..., pc[n]$ be a code of each position in the metaposition respectively. Then a code of the metaposition mc can be represented as:

$$mc = h(cc, n, pc[])$$

$h()$ is a kind of a hash function, which hashes all of cc, n, $pc[1], ..., pc[n]$ on a certain integer. This is classified as one-way hash functions. $h()$ should be selected such as its computation is fast and it minimizes collisions[7].

There are several alternatives that can be used for such a hash function: simple arithmetic sum, simple exclusive-OR sum, cyclic redundancy code (CRC), secure hash functions, and so on.

3.2.1 Simple Arithmetic Sum

For n random integers $r[1]$ to $r[n]$, if we can assume the following conditions (**Condition A**), we could use a simple arithmetic sum to represent all random integers.

1. Arithmetic addition is calculated by modulo 2^w when w is the number of bits of each random integer.
2. Random numbers are uniformly distributed within the range 0 to $2^w - 1$.
3. All $r[1], ..., r[n]$ are independent.

The resulting sum is also uniformly distributed within the range 0 to $2^w - 1$. This can be proved easily by induction. First, if n is 1, it is obvious. Secondly, we assume

$$s_n = \sum_{i=1}^{n} r[i]$$

is uniformly distributed within the range 0 to $2^w - 1$. Since $r[n+1]$ is uniformly distributed within the range 0 to $2^w - 1$, $s_{n+1} = s_n + r[n+1]$ is also uniformly distributed.

If we can nearly assume the above conditions with respect to cc, n and $pc[i]$, we can use this for encoding a metaposition. This encoding has a commutative feature for each position in a metaposition. Therefore, it doesn't need sorting positions in a metaposition before encoding.

3.2.2 Simple Exclusive-OR Sum

Similarly as above, for n random integers $r[1]$ to $r[n]$, if we can assume the following conditions (**Condition E**), we could use a simple exclusive-OR sum to represent all random integers.

1. All bits of each random integer have a possibility of 1/2 as 0, 1/2 as 1.
2. All $r[1], ..., r[n]$ are independent.

All bits of the resulting sum have a possibility of 1/2 as 0, 1/2 as 1. Hence, the resulting sum is also uniformly distributed within the range 0 to $2^w - 1$ when w is the number of bits of each random integer. This can be proved easily by induction, too. This encoding also has a commutative feature for each position in a metaposition. Therefore, it doesn't need sorting positions in a metaposition before encoding.

3.2.3 Cyclic Redundancy Code (CRC)

Cyclic redundancy code (CRC) is used especially for the error detecting of transmission[3, 21]. It can distinguish all errors such that the received value is several bits different from the original value. However, CRC can be also used for representing the sequence of integers. This encoding doesn't have a commutative feature for each position in a metaposition. Therefore, it is necessary to sort positions in a metaposition before encoding in some particular order.

There are several standards of CRC codes depending on the polynomial concerned: CRC-12, CRC-16, CRC-CCITT, CRC-32. We can select one of them for encoding in accordance with the target domain.

3.2.4 Secure Hash Functions

Secure hash function such as MD5 or SHA-1 is used for data-encryption and has an excellent collision-resistant feature[9]. However, it takes much time to compute these codes because these functions are specially designed to defend against possible decoding attacks by humans or man-made programs.

For encoding a metaposition, because it is meaningless to defend against decoding, we cannot get much advantage but only lose the efficiency of search by using these functions. If we have to encode metapositions each of which is consisted of nearly same codes of positions and consequently we have to be very sensitive to collisions, these encoding might be necessary.

3.3 Domain-specific consideration: encoding a metaposition in Tsuitate-Tsume-Shogi

As an application of Uncertainty Paradigm to the adversary-agent problems with uncertainty, we have chosen the domain of *Tsuitate-Tsume-Shogi* (TTS). Tsuitate-Shogi is a Shogi variant which is one of the best known and most popular among all variants as well as Kriegspiel[15] being in chess. Tsuitate-Tsume-Shogi is a mating problem of Tsuitate-Shogi[5], which is also a variant of Tsume-Shogi, a mating problem of Shogi[4]. The detailed rules of TTS have been given in [19] or [16]. We have also shown the simple explanation and a sample problem of TTS in the Appendix.

First, we should mention the encoding method of a position. As described in a previous section, the Zobrist method has been successfully applied for years in the domain of computer chess. However, in the case of Shogi, there is additional factor than chess. Different from that of chess, it is necessary to represent a position of Shogi in terms of its pieces in the board as well as pieces in hand of both sides. Though pieces in the board can be coded efficiently using the Zobrist method same as chess, pieces in hand of both sides cannot be represented by this encoding. Therefore, in most Shogi programs including ours, a position is coded as a code of its board as well as a code of pieces in hand of one player. (If both a code of the board and a code of pieces in hand of one player in a certain position are same as those in another position, two positions can be recognized as to be same.)

The board code *bp* of a position is represented as:

$$bp = \sum_{i}^{\oplus} rsp[i] \;\oplus\; \sum_{i}^{\oplus} rdp[i]$$

where $rsp[i]$ is a random integer corresponding to the pieces of one player, and $rdp[i]$ is a random integer corresponding to the pieces of the other player.

We have had to pay careful attention to the selection of random numbers. Many generators of pseudo-random numbers have been developed and studied deeply[6]. Here we have adopted the pseudo-random numbers based on pseudo-DES algorithm[14]. Each random number is a 64-bit integer generated by this algorithm and it is assigned to each placement possibility. Each board position is computed using the above Zobrist method.

Pieces in hand of one player are represented as an array of the number of pieces according to a type of piece, *npi*[for all types of pieces in hand]. This can be recognized as a set:
{ *tpi* | *tpi* is one type of pieces in hand }.
For example:

npi	Rook(飛)	Bishop(角)	Gold(金)	Silver(銀)	Knight(桂)	Lance(香)	Pawn(歩)
	1	0	2	1	0	3	4

is equivalent to a set:
{ Rook, Gold, Gold, Silver, Lance, Lance, Lance, Pawn, Pawn, Pawn, Pawn }.

npi[] can be packed into an integer *cpi* of a certain bits (32 bits) with perfect information. *cpi* is a code of pieces in hand.

In our program of solving TTS, each board position is coded into a 64-bit integer. We can assume both the equi-distribution within the range of 0 to $2^{64} - 1$ and the equi-possibility (0 or 1) of each bit for this random number. In TTS, since the solver of course knows the placements of his/her own pieces, he/she can distinguish two positions both in which the placements of his/her pieces differ. Consequently, all positions in a metaposition have the same placements with respect to pieces of the solver. Therefore, we have separated pieces of both sides and computed a board code of each side respectively.

The board code *sbp* of a position with respect to the solver's pieces and the board code *dbp* of a position with respect to the opponent's pieces are represented as:

$$sbp = \sum_{i}^{\oplus} rsp[i]$$

$$dbp = \sum_{i}^{\oplus} rdp[i]$$

where $rsp[i]$ is a random integer corresponding to the pieces of the solver, and $rdp[i]$ is a random integer corresponding to the pieces of the opponent (defender).

Secondly, let us proceed to the encoding method of a metaposition. We have also separated a code with respect to board pieces and a code with respect to pieces in hand of the solver. Since the code with respect to pieces in hand of the solver is common for all positions in a metaposition, we can handle this code exactly same in normal Shogi programs. Codes with respect to board pieces are different among positions in a metaposition, so we have to sum up these codes.

Let spb be a random integer corresponding to the board pieces of solver in a metaposition and f be the number of fouls that have been committed so far, both of which are codes of the common properties for all positions in the metaposition, and let n be the number of positions in the metaposition, $dpb[i]$ be a random integer corresponding to the pieces of the opponent for each position in the metaposition. Then, a code of the metaposition mc can be represented as:

$$mc = h(spb, f, n, dpb[])$$

Please notice that each $dpb[i]$ is not independent. Let $gcdpb$ be a greatest common code for all $dpb[i]$, and $deladpb[i]$ be a characteristic code for each $dpb[i]$, $dpb[i]$ is represented as:

$$dpb[i] = gcdpb \ \oplus \ deltadpb[i]$$

Because codes of metapositions are not so collision-sensitive in this domain, we reasonably think we don't have to examine some secure hash functions such as MD5. Therefore, we have examined three methods discussed in the previous section: simple arithmetic sum, simple exclusive-OR sum, and CRC.

First, let us consider encoding by means of a simple arithmetic sum. We can nearly assume the equi-distribution of spb and $dpb[i]$ within the range from 0 to $2^{64} - 1$. Moreover, though each $dpb[i]$ is dependent in the relation of exclusive-OR, we can roughly assume their independence with respect to arithmetic addition. Consequently, the **Condition A** roughly holds in this case, thus we can use a simple arithmetic sum for encoding metaposition.

Secondly, let us consider encoding by means of a simple exclusive-OR sum. We can nearly assume the equi-possibility (0 or 1) of each bit in spb and $dpb[i]$. However, since each $dpb[i]$ is dependent in the relation of exclusive-OR, the **Condition E** obviously does not hold in this case. Therefore, encoding a metaposition by means of a simple exclusive-OR sum may be not so appropriate.

Thirdly, Cyclic redundancy code (CRC) can be used for encoding a metaposition after sorting positions in the metaposition in some particular order. We have used two CRC-32 codes corresponding to high 32 bits and low 32 bits respectively.

4 Searching with a transposition table

Iteration has to be doubly nested to find the solution with the shortest steps and the least fouls. The outer iteration is that of the search depths of checks and responses, while the inner iteration is that of the allowed count of fouls. After deciding the solvable steps and fouls, the search with the multiple iterative deepening at OR nodes is performed to determine the best solution sequence of metamoves. A sequence is recognized better when

> The number of steps of one sequence is less than that of another sequence.
> If the above is same, the number of fouls is less than that of another sequence.
> If the above is same, the number of pieces in hand at the mated position is larger than that of another sequence.

A metaposition that has the best sequence is selected at every OR node, while a metaposition that has the worst sequence is selected at every AND node or at every AND-splitting.

Table 1: Performance results of solving TTS problems

	NOTP		ADD		XOR		CRC	
correct answer	24		24		23		24	
code collision	-		-		Found		-	
maximum time	199839	(1)	14260	(0.071)	13895	(0.070)	14387	(0.072)
arithmetic mean time	10516	(1)	867.3	(0.082)	928.3	(0.088)	875.1	(0.083)
geometric mean time	64.49	(1)	46.17	(0.716)	47.47	(0.736)	46.93	(0.728)
maximum nodes	28345500	(1)	7340640	(0.259)	7150850	(0.252)	7340640	(0.259)
arithmetic mean nodes	2445750	(1)	758924	(0.310)	746951	(0.305)	758924	(0.310)
geometric mean nodes	134419	(1)	97704	(0.727)	97275	(0.724)	97704	(0.727)

”NOTP” represents a program with no transposition table. Similarly, ”ADD”, ”XOR” and ”CRC” represent a program with the transposition table using the following encoding method: simple arithmetic sum, exclusive-OR sum, and CRC-32, respectively. The numbers in the parenthesis indicate the ratios of the values to the case without the transposition table.

4.1 Experimental Results and Discussions

Experiments have been performed using a test set that contains 39 problems from the source[5]. Here we have tried to solve 24 problems among them. They vary from some easy problems with 7 or 9 steps to some hard problems with 19 steps within 3 permitted fouls. We have tried to solve 24 problems of TTS using a simple depth-first full-width search with iterative deepening, in which a transposition table is exploited.

Experiments have been done under the following environment:

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

The summary of results of the experiments are shown in Table 1. For reference, the results by our previous program without the transposition table are also included. The programs have solved most problems and shown the correct solution sequences except one problem, in which it is necessary to omit the useless interposing move.

All programs could solve all tested problems and give the correct answers. The only exception was that the XOR-encoding (denoted by ‘XOR’ in Table 1) program was unable to solve one problem correctly. This is caused by the collisions of codes of metapositions. As expected, XOR-encoding is not proper encoding method for the domain of TTS problems.

As for the efficiency of the search, the methods exploiting the transposition table is faster than one without the transposition table by a factor about from 1.4 to 14. Both the solving time by the program without the transposition table and that by the program with ADD-encoding (denoted by ‘ADD’ in Table 1) are plotted in Figure 1. The total number of metapositions generated in the search exploiting the transposition table is also fewer by a factor about from 1.4 to 4.

As for the encoding methods, the XOR-encoding is out of the question because of the collisions of codes. Moreover, the ADD-encoding has a little advantage over the CRC-encoding (denoted by ‘CRC’ in Table 1) with respect to solving speed by roughly one percent. This is obviously caused the difference of quantity of computing those codes and sorting positions required only before the CRC-encoding. Those results show that the ADD-encoding is superior to other encoding methods for a metaposition of TTS problems.

In addition to the above 24 problems, we have tried to solve another problem with 43 steps and 4 fouls, which is shown in Figure 2. The program without the transposition table could not solve this problem due to deep steps. However, the programs with the transposition table have solved it with the search of 37 steps in 70 seconds and settled the solution sequence with the search of 39 steps in about 2850 seconds. This is entirely owing to the power of the transposition table.

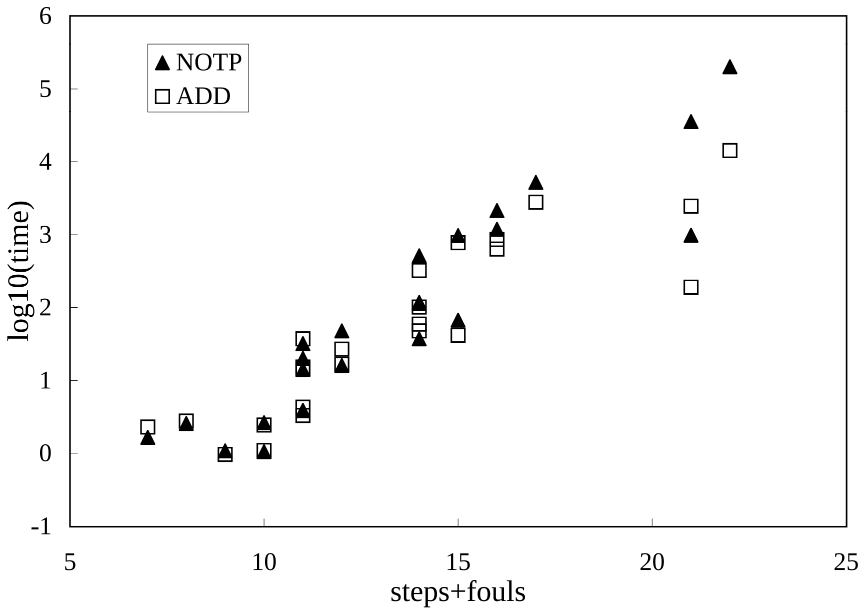

Figure 1: Steps+fouls versus the logarithms to the base 10 of the seconds of solving time

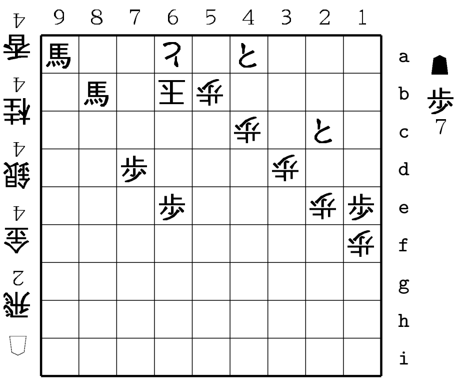

Figure 2: A problem with 43 steps and 4 fouls

5 Conclusions

We have extended a recently proposed paradigm for solving problems with uncertainty, Uncertainty Paradigm, by encoding a metaposition and exploiting the transposition table in searching.

As to the encoding methods of a metaposition, we have considered several methods and tested ADD-encoding, XOR-encoding and CRC-encoding in the domain of Tsuitate-Tsume-Shogi. By using the transposition table, the efficiency of search has been enhanced by a factor about from 1.5 to 14. We have confirmed that ADD-encoding method outperforms the other encoding methods considered in this paper. Moreover, no collision of codes has detected with this encoding.

Though our programs have succeeded to solve all the problems in the test set of Tsuitate-Tsume-Shogi, there still are other hard problems that have not yet been solved. These are problems with quite long steps, or problems that have considerably large branching factors. To solve such problems in a short time, we should examine the other search algorithms that have best-first manners using the transposition table, which are PDS, and df-pn. We have examined these algorithms to the adversary-agent problem with complete information, on 6x6 Othello and Tsume-Shogi and confirmed the excellence of these searches[17, 18]. Since UPS is also the AND/OR tree search same as the search of the adversary-agent problem with complete information, we think these algorithms represented by PDS would also work efficiently in UPS. However, some modifications or specialization of these algorithms will be required in UPS because there are additional factors derived from uncertainty. Now that we have implemented the encoding of a metaposition, we are ready to examine these searches in UPS. This should be the next subject.

Acknowledgements

The authors thank Jonathan Schaeffer for his useful advice on the implementation of the metaposition. We would also like to thank Tetsu Kato who has pleasantly provided us with the problems and explanations of TTS. This research was supported by the grant-in-aid of the Ministry of Culture in Japan (grant no.10780225).

References

[1] L. V. Allis. *Searching for Solutions in Games and Artificial Intelligence.* PhD thesis, Department of Computer Science, University of Limburg, Netherlands, 1994.

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

[3] L. P. Deutsch. GZIP file format specification version 4.3. Request for Comments:1952, 1996.

[4] R. Grimbergen. A survey of Tsume-Shogi programs using variable-depth search. In H. J. van den Herik and H. 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.

[5] T. Kato (Ed.). Collection of Kapitein documents No.1. Explanations and Problems of Tsuitate-Tsume-Shogi (in Japanese), 1995.

[6] D. E. Knuth. *Seminumerical algorithms*, volume 2 of *The art of computer programming.* Addison-Wesley, third edition, 1997.

[7] D. E. Knuth. *Sorting and searching*, volume 3 of *The art of computer programming.* Addison-Wesley, second edition, 1998.

[8] D. Levy and M. Newborn. *How computers play chess.* Computer Science Press, New York, 1991.

[9] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of applied cryptography.* CRC Press, Boca Raton, Florida, 1997.

[10] A. 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, Nov. 1998.

[11] A. Nagai. A new depth-first-search algorithm for AND/OR trees. M.Sc. thesis, Department of Information Science, University of Tokyo, Japan, 1999.

[12] A. Nagai and H. Imai. Application of df-pn+ to Othello endgames. In *Proceedings of Game Programming Workshop in Japan '99*, pages 16–23, Hakone, Japan, Oct. 1999.

[13] G. Owen. *Game Theory.* Academic Press, New York, third edition, 1995.

[14] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C.* Cambridge University Press, New York, second edition, 1992.

[15] D. B. Pritchard. *The Encyclopedia of Chess Variants.* Games & Puzzles Publications, Godalming, UK, 1994.

[16] M. Sakuta and H. 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, Oct. 1999.

[17] M. Sakuta and H. Iida. An empirical comparison of depth-first AND/OR tree search algorithms on 6x6 Othello and Tsume-Shogi. In H. J. van den Herik, editor, *Advances in Computer Chess 9.* 2000. in press.

[18] M. Sakuta and H. Iida. An empirical comparison of innovative AND/OR-tree search algorithms on Tsume-Shogi (in Japanese, with English abstract). Studies in Information 5, Shizuoka University, 2000. in press.

[19] M. Sakuta and H. Iida. Solving problems under Uncertainty Paradigm. CGRI Technical Report 2000-01, Computer Games Research Institute, Shizuoka University, Japan, 2000.

[20] M. Seo, H. Iida, and J. W. H. M. Uiterwijk. The PN*-search algorithm: Application to Tsume-Shogi. CGRI Technical Report 99-12, Computer Games Research Institute, Shizuoka University, Japan, 1999.

[21] R. N. Williams. A painless guide to CRC error detection algorithms. available at ftp.adelaide.edu.au/pub/rocksoft/crc_v3.txt, 1993.

[22] A. L. Zobrist. A new hashing method with application for game playing. Technical Report 88, Computer Science Department, The University of Wisconsin, Madison WI, USA, 1970. Reprinted in: *ICCA Journal*, 13(2):69–73, 1990.

A simple explanation and a sample problem of TTS

Here we give a summary of rules of Tsuitate-Tsume-Shogi (TTS). The rules of TTS are fairly similar to those of Tsume-Shogi except several cases. There are two agents in TTS: the attacker as the *solver* and the defender as the *opponent*. In TTS, the attacker is unable to see the opponent's pieces and response except the initial position of a mating problem considered. The goal of solving a TTS problem is to mate the opponent's king after the sequence of check and its response as well as in Tsume-Shogi. However, in TTS, 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. The attacker is allowed to try the foul moves in a certain number of times, typically eight times or less. The foul move is defined as the subset of the illegal move. It is the move that seems to be legal on the board only with the own pieces but is illegal on the board with pieces of both sides. Thus, the foul moves are:

1. The sliding piece (rook, bishop, lance) jumps over the opponent piece.
2. Drop a piece into the square where there is the opponent piece.
3. The pawn-drop move that causes the dropped pawn mate.
4. The king of the attacker remains in check or come to be in check after the move, for problems with double kings.

Please notice that the attacker cannot cancel the move that has been once tried and it has to be a check move if it has turned out to be a legal move. The attacker should try to guess the position of the defender's pieces as the mating search progresses by trying moves that can be either legal or illegal with respect to the full position.

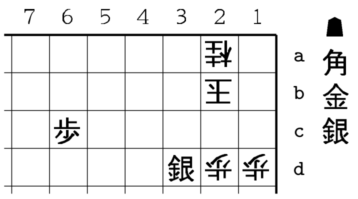

Figure 3: A sample problem

Next, let us show a sample TTS problem and its solution. A sample TTS problem is shown in Figure 3. This is a definite initial position. The attacker first makes the silver-dropping move **1. 銀*2c**. There are three moves as its response, **1. ...玉1a**, **1. ...玉1c**, and **1. ...玉3a**. The attacker should make the move that checks the defender's king (玉) for all three possibilities. So the attacker makes the move **2. 角*2b**. If the king is at **1a** or **1c**, the position is turned out to lead to checkmate. Otherwise, the defender has two possible moves **2. ...玉4a** or **2. ...玉4b** as the response. Then the attacker makes the move **3. 角3a+** that checks the king for both possibilities. There are three responses, **3. ...玉x3a**, **3. ...玉5a**, or **3. ...玉5b**. If the defender choices the move **3. ...玉x3a**, the attacker knows it because the promoted bishop at **3a** disappears. So the attacker can make the final move **4. 金*3b** and checkmate the king. Otherwise, the attacker knows that the defender moves the king either to **5a** or to **5b**. Then the attacker can make the final move **4. 金*6b** and checkmate the king. The solution sequence with 7 steps is either of the below:

1. 銀*2c (–) 2. 角*2b (–) 3. 角3a+ (3a) 4. 金*3b mate.
1. 銀*2c (–) 2. 角*2b (–) 3. 角3a+ (–) 4. 金*6b mate.

'(–)' indicates a response of the defender after which no capturing has occurred, and '**(3a)**' indicates a response after which the piece at **3a** has been captured by some defender's piece.