Complex Games Lab Workshop

November 10, 1998

Electrotechnical Laboratory
Machine Inference Group
Umezono 1-1-4, Tsukuba
Ibaraki, JAPAN 305

Organising Committee

Schedule

Session 1

1:00 – 1:05
Opening Comments

1:05 – 1:30
Move Evaluation Tree System
by Hiroto Yoshii

1:30 – 1:55
Memory-Based Approach in Go-program Katsunari
by Shinichi Sei and Toshiaki Kawashima

1:55 – 2:20
Succinct Games and Exotic Numeration Systems
by Aviezri Fraenkel

2:20 – 2:45
Zero Sum Games As Distributed Cognitive Systems
by Robert L. West

2:45 – 3:15: Coffee Break

Session 2

3:15 – 3:40
Evaluation of Tsume-shogi with the Method of Least Square
by Toshinori Kasuga, Tsuyoshi Suzuki, andYoshiyuki Kotani

3:40 – 4:05
How players learn at KANSO-SEN
by Takeshi Ito

4:05 – 4:30
Incremental Generation of Possible Moves in Shogi
by Tsuyoshi Suzuki, Nobuo Inui, and Yoshiyuki Kotani

4:30 – 4:55
A new AND/OR Tree Searching Algorithm Using Proof Number and Disproof Number
by Ayumu Nagai

Move Evaluation Tree System

Hiroto Yoshii
hiroto-yoshii@mrj.biglobe.ne.jp

Abstract

This paper discloses a system that evaluates moves in Go. The system—Move Evaluation Tree System (METS)—introduces a tree architecture algorithm, which is a popular algorithm in the field of pattern recognition. Using the METS algorithm, we can get emergency values of every empty position at any situation of the game. The experiment using a large database shows that the METS algorithm has a great ability to recognize a configuration of stones and evaluate the significance of empty positions.

1 Introduction

Among various board games, game of Go is one of the most difficult games because of its huge search space. Many researchers don't believe that Go can be solved by an exhaustive search such as that carried out by Chess algorithms: we must select good moves among all possible—hundreds of—moves in some other way. Some researchers have challenged this problem and showed successful results [1], [2], however their algorithms couldn't cope with selecting problem as a general pattern recognition problem and seem to lack strong theoretical supports. In this paper, we re-define Go as a pattern recognition problem and propose a novel pattern recognition algorithm—Move Evaluation Tree System (METS). Then this paper gives a full description of the new algorithm and an evaluation of its significance.

2 Description of the Algorithm

2.1 Pattern Recognition Problem in the Game of Go

Before describing an algorithm, we must define the pattern recognition problem in the game of Go. In our approach, training data is *N* game records that contain totally *X* moves. Of course, game states are sequentially changing, and players may decide their moves in the series of moves. However, in this paper, we simplify the training data as snap-shots: we treat the training data as a just assembly of information which consists of a game state and a move point. Then the question is “where should we put a new move in a given unknown game situation?”.

An outline of the METS algorithm is as follows: at first, we collect patterns around only move positions from training data; when training data contains *X* moves, the number of patterns are *X*. Next, we cluster them using a system resembling a “decision tree classifier”. Finally, we give every cluster a score, or priority. In the following sub-sections, we define training patterns, and next we describe the tree making phase and the score making phase of the algorithm.

2.2 Training Patterns

Training patterns are patterns around move positions, which have two-dimensional topology. The maximum range of pattern is a 37x37 square, because the size of a board is 19x19 and the pattern around the upper-right corner of the board must contain the lower-left corner of the board, for example. Practically, we don’t need such wide range of patterns; we limit the size of patterns to 17 Manhattan distance around the move position. Each pattern consists of digits, where each digit is digit represents either a white stone ( ), a black stone ( ), empty ( ) or off-board ( ); strictly speaking, we deal with black turn and white turn symmetrically, and stone types are not white or black but self or opponent. For example, a pattern around the first-move position is a pattern that consists of only digits of either empty or off-board. A pattern around the position A of the Fig. 3 has four black stones and four white stones in the near side and seven black stones and seven white stones totally. Notice that the digits that construct patterns themselves are just digits with two-dimensional topology and they have no order. Order of digits is put through the tree making phase.

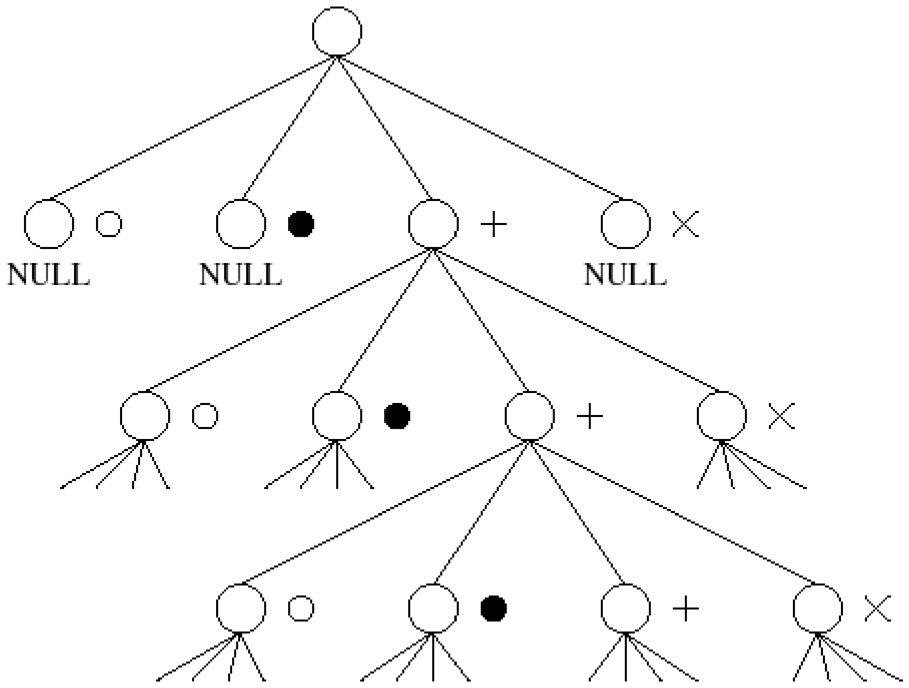


Fig. 1: A Decision Tree
Circles with lines indicate “nodes”. The apex node is the “root node”.
States of nodes are shown right-hand side of circles (white, black, empty, and off-board from left to right). The three nodes below the root node are “null nodes”.

2.3 The Tree Making Phase

The algorithm, which we call the Move Evaluation Tree System (METS), is like a ‘decision tree’ classifier (see, for example, [3]). The final result of the algorithm is a tree such as that of Fig. 1; the tree divides all training patterns step by step. The tree consists of three kinds of nodes; internal nodes, leaf nodes and null nodes. At each internal node, we put a decision about a state of digit—“which state the digit is either white, black, empty, or off-board, at **the target position**”—which results in four branches. For example, at the apex node—the root node—in Fig. 1, all training patterns remains and we must divide them. Each internal node includes some training patterns and we divide or cluster them into child nodes if necessary. The full algorithm is as follows.

```
step 1. At each node, select one position—the target position—to watch
step 2. Divide training patterns into four groups (white, black, empty, and off-board)
        according to the state of the digit at the chosen position in step 1
step 3. At each child node
    step 3.1 If a child node has no training patterns, the node become null node.
    step 3.2 If a child node has less training patterns than the threshold, the node
             become leaf node.
    step 3.3. If neither of the above case hold, goto step 1
When there remains no internal node, we stop making the tree
```

We pass through *n* decisions until we reach an internal node in the *n*-th depth of the tree: i. e. the internal node can be identified with the pattern that consists of digits of passed decisions. Finally, a leaf node can be identified with a pattern with digits of all passed decisions, and there are less training patterns than the threshold, which have the same pattern.

Still unsettled problem is which position we should choose at **step 1**. Choosing process is illustrated in Fig. 2. In Fig. 2, the move position is "kakari" position against a white "hoshi" stone. A priori, we group positions in terms of Manhattan distances: the first group consists of one position—the move position—and the second group consists of four positions, and the *n*-th group consists of *4*(n-1)* positions.

The algorithm has two rules—range constraint and entropy constraint—to choose position at **step 1**. The former is that we surely select positions in accordance with the above grouping: i. e. we can not select positions in (*n+1*)-th group until all positions within *n*-th group have been selected. This constraint is derived from the intuition that the closer a board intersection is to the center of a patter, the more critical its state becomes. Note that at the root node, we necessarily watch the move position, i. e. empty position, which results in the three null nodes of Fig. 1.

The latter is that we select a position among a group in terms of an entropy, which can be calculated as follows; $entropy = -\sum p_{branch} \log(p_{branch})$, where P_{branch} is the probability of a branch. This constraint is for the purpose that we want to make a tree as compact as possible.

Through the tree making phase, we can get a tree system that clusters training patterns. If the number of all leaf nodes is *L*, training patterns turn out to be divided into *L* clusters. Next, we put scores on each leaf nodes.

2.4 The Score Making Phase

Almost every empty position can be classified into a leaf node of the tree. In the score making phase, we use this phenomenon. We scan all *X* game states in training data: at each state, all empty positions are classified by the tree which was made in the tree making phase. Thus each positions have own leaf node numbers, otherwise drop into null node—we ignore positions which were classified into null nodes. Then if a position is move position, we increment a "***positive count***" of the leaf node, and if a position is not-move position, we increment a "***negative count***". After we scan all empty position in the training data, we can get total positive and negative count at each leaf nodes. Finally, we give a leaf node a $score = \frac{negativecount}{positivecount}$, where a smaller score is better. Note that positive count must be more than zero, because a leaf node contains some training patterns. The score reflects a kind of negative emergency: if a position which has a high score, the position has remained untouched through a considerable amount of states in the training data.

When we give all leaf nodes scores, we can put each positions scores at a given unknown game situation, and positions are sorted increasingly in terms of the scores. Now we can put a new move according to the priority.

3 Experimental Results

3.1 Tree Structures

In the experiments, training data consists of 34,266 game records and 7,067,744 moves, and test data consists of other 346 game records and 70,817 moves from "Kifu Database 96". By changing the threshold of **step 3.2** of the algorithm, we can get different trees with various sizes. We made three kinds of METS in the experiments: METS A with the threshold 50, METS B with the threshold 500, and METS C with the threshold 5,000. Tree sizes and numbers of whole leaf nodes of each METS are shown in Table 1. Approximately, sizes of trees are inversely proportional to the values of threshold.

Examples of leaf nodes in METS A are shown in Fig. 3. Both A and B position are on black turns. The leaf number of position A is #536959 whose score is 0 which means high emergency, while the leaf number of position B is #314275 whose score is 28656 which means low emergency. Before reaching each nodes, we check all positions within the closed area by dotted lines: i. e. each nodes can be identified with patterns within the area.

3.2 Recognition Rates

To evaluate significance of METS, we check recognition rates for the test data. At all situations of the test data, we put scores on all positions by METS and sort positions increasingly in terms of scores: note that the score indicates negative emergency, and positions with low scores are high emergency points. In fact, each situation in the test data has a move position—i. e. each situation has an answer position. If the position with the lowest score hit the answer, pattern recognition succeeded. Fig. 4 shows cumulative recognition rates of METS A, B and C; "*n*-th" means that the target move is contained in the *n* moves with lowest scores. Recognition performance increases from METS C to METS A.

3.3 Discussion

At first, we must notify that METS classifies patterns with extremely high speed. For example, an average depth of leaf nodes in METS A is 46.86, which means that an average of matching time per a position become less than 47*(read time + seek time). The time becomes in the order of msec in 200Mhz CPU computers. The system mainly owes the high recognition speed to the tree structure, and the speed doesn't decrease so much however large a tree becomes.

Secondly, METS gives us objective and quantitative significance of positions: the METS algorithm scores all positions at any situation of the game of Go. Though some researchers succeeded in selecting positions by using database, they seem to have difficulty in giving scores and the algorithms of scoring are a little ad-hoc [1], [2]. In fact, we tried some values within the structure of trees—number of training patterns of leaf nodes or depth of leaf nodes, however the values didn't give us good scoring. Consequently, in order to give a leaf node an adequate score, we must consider negative data—patterns around not-move positions.

Thirdly, we discuss recognition rates of METS. The recognition rate—25%—of METS A seems a very high recognition rate, however it is difficult to evaluate recognition rates which METS showed in the experiments; an experiments of this type—using a huge database—is very unique and we have no references. Qualitatively speaking, the recognition rates of METS are not enough for program of Go but very helpful for selecting candidates. The program—"Monkey Jump"—decides a new move using almost only METS, i. e. "Monkey Jump" always put moves at the position with the lowest score, and fought at the third FOST cup ranking 26-th (program number is No. 25, 4 wins and 6 loses) [4]. Indeed "Monkey Jump" played well in the beginning of games, but in the middle of games it started giving very wrong moves. The fact means that stone configurations are not enough for deciding moves in the middle of games. The system must watch structures of Go; for example strength of stone groups, life and death of stones, etc. I have improved METS and developed a new system that overcomes the deficits, which will be disclosed in the near future.

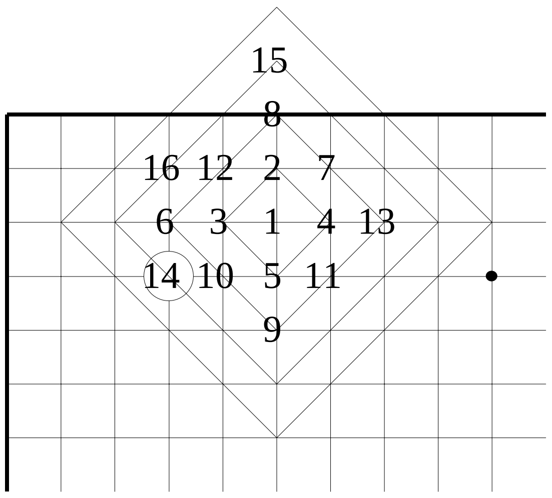
References

[1] S. Sei and T. Kawashima, "The Experiment of Creating Move from "Local Pattern" Knowledge in Go Program, *Proceedings of Game Programming Workshop in Japan '94*, pp. 97—104, 1994 (in Japanese)

[2] T. Kojima, K Ueda, and S. Nagano, "An Evolutionary Algorithm Extended by Ecological Analogy and its Application to the Game of Go", *Proceedings of IJCAI'97*, pp. 684—689, 1997

[3] S. R. Safavian and D. Landgrebe, "A Survey of Decision Tree Classifier Methodology", *IEEE Trans. SMC*, Vol. 21, No. 3, pp. 660—674, 1991

[4] A. Yoshikawa, "Report of the third FOST cup", *bit Vol. 29, No. 12*, pp. 12—18, 1997 (in Japanese)



the 1st group --- 1
the 2nd group --- 2,3,4,5
the 3rd group --- 6,7,8,9,10,11,12,13
the 4th group --- 14,15,16.....

Fig. 2
Target position is "kakari" against "hoshi". Positions are grouped in terms of Manhattan distances. Numbers of positions are an order for watching for example.

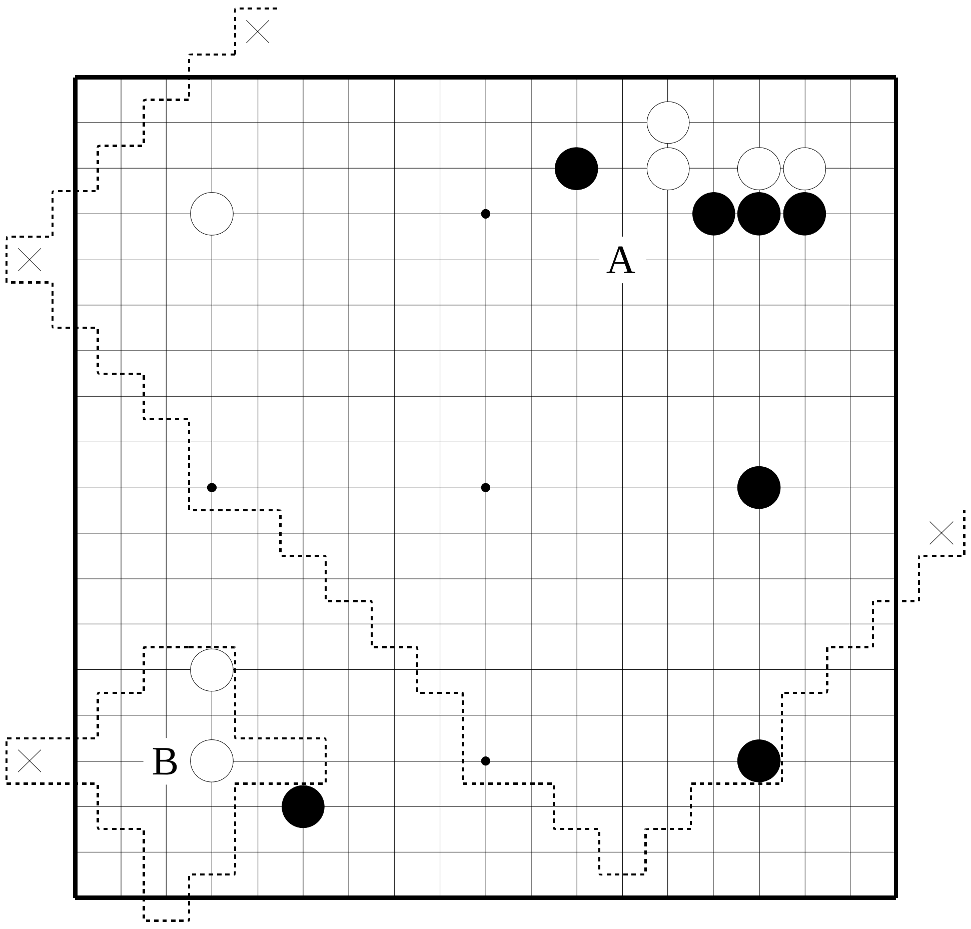


Fig. 3
Examples of leaf nodes; a pattern around position A is classified to a leaf node #536959 and B to #314275. Areas closed by dotted lines indicate areas to watch in each leaf nodes.

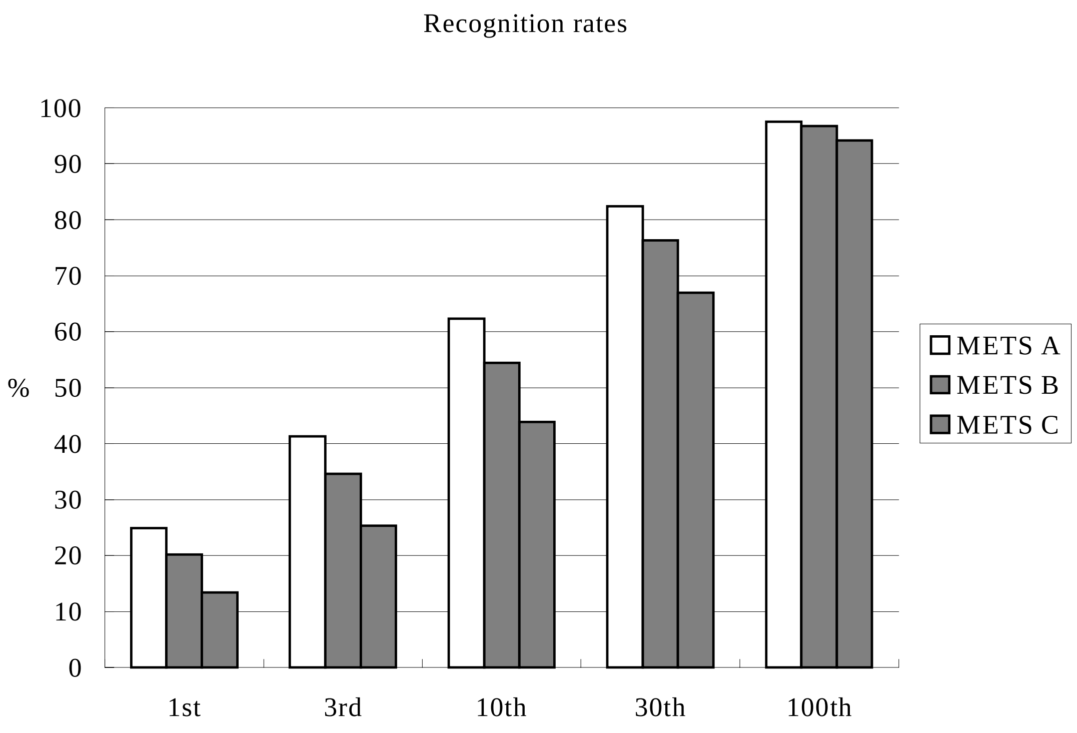


Fig. 4
In the graph, a recognition rate of *n*-th means a ratio in which move positions are within *n*-th candidates

	METS A	METS B	METS C
threshold	50	500	5000
tree size (byte)	16,260,496	1,540,668	129,280
number of leaf nodes	558,785	70,705	7,022

Table 1: the list of the tree size and the number of leaf nodes

Memory-Based Approach in Go-program “KATSUNARI”

Shinichi Sei ssei@ssl.fujitsu.co.jp
Toshiaki Kawashima kawkaw@ssl.fujitsu.co.jp
Fujitsu Social Science Laboratory Ltd.

Abstract

To make a strong Go-program, programmers analyze the thinking process of expert players and devise some algorithms to make good moves based on the result. However, such work is very difficult and there isn't a strong Go-program. We had proposed a new method which makes good move by using a large quantity of pattern knowledge extracted from professional players' games. And we had developed a Go-program “KATSUNARI”[1] which included the method. Because our method can make good moves by using pattern knowledge directly, we don't need difficult work of usual way. To make KATSUNARI stronger, we are improving mainly its database which has pattern knowledge. We classified pattern knowledge into two categories: the knowledge which show basic skill and the knowledge difficult to understand validity. Examples of former knowledge are pattern of JOSEKI and pattern of local competition and examples of latter knowledge are professional players' moves. We made two types of databases which included each knowledge. By using these databases properly, KATSUNARI can make good moves under any board situation than before. In this paper, we describe about new databases of KATSUNARI and how KATSUNARI uses them.

1 Introduction

The chess program “DeepBlue” defeated the human world champion Kasparov in 1997. However, the strongest Go-program has only strength of intermediate player of amateur[2]. Chess program became strong mainly by using search method, but the same method isn't suitable to Go-program. Reasons of it are the very large search space of Go-game compared with Chess and the difficulty of the evaluation of the board situation. Therefore, the new method to make strong Go-game program is necessary.

Most of Go-programs adopt the approach of imitating human strong player's thinking process. Programmers of these Go-programs analyze how human expert players recognize board situation and how they make moves. And they devise original algorithms to make move like expert players' and implemented those algorithms in their programs [Fotland, 1993][Chen, 1990][Sanechika, 1988][Fost, 1997][Fost, 1998]. However, this approach has a problem that all knowledge to make good move cannot be represented because the analysis of expert players' thinking process is too difficult. Although moves created by their original algorithms are sometimes good in typical situations, they aren't good in complicated situation like actual games. From these reasons, Go-programs have only strength of intermediate player of amateur.

Recently, there are some programs that adopt approach using learning functions[Brügmann, 1994] [Cazenave, 1996][Enderton, 1991][Stoutamire, 1991], but these programs are not strong yet.

We had proposed a new method which makes good move by using a large quantity of pattern knowledge extracted from professional players' games[Sei, 1994]. And we had developed a Go-program KATSUNARI which included the method[Sei, 1996]. A pattern knowledge consists of the professional players' move and local arrangement of stones around the move. We collected a large quantity of patterns automatically from professional players' games and make database retained them. KATSUNARI's process to make move is (1) compare stone arrangement on board with each pattern (2) propose written move in similar pattern as candidates (3) evaluate each candidate (4) select the best candidate by result of evaluation. Because KATSUNARI makes good move by using directly pattern knowledge extracted from professional players' games, we don't need analysis and devising original algorithms of usual way.

KATSUNARI participated in 1996 and 1997 World Open Computer Go Championship, FOST-CUP, to evaluate our method. However, we couldn't leave good

[1]This name is in Japanese.
[2]The 1997's champion program “HandTalk” and 1998's champion program “Silver IGO” were given a 3-kyu diploma by Nihon-Kiin.

records: ranking were 13th in 19 programs (4 wins and 5 loses) in 1996, and 20th in 38 programs (5 wins and 5 loses) in 1997. As a result of our investigation about why records were not good, we found that our method has some defects[Sei, 1998]. To make KATSUNARI stronger, we are improving mainly database which has pattern knowledge. In this paper, we describe about new databases of KATSUNARI and how KATSUNARI use them.

2 Memory-Based Approach

To make a strong Go-program, programmers analyze the thinking process of expert players and devise some algorithms to make good moves based on the result. However, such work is very difficult and there isn't a strong Go-program. It is easy to devise only some algorithms to accomplish single-purpose such as to surround territory or to capture stones. However, those algorithms can't frequently make a suitable move at complicated situation like actual game. Moreover, it is difficult to devise some algorithms to accomplish multi-purpose at same time. Therefore, the new method to make a strong Go-game program is necessary.

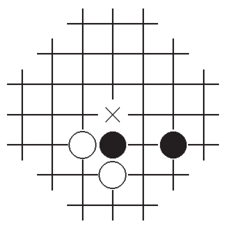
We had proposed to apply memory-based approach to make a strong Go-program. Memory-based approach means directly using a lot of knowledge which consists of problem and its solution to solve problems. Typical example which adopted this approach is Memory-Based Reasoning(MBR)[Stanfill, 1986]. The method retains a lot of previous experiences, retrieves the best similar experience from a collection and outputs it as the solution of a given problem. MBR has the feature that it can outputs good answer in a field where methods to solve problems are not established. There are some research reports using MBR in the pronunciation of word[Stanfill, 1986], machine translation[Kitano, 1993][Sato, 1993], and so on.

2.1 Pattern Knowledge

It is said that to make good stone arrangement is one of important tactics in Go-game. And we also know that strong human players make moves by considering local arrangement of stones. In Tsumego problem, there is a report that strong players use pattern knowledge that is the pair of move and local arrangement of stones around of move[Yoshikawa, 1996]. Therefore, we considered that using pattern knowledge is effective to make strong Go-program.

We show a example of KATSUNARI's pattern knowledge in Figure1. We defined a octagon shape as shape of pattern[3]. Reasons why we decided this shape are followings. Strong players usually don't consider positions of long distance from the position of candidate. Many of words(Keima, Ogeima, Ikken-Tobi, Niken-Tobi,...) which show the relation of stones in Go-terms are included in this shape. And we set center of pattern as the the position of candidate.

[3]This shape is the shape of improving KATSUNARI. The shape of old KATSUNARI is a little different from this.



: Position of candidate
at Black's turn
It is the position where
professional player put
in actual game.

Figure 1: Example of Pattern

2.2 Database Creation

Programs which adopted memory-based approach need a lot of knowledge. However, it is difficult to represent strong player's various knowledge into pattern. Moreover, we can't write down a lot of pattern knowledge by ourselves.

Before creating database, we classified pattern knowledge into two categories(Figure2): the knowledge which show basic skill and the knowledge difficult to understand validity. Examples of former knowledge are pattern of JOSEKI and pattern of local competition. Examples of latter knowledge are professional players' moves because those moves are so advanced that we can't understand their meaning and worth. We collected former knowledge from some textbooks and dictionaries about Go-game. We extracted latter knowledge from many professional players' games. We made the program to extract patterns automatically and collected about 50,000 patterns from about 400 games by professional players.

Patterns

Patterns which represent basic skill

- FUSEKI : standard sequence in opening stage
- JOSKEI : standard sequence in corner
- TESUJI : standard sequence in fighting
- Local Competition : Cut,Connect,HANE,$\cdots$

Patterns difficult to understand validity

Moves by expert players : They are collected from professional players' games.

Figure 2: Classification of Pattern

3 Go-program KATSUNARI

In this section, we describe about method of KATSUNARI which adopted memory-based approach.

3.1 Process to Make Move

We show the KATSUNARI's process to make move in Figure3. The detail of process is following.

1. *Pattern Candidate* Creation
   for each empty point on board
   (a) compare with arrangement of stones around the point and arrangement of stones of each pattern in database, and find out same pattern
   (b) propose this point as pattern candidate, if same arrangement of stones is found
2. *Capture Candidate* Creation
   for each stone on board
   (a) investigate status of stone (alive/dead/neutral)
   (b) propose move to capture/escape the stone, if status is neutral
3. Next Move Selection
   for each candidate
   (a) image a board where a candidate is temporary put, and estimate the board situation
   (b) adjust the value from considering the degree of importance of stone
   (c) select candidate with the highest value as next move

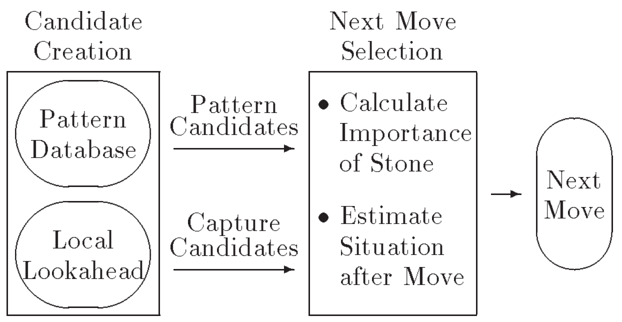


Figure 3: Process to Make Move

3.2 Candidates

KATSUNARI creates two kinds of candidates, these are *Pattern Candidate* and *Capture Candidate*. *Pattern Candidate* means candidates which created by using pattern knowledge, and it makes good arrangement of stones. *Capture Candidate* means candidates to capture enemy's stones and the candidates to escape family stones. And KATSUNARI creates some kinds of *Pattern Candidate*, these are *JOSEKI Candidate*, *FUSEKI Candidate*, *TESUJI Candidate*, *Small Pattern Candidate* and *Large Pattern Candidate*.

Capture Candidate
Capture Candidate means candidates to capture enemy's stones and the candidates to escape family stones. This candidate is created by local lookahead. Although some of candidates to capture/escape stones are created in creating *Pattern Candidates*, most of candidates to capture/escape stones aren't created. Because, KATSUNARI doesn't need to mind to make good arrangement of stones but to reduce DAME(liberty points) of stones to create *Capture Candidate*.

FUSEKI Candidate
FUSEKI is one of Go-terms and it is a standard sequence in the opening stage. FUSEKI is so analyzed in detail by human player and it is published as FUSEKI dictionary. We collected FUSEKI patterns from published FUSEKI dictionary and made FUSEKI pattern database. The bounds of FUSEKI pattern is larger than another kinds of pattern because program needs to considers wide scope(the size of the most large pattern is same as whole board). This candidate is established to overcome one of defects in old KATSUNARI. Old KATSUNARI was weak in the opening stage because it didn't have such large pattern.

JOSEKI Candidate
JOSEKI is one of Go-terms and it is a established or standard sequence at corner. It is so analyzed in detail by human player and it is published as JOSEKI dictionary as FUSEKI. We collected JOSEKI patterns from published JOSEKI dictionary and made JOSEKI pattern database.

TESUJI Candidate
TESUJI is one of Go-terms and a standard sequence to accomplish specific purpose. We collected TESUJI patterns from published TESUJI dictionary and some textbooks.

KATSUNARI estimates the worth of each candidate to select the best one. For estimation, KATSUNARI images a board where the candidate move is temporarily put and calculates how many points KATSUNARI leads. However, in this method, it is difficult to find out the move where the effect appears afterwards, such as sacrifice move. Then, to calculate the worth accurately, we added the pattern for evaluation to TESUJI pattern. We show examples in Figure4. When KATSUNARI estimates this candidate, it images the board where the pattern for evaluation is set. This candidate is established to overcome one of defects in old KATSUNARI, too.

Small Pattern Candidate
This is the candidate for local competition, e.g. Cut, Connect, HANE, NOBI, OSAE, TSUKIDASHI, FUKURAMI, etc. To make such moves, human players usually consider only arrangement of stones in small area. Then, we prepared pattern with small bounds to create these moves. We designed square(3×3) as the shape of pattern, and we set position of move at center of shape.

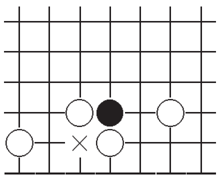
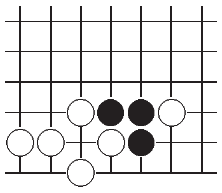
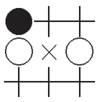
Figure of Black's Candidate Position	Pattern for Evaluation
	

Figure 4: TESUJI Pattern

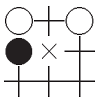
This candidate is established to overcome one of defects in old KATSUNARI, too. When there are many stones in small area, the size of old KATSUNARI's pattern is too wide to find same pattern. It often occurs especially in middle stage and end stage.

We added *the degree of emergency* to each *Small Pattern*. The degree is used to prune unnecessary candidates. KATSUNARI saves *Small Pattern Candidates* with high degree and abandons another. To calculate *the degree of emergency* accurately, KATSUNARI checks several items, these are arrangement of stones, amount of liberty of each stone, status of each stone(alive/dead/neutral) and each stone's distance from edge. We show examples in Figure5.

The technique of these moves is basic in Go-game. Even amateur player knows their meaning and worth. We could write all patterns by ourselves because the amount is small (about 1,000).



```
if( Left white's liberty $\leq$ 2 ){
    Emergency = 80;
}else if( Top left black's liberty $\leq$ 2 ){
    Emergency = 0;
}else{
    Emergency = 60;
```



```
if( Bottom line is edge ){
    Emergency = 80;
}else{
    Emergency = 60;
```

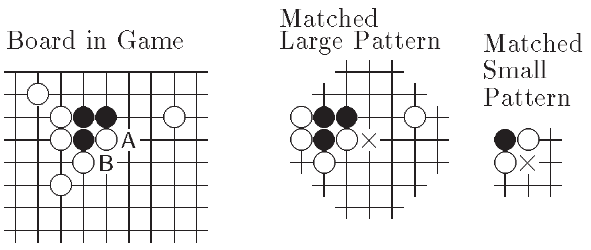
Figure 5: Small Pattern

Large Pattern Candidate
This patterns are extracted from professional players' games. We described about this candidate in previous section and showed a example pattern in Figure1.

Pattern Candidate Pruning
After KATSUNARI creates many *Pattern Candidate*, it prunes unnecessary candidates before evaluating each candidate.

The bounds of pattern is used to decide whether to prune or not. We explain about it in Figure6. When the left figure's board situation is given, KATSUNARI creates two kinds of different *Pattern Candidate*, there are *Large Pattern Candidate* at position A and *Small Pattern Candidate* at position B. However, if KATSUNARI don't create *Large Pattern Candidate* at B, *Small Pattern Candidate* of B is pruned. We can expect that the move created by considering wide scope is better than the move created by considering small scope. KATSUNARI considers that the candidate of B is not better than the candidate of A, because the bounds of *Large Pattern Candidate* of A covers *Small Pattern's* of B completely in this case.

KATSUNARI does this pruning for another kinds of candidate, too.



A: Position of *Large Pattern Candidate*
B: Position of *Small Pattern Candidate*
KATSUNARI prunes candidate of B

Figure 6: Pattern Candidate Pruning

4 Evaluation of KATSUNARI

4.1 FOST CUP

Because we are in the middle of improvement of KATSUNARI, we can't evaluate our method at the present. Indeed, a part of *Small Pattern Candidate* was implemented on 1997, and *FUSEKI Candidate* and *TESUJI Candidate* were implemented on 1998. Pruning function hasn't been implemented yet. We are still collecting pattern knowledge, there are only about 400 FUSEKI patterns and about 200 TESUJI patterns in KATSUNARI.

However, KATSUNARI participated in 1998 World Open Computer Go Championship, FOST-CUP, to evaluate the strength at that time. Although the record of present KATSUNARI isn't good, the record becomes better every year(Table1). We can expect that KATSUNARI become stronger after improvement.

Table 1: Results in FOST-CUP

Rank	wins - loses	year
1996	13th in 19 programs	4 - 5
1997	20th in 38 programs	5 - 5
1998	15th in 38 programs	4 - 2

4.2 Versus "The Strongest Game of Go"

"The Strongest Game of Go" is the name of commercial version of "Go4++". "Go4++" is one of top class Go-programs, 2nd in 1996 and 3rd in 1997 World Open Computer Go Championship. We did test matches with it in several times. Although KATSUNARI could win a few times and the average of score was about 20 points behind. But, the strength of advanced player commented that the difference of 20 points is small difference. We show one of scores in Figure7.

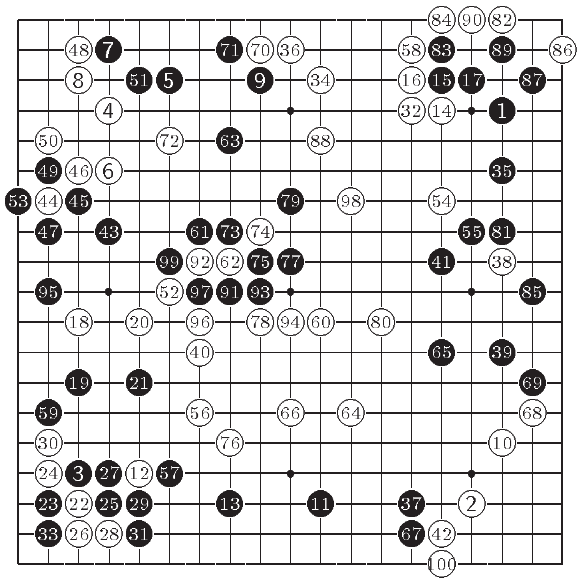


Figure 7: KATSUNARI(Black) vs The Strongest Game of Go

5 Related Go-Programs

Although most of Go-programs use a kind of original pattern knowledge, the aim of it is limited. It is generally used to create JOSEKI candidate or to create move to make/break *eyes*.

Recently, there are several programs which adopted memory-based approach like KATSUNARI. "Monkey Jump" and "KuruKuru" have a huge decision tree created from a lot of professional players' games, and they make move by using them[Fost, 1998][Fost, 1997]. To our regret, detailed reports about the method haven't been published yet.

6 Conclusion

We are improving Go-program KATSUNARI which adopted memory-based approach. KATSUNARI has various pattern databases created from different type of knowledge and makes good move by using these databases properly.

KATSUNARI's record in Computer Go Championship became better by our improvement every years. We can expect that KATSUNARI to become stronger after improvement.

Acknowledgment

We would like to thank Yukiyo Okano and Yasuhiro Ike for their helps in programming.

References

[Brügmann, 1994] Brügmann, B. Monte Carlo Go. available from Go Archive as Go/comp/mcgo.tex.Z, 1994

[Cazenave, 1996] Cazenave, T. Automatic Acquisition of Tactical Go Rules. *Proc. of the 3rd Game Programming Workshop*, pp.10-19, 1996

[Chen, 1990] Chen, K. The Move Decision Process of Go Intellect. *Computer Go*, No.14, pp.9-17, 1990

[Enderton, 1991] Enderton, H.D. The Golem Go Program. *CMU-CS-92-101*, Carnegie Mellon University, 1991

[Fost, 1997] *The Profile*, FOST-CUP, 1997

[Fost, 1998] *The Profile*, FOST-CUP, 1998

[Fotland, 1993] Fotland, D. Knowledge Representation in The Many Faces of Go. available from Go Archive as Go/comp/mfg.Z, 1993

[Kitano, 1993] Kitano, H. A Comprehensive and Practical Model of Memory-Based Machine Translation. *Proc. of 13rd International Joint Conference on Artificial Intelligence*, pp.1276-1282, 1993

[Sanechika, 1988] Sanechika, N. et al. "Go Generation" A Go Playing System. *ICOT Technical Report*, TR-545 1990

[Sato, 1993] Sato, S. Example-Based Translation of Technical Terms. *5th Int. Conf. on Theoretical and Methodological Issues in Machine Translation*, 1993

[Sei, 1994] Sei, S. and Kawashima, T. The Experiment of Creating Move from "Local Pattern" Knowledge in Go Program. *Proc. of the 1st Game Programming Workshop*, pp.97-104, 1994 (in Japanese)

[Sei, 1996] Sei, S. and Kawashima, T. The Experiment of the Go-program "KATSUNARI" using Memory-Based Reasoning. *Proc. of the 3rd Game Programming Workshop*, pp.115-122, 1996 (in Japanese)

[Sei, 1998] Sei, S. and Kawashima, T. Evaluation of the Method to Create Candidates using Pattern Knowledge Extracted from Professional Player's Scores in Go-Game. *CGF Journal*, Vol.2, 1998 (in Japanese)

[Stoutamire, 1991] Stoutamire, D. Machine Learning Applied to Go. *MS thesis*, Case Western Reserve University, 1991

[Stanfill, 1986] Stanfill, C and Waltz, D. Toward Memory-Based Reasoning. *Communications of the ACM*, Vol.29, 1986.

[Yoshikawa, 1996] Yoshikawa, A. and Saito, Y. Can not solve Tsume-Go Problems without looking ahead? *Proc. of the 3rd Game Programming Workshop*, pp.76-83, 1996 (in Japanese)

Succinct Games and Exotic Numeration Systems

Aviezri S. Fraenkel

Department of Applied Mathematics and Computer Science
The Weizmann Institute of Science
Rehovot 76100, Israel
`fraenkel@wisdom.weizmann.ac.il`
`http://www.wisdom.weizmann.ac.il/~fraenkel`

Extended Abstract

1 Introduction

For simplicity, we restrict attention to 2-player, 0-sum perfect information games without chance moves which are *acyclic*. Further, we consider normal play, i.e., the player making the last move wins, and the opponent loses. The tool for providing a polynomial strategy for this class of games is the *Sprague-Grundy* function [2], g-function for short: for a general digraph with n edges, the g-function can be computed in $O(n)$ steps.

A game is *succinct* if its input size is $O(\log n)$ rather than $O(n)$. An example of a succinct game is *Nim*, in which a finite number n of *tokens* (marbles, stones or the like) is arranged in $k \geq 1$ heaps (piles). A move consists of selecting a heap and removing from it a positive number of tokens, possibly the entire heap. The input size is $\sum_{i=1}^{k} \log n_i$, where $n_1, \ldots, n_k$ are the sizes of the k heaps. Because of succinctness, an additional property of the g-function is required to reduce the complexity from $O(\sum_{i=1}^{k} n_i)$ to $O(\sum_{i=1}^{k} \log n_i)$. The fact that the g-values of Nim are arranged in a simple arithmetic sequence constitutes this additional property. For the polynomial subclass of the class of *octal* games [9], polynomiality is usually established by showing that the g-function is periodic, though the period and/or preperiod can sometimes be very large [8].

In this note we show that for certain succinct games for which the g-function is highly chaotic, polynomiality can nevertheless be established by resorting to special numeration systems. We close with a question.

In any game, an N-position is any position from which the *N*ext (first) player can win, independent of the moves of the opponent. A P-position is any position such that the *P*revious (second) player can win, independent of the moves of the opponent. The set of all N-positions of a game is denoted by $\mathcal{N}$, and the set of all P-positions by $\mathcal{P}$.

2 An Example

Denote by $\mathbb{Z}^0$ the set of nonnegative integers, and by $\mathbb{Z}^+$ the set of positive integers. Define a family of succinct games, played on two heaps of tokens, which depends on two parameters $s, t \in \mathbb{Z}^+$. There are two types of moves: I. Take any positive number of tokens from a single heap, possibly the entire heap. II. Take $k > 0$ and $l > 0$ from the two heaps, where, say, $0 < k \le l$. This move is constrained by the condition $0 < k \le l < sk + t$, which is equivalent to $0 \le l - k < (s-1)k + t, \quad k \in \mathbb{Z}^+$. The case $s = t = 1$ is known as *Wythoff's game* [10], [4], [11].

For $s = t = 2$, the first few P-positions are listed in Table 1. What's its next entry?

Table 1: The first few P-positions for $s = t = 2$.

n	A_n	B_n
0	0	0
1	1	4
2	2	8
3	3	12
4	5	18
5	6	22
6	7	26
7	9	32
8	10	36
9	11	40
10	13	46
11	14	50
12	15	54
13	16	58

Let S be any finite subset of $\mathbb{Z}^0$. Define $\operatorname{mex} S = \mathbb{Z}^0 \setminus S$ = least nonnegative integer not in S. In [7], where the game has been proposed and analyzed, the following was proved:

Theorem 1. *$\mathcal{P} = \bigcup_{i=0}^{\infty}\{(A_i, B_i)\}$, where $A_n = \operatorname{mex}\{A_i, B_i : 0 \le i < n\}$ and $B_n = sA_n + tn$ $(n \ge 0)$.*

It is easy to see that if $A = \bigcup_{n=1}^{\infty} A_n$, $B = \bigcup_{n=1}^{\infty} B_n$, then A and B are *complementary*, i.e., $A \cup B$ = set of all positive integers, and $A \cap B = \emptyset$.

Given any two heaps of the game, containing x and y tokens with $x \le y$. The complementarity of A and B implies that either $x = A_n$ or $x = B_n$ for some n. Hence Table 1 has to be computed only up to the encounter of x. Moreover, it is not hard to see that $n \le x$, and if $x = A_n$, then $x/2 < n$, so the table has to be computed up to at most $\Omega(x)$, which implies a strategy computation linear in x. But because of succinctness, this is an exponential strategy! We now show that a linear strategy of complexity $O(\log x)$ exists, which is based on a special numeration system.

For fixed $s, t \in \mathbb{Z}^+$, put $u_{-1} = 1/s$, $u_0 = 1$, and let $u_n = (s+t-1)u_{n-1} + su_{n-2}$ $(n \geq 1)$. Denote by $\mathcal{U}$ the numeration system with bases u_0, u_1, ... and digits $d_i \in \{0, \ldots, s+t-1\}$, with the additional requirement $d_{i+1} = s+t-1 \Rightarrow d_i < s$ $(i \geq 0)$. Every positive integer has a unique representation over $\mathcal{U}$, [5].

Consider the special case $s = t = 2$. Then $u_{-1} = \frac{1}{2}$, $u_0 = 1$, $u_1 = 4$, $u_2 = 14$, $u_3 = 50$, $u_4 = 178$, ... The representations of the integers 1 to 60 in this numeration system are displayed in Table 2.

Table 2: A quaternary representation of the first few integers in $\mathbb{Z}^+$.

50	14	4	1	n	14	4	1	n
	2	0	3	31			1	1
	2	1	0	32			2	2
	2	1	1	33			3	3
	2	1	2	34		1	0	4
	2	1	3	35		1	1	5
	2	2	0	36		1	2	6
	2	2	1	37		1	3	7
	2	2	2	38		2	0	8
	2	2	3	39		2	1	9
	2	3	0	40		2	2	10
	2	3	1	41		2	3	11
	3	0	0	42		3	0	12
	3	0	1	43		3	1	13
	3	0	2	44	1	0	0	14
	3	0	3	45	1	0	1	15
	3	1	0	46	1	0	2	16
	3	1	1	47	1	0	3	17
	3	1	2	48	1	1	0	18
	3	1	3	49	1	1	1	19
1	0	0	0	50	1	1	2	20
1	0	0	1	51	1	1	3	21
1	0	0	2	52	1	2	0	22
1	0	0	3	53	1	2	1	23
1	0	1	0	54	1	2	2	24
1	0	1	1	55	1	2	3	25
1	0	1	2	56	1	3	0	26
1	0	1	3	57	1	3	1	27
1	0	2	0	58	2	0	0	28
1	0	2	1	59	2	0	1	29
1	0	2	2	60	2	0	2	30

A question we just might ask at this point is whether there is any connection between Tables 1 and 2. If we scan the first few entries of both, we may be tempted to conclude that all the entries under A_n in Table 1 have representations ending in no 0 in Table 2. But then 14 is a counterexample, whose representation ends in two 0s. Also it appears that the B_n all have representation ending in a single 0. But 50, with representation 1000 is a counterexample, in fact, the only counterexample in the range of the two tables.

It turns out, however, that the following two remarkable, æsthetically pleasing, properties hold in general:

a. All the A_n have representations ending in an *even* number of 0s, and all the B_n have representations ending in an *odd* number of 0s.

b. For every $(A_n, B_n) \in \mathcal{P}$, the representation of B_n is the "left shift" of the representation of A_n.

Thus $(1,4)$ of Table 1 has representation $(1,10)$, and $(6,22)$ has representation $(12,120)$: 10 is the "left shift" of 1, 120 the left shift of 12. We remark that the second part of **a** follows from its first part, since A and B are complementary.

We leave it to the reader to show that these observations lead to an easy polynomial strategy for our class of games. We remark that for Wythoff's game ($s = t = 1$), and even for a generalization thereof [6] ($s = 1,\ t \geq 1$), there is another polynomial strategy, based on the fact that then $A_n = \lfloor n\alpha \rfloor$, $B_n = \lfloor n\beta \rfloor$, where $\alpha = (2 - t + \sqrt{t^2+4})/2,\ \ \beta = (2 + t + \sqrt{t^2+4})/2$. But for $s > 1$, there exist no such α, β, and we don't know whether there is a polynomial strategy other than the one based on the above numeration system.

3 Another Numeration System

A special case of a numeration system considered in §4 of [5] is based on the even-indexed Fibonacci numbers, namely, $1, 3, 8, 21, 55, 144, \ldots$. They satisfy the recurrence $p_n = 3p_{n-1} - p_{n-2}$ $(n \geq 1)$, where $p_{-1} = 0$, $p_0 = 1$. Every positive integer N has a unique representation in a ternary numeration system of the form $N = \sum_{i \geq 0} d_i p_i$, where the digits satisfy $d_i \in \{0, 1, 2\}$ $(i \geq 0)$, and the additional condition: if for some $0 \leq k < l \leq n$, $d_l = d_k = 2$, then there is j with $k < j < l$ such that $d_j = 0$. The representations of the first few positive integers are given in Table 3. This numeration system has also been used in [3].

In Table 4 we display the first few positive integers A_n whose representation in this ternary numeration system ends in an even number of 0s, and the numbers B_n whose representation ends in an odd number of 0s.

The following property seems to hold: $A_n = \operatorname{mex}\{A_i, B_i : 0 \leq i < n\}$, $B_n = 2A_n + n - r_n$, where $r_n = |\{k : B_k - B_{k-1} = 2,\ k \leq n\}|$. Is there a game with two heaps and *simple* rules such that $\mathcal{P} = \bigcup_{i=0}^{\infty}\{(A_i, B_i)\}$? That is, the rules should be independent of the *size* of either heap.

Remarks.

- The special case $s = t = 2$ of the (s,t)-sequences considered above was recently put into Neil Sloane's *On-Line Encyclopædia of Integer Sequences*

Table 3: A ternary representation of the first few integers in $\mathbb{Z}^+$.

55	21	8	3	1	n	21	8	3	1	n
	1	1	0	0	29				1	1
	1	1	0	1	30				2	2
	1	1	0	2	31			1	0	3
	1	1	1	0	32			1	1	4
	1	1	1	1	33			1	2	5
	1	1	1	2	34			2	0	6
	1	1	2	0	35			2	1	7
	1	1	2	1	36		1	0	0	8
	1	2	0	0	37		1	0	1	9
	1	2	0	1	38		1	0	2	10
	1	2	0	2	39		1	1	0	11
	1	2	1	0	40		1	1	1	12
	1	2	1	1	41		1	1	2	13
	2	0	0	0	42		1	2	0	14
	2	0	0	1	43		1	2	1	15
	2	0	0	2	44		2	0	0	16
	2	0	1	0	45		2	0	1	17
	2	0	1	1	46		2	0	2	18
	2	0	1	2	47		2	1	0	19
	2	0	2	0	48		2	1	1	20
	2	0	2	1	49	1	0	0	0	21
	2	1	0	0	50	1	0	0	1	22
	2	1	0	1	51	1	0	0	2	23
	2	1	0	2	52	1	0	1	0	24
	2	1	1	0	53	1	0	1	1	25
	2	1	1	1	54	1	0	1	2	26
1	0	0	0	0	55	1	0	2	0	27
1	0	0	0	1	56	1	0	2	1	28

Table 4: The first few P-positions — of which game?

n	A_n	B_n
1	1	3
2	2	6
3	4	11
4	5	14
5	7	19
6	8	21
7	9	24
8	10	27
9	12	32
10	13	35
11	15	40
12	16	42
13	17	45
14	18	48

at http://www.research.att.com/˜njas/sequences/index.html , sequence numbers A045671, A045672.

- A sequence defined quite differently, namely A026366, turns out to be equivalent to an (s,t)-sequence with $s = 2$, $t = 1$. In [7] this definition was generalized to any $s, t \in \mathbb{Z}^+$, and the equivalence to the above (s,t)-sequences was proved.

- In [1], an interpretation for the recurrence $f_{n+1} = 6f_n - f_{n-1}$ was found. Other interpretations, in terms of numeration systems belonging to the same family as the system considered last, can be given. We give a few examples. Consider this recurrence with $f_0 = 1$ throughout. If we put $f_1 = 7$, we get a numeration system with digits satisfying $0 \le d_i \le 5$ $(i \ge 1)$, $0 \le d_0 \le 6$ and: if d_k and d_l are maximal, then there is j with $k < j < l$ such that $d_j < 4$. If $f_1 = 6$, the numeration system digits satisfy $0 \le d_i \le 5$ $(i \ge 0)$and the same additional condition. If $f_1 = 9$, then $0 \le d_i \le 5$ $(i \ge 1)$, $0 \le d_0 \le 8$ and the same additional condition.

References

[1] E. Barcucci, S. Brunetti, A. Del Lungo and F. Del Ristoro 1998, A combinatorial interpretation of the recurrence $f_{n+1} = 6f_n - f_{n-1}$, *Discr. Math.* **190**, 235–240.

[2] E. R. Berlekamp, J. H. Conway and R. K. Guy 1982, *Winning Ways* (two volumes), Academic Press, London.

[3] F. R. K. Chung and R. L. Graham 1981, On irregularities of distribution of real sequences, *Proc. Nat. Acad. Sci. U.S.A.* **78**, 4001.

[4] H. S. M. Coxeter 1953, The golden section, phyllotaxis and Wythoff's game, *Scripta Math.* **19**, 135–143.

[5] A. S. Fraenkel 1985, Systems of numeration, *Amer. Math. Monthly* **92**, 105–114.

[6] A. S. Fraenkel 1982, How to beat your Wythoff games' opponents on three fronts, *Amer. Math. Monthly* **89**, 353–361.

[7] A. S. Fraenkel 1998, Heap games, numeration systems and sequences, to appear in *Annals of Combinatorics.*

[8] A. Gangolli and T. Plambeck 1989, A note on periodicity in some octal games, *Internat. J. Game Theory* **18**, 311–320.

[9] R. K. Guy and C. A. B. Smith 1956, The G-values of various games, *Proc. Camb. Phil. Soc.* **52**, 514–526.

[10] W. A. Wythoff 1907, A modification of the game of Nim, *Nieuw Arch. Wisk.* **7**, 199–202.

[11] A. M. Yaglom and I. M. Yaglom 1967, *Challenging Mathematical Problems with Elementary Solutions*, translated by J. McCawley, Jr., revised and edited by B. Gordon, Vol. II, Holden-Day, San Francisco.

Zero Sum Games As Distributed Cognitive Systems

Robert L. West (rwest@hkucc.hku.hk)
Department of Psychology, University of Hong Kong, Hong Kong, PRC.

Abstract

In the case of two individuals in a competitive situation, or "game," the game itself (i.e. the players, the rules, the equipment) can be considered to constitute a distributed cognitive system. However, the dominant model of competitive behavior is game theory (VonNeumann & Morgenstern, 1944), which has traditionally treated individuals as isolated units of cognition. By simulating game playing with neural networks, and also by using human subjects, it is demonstrated that the interaction between two players can give rise to emergent properties which are not inherent in the individual players.

Recent work in distributed cognition (e.g. Hutchins, 1994; Norman, 1993; Zhang, 1997; Zhang & Norman, 1994) has indicated that cognitive processing can take place across distributed systems composed of multiple, interacting cognitive systems. For example, navigating a large ship, such as a naval vessel, is accomplished through interactions amongst specially trained humans and specialized equipment (Hutchins, 1994). Distributive systems involving more than one agent are prototypically cooperative in nature, in that the agents involved benefit from the function of the distributed system (e.g. a ship avoids sinking). However, distributed systems may also result in situations in which some individuals benefit at a cost to others. The simplest example of this is the case of two individuals in a zero sum game (i.e. a game in which only one player can win). Games such as this can be thought of as distributed cognitive systems with the goal of choosing one player as the winner.

Although game playing clearly involves interactions between the players, it does not necessarily follow that we need to consider the distributed properties of a game in order to understand the behavior of a player. This depends on whether the functionality of the cognitive mechanism used by an individual player can be understood in isolation, or needs to be interpreted in terms of the role it plays in the distributed system. The answer to this question will depend to some degree on our assumptions concerning the game playing process. For example, game theory (VonNeumann & Morgenstern, 1944) describes how rational players should behave in a competitive situation prescribed by rules and with payoffs for certain results. However, in order to do this it is necessary to make assumptions concerning the cognitive mechanisms available to the players. One assumption that is frequently made is that players have the ability to generate random responses (i.e. to draw responses at random from a predetermined distribution). For example, the game theory solution for Paper, Rocks and Scissors (hence forth PRS) is to play randomly, 1/3 paper, 1/3 rocks, and 1/3 scissors (in PRS play: paper beats rocks, rocks beats scissors, and scissors beats paper). With this assumption in place there is nothing to be gained by viewing PRS as a distributed system because players' interactions are limited to tossing out and receiving random responses. However, the assumption of random responses is problematic for two reasons. The first is that people are normally quite bad at generating random responses (see Tune, 1964, and Wagenaar, 1972 for reviews), and the second is that when people guess what is coming next in a series they attempt to capitalize on sequential dependencies, regardless if they are present or not (e.g., Anderson, 1960; Estes, 1972; Restle, 1966; Rose & Vitz, 1966; Vitz & Todd, 1967; Ward, 1973; Ward & Li, 1988).

Given the above research, a more realistic model of PRS play would have players trying to detect each others sequential dependencies. Note that the story is now different if we consider the players in isolation or if we consider them within the context of the distributed system formed by the game. Taken in isolation, a player's strategy appears passive, limited to searching for sequential dependencies in their opponents responses. However, from the distributed perspective the situation is highly interactive as each player both drives, and is driven by, their opponent's responses (i.e. my behavior would be based on my beliefs about sequential dependencies in my opponents play, which would be driven by my opponents behavior, which in turn is driven by my behavior in a similar way) . The question is, whether this highly interactive situation can impart an alternative functional significance to a sequential detection mechanism?

The Decoy Strategy

Given an opponent who is using the strategy of searching for sequential dependencies, we can ask the game theory question of how a rational opponent should respond. Generating random responses will certainly avoid any disadvantage, but it will also fail to produce an advantage. The ideal strategy under these conditions would be to use one's own responses to lure the opponent into a predictable pattern of play which could somehow be exploited. Interestingly, this agrees well with peoples' reports of how they play games such as PRS. Aside from a minority who claim to respond randomly, most people claim to deceive their opponents by allowing them to detect biases which are, in reality, decoys drawing their opponents into a predictable pattern of play. This strategy of using ones own pattern of responses to exert control over one's opponent's responses will be referred to as the decoy strategy.

What I will endeavor to show in this paper is that the function of sequential detection mechanisms within a game situation is not to passively detect sequential dependencies, but to execute the decoy strategy. Furthermore, it will be demonstrated how the ability to do this is mediated by working memory.

Simulating the Decoy Strategy

The sequential detection mechanisms assumed to be used by human game players were modeled using two layer neural networks with one layer for input and one for output (i.e. perceptrons, Rosenblatt, 1962). The output layer consisted of three nodes, to represent paper, rocks, and scissors. The input layer consisted of a variable number of three node sets. Each set represented the previous outputs of the opponent network at a particular lag, with the three nodes in each set again representing paper, rocks, and scissors. Thus the networks could be set to "remember" any number of trials back from the current trial. To represent this the networks are be referred to in terms of how many lags back they could recall (i.e. a lag1 network can remember one trial back, and a lag2 network, two trials back). Outputs were determined by summing the weights associated with the activated connections. If two or more output nodes were equally weighted the tie was resolved through random selection. Learning was accomplished through back-propagation in which a win was rewarded by adding 1 to the activated connections leading to the node representing the winning response, and a loss was punished by subtracting 1 (ties were treated as losses). In all trials, both networks began with all weights set to zero.

The neural network mechanisms used in this study were deliberately made as simple as possible in order to keep the process as transparent as possible. Also, the use of perceptrons means that the individual networks can be treated as linear systems, an important consideration for game theorists.

Simulation Results

The effect of memory was clear, networks that could remember more always won in the long term. Figure 1 displays a representative result of a lag2 network versus a lag1 network. However, as would be expected by symmetry, when a lag2 was pitted against another lag2 network no advantage emerged.

According to the decoy strategy a player wins through controlling the opponent's responses. This strategy can be seen in the causal nature of the simulation results. The network with the higher lag factor was able to win not by passively detecting sequential dependencies but by creating them. Unlike humans who might be predisposed to generating sequential dependencies, the networks based their responses solely on each others play. Thus any tendencies for one network to be predictable were *caused* by the other network.

The Decoy Strategy in Humans

The next step was to find out if human subjects could execute the decoy strategy. To do this, human subjects played PRS against a lag1 network. There were two reasons for having them play against the network instead of against each other. First, it seemed likely that they would be approximately equal in ability which, according to the simulation results above, would produce an unremarkable outcome in the long run (i.e. a 50/50 chance of winning). Second, under these conditions the only sequential dependencies present in the computer would be ones *created* by the subject. In order to win subjects would have to both create and exploit sequential dependencies in the lag1 network.

Method

Subjects The subjects were 13 volunteers from the University of British Columbia and the University of Hong Kong.

Apparatus Subjects played against a lag1 network implemented in Visual Basic (the simulations were also done using the same program). Subjects selected their PRS outputs by using a mouse to click on three different icons. Following this they clicked on a button to reveal the computer's response. The score and the number of trials were displayed so subjects could monitor their progress.

Procedure Each subject played for approximately 20 minutes. The number of trials varied based on each subject's playing speed. All subjects played at least 250 trials (mean number of trials = 441). Subjects were instructed that the computer was programmed to play like a human, and that it was possible to beat it. They were also told that the program was very complex and that they should play by intuition.

Results

Figure 2 displays the subject's score minus the computer's score across trials. The data was combined so that each subject's game picks up where the previous subject left off (e.g. subject 1 finished with a lead of 44 points after 800 trials so subject 2 was plotted as though he began with a 44 point lead starting at trial number 801). This was done in order to get a sufficient number of trials (total number of trials = 5727) to indicate an unambiguous trend. The upward trend in Figure 2 is very clear and demonstrates that the human subjects were able to execute the decoy strategy.

Discussion

The neural networks used in this study were designed to passively detect sequential dependencies. The decoy strategy was not implicit in the design of these networks, but emerged from the *interaction* between them. Although it is possible that the human subjects were able to win by some other means it is unclear how this could be achieved. Also, it is doubtful if other explanations could achieve the same level of parsimony, or consistency with previous research.

The implications of these results go far beyond describing a good strategy for playing PRS, as it is possible that a considerable amount of competitive behavior is based on this type of process. More generally, the results of this study are consistent with the view that human cognition needs to be understood within in the environmental context

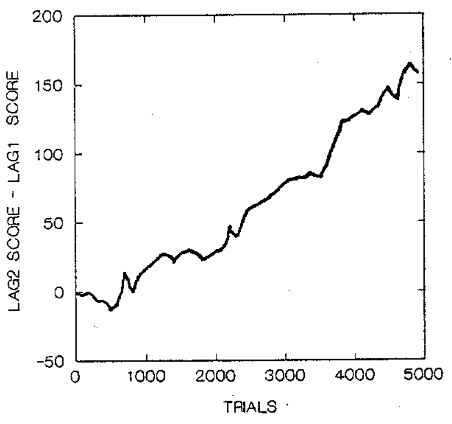


Figure 1: A Lag2 network versus Lag1 network

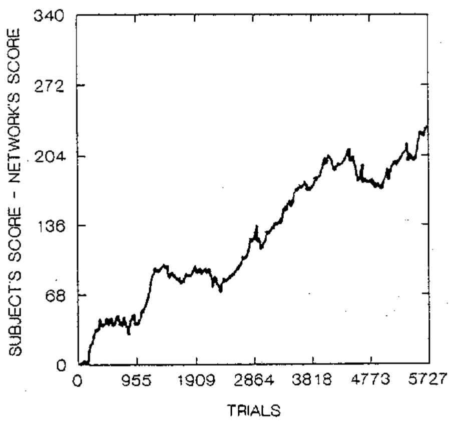


Figure 2: Human subjects versus a Lag1 network

in which it developed (e.g. Gibson, 1986). For humans this entails understanding an individual within a social context that is both cooperative and competitive. As demonstrated in this study, the benefit of an individual cognitive system may reside in the type of distributed system it creates when joined with other systems, rather than in its function as an isolated unit.

The advantage of the methodology used in this study is that it reconciles distributive and individual cognitive research for this type of behavior. Using the detailed findings of traditional cognitive research on individuals, tentative models can be constructed and placed in interactive situations. The emergent patterns from such simulations can then be compared to simulations in which one of the simulated agents is replaced with a human agent, or to interactions between two humans. In this way we can begin to understand the relationship between individual cognitive agents and the emergent, distributed systems in which we live.

References

Anderson, N. H. (1960). Effect of first-order probability in a two choice learning situation, *Journal of Experimental Psychology, 59,* 73-93.

Estes, W. K. (1972). Research and theory on the learning of probabilities. *Journal of the American Statistical Association, 67,* 81-102.

Gibson, J. J. (1986). *The ecological approach to visual perception.* Hillsdale, NJ: Erlbaum.

Hutchins, E. (1994). *Cognition in the wild.* Cambridge, MA: The MIT Press.

Norman, D. A. (1993). *Things that make us smart.* Reading MA: Addison-Wesley.

Restle, F. (1966). Run structure and probability learning: Disproof of Restle's model. *Journal of Experimental Psychology, 72,* 382-389.

Rose, R. M., & Vitz, P. C. (1966). The role of runs of events in probability learning. *Journal of Experimental Psychology, 72,* 751-760.

Rosenblatt, F. (1962). *Principles of neurodynamics.* New York: Spartan.

Tune, G. S. (1964). A brief survey of variables that influence random generation. *Perception and Motor Skills, 18,* 705-710.

Vitz, P. C., & Todd, T. C. (1967). A model of learning for simple repeating binary patterns. *Journal of Experimental Psychology, 75,* 108-117.

VonNeumann, J., & Morgenstern, O. (1944) *Theory of games and economic behaviour.* Princton, N. J.: Princton University Press.

Wagenaar, W. A. (1972). Generation of random sequences by human subjects: A critical survey of the literature. *Psychological Bulletin, 77,* 65-72.

Ward, L. M. (1973). Use of markov-encoded sequential information in numerical signal detection. *Perception and Psychophysics, 14,* 337-342.

Ward, L. M., Livingston, J. W., & Li, J. (1988). On probabilistic categorization: The markovian observer. *Perception and Psychophysics, 43,* 125-136.

Zhang, J. (1997). A distributed representation approach to group problem solving. *Journal of the American Society of Information Science,* in press.

Zhang J. & Norman, D. A. (1994). Representation in distributed cognitive tasks. *Cognitive Science, 18 (1),* 87-122.

Evaluation of Tsume-shogi with the Method of Least Squares

Toshinori KASUGA, Tsuyoshi SUZUKI, Yoshiyuki KOTANI
Tokyo University of Agriculture and Technology
{kasuga, go, kotani}@fairy.ei.tuat.ac.jp

Abstract

It's very difficult for computers to deal algorithmically with the human sense of the 'artistic value' of Tsume-shogi (shogi end game) problems. We propose an objective method for automatically producing evaluations similar to those of humans. We use the least square method, whose input consists of the factors: sacrificed piece numbers, the number of a king's move, etc.. All the weights attached to the factors are calculated by the method, using human scores of tsume-shogi problems in "Tsume-shogi Paradise" magazine as training values. We examined the difference between our automatic evaluation values and score by human, and verified the correctness of the result.

1 Introduction

Tsume-shogi is a puzzle game to find the checkmate move sequences when both players are assumed to play optimally. Tsume-shogi has a 400-years history in Japan as an artistic problem. Evaluation of tsume-shogi problems is determined by human sense. So far there is proposal of describing an evaluation value with the weighted linear sum. But the all weights for the value and human evaluation factors are determined by human sense [1]. We will derive the weights from parameters surveyed from the problems of "tsume-shogi paradise" journal. After that, we put it into practice to evaluate tsume-shogi using those parameters. To calculate the all weight, we make an experiment with the method of least squares.

2 The Method of Evaluation

This chapter shows how to calculate the evaluation value, and what factors are used for the purpose.

2.1 The Factor to Evaluate Tsume-shogi

We consider the following to evaluate a tsume-shogi problem:

(1) Constant
(2) The free area around King in the first position
(3) The free area around King in the end position
(4) The number of dropped sacrificed pieces
(5) The number of non-dropped sacrificed pieces
(6) The number of offence pieces on the first position
(7) The number of defence pieces on the first position
(8) The number of pieces in offence's hand in first position
(9) The extent of position pieces put on
(10) The number of King moves
(11) The number of check moves at start
(12) The number of pieces offence captures
(13) The number of unpromoted moves
(14) The number of double checks
(15) The number of discovered checks

"The free area" means the number of squares around King no having any pieces.
"Sacrificed pieces" means that defence gets the piece after the offence moves the check. This move is frequently used in Tsume-shogi.

2.2 The Method to Calculate the Value of Evaluation

The position is denoted by x_j. The evaluation value calculated by the evaluation function is $F(x_j)$. The factor i of position x_j has the value $f_i(x_j)$, the weight of the factor i is w_i. The evaluation value $F(x_j)$ is determined by the following linear sum:

$$F(x_j) = \sum_{i=0}^{n} w_i f_i(x_j)$$

The value of "Tsume-shogi paradise" journal is y_j, the sums of the squares of the differences between $F(x)$ and y_j is Q.

$$Q = \sum_{j=1}^{m} \left[f(x_j) - y_j \right]^2$$

The both sides of the above expression were differentiated by w_i, that value is made equal to 0.

$$\frac{\partial Q}{\partial w_i} = 0$$

The differential equation is solved, we get the linear equation. Calculating this linear expression about w, we can find the weight of all the factors.

3 Evaluation of Tsume-shogi

This chapter shows the results of evaluating Tsume-shogi.

3.1 Find the Weight

For determining the weights, we used 80 tsume-shogi problems from the shogi magazine "Tsume-shogi Paradise". All the problems have solution lengths of between 5 and 7 moves, and come from either the "Primary School" a "short solution contest" in the magazine.

Table 1. The result of calculation of weights

Evaluation Factors	Value of Weight
Constant(1)	1.187059
The free area around King at start	0.063114
The free area around King at end	-0.047179
The number of dropped sacrificed pieces	0.077269
The number of non-dropped sacrificed pieces	0.095142
The number of offence pieces on start position	0.062577
The number of defense pieces on start position	0.010568
The number of pieces in offence's hand	-0.004848
The extent of position pieces put on	0.002101
The number of King moves	0.018093
The number of checkmate moves at start	0.009809
The number of pieces offence captures	-0.120283
The number of unpromoted moves	0.043834
The number of double checks	0.026583
The number of discovered checks	0.116201

3.2 Evaluation of Tsume-shogi

We used the weights determined in Sec3.1 to calculate evaluations for 30 new problems that were not used in the training process.

The evaluation value of a tsume-shogi problem was determined in "Tsume-shogi" paradise using the way: readers evaluate a problem, evaluate it and choose one of the values: 3, 2 and 1. The problem thought best is valued at 3. We created a single value by averaging all the numbers chosen by about 140 readers who solved the problem.

The results to compare the true value and the value calculated by our system are shown in figure 1.

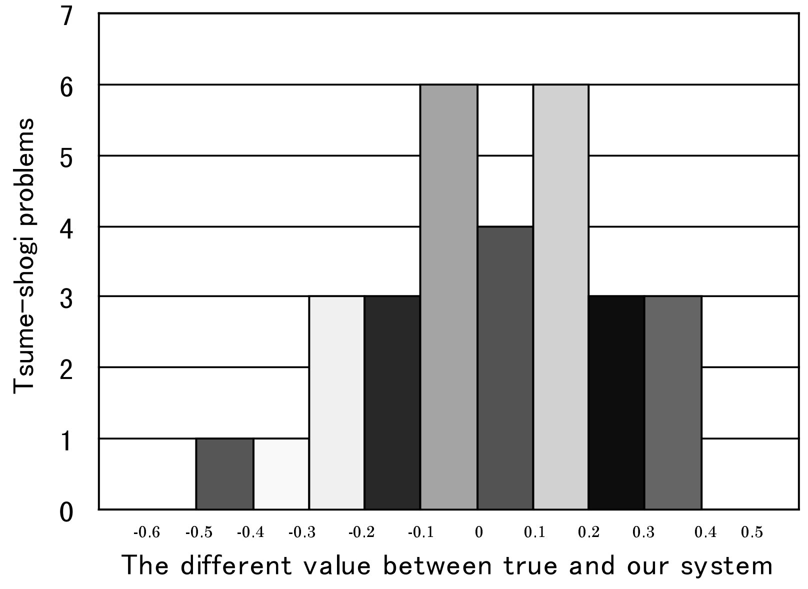


Figure 1. a distribution map of difference

The examples of the results are shown in appendix.

4 Discussion

The evaluation that our system calculated has a lot of difference from the human evaluation of "Tsume-shogi paradise" journal, in spite of obvious similarities. This interests us, since the factors reflect our common sense of tsume-shogi property.

The results showed the values our system calculated was lower than true values. It may be caused by the free area around King at end. Usually, it makes a favorable impression that the pieces around King are fewer. But that weight reduces a value. The more freedom the area around King has, the lower the value becomes.

Generally, it's supposed to be advantageous if the number of defense piece is bigger than the offence one. However, our system had the results opposite. The more pieces the offense has, the higher the value the problem would be.

When true values were very high or low in problems having as same factors, our system score was different from them. Human sense evaluates an aim of a tsume-shogi problem, too. If a system can understand it and obtain a good value, our system may be able to evaluate problems nearer true value.

5 Conclusion

We proposed a method of evaluation of tsume-shogi problems with the method of least square, and calculated evaluation values for the problems of Tsume-shogi paradise journal. A number of particular patterns were found, which relatively differ from human common sense. One of the reasons probably is that the set for the experiment was well selected, and did not reflect primitive quality difference. This may rather reflect the sense of enthusiasts' view.

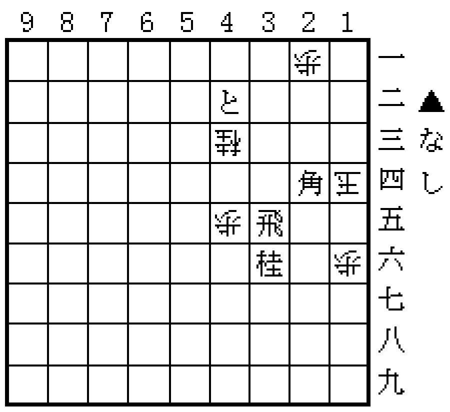
6 References

[1] The editorial department of Tsume-shogi Paradise : Tsume-shogi Paradise,1996.5 – 1998.5.

[2] K.Koyama: The Database and Evaluation by Human Sense of Tsume-shogi, Advance in Computer Shogi, kyoritsu publ., pp.90-124, 1996.

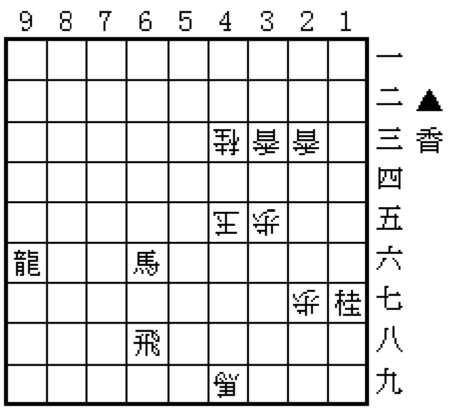
[3] M.Hirose, T.Ito, H.Matsubara : Automatic Composition of Tsume-Shogi by Reverse Method, Game Programming Workshop'96, pp.34-43, 1996.

Appendix



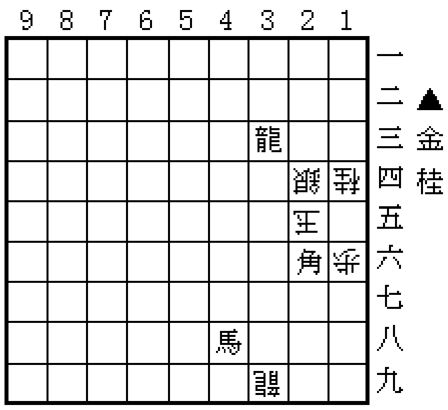
NO.1

True value	1.396226
Our system	1.769568



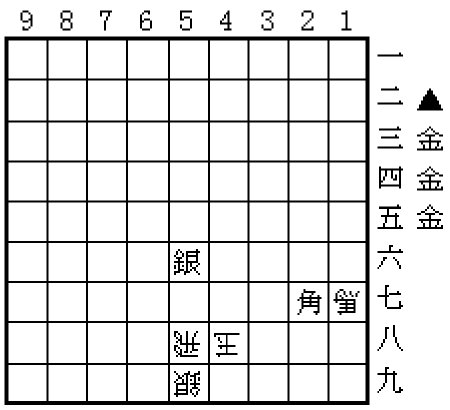
NO.2

True value	2.773333
Our system	2.612687



NO.3

True value	2.595960
Our system	2.236203



NO.4

True value	1.950355
Our system	1.962546

How players learn at "KANSO-SEN"

Takeshi Ito, Teiji Furugori
Division of Information Engineering, The University of Electro-Communications

1 Introduction

This paper reports some of our work on understanding how players learn at KANSO-SEN. KANSO-SEN is that the players exchange their opinions about each other's impressions on the game after the game is over. Most of IGO and SHOGI players do KANSO-SEN. We regard the KANSO-SEN as one of co-operative learning process. Co-operative learning is common works in educational or cognitive researches. For example, co-operative learning are not very effective at some small groups in mathematics (Mulryan, 1992). Another works that co-operative learning is more effective than individual learning (Ellison & Boykin, 1994). However, most of these works are learning on easy or convergent problems. These are not suitable for researching the effect of co-operative learning.

In our research we are interested in SHOGI and the KANSO-SEN as a difficult problem to understand how human learn at co-operative situation. It's broadly known from experience that KANSO-SEN is good to improve the players' SHOGI technique. The current study is an investigation of subjects' learning of SHOGI at KANSO-SEN between novices, experts and both.
In this way, we are assured that the behaviors we observe are many levels of players' co-operative learning processes.

2. The features of KANSO-SEN

We have already explained that KANSO-SEN is that the players exchange their opinions about each other's impressions on the game after the game is over. We must add that it is allowed that someone who watches the game joins the discussion once in a while. It goes without saying that players don't speak with each other while playing the game. Players' exchanging a few words with someone who watches the game is prohibited, too. Most players therefore have the motivation that want to know if their moves that they chose while playing the game were correct. Furthermore, at games of face-to-face type like SHOGI, chess, checker and so on each player has each viewpoint of the opposite side. So, players are present at KANSO-SEN with a great thrill of expectation that might come in contact with another viewpoint.

It is almost impossible that we find out a correct move at a situation of difficult games like SHOGI. We often use a kind of intuition to difficult situations like SHOGI. We call the intuition "TAIKYOKUKAN". It is important for players to have better TAIKYOKUKAN. To improve it, players need to discuss their opinions of many situations about games.

These expectations give KANSO-SEN the tone for ideal co-operative learning. We interpreted KANSO-SEN as a good example of co-operative learning. It is because that it gives two players different viewpoints naturally and makes the players motivated very much.

3. Experiments

3.1. Objectives

By examining players learning processes for KANSO-SEN of SHOGI, what kinds of information are exchanged and how they learn are investigated. We designed two conditions to examine the influence of partner's level. Learning processes of novice SHOGI player with who were the same level partner and the higher level partner at KANSO-SEN were observed.

3.2. Method

The subjects in this experiment were two students in the SHOGI club of our university. They were novice players of SHOGI (amateur 5 kyuu). We prepared another player for their partner at KANSO-SEN who was a middle-grade player of SHOGI (amateur 4 dan).

The actual experiments were conducted in the following order.

(1) Playing with a computer progam

In order to keep the equal partner's level of the SHOGI, a subject played a game with a computer program. The subject was allowed unrestricted thought time for the game. The partner for KANSO-SEN watched the game.

(2) Self-explaining the game

The subject was parted from the partner for KANSO-SEN for a while after the game is over.

The subject was asked to think aloud while reconstructing the game with using the computer replay function, and all performances are recorded by a video.

(3) Filling out the first questionnaire

The subject was asked to fill a questionnaire after he finished explaining the game. The contents of the questionnaire were " What is a point on this game? ", " What are good moves on this game? " and so on.

(4) Playing KANSO-SEN

The subject and the partner for KANSO-SEN were asked to discuss about the game, and all performances are recorded by a video.

(5) Filling out the second questionnaire

The subject was asked to fill an another questionnaire after he finished the KANSO-SEN. The contents of it were the same of the first questionnaire.

We designed two conditions. One was that novice subject discussed with novice partner (NN). Another was that novice subject discussed with middle-grade partner (NM). We examined two conditions each 9 times for 2 months and analyzed these verbal-protocol data above (2), (4).

3.3. Results and Discussion

<Result 1> One subject could beat the computer program that he had never beaten.

This result indicates that there are some effective learning processes that can't obtain in ordinary playing. We can't assert that the process is KANSO-SEN. It might be saying that speaking aloud about own game has some significance for effective learning. The subjects were observed that reflected the trace of their own games to learn a lesson from it.

<Result 2> In both conditions, subjects changed their opinions before and after KANSO-SEN.

This result was acquired from examining two questionnaires before and after KANSO-SEN. It indicates that most subjects are changed their thought by KANSO-SEN. examining process was observed at most KANSO-SEN, while it wasn't seen at Self-explaining process. A subject and the partner divided their roles of observer and inventor very neatly. That helped the examining operation.

<Result 3> they were talking on different wavelengths at NM-condition, while they made a perfect pair at NN-condition.

This is because novice and middle-grade players' intuitions are too different. The ideas that middle-grade player naturally generated weren't almost understood by novice player. On the other hand, the conversation grew lively between novice players.

<Result 4> In both conditions, players winnowed the candidates of moves down about from 2 to 5.

Many researches on problem solving have insisted that novice searches all possibilities for drugs. However, novice as well as middle-grade players could select a few candidates before they decided their moves. This result indicates that novice learn the way of thinking about SHOGI at first. The outline of thinking way to select their move is a sequential of thinking that they generate some candidates at a situation and read the tips to select a move. We call it "SHOGI players' script". This is because that they must winnow the candidates of move as players can't search for drugs in difficult problems like SHOGI.

<Results 5> Meta-cognitive utterances are observed at the latter period of this experiment.

Subjects refer to the meta-cognitive utterance like "I tend to counter with a defensive fall...” "Such move is particular to the computer..." and so on at the latter period of this experiment. This is because that the experiences that they replay and reflect their own game help their meta-cognitive views of SHOGI.

4. SHOGI players' script

Based on the results of the experiment a cognitive model "SHOGI players' script (SPS)" is proposed which simulates SHOGI move selecting. SHOGI players must learn the rules of SHOGI at first time. They begin to learn SPS at the same time, too.

From the results of the experiment the SPS consist of three stages of "Generating the candidates process", "Winnowing the candidates process" and "Selecting one move process".

In generating the candidates process the candidates are generates using intuition. At such difficult problems like SHOGI human always use intuition naturally. Then in the next winnowing the candidates’ process, players search the many situations from the candidates. Finally, in the Selecting one move process, these candidates are compared with using players' evaluation standard.

Using SPS model helps explain the learning processes at KANSO-SEN. Learning processes are regarded as refining processes of intuition and evaluation standard.

5. Conclusion

In this research by observing KANSO-SEN through a psychological experiment, a cognitive model SPS has been proposed. It has been explained the learning process with SPS model.

In the experiment of this research 5-kyu-grade novice has been used to obtain preliminary information about learning processes at KANSO-SEN. But slightly more low-grade novice expects not to have the SPS model. In the future it is hoped that psychological experiments how the lower-grade players' learn the SPS model.

REFFERENCE

1. Ellison,C.M.& Boykin, A.W. :" Comprising outcomes from differential cooperative and individualistic learning methods.", Social Behavior and Personality,22,pp.91-104, 1994
2. Mulryan,C.M. :"Student passivity during cooperative small groups in mathematics.", Journal of Educational Research, 85, pp.261-273,1992

Incremental generation of possible moves in Shogi

Tsuyoshi SUZUKI, Nobuo INUI and Yoshiyuki KOTANI,
Tokyo University of Agriculture and Technology
2-24-16 Nakamachi, Koganei, Tokyo, 184-8588 JAPAN
go@fairy.ei.tuat.ac.jp, nobu@fairy.ei.tuat.ac.jp, kotani@cc.tuat.ac.jp

ABSTRACT

In othello and chess programs, the cost of generating possible moves is not a matter of importance because possible moves are not many on each position. Since there are many more possible moves in Shogi than the ones in chess, it is necessary to reduce the cost of generating possible moves. It is advantageous to search more nodes within the limited amount of time to reduce the cost of the generation. Presently, in Shogi programs, possible moves have been generated without using any previous data of move generation.

This paper proposes a new method to reduce the cost of the generation by using the data of moves in previous position. Firstly, we copy the previous data to the current data, and then we compute the difference between the previous position and the next position.

We examined this method in a Shogi program. Compared to a benchmark program that used a conventional move generation algorithm, our incremental technique produced a speed improvement of about 28%.

1 Introduction

The game of Shogi is a two-person complete information game like chess. Shogi is more complicated than chess. The reasons are as follows:

(1) 9 x 9 board
(2) dropped pieces can be reused
(3) there are many possible moves (max:600, average:80)
(4) each game lasts about 120 moves

A position is hard to evaluate, and a good evaluation function requires a great deal of knowledge. In spite of the effort in developing Shogi programs, the level of the best programs still remains low on the human scale.

An important guideline in developing a program is time constraints. This is crucial when there are many possible moves. The aim of this paper is to show the incremental generation of possible moves.

In section 2, we show the reason of using incremental generation. Presently, in Shogi programs, possible moves have been generated without using the previous position data. In our method, possible moves are generated by using the previous position data. We show the main idea of the incremental generation of possible moves in section 3. We also show the results of the experiment in section 4, and the conclusion is given in section 5.

2 Why is it necessary to generate incrementally?

Considering the average length of move sequence in actual games L and average branching factor B, there are $C=B^L$ states in games. In chess B=40, L=80 and $C=10^{128}$, but in Shogi B=80 L=120 and $C=10^{228}$, which shows complexity of games, and difficulty of developing the program.

Since there are many more possible moves in Shogi than the ones in chess, it is necessary to reduce the cost of generating possible moves. It is advantageous to search more nodes within the limited amount of time to reduce the cost of the generation.

Making observation of human Shogi players is very helpful in order to develop a Shogi program. For example, human players do not re-calculate each situation of the board after each move. In fact, he may calculate only situations that are affected by the last move. This remark was a motivation to find a mechanism about the incremental generation of possible moves.

The incremental generation of possible moves computes the difference between the previous position and the next position. Presently, in Shogi programs, possible moves have been generated each position fully. In Shogi, the cost of possible moves is not low. Since there is a little difference between the previous position and the next position, there is also a little difference between possible moves in the previous position and the next position.

Because the incremental generation of possible moves computes the difference between the previous position and the next position, it requires to compute less.

3 How to generate incrementally

In this section, we describe the method of the incremental generation of possible moves.

3.1 Previous method

Presently, in Shogi programs, possible moves have been generated without using the previous position data (we call this type-1).

Figure 1(1),(2) shows that 金 moves from 2三 to 3二, and Figure 2(1),(2) shows possible moves in Figure 1(1),(2) respectively.

There are eight moves in Figure 1(1). After 3二金 moves, there are nine moves in Figure 1(2). The previous method generating possible moves in Figure 1(1), and generate possible moves in Figure 1(2) without using the Figure 1(1) data after last move.

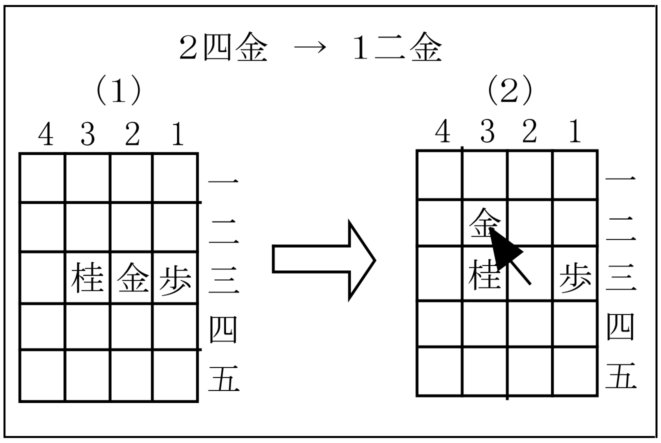


Figure 1: Example of change of possible moves

However, when we compare the list of possible moves in Figure 1(1) and the list of possible moves Figure 1(2), we find that:

(1) position is similar

(2) possible moves are similar

(3) move-ordering is similar

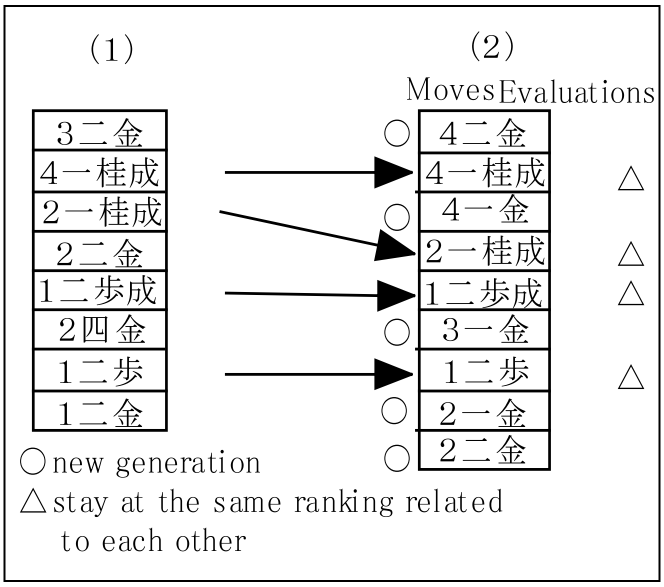


Figure 2: Possible moves in Figure 1

Hence, (3) helps alpha-beta search efficiently.

There is few difference between the previous position and the next position. Should we use the previous data?

3.2 Incremental generation of possible moves

As we point out in 3.1, there is a little difference of position, possible moves and move-ordering between the previous position and the next position. Average of possible moves in Shogi is about eighty, while last move changes a few parts of possible moves. Therefore, idea of the incremental generation of possible moves is that possible moves are generated by utilizing the previous position data.

The incremental generation of possible moves uses Black's possible-move-list and White's possible-move-list. These lists are updates after each move.

The incremental generation of possible moves is following (we call this type-2):
(1) possible moves in next position from the list of previous possible moves are selected
(2) moves from (1) are copied to the list of next possible moves
(3) the list of the new possible moves after the last moves is generated
(4) moves from (3) are added to the list of the next possible moves
(5) the list of the next possible moves is sorted out according to their evaluation

Figure 3 shows this operation in Figure 1. Firstly we select the possible moves in the list of the previous possible moves by the last one in the next position (Fig.3(1)) and their move are copied to the list of the next possible moves (Fig.3(2)). Then we generate the new possible moves after the last one, and their moves are added to list of the possible moves in next position. Finally we sort out the list of the next possible moves according to their evaluation.

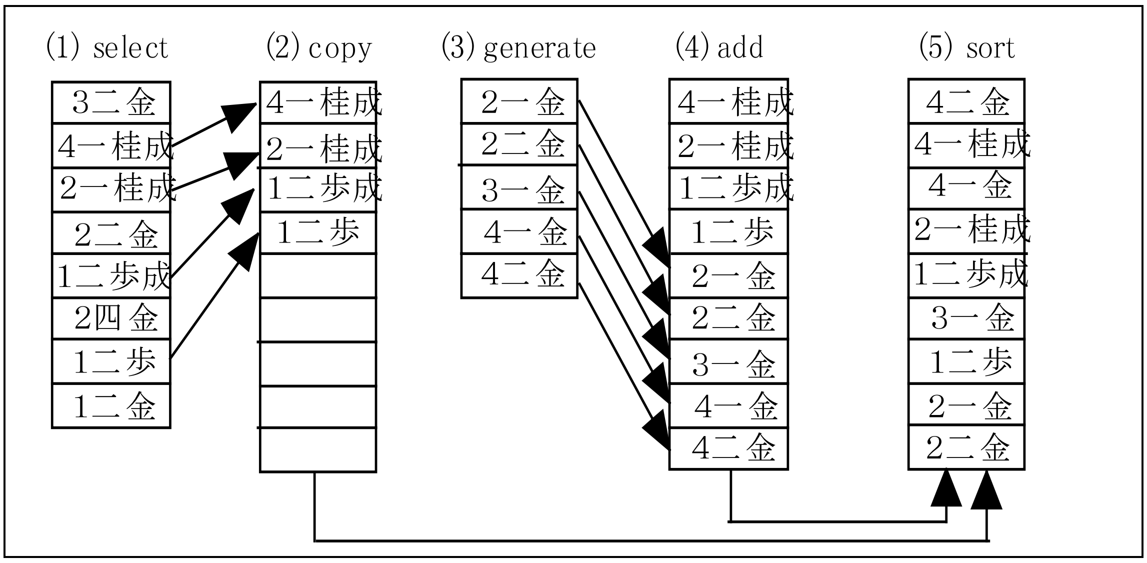


Figure 3: type-2 technique

3.3 Improvement on incremental generation of possible moves

We remark that the copied possible moves in the list of the next position are kept in their ranking. This helps sorting for the list of possible moves when to utilize alpha-beta game tree search. In general, the moves related in last moves obtain a high evaluation. By this fact, we get a following simple improvement (we call this type-3):
(1) possible moves in next position from the list of previous possible moves are selected
(2) moves from (1) is copied to the list of next possible moves
(3) the list of the new possible moves after the last moves is generated
(4) moves form (3) are added to top of the list of the next possible moves
(5) the list of the next possible moves is sorted out according to their evaluation

In Figure 3 the new generation moves added to last of the list of the possible moves. However in Figure 4 the new generation moves added to top of the list of the possible moves. By this means the sorting of the list of possible moves has more efficiency.

Most Shogi program searches top of the list of possible move by uses pre-pruning. Therefore we should sort the list of possible moves in the part of top of the list, which is for the next improvement.

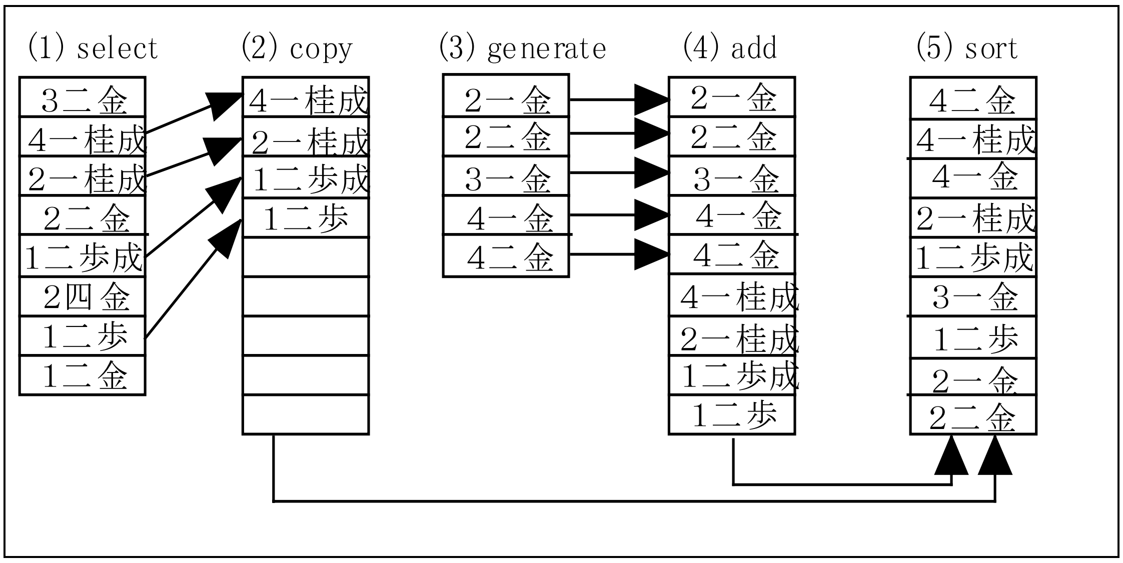


Figure 4: type-3 technique

We also propose that we first sort the new generated moves according to its evaluation, next we merge the new generated moves and the previous possible moves by the merge sort (we call this type-4).

(1) possible moves in next position from the list of previous possible moves are selected

(2) the list of the new possible moves after the last moves is generated

(3) the list of the new generation moves is sorted

(4) the new generated moves and the previous possible moves are merged

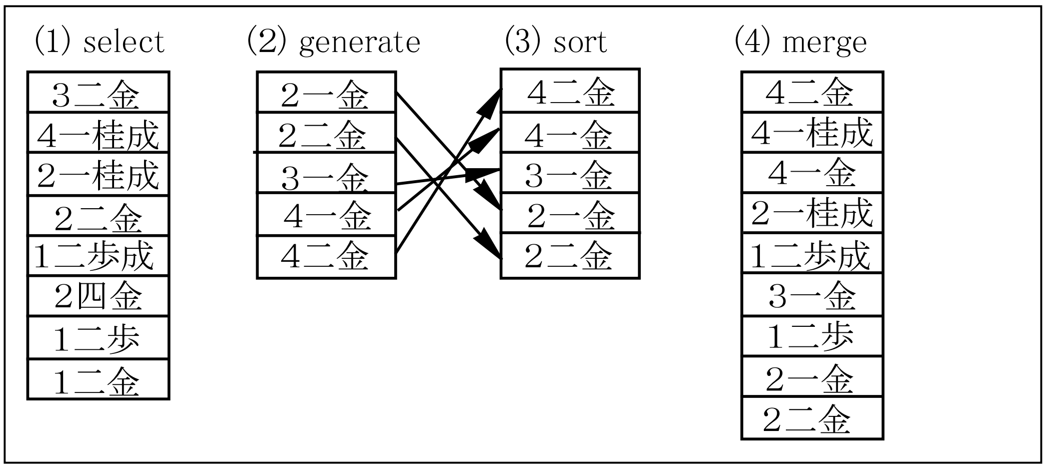


Figure 5: type-4 technique

Figure 5 shows this method in Figure 1. Moreover, we can also sort top of the list.

4 Results

We examined the type-1, type-2, type-3 and type-4 techniques by implementing them in a Shogi program. To allow our tests to evaluate solely the differences due to each move generation technique, we used an evaluation function that simply returned random numbers. We examined the experiments 50 games (about 5000 positions). Average length of move sequence in games is 97, average branching factor is 82 and 3-7 depths. The evaluation function consists of the value of pieces.

Figure 6 shows the result. In We define a unit as number of searched positions by type-1 program in a second (about 45000 nodes/sec).

Type-2, type-3 and type-4 generated the possible moves efficiently. There is little difference among type-2, type-3 and type-4. This means that sorting time of the possible moves is not influenced on search time.

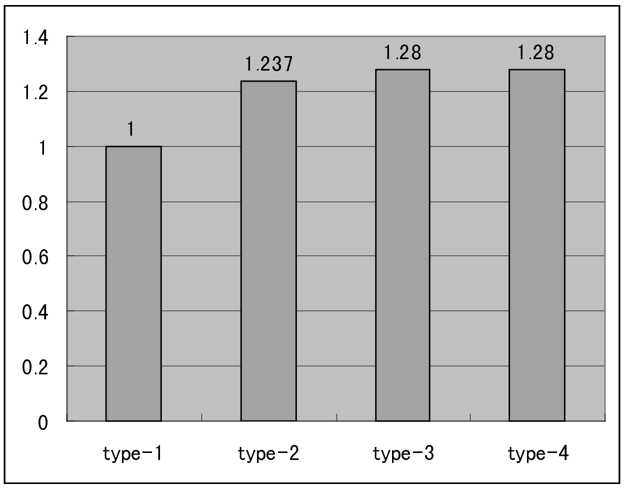


Figure 6: Result

Evaluation function requires the most time. Our method is no longer efficient for Shogi programs which search a few positions in a second, because our method is less used in game tree searching, but efficient for Shogi program which searches more 10000 position in a second. As shown in the figure, the number of the search positions in type-3 and type-4 increased about 28% compared to type-1.

5 Conclusion

This paper proposed the incremental generation of possible moves, and showed the result of the experiment. The incremental generation of possible moves is

(1) to utilize the previous position,

(2) to compute the difference between the previous position and the next position.

As the result of experiment, we can show that our method reduces the cost of move generation. Though we do not research the detail, our method is more effective for positions that have a lot of possible moves.

Our program that is used for the experiment is still under development, we will examine the experiment which use the program with pre-pruning. We conjecture that our method is still efficient for such a program.

Reference

[1] Y.Kotani et al.: Computer Shogi (in Japanese), Science, Japan(1990)

[2] D.Levy and M.Newborn: How computer plays chess, Freeman and Company, New York (1990).

[3] H.Iida et al.: A Report on SHOUCHAN Project (in Japanese), Tokyo Univ. of A&T. (1994)

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

Ayumu Nagai
Department of Information Science,
University of Tokyo

Abstract

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

1 Introduction

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

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

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

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

2 Proof number and disproof number

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

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

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

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

The concrete method to calculate the proof and disproof number is as follows. (In the following, *n.pn* and *n.dn* stands for the proof number and the disproof number at node *n* respectively)

1. For a terminal node *n*
   (a) When game-theoretical value is already known
      i. When game-theoretically **proof**

$$\begin{aligned} n.pn &= 0 \\ n.dn &= \infty \end{aligned}$$

      ii. When game-theoretically **disproof**

$$\begin{aligned} n.pn &= \infty \\ n.dn &= 0 \end{aligned}$$

   (b) When game-theoretical value is not known yet

$$\begin{aligned} n.pn &= 1 \\ n.dn &= 1 \end{aligned}$$

2. For an internal node *n*
   (a) For an OR node

$$\begin{aligned} n.pn &= \operatorname*{Min}_{n_i \in \text{children of } n} n_i.pn \\ n.dn &= \sum_{n_i \in \text{children of } n} n_i.dn \end{aligned}$$

   (b) For an AND node

$$\begin{aligned} n.pn &= \sum_{n_i \in \text{children of } n} n_i.pn \\ n.dn &= \operatorname*{Min}_{n_i \in \text{children of } n} n_i.dn \end{aligned}$$

3 Classification of AND/OR tree search algorithms

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

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

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

Table 1: Classification of AND/OR tree search algorithms

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

3.1 Allis's Algorithm

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

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

3.2 Elkan's Algorithm

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

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

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

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

3.3 Seo's Algorithm

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

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

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

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

3.4 The Necessity of a new Algorithm

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

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

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

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

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

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

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

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

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

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

5 Performance measures on random trees

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

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

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

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

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

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

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

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

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

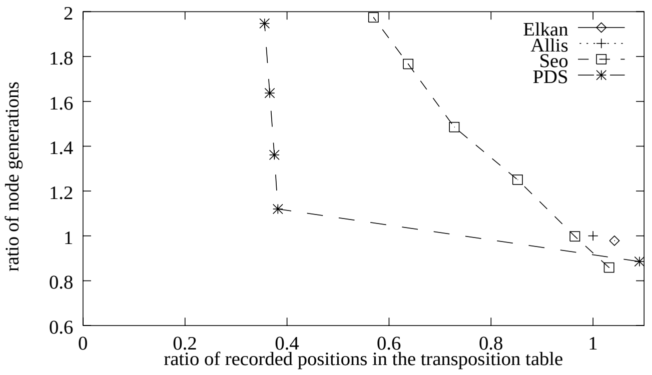


Figure 1: table-recorded positions and node expansions with proof solutions

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

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

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

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

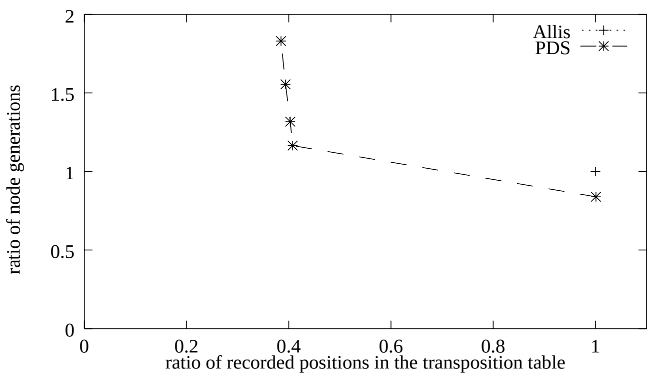


Figure 2: table-recorded positions and node expansions with disproof solutions

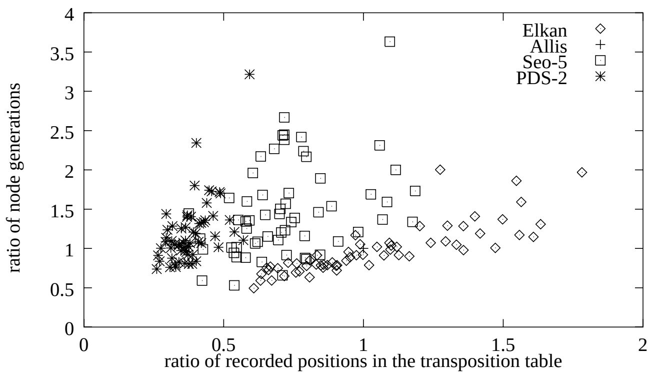


Figure 3: table-recorded positions and node expansions with proof solutions

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

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

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

6 Conclusion

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

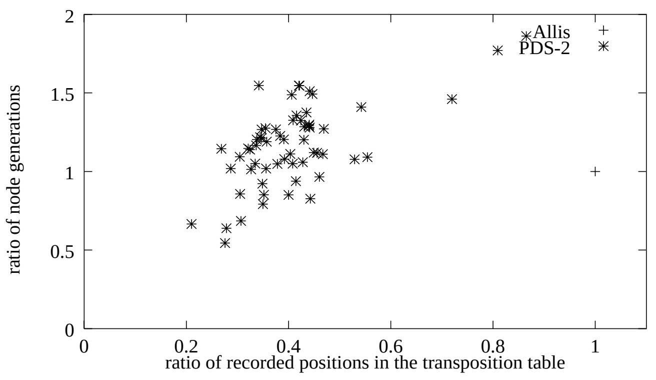


Figure 4: table-recorded positions and node expansions with disproof solutions

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

References

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

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

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

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

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

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

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

[Seo, 1995] Masahiro Seo. *The C* Algorithm for AND/OR Tree Search and its Application to a Tsume-Shogi Program.* M.Sc. Thesis, Department of Information Science, University of Tokyo, Japan, 1995.

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