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.

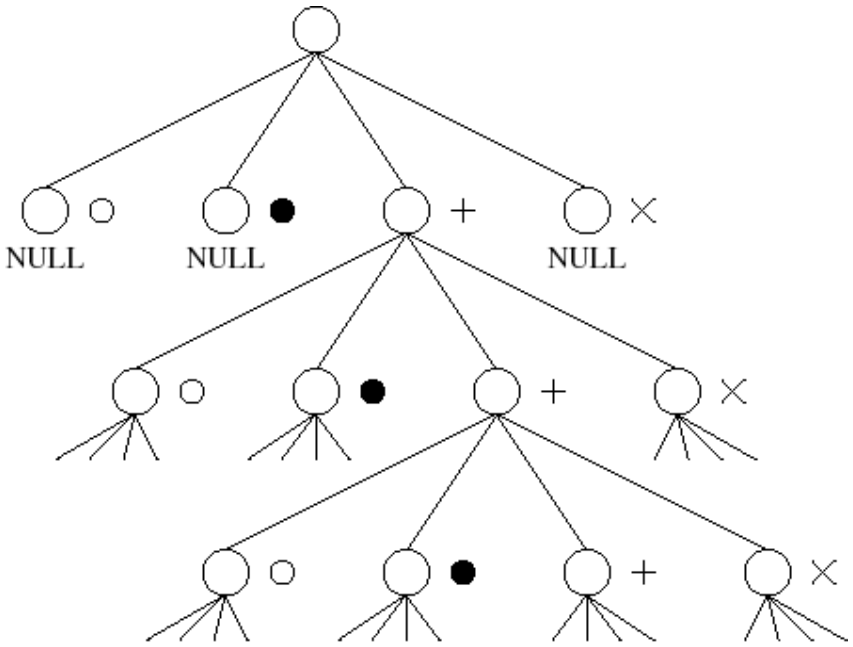


Fig. 1: A Decision Tree

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.

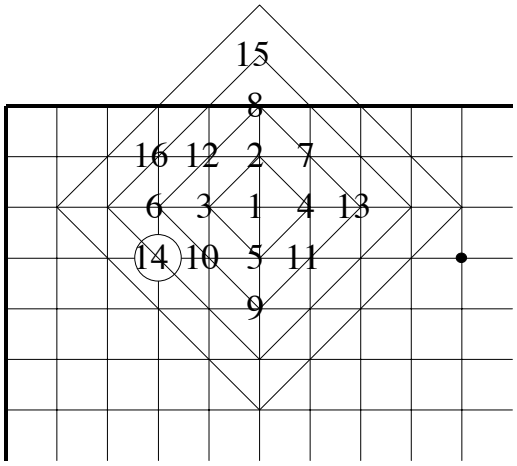
References

[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.

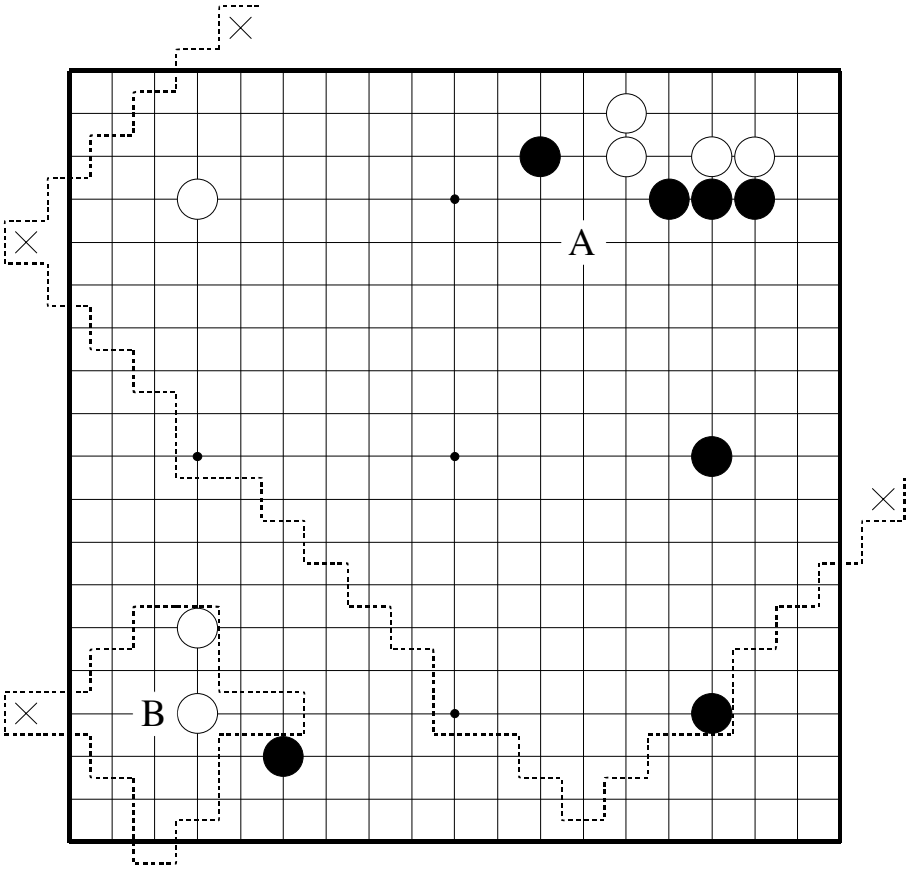


Fig. 3
Examples of leaf nodes; a pattern around position A is classified to a leaf node #536959 and B to #314275. Areas closed by dotted lines indicate areas to watch in each leaf nodes.

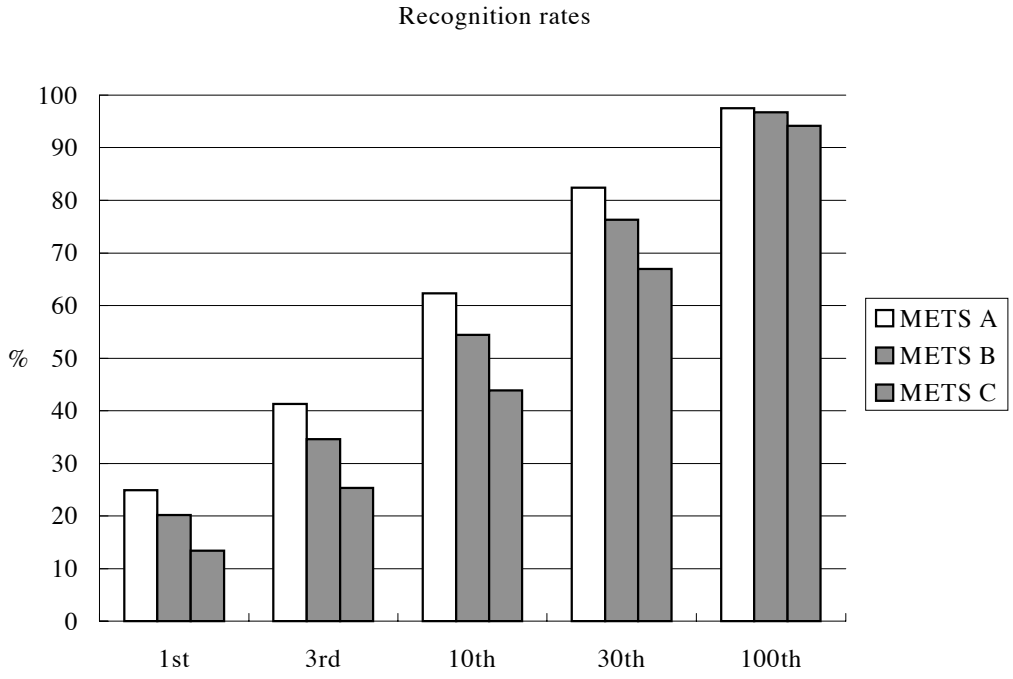


Fig. 4
In the graph, a recognition rate of *n*-th means a ratio in which move positions are within *n*-th candidates

	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