Knowledge Acquisition from Game Records

Takuya Kojima **Atsushi Yoshikawa**
NTT Communication Science Labs.
3-1 Morinosato Wakamiya, Atsugi, 243-0198 JAPAN
kojima@rudolph.brl.ntt.co.jp

Abstract

A large amount of knowledge is considered very useful for systems playing such games as Go and Shogi (Japanese chess) which has a much larger search space than chess. Knowledge acquisition is therefore necessary for systems playing these games. This paper explains two approaches to acquire Go knowledge from game records: a deductive approach and an evolutionary one. The former is taken to acquire strict knowledge; several rules are acquired from a single training example. The latter is taken to acquire a large amount of heuristic knowledge from a large amount of training examples. Tsume-Go problems are solved in order to compare the performance of the algorithm with others.

1 INTRODUCTION

1.1 PURPOSE

Game studies have mainly focused on search techniques, and these techniques enable systems playing most games — such as chess, checkers, and Othello — to perform as well as human experts. It is very difficult, however, for systems implemented using only search techniques to perform well when playing such games as Go and Shogi (Japanese chess) for which the search space is much larger than that for chess. The implementation of systems that play Go at an expert level is an especially hard problem. Although knowledge similar to the knowledge human experts use is expected to be useful, no systems having such knowledge exist.

The knowledge of human experts has two important features: quality and quantity. As for quality, it is *flexible*: human experts have many types of knowledge, such as verbal and visual, and there is little limitation in size and shape. As for quantity, it can be of *large amount*: a chess master is estimated to have about 50,000 chunks of knowledge of chess [4, 12].

Most computer systems which play games nowadays have much less knowledge than human experts. "Many Faces of Go" [2], for example, which has more knowledge than most Go systems, has about a thousand patterns [3], which the programmer manages to input, while human experts are said to have much more knowledge. The most effective way for systems to have almost the same amount of knowledge as human experts may be for themselves to acquire knowledge automatically.

Although some systems have tried to acquire Go knowledge, most of them acquire only fixed-shaped knowledge (square or diamond shape), which is quite different from what human experts have. A new algorithm which yields more flexible knowledge is therefore necessary.

1.2 CLASSIFICATION OF GO KNOWLEDGE

We classify Go knowledge according to two criteria: form and degree of validity. First we classify it into three types according to form: patterns, sequences of moves, and maxims. A *pattern* is a rule whose condition part is a part of the board configuration and whose action part is a move, a *sequence of moves* is a rule suggesting several moves, and a *maxim* is a verbal rule consisting of Go terms. We then classify Go knowledge into two types according to its degree of validity: strict knowledge and heuristic knowledge. *Strict* knowledge is the knowledge that can always be applied and whose validity can be proven rather easily. For example, in *tsume-go*, or life and death problems, why and how stones are killed can be shown. *Heuristic* knowledge, on the other hand, is the knowledge whose effectiveness is hard to show and whose validity depends on situations. For example the maxim, "When

the opponent's move is *tsuke*, then play *hane*," is not always true and sometimes it is better to play *nobi* when the opponent's move is *tsuke*.

1.3 TWO APPROACHES

In this paper, acquisition of pattern knowledge is focused. Two approaches to acquire Go knowledge are introduced in this paper: a deductive approach and an evolutionary approach.

A deductive approach is taken to acquire strict knowledge. By observing the process that human Go novices and intermediates learn some Go knowledge, it is observed that they learn it in a certain deductive way. The most major approach of AI for deductive learning is Explanation-Based Learning (EBL) [1] or Explanation-Based Generalization (EBG) [11]. EBG is applied to make a model of acquisition of strict knowledge. In this approach, several rules are acquired from a single training example.

We chose an evolutionary approach of AI to acquire a large amount of heuristic knowledge from a large amount of training examples. Exhaustive search, which searches the whole search space, is impossible for such a large search space as Go. On the other hand evolutionary approaches are said to be able to acquire approximate solutions from a large search space without searching the whole search space.

Ordinary evolutionary algorithms, however, have two weaknesses to acquire a large amount of flexible knowledge. One is in quantity: they tend to get similar rules, which are optimal, thus they acquire few types of rules. The other is in quality: they tend to get fixed form of knowledge, such as fixed-length binary strings in ordinary Genetic Algorithms. In this new algorithm, activation values to individuals and a new variation-making operator, splitting, are introduced to overcome these two weaknesses.

2 RULES OF GO

Some rules of Go are explained, which are used in this paper. Go is a board game between two players, Black and White. The players take turns, and on each turn each player puts a stone of his/her color(Black or White) on a board. Stones are placed on intersections of vertical and horizontal lines on the board, including edges and corners. The board used in Go is usually 19 $\times$ 19, and smaller boards such as 9 $\times$ 9, 11 $\times$ 11 or 13 $\times$ 13 are used, usually by beginners in order to reduce the difficulties.

Stones of the same color adjacent to each other horizontally and vertically form a **string**. A vacant intersection adjacent to a stone or a string is called a **liberty**. A stone or a string must have at least one liberty. When a stone or a string has no liberty, it is removed from the board, called **captured**. When a stone or a string has only one liberty, it is called in **atari**.

Take Figure 1 as an example. In the right figure, stones marked with the same letter **a**,**b** and **c** belong to the same string. Liberties of a stone or a string are marked by **L**. In Figure 2, all White stones are in atari and the places marked **a** are the last liberties for each stone or string.

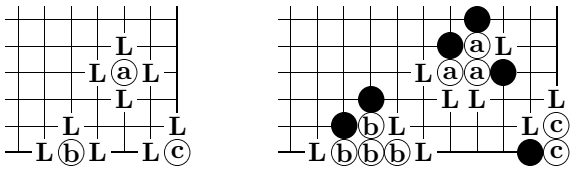

Figure 1: Examples of liberties. **L** stands for liberties of White stones.

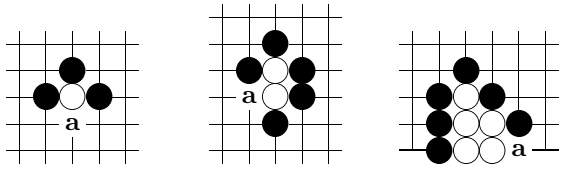

Figure 2: Examples of atari. All the White stones are in atari and **a** is the last liberties.

A player can pass at any time, and a game is over when both players continuously pass. The object of the game is to surround territory and capture opponent's stones. An intersection a player surrounds and a stone a player captures count as a point, and the player with the more points is the winner.

3 A DEDUCTIVE APPROACH

3.1 SYSTEM

This system tries to acquire Go knowledge deductively after playing Go against someone or being given Go problems and their answers. When the system loses against someone or detects a move which the system does not know, it learns some rules. When it realizes that some rules are inaccurate, it refines the rules. This system is implemented in Prolog.

The target here is to acquire knowledge to capture opponent's stones. The goal of the game here is to capture opponent's stones. The player who captures opponent's stones earlier wins.

3.1.1 Overview of This System

Figure 3 is an overview of this system. This system consists of four modules: Knowledge Base (KB), Decision Maker, Rule Acquisition Module, and Rule Refinement Module.

The system plays Go against someone. During a game Decision Maker searches appropriate rules in KB, and applies them if it finds them; otherwise it puts stones on a vacant place randomly. After the game, if the system loses, Rule Acquisition Module acquires some rules and adds them to KB. When the system detect inaccurate rules, Rule Refinement Module refines them. In the refinement, it extracts the inaccurate rules from KB and refines them and adds new refined rules to KB.

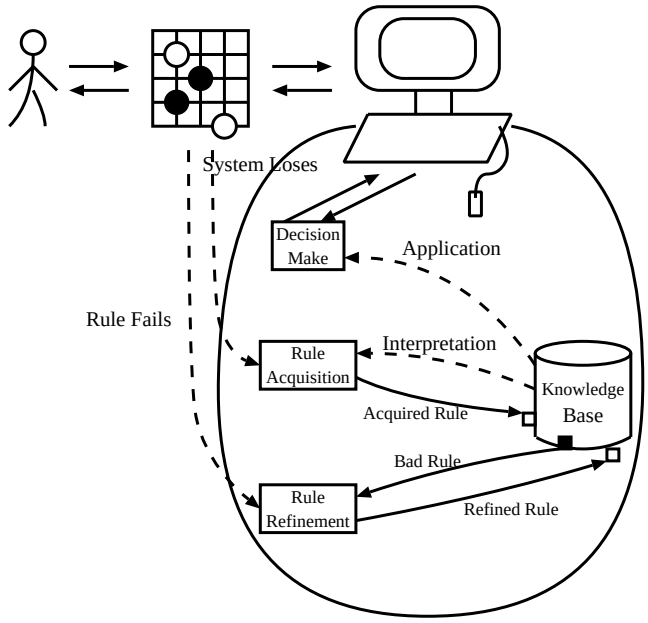

Figure 3: Overview of this system.

3.1.2 Terminology

Some terms used in this paper are introduced here.

last move M is the number of moves in a game (the last move number).

color In Go, there are two colors, Black(B) and White(W). $C = \{B, W\}$.

board size A board size BS is the number of horizontal and vertical lines on the board. It is a constant through a game. In ordinary Go a board is 19×19, $BS = 19$. Sometimes it is set to 9, 11 or 13.

board configuration Board configurations consist of stones, vacant places, and edges of the board ($BC \subseteq (S \cup V \cup E)$). BC_m represents a board configuration after Move m ($BC_m = \{s_0..s_m, v_{x_1,y_1}..v_{x_i,y_i}, e_{x_1,y_1}..e_{x_j,y_j}\}$). BC_0 is the initial board configuration.

- **stone** S is a set of stones and S_m ($\subseteq S$) is a set of those placed from the initial position to Move m. S_0 is a set of stones which are placed in the initial position. $S_m = S_0 \cup \{s_i | 1 \leq i \leq m\}$, where s_i is a stone which is put on Move i. A stone s_i is a 2-tuple: $s_i = \langle [x_i, y_i], c_i \rangle$ ($1 \leq x_i \leq BS$, $1 \leq y_i \leq BS$, $c_i \in C$), where $[x_i, y_i]$ is a coordinate of the place on which the stone is put and c_i is a color of the stone.
- **vacant** V is a set of vacant places and $V_m (\subseteq V)$ is that after Move m. $V_m = \{v_{x,y} | 1 \leq x \leq BS,\ 1 \leq y \leq BS,\ s(= \langle [x, y], c \rangle) \notin S_m, c \in C\}$, where $v_{x,y}$ represents a vacant place at [x,y].
- **edge** E is a set of edges. $E = \{e_{x,y} | (0 \leq x \leq BS + 1) \wedge (0 \leq y \leq BS + 1) \wedge (x = 0 \vee x = BS + 1 \vee y = 0 \vee y = BS + 1)\}$, where $e_{x,y}$ represents an edge at [x,y].

move δ is a function which represents a move. $\delta(BC_{m-1}, s_m) = BC_m$, for example, represents that in a board configuration after Move $m - 1$ (BC_{m-1}), Move m yields BC_m by putting Stone s_m.

don't care "*" mark represents 'don't care'.

3.1.3 Knowledge Base: Rules

Knowledge Base has two kinds of rules: *basic rules* and *forcing rules*. A basic rule is a definition of a basic term of Go, such as 'capture' and '*atari*'. Forcing rules are defined as follows: a player always wins if he/she applies forcing rules. They are represented as follows:

$rule_n^{for}$: **if** $cond_1 \wedge .. \wedge cond_m$ **then** $\delta(BC_*, s)(s = \langle [x, y], c \rangle)$,

where $cond_i \in (BC \cup BR)$, and n is the rule number.

An example of forcing rules is shown below:

if r_color(t)=$t' \wedge \ \ s_k = \langle [*, *], t' \rangle \wedge \text{atari}_{s_k}^{x,y}$ **then** $\delta(BC, s_m)$ $(s_m = \langle [x, y], t \rangle)$ (f1)

which means as follows: **if** opponent's stones are in atari, **then** put a stone on their last liberty.

In the initial state, the only given forcing rule to the system is Rule f1 shown above. Other forcing rules are acquired by Rule Acquisition Module.

There are two ways to apply forcing rules: one is to apply them for attack and the other to apply for de-

fense. When a player applies a forcing rule for the sake of his/her own attack, it is called an *application for attack*. When he/she recognizes that his/her opponent can apply a forcing rule in the next turn, a player may make a move to escape opponent's application for attack; that is called an *application for defense*. Take Figure 2 for example. In the figure, all the White stones are in atari and all the places marked **a** in the figure are the last liberty of White stones. Thus, when Black puts a stone on **a**, the move is an application of Rule f1 for attack; when White puts a stone on **a**, the move is an application of Rule f1 for defense.

3.1.4 Decision Maker

Decision Maker applies forcing rules. If there is no forcing rules to be applied, a stone is put randomly in a vacant place. Decision Maker makes a move according to the following algorithm:

Algorithm 1 Decision Making
1 **if** able to apply Rule f1 for attack
  then apply Rule f1 for attack
2 **if** opponent is able to apply Rule f1 for attack
  then apply Rule f1 for defense
3 **if** able to apply a forcing rule other than Rule f1
  then apply the rule for attack
4 **if** opponent is able to apply a forcing rule other than Rule f1
  then apply the rule for defense
5 Choose a place to put a stone randomly in a vacant place.

3.2 RULE ACQUISITION

3.2.1 Terminology

Some terminologies used below are introduced here.

interpretation i_m is an interpretation of Move m and it consists of a list `[`m`,`r`,`ad`,`wl`]` where:

- m**:** represents that this interpretation is that of Move m.
- r**:** denotes the kind and the number of a rule of which the move is interpreted as application of a rule. It is usually '`f`'(forcing rule)+i(the number of a forcing rule), such as "`f3`".
- ad**:** $ad \in \{$`a`,`d`$\}$, where `a` represents application for attack and `d` that for defense.
- wl**:** $wl \in \{$`w`,`l`$\}$, where `w` represents that the move is interpreted as application of the rule by a winner and `l` by a loser.

Interpretation Sequence IS is a list of interpretations of moves.

$$IS = [i_M, i_{M-1}, \ldots, i_m](1 \leq m \leq M).$$

Note that interpretations are lined in reverse order of the moves in the game.

relevant parts RP ($\subseteq BC$) is a set of relevant parts of a board configuration. RP_m is a set of parts relevant to Move m, and it is a part of a board configuration after Move m($RP_m \subseteq BC_m$).

3.2.2 Rule Acquisition Algorithm

Rule Acquisition Algorithm is shown below:

Algorithm 2 Rule Acquisition
$m \leftarrow M$, $IS \leftarrow \phi$, $RP \leftarrow \phi$, j =the number of forcing rules
Loop 1
  Loop 2
1 $i_m \leftarrow$ Interpret($\delta(BC_{m-1}, s_m) = BC_m$)
2 **if** i_m=ϕ **then exit Loop 2**
3 **if** i_m=`[`m`,f*,a,l]` **then Refinement**
4 $IS \leftarrow$ Append($IS, [i_m]$)
5 $RP_m \leftarrow$ Explain(i_m)
6 $RP \leftarrow RP \cup RP_m - \{s_m\}$
7 $m \leftarrow m - 1$
  end Loop 2
8 **if** $i_{m+1} \neq$ `[`$m+1$`,f*,d,l]` **then exit Loop 1**
9 $RP \leftarrow RP \cup \{v_{x,y}(\in V)\} - \{s_m(= \langle [x, y], c \rangle)\}$
10 pre-rule $\leftarrow \langle RP \subseteq BC_i \rightarrow s_m \rangle$
11 $rule^{for}_{j+1} \leftarrow$ Generalize(pre-rule)
12 add $rule^{for}_{j+1}$ to KB
end Loop 1

The proposed algorithm consists of three stages. The first stage (Step 1 to 7) chooses good moves to be learned from a sequence of moves in a game by interpreting moves using forcing rules. The second stage (Step 5) extracts relevant parts from the board configuration where the learning moves are made by using EBG on each move. The last stage (Step 9 to 11) generalizes the position and the move by changing their coordinates from constants to variables. See Figure 4.

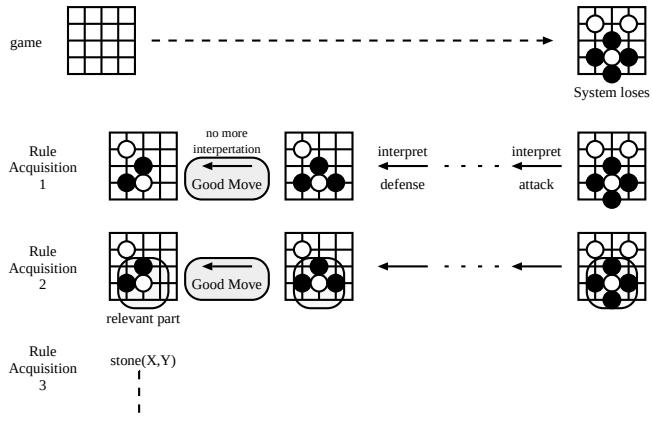

Figure 4: Overview of Rule Acquisition algorithm.

Stage 1: Finding a move to be learned

In the first stage (from Step 1 to 7 = Loop 2), the move to be learned by the system is detected. It investigates moves from the last move to the initial one, move by move. The system interprets moves of both players; it tries to understand moves as the application of a forcing rule. Take Figure 5 as an example. Move 5 is interpreted as the application of Rule f1 for attack by Black (f1: attack), since before Move 5 all White stones are in atari and Black to move. Move 4 is interpreted as the application of Rule f1 for defense by White (f1: defense). Move 3, however, cannot be interpreted by the forcing rule the system has. The move which cannot be interpreted by the forcing rules of the system is the move to be learned (Step 2).

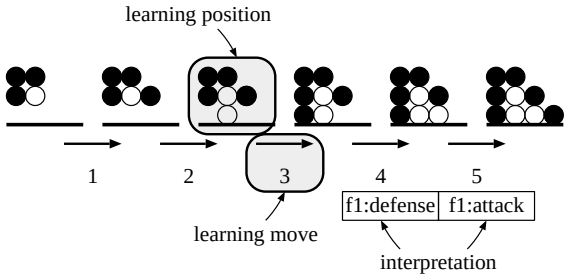

Figure 5: Finding a move to be learned.

A reason why the move should be learned is briefly explained (detailed explanation is in [8]). Since the system is beaten, the last move (Move 5 in Figure 5) is interpreted as attack by a winner and the move before the last (Move 4) is interpreted as defense by the system. Other moves are also interpreted alternately – attack by a winner and defense by the system. The last move which the system can interpret (the last interpretable move, LIM, Move 4) is a defense by the system. Therefore, the move just before LIM (Move 3) was actually an attack by a winner and the move cannot be pretended. Thus, the move just before LIM is a forcing rule to be learned.

Stage 2: Extracting the essential part

The essential part to capturing stones is extracted from the whole board in order for the rule to be used in future. We utilize EBG, which can extract an essential part by building an explanation tree which explains why a given example is a positive example (Step 5).

Take Figure 6 for example. EBG explains the reason why the system interprets Move 5 as the application of Rule f1 for attack by the winner. The reason for the interpretation is that three White stones are in atari. Why the White stones are in atari? Because a White stone is on [5,1], [6,0] is an edge and there is no stone

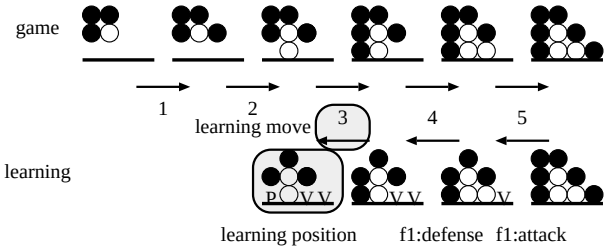

Figure 6: A process of extracting relevant parts.

on [7,1] etc. (Figure 7).

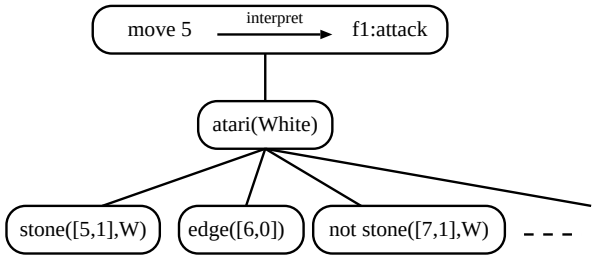

Figure 7: An explanation tree for Move 5.

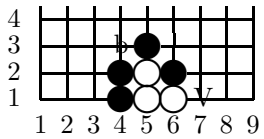

Figure 8: Explanation of Move 5.

Thus the output of EBG is as Figure 8, where the place denoted **b** is vacant because the Black stone on place **b** does not help Black to make White in atari. At the same time, the place where Move 5 was put is marked as **V**, which means that the place should be vacant to apply the rule (Step 9).

The same procedure is done to the fourth move. Since the third move is the move to be learned, the move is replaced by **P**, which means that this place is vacant and you should play here (Step 10).

The coordinate of the rule is generalized (Step 11) and the rule (f2) is added to Knowledge Base (Step 12). As a consequence, third move is interpreted as an application of Rule f2 for attack, and the same procedure is done again, and the other rule (f3) is also acquired (Figure 9). In this example, two rules are acquired from an example. Rule f3 is very famous pattern, which beginners learn, and it is a valid rule.

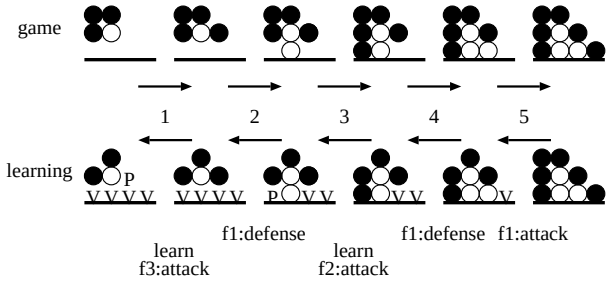

Figure 9: Acquisition of two rules from an example.

Other example is Geta. Geta is a famous move, a net trap, which is learned by novices (Figure 10). Although this is not an immediate threat and more difficult than the previous example, this algorithm acquires a proper rule as shown in Figure 11.

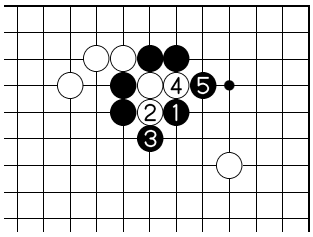

Figure 10: A problem and its answer of Geta.

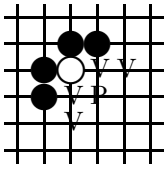

Figure 11: An acquired rule of Geta.

This system automatically detects inaccurate rules and refines them, which is not explained in this paper. See [8].

4 AN EVOLUTIONARY APPROACH

This approach is taken to acquire a large amount of heuristic knowledge from a large amount of training examples.

4.1 ALGORITHM

In this section, the proposed algorithms are explained. See [8, 9, 10] for details.

4.1.1 Overview of the Algorithm

The algorithm's aim is to acquire useful rules, herein *individuals*, which match the given training data, herein *food*. Each rule takes the form of a production rule, consisting of an IF-part (consisting of conditions) and a THEN-part (consisting of an action), and has an *activation value*. There are no rules in the initial state. Rules matching a training datum get food, and their activation values increase. If there are no rules matching a given datum, a new rule matching the datum with only one condition is created. The number of rules with only one condition thus increases at an early stage.

After being fed the rules with activation value over a threshold are *split* into two rules: the original one and a more complex one. More complex rules are thus created by splitting, and the number of rules increases.

Every rule consumes food at each step, and decreases one activation value. Rules whose activation value is 0 *die*. Thus, rules that are too complex and get food too rarely, die. As a result, all the rules are expected to get food at almost the same frequency. The procedure of this algorithm is shown below.

Algorithm 3 Pattern Acquisition Algorithm

```
1   step ← 1
    while step ≤ the number of iterations
2       choose a random game record from the game database
3       move ← 1
        while move ≤ the number of moves in the game
4           a training datum ← move-th move
5           if no rule matches the datum
                then create a new rule
                else feed matched rules
            for all rules
6               if activation of a fed rule > threshold
                    then split the rule
7               activation of a rule ← activation of a rule − 1
8               if activation of a rule = 0
                    then the rule dies
            end for
9           move ← move + 1
10          step ← step +1
        end while
    end while
```

4.1.2 Details of the Algorithm

Rules

Each rule takes the form of a production rule consisting of an IF-part and a THEN-part.

A rule is described in relative terms so that it can be executed by either player. Eight equal board configurations (4 rotations (90 degrees) times 2 reflections) are converted to one of the configurations. Rules are described as follows:

IF exist($[x_1, y_1], obj_1$) $\wedge$ ...$\wedge$ exist($[x_n, y_n], obj_n$)
THEN play([0,0]),

where $[x_i, y_i]$ represents relative coordinates from the

action place [1], whose coordinate is always [0,0], and obj_i is one of the four objects shown below:

1. SAME: the same color stone as the action stone [2]
2. DIFF: a stone with different color from the action stone
3. EDGE: an edge of the board [3]
4. $-n$: a previous move [4] ($n \in N, n > 0$)

The following rule is an example: "**IF** exist([-1,-1], -1) $\wedge$ exist([0,-1], SAME) $\wedge$ exist([-2,-1], DIFF) $\wedge$ exist([0,-5], EDGE) **THEN** play([0,0])". This rule is shown in Figure 12. The active player is always shown as Black in this paper. The action stone and the move are presented as the stone with the largest number. This stone is "2" in this example. This rule means that **if** the previous move (shown as "1") is played to point [-1,-1] (relative to the action place) and a stone of the *same* color as the action stone (Black) exists at point [0,-1] and a stone of a *different* color from the action stone (White) exists at point [-2,-1] and an *edge* exists at point [0,-5], **then** put a stone on [0,0].

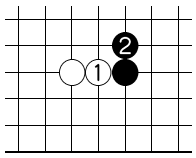

Figure 12: An example of rules: **IF** exist $([-1,-1],-1)$ $\wedge$ exist([0,-1],SAME) $\wedge$ exist([-2,-1],DIFF) $\wedge$ exist([0,-5],EDGE) **THEN** play([0,0])

Feeding Rules

Among matched rules, those that do not have rules more specific than themselves are *fed*. An activation value equal to Parameter $FOOD$ is equally shared among the fed rules. The following is an example.

Suppose that the following five rules matched a given training datum.

1. **IF** C_1 **THEN** A_1
2. **IF** $C_1 \wedge C_2$ **THEN** A_1
3. **IF** $C_2 \wedge C_3$ **THEN** A_1
4. **IF** $C_4 \wedge C_5$ **THEN** A_1
5. **IF** $C_2 \wedge C_3 \wedge C_4$ **THEN** A_1

[1]The place where a stone is placed according to the rule.

[2]The stone to be placed according to the rule.

[3]An edge is located just outside the board, not inside the board.

[4]"-3" represents the move played three moves before. This is used to acquire sequences of moves.

Rules 1 and 3 are not fed because they have more specific rules than themselves (Rules 2 and 5, respectively). The others (Rules 2, 4, and 5) share food, and each gets one third of the food.

As a consequence, the more general the rules are, the more frequently they are matched: however, because of this feeding method, the probability that they are fed falls although they are matched. As a result, too general rules are not acquired because they often match with food, but rarely get food as long as more specific rules exist.

Splitting and Creating Rules

A fed rule whose activation value is greater than a certain threshold value is split into two rules: one, called *parent*, is the original rule and the other, called *child*, is created by adding a new condition to the IF-part of the original rule. For example, rule, **IF** $C_1 \wedge C_2$ **THEN** A_1, is split into the original rule and a newly created rule, **IF** $C_1 \wedge C_2 \wedge C_3$ **THEN** A_1. The new condition, C_3, is chosen at random from among the objects on the current board (stones, edges of the board, and the previous moves).

A child extracts a certain amount of the activation value from its parent. As a result, after the split, the activation value of the rule changes as listed in Table 1. The total activation value, however, does not change after the split.

Table 1: Change of activation value after splitting.

	before	after
parent	P_ACT	P_ACT − INI_ACT
child		INI_ACT

If no rule is matched, a new rule is created which has only one predicate chosen randomly from among the objects on the current board. A newly created rule that is the same as a rule already in the rule set is deleted. There is thus no duplication of rules.

Examples of Acquired Rules

Figure 13 lists examples of acquired rules.

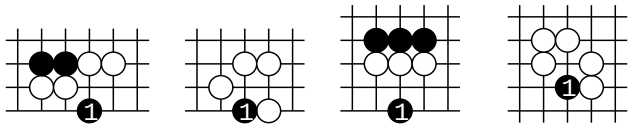

Figure 13: Examples of acquired rules of Tsume-Go.

4.2 APPLICATION TO TSUME-GO

Although the proposed algorithm can be applied to normal Go games, it is applied to Tsume-Go problems (life and death problems) in order to compare the proposed algorithm with other algorithms.

4.2.1 Other Algorithms

The compared algorithms are Fixed Algorithm and Semi-Fixed Algorithm. These algorithms are explained here. As compared to these algorithms, the algorithm proposed in Subsection 4.1 will be called Flexible Algorithm below.

Fixed Algorithm

Fixed Algorithm acquires fixed patterns. The shape of the acquired patterns is the same, but the size can change. A pattern includes all the squares within d Manhattan distance from the center and their adjacent 8 squares. The IF-part is all the squares except the center, and the THEN-part is the center square, so when a pattern is matched, the move is to the center square. Figure 14 shows the shapes of patterns. In the figure, the numbers indicate the Manhattan distance from the center square, 'a' means their adjacent squares, and '1' means the center square. A square is occupied by one of the following five objects: SAME, DIFF, EDGE, EMPTY(the square is not occupied by any stone), and OB(the square is out of the board). Note that empty squares are explicitly described in this algorithm. In this paper, d ranges from 0 to 3. The number of conditions according to d is as listed in Table 2.

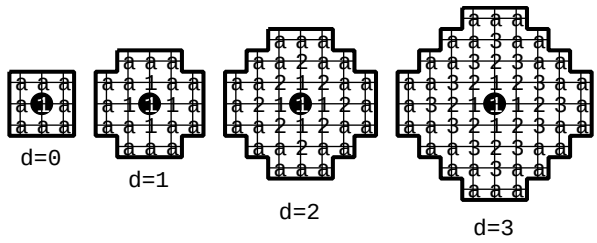

Figure 14: Shapes of patterns acquired by Fixed Algorithm.

Table 2: Then number of conditions in a rule.

d	0	1	2	3
The number of conditions	8	20	36	56

Patterns are acquired by a simple mechanism. When a training datum is given, a pattern for each d, whose center is the training move, is simply stored into a database. A total four patterns ($d = 0, 1, 2, 3$) are stored. Eight rotations are considered and only one of them is stored and the others will not be stored.

When the patterns are used, only the pattern with the largest d is used, which is considered as "fed" in Subsection 4.1.2. For example, if two patterns with $d = 0$ and $d = 2$ match a board configuration, only the pattern with $d = 2$ is considered as "fed".

Semi-Fixed Algorithm

Patterns acquired by Semi-Fixed Algorithm are almost the same as those acquired by the Fixed Algorithm. The difference is that it mainly considers empty squares. In our cognitive experiment [13] using an eye mark recorder, experts mainly solve Tsume-Go problems by seeing empty squares and stones adjacent to the empty squares. This mechanism is introduced in the Semi-Fixed Algorithm. The Manhattan distance is how many steps you can walk vertically and horizontally from the center. In this algorithm you walk only on empty squares. That is, when you count d, only empty squares are counted and squares occupied by a stone are not counted. The counted squares and squares adjacent to the counted ones are considered in a rule. As a result, the number of objects in the IF-part varies. When there is no stone within d Manhattan distance, the rule is the same as that acquired by the Fixed Algorithm. An example is given in Figure 15.

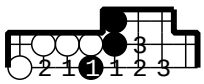

Figure 15: An example of rules acquired by Semi-Fixed Algorithm. $d = 3$.

Mechanism of pattern acquisition is the same as that in the Fixed Algorithm.

4.2.2 Priority Assignment Algorithm

In the pattern acquisition, only patterns are acquired. When you solve Tsume-Go, many patterns are matched. Thus conflict resolution mechanisms are required. In this paper, two mechanisms are introduced.

Assignment of Weights to Rules

In this algorithm, weights, indicating how important the rules are, are automatically assigned to the rules acquired in the first step.

Methods: A weight is an integer. In the initial state, all rules are given the same weight, 200. When the value is larger that this, it takes more time to assign

the weights. When the value is too small, most rules tend to have similar values and discrimination does not work properly. The initial value of weights, 200, was decided after some preliminary experiments.

The points of Move m are calculated by summing all the weights of rules that are fed by Move m. After all the points of possible moves are calculated, the moves are ranked in descending order of points. A move that duplicates a training datum is called a correct move.

Initially given weights are automatically adjusted in the following way. The value of one is subtracted from the weights of all the rules fed by moves ranked higher than the correct move, and the value of S/n is added to the weights of all the rules fed by the correct move, where S is the total subtracted values and n is the number of rules fed by the correct move.

The adjustment of the weights does not change the total amount of the weights of all rules, because the total subtraction of weights is S, and the total addition is $S/n \times n = S$.

We use Table 3 as an example to explain the adjustment. Each $Move_i$ is a candidate move, ranked as the i-th rank by the summation of all the current weights of rules, R_{ij}, which are fed by $Move_i$. Numbers in the table indicate the weights of each rule. In this example the correct move is in the fourth rank; thus, all the weights of the rules fed by Move 1, 2, and 3, which are all negative examples, are decreased by one and the total subtraction is 8. The number of rules fed by the correct move is two, thus $8/2 = 4$ is added to each rule. As a result, the correct move will be in the third rank. Repeating these procedures is expected to make the weights more appropriate.

Table 3: Adjustment of weights of rules.

		weights	
candidates	rules	before	after
$Move_1$	R_{11}	230	229
	R_{12}	210	209
	R_{13}	180	179
	Total	620	617
$Move_2$	R_{21}	200	199
	R_{22}	180	179
	R_{23}	170	169
	Total	550	547
$Move_3$	R_{31}	200	199
	R_{32}	180	179
	Total	380	378
$Move_4$	R_{41}	195	199
(correct)	R_{42}	180	184
	Total	375	383

This realizes automatic adjustment of the degree of change because a larger amount is given to the rules fed by the move when the rank of the correct move is low, and a smaller amount is assigned when the rank of the correct move is high.

Probability of Rule Accuracy

This algorithm calculates the probability of the accuracy of each rule. When only the IF-part is matched and the THEN-part is not matched, it is called a"match" , which is a negative example. When both the IF-part and the THEN-part are matched, it is called a "hit", which is a positive example. The probability of accuracy (p) for a rule, i, is calculated as follows: a_i = No. of hit / No. of match.

The probability of a move, $m, (A_m)$ is calculated as follows: $A_m = 1 - \prod_{i=1}^{n}(1 - a_i)$, where n is the number of hit rules. When $n = 1, A_m = a_1$. When Tsume-Go problems are solved, probability of all moves is calculated and the highest one is selected.

4.2.3 Solving Tsume-Go

Methods

1039 Tsume-Go problems and their answers (total 3,993 moves) are given as the training examples. Patterns were acquired in the first step and then refined in the second step. Using the refined patterns, Tsume-Go problems were solved. We tested all six combinations of the three algorithms for the first step and two for the second step.

Three kinds of problems were used as test problems: basic problems [7], problems for 3 dan [6], and problems for 5 dan [5]. Each kind of problem contained 100 problems, and a total of 300 problems were solved. These problems were not part of the training examples.

Overall Results

The correct rates of solving Tsume-Go for the six combinations are listed in Table 4. The results show that as the first step algorithm, Flexible Algorithm is much better than either Semi-Fixed Algorithm or Fixed Algorithm. Semi-Fixed Algorithm seems a bit better than Fixed Algorithm but the difference is not so clear.

As for the second step, Probability Algorithm seems a bit better than Weights Algorithm with Fixed and Semi-Fixed Algorithms, but Weights Algorithm is better for Flexible Algorithm. This is a very interesting result. It is said that combinations of First step and Second step are very important.

The time it takes to solve problems is very short, less than 10 seconds for solving all the 300 problems for

Table 4: Correct rates of solving Tsume-Go for six conditions.

		first step		
		Flexible	Semi-Fixed	Fixed
second step	weights	31.0	13.3	11.0
	probability	25.0	15.0	13.3

any combination using an Ultra Sparc Station.[5]

Results of Flexible Rules with Weights

Using Flexible Algorithm (proposed in 4.1) and Weights Algorithm produces the best performance. In this section the result is shown in detail. Table 5 lists the detailed results of solving Tsume-Go using the acquired rules with weights. The number indicates the total percentage of correct moves for candidates. For example, solving *Basic* Tsume-Go, the correct percentage of the *1st* rank answer of the system is *36%*, and the correct percentage of the *1st and 2nd* rank answers by the system is *51%*.

Table 5: Correct percentage of solving Tsume-Go using acquired rules with weights. (%)

Rank	Basic	3 dan	5 dan	Average
1st	36 (36)	31 (31)	26 (26)	31 (31)
2nd	15 (51)	26 (57)	20 (46)	20 (51)
3rd	12 (63)	10 (67)	9 (55)	11 (62)
4th	10 (73)	7 (74)	14 (69)	10 (72)
5th	6 (79)	6 (80)	8 (77)	7 (79)

We made an experiment, where human subjects solve the same 300 problems as the system. Subjects were given a Tsume-Go problem and replied with the first move of the solution within three or four seconds. Detailed procedures for the cognitive experiment are explained in [14, 13]. By comparing the average correct rate of the first-ranked move of the system (31%) and the average correct rates of human players, the performance of this system almost equals that of 1 dan human players.

Performance for the difference search space

In order to see whether the performance of the system differs according to the size of search space, the 300 test problems were divided into two groups according to the number of candidate moves: the problems of one group had 10 or more candidate moves (165 problems) and those of the other group had less than 10 candidate moves (135 problems). While the percentage of correct answers of the system for the former set was 29.1%, that for the latter set was 33.3%. This result shows that the system is effective in selecting the first move even if the problem has a lot of candidates. Hence this system will help search-based Tsume-Go solvers when the problems have many candidates, a situation which is very difficult for current search-based solvers.

[5] Almost the same speed as a Pentium II 400 MHz.

5 CONCLUSIONS

Two approaches to acquire Go knowledge from game records were explained: a deductive approach and an evolutionary one. The former approach was taken to acquire strict knowledge. Several rules were acquired from a single training example, and it was shown that proper knowledge can be acquired. The latter approach was taken to acquire a large amount of heuristic knowledge from a large amount of training examples. Tsume-Go problems were solved in order to compare the performance of the algorithm with others. As a result, the proposed algorithm performed best and the performance is as good as 1 dan human players.

Acknowledgments

The authors would like to thank Dr. Kazuhiro Ueda for his helpful comments and discussions. Support for this research is gratefully acknowledged from Dr. Norihiro Hagita, the executive manager of the Media Information Laboratory in NTT Communication Science Laboratories. Members of the Dialog Understanding Research Group gave us invaluable comments on this paper.

References

[1] G. DeJong and R. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176, 1986.

[2] D. Fotland. Knowledge representation in the Many Faces of Go. Available from `ftp://igs.nuri.net/Go/computer/mfg.Z`, 1993.

[3] D. Fotland. Biography of the Many Faces of Go. This is the article reachable by World Wide Web: `http://cgl.ucsf.edu/go/Programs/ManyFaces.html`, 1994.

[4] F. Gobet and H. A. Simon. Recall of random and distorted chess positions: Implications for the theory of expertise. *Memory & Cognition*, 24(4):493–503, 1996.

[5] Y. Ishida. *Godan Toppa-no Tsume-Go 100*. Tsuchiya Shoten, 1989.

[6] Y. Ishida. *Sandan Chosen-no Tsume-Go 100*. Tsuchiya Shoten, 1989.

[7] Y. Ishida. *Kihon Tsume-Go 100 Dai*. Nihon Bunge-sha, 1994.

[8] T. Kojima. *Automatic Acquisition of Go Knowledge from Game Records: Deductive and Evolutionary Approaches*. PhD thesis, The University of Tokyo, 1998.

[9] T. Kojima, K. Ueda, and S. Nagano. An evolutionary algorithm extended by ecological analogy and its application to the game of Go. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 684–689. Morgan Kaufmann, 1997.

[10] T. Kojima and A. Yoshikawa. A two-step model of pattern acquisition: Application to Tsume-Go. In H. J. van den Herik and H. Iida, editors, *Computers and Games*, number 1558 in LNCS, pages 146–166. Springer-Verlag, 1999.

[11] T. M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-Based Generalization: A unifying view. *Machine Learning*, 1(1):47–80, 1986.

[12] H. A. Simon and K. Gilmartin. A simulation of memory for chess positions. *Cognitive Psychology*, 5:29–46, 1973.

[13] A. Yoshikawa and Y. Saito. Perception in tsumego under 3 second time pressure. In *Game Programming Workshop in Japan '95*, pages 105–112, 1995. in Japanese.

[14] A. Yoshikawa and Y. Saito. Perception in tsumego under 4 seconds time pressure. In *Proceedings of the 18th Annual Conference of the Cognitive Science Society*, page 868, 1996.