Preface

A large international conference, Intelligent Systems and Computer Engineering, was held in Hong Kong, March 21–23, 2007, under the International MultiConference of Engineers and Computer Scientists (IMECS) 2007. The IMECS 2007 is organized by the International Association of Engineers (IAENG), a nonprofit international association for engineers and computer scientists. The IMECS conferences serve as good platforms for the engineering community to meet with each other and to exchange ideas. The conferences also strike a balance between theoretical and application development. The conference committees have been formed with over two hundred committee members who are mainly research center heads, faculty deans, department heads, professors, and research scientists from over thirty countries. The conferences are truly international meetings with a high level of participation from many countries. The response that we have received for the multiconference is excellent. There have been more than one thousand one hundred manuscript submissions for the IMECS 2007. All submitted papers have gone through the peer review process and the overall acceptance rate is 58.46%.

This volume contains revised and extended research articles on intelligent systems and computer engineering written by prominent researchers participating in the multiconference IMECS 2007. There is huge demand, not only for theories but also applications, for the intelligent systems and computer engineering in the society to meet the needs of rapidly developing top-end high technologies and to improve the increasing high quality of life. Topics covered include automated planning, expert systems, machine learning, fuzzy systems, knowledge-based systems, computer systems organization, computing methodologies, and industrial applications. The papers are representative of these subjects. The book offers state-of-the-art tremendous advances in intelligent systems and computer engineering and also serves as an excellent reference work for researchers and graduate students working with intelligent systems and computer engineering.

Sio Iong Ao, Oscar Castillo, and Li Xu

July 2007

Hong Kong, Mexico, and China

Chapter 2
Automatically Defined Groups for Knowledge Acquisition from Computer Logs and Its Extension for Adaptive Agent Size

Akira Hara, Yoshiaki Kurosawa, and Takumi Ichimura

2.1 Introduction

Recently, a large amount of data is stored in databases through the advance of computer and network environments. To acquire knowledge from the databases is important for analyses of the present condition of the systems and for predictions of coming incidents. The log file is one of the databases stored automatically in computer systems. Unexpected incidents such as system troubles as well as the histories of daily service programs' actions are recorded in the log files. System administrators have to check the messages in the log files in order to analyze the present condition of the systems. However, the descriptions of the messages are written in various formats according to the kinds of service programs and application software. It may be difficult to understand the meaning of the messages without the manuals or specifications. Moreover, the log files become enormous, and important messages are liable to mingle with a lot of insignificant messages. Therefore, checking the log files is a troublesome task for administrators.

Log monitoring tools such as SWATCH [1], in which regular expressions for representing problematic phrases are used for pattern matching, are effective for detecting well-known typical error messages. However, various programs running in the systems may be open source software or software companies' products, and they may have been newly developed or upgraded recently. Therefore, it is impossible to detect all the problematic messages by the predefined rules. In addition, in order to cope with illegal use by hackers, it is important to detect unusual behavior such as the start of the unsupposed service program, even if the message does not correspond to the error message. To realize this system, the error-detection rules depending on the environment of the systems should be acquired adaptively by means of evolution or learning.

Genetic programming (GP) [2] is one of the evolutionary computation methods, and it can optimize the tree structural programs. Much research on extracting rules from databases by GP has been done in recent years. In the research [3–5],

© Springer Science+Business Media, LLC 2008

the tree structural program in a GP individual represents an IF-THEN rule. In order to acquire multiple rules, we had previously proposed an outstanding method that united GP with cooperative problem-solving by multiple agents. We called this method automatically defined groups (ADG) [6, 7]. By using this method, we had developed the rule extraction algorithm from the database [8–12]. In this system, two or more rules hidden in the database, and respective rules' importance can be acquired by cooperation of agents. However, we meet a problematic situation when the database has many latent rules. In this case, the number of agents runs short for search and for evaluation of each rule because the number of agents is fixed in advance. In order to solve this problem, we have improved ADG so that the method can treat the variable number of agents. In other words, the number of agents increases adaptively according to the acquired rules.

In Sect. 2.2, we explain the algorithm of ADG, and the application to rule extraction from classified data. In Sect. 2.3, we describe how to extract rules from log files by ADG, and show a preliminary experiment using a centralized control server for many client computers. In Sect. 2.4, we describe an issue in the case where we apply the rule-extracting algorithm to a large-scale log file, and then we propose the ADG with variable agent size for solving the problem. We also show the results of experiments using the large-scale log files. In Sect. 2.5, we describe conclusions and future work.

2.2 Rule Extraction by ADG

2.2.1 Automatically Defined Groups

In the field of data processing, to cluster the enormous data and then to extract common characteristics from each cluster of data are important for knowledge acquisition. In order to accomplish this task, we adopt a multiagent approach, in which agents compete with one another for their share of the data, and each agent generates a rule for the assigned data; the former corresponds to the clustering of data, and the latter corresponds to the rule extraction in each cluster. As a result, all rules are extracted by multiagent cooperation. However, we do not know how many rules subsist in the given data and how data should be allotted to each agent. Moreover, as we prepare abundant agents, the number of tree structural programs increases in an individual. Therefore, search performance declines.

In order to solve these problems, we have proposed an original evolutionary method, automatically defined groups. The method is an extension of GP, and it optimizes both the grouping of agents and the tree structural program of each group in the process of evolution. By grouping multiple agents, we can prevent the increase of search space and perform an efficient optimization. Moreover, we can easily analyze agents' behavior group by group. Respective groups play different roles from one another for cooperative problem-solving. The acquired group structure is utilized

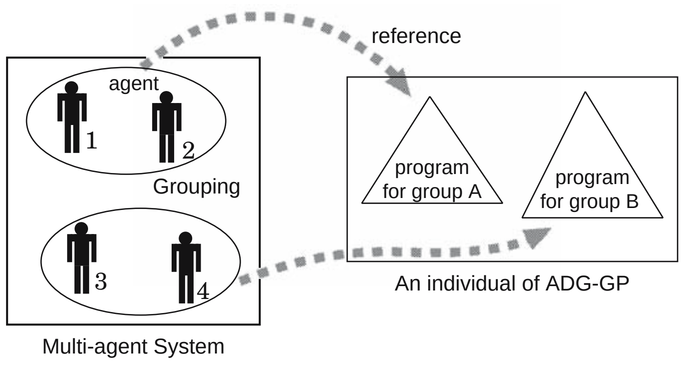


Fig. 2.1 Concept of automatically defined groups

for understanding how many roles are needed and which agents have the same role. That is, the following three points are automatically acquired by using ADG.

- How many groups (roles) are required to solve the problem?
- To which group does each agent belong?
- What is the program of each group?

In the original ADG, each individual consists of a predefined number of agents. The individual maintains multiple trees, each of which functions as a specialized program for a distinct group as shown in Fig. 2.1. We define a group as the set of agents referring to the same tree for the determination of their actions. All agents belonging to the same group use the same program.

Generating an initial population, agents in each GP individual are divided into several groups at random. Crossover operations are restricted to corresponding tree pairs. For example, a tree referred to by agent 1 in an individual breeds with a tree referred to by agent 1 in another individual. This breeding strategy is called restricted breeding [13–15]. In ADG, we also have to consider the sets of agents that refer to the trees used for the crossover. The group structure is optimized by dividing or unifying the groups according to the inclusion relationship of the sets.

The concrete processes are as follows. We arbitrarily choose an agent for two parental individuals. A tree referred to by the agent in each individual is used for crossover. We use T and T' as expressions of these trees, respectively. In each parental individual, we decide a set $A(T)$, the set of agents that refer to the selected tree T. When we perform a crossover operation on trees T and T', there are the following three cases.

(a) If the relationship of the sets is $A(T) = A(T')$, the structure of each individual is unchanged.
(b) If the relationship of the sets is $A(T) \supset A(T')$, the division of groups takes place in the individual with T, so that the only tree referred to by the agents in

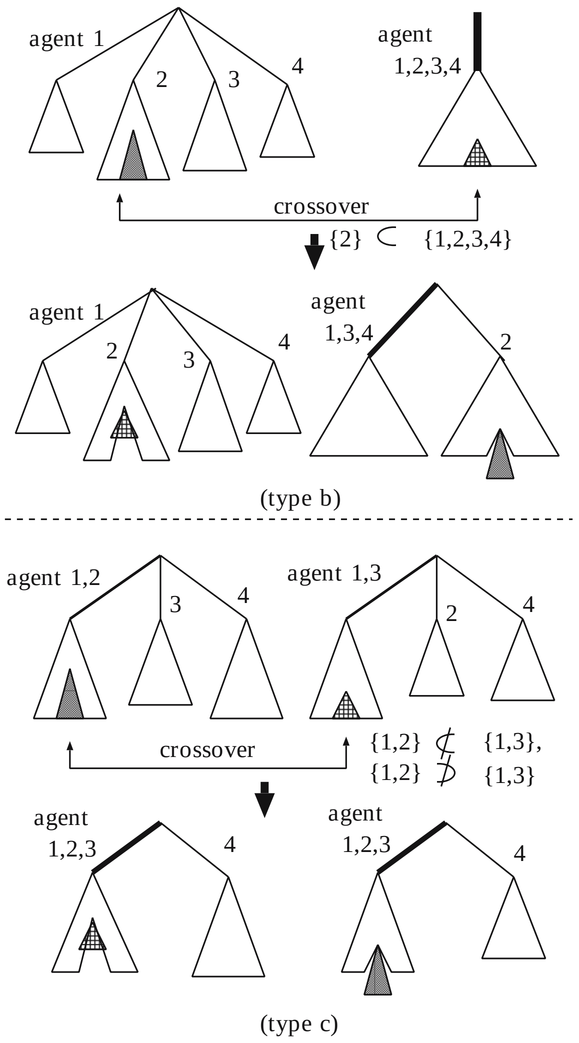


Fig. 2.2 Examples of crossover

$A(T) \cap A(T')$ can be used for crossover. The individual which maintains T' is unchanged. Figure 2.2 (type b) indicates an example of this type of crossover.

(c) If the relationship of the sets is $A(T) \not\supset A(T')$ and $A(T) \not\subset A(T')$, the unification of groups takes place in both individuals so that the agents in $A(T) \cup A(T')$ can refer to an identical tree. Figure 2.2 (type c) shows an example of this crossover.

We expect that the search works efficiently and the adequate group structure is acquired by using this method.

2.2.2 Rule Extraction from Classified Data

In some kinds of databases, each datum is classified into the positive or negative case (or more than two categories). For example, patient diagnostic data in hospitals are classified into some categories according to their diseases. It is an important task to extract characteristics for a target class. However, even if data belong to the same class, all the data in the class do not necessarily have the same characteristics. A part of a dataset might show a different characteristic. It is possible to apply ADG to rule extraction from such classified data. In ADG, multiple tree structural rules are generated evolutionally, and each rule represents the characteristic of a subset in the same class of data. Figure 2.3 shows a concept of rule extraction using ADG. Each agent group extracts a rule for the divided subset, and the rules acquired by multiple groups can cover all the data in the target class. Moreover, when agents are grouped, the load of each agent and predictive accuracy of its rule are considered. As a result, a lot of agents come to belong in the group with the high use-frequency and high-accuracy rule. In other words, we can regard the number of agents in each group as the important degree of the rule. Thus, two or more rules and the important degree of respective rules can be acquired at the same time. This method was applied to medical data and the effectiveness has been verified [8–11].

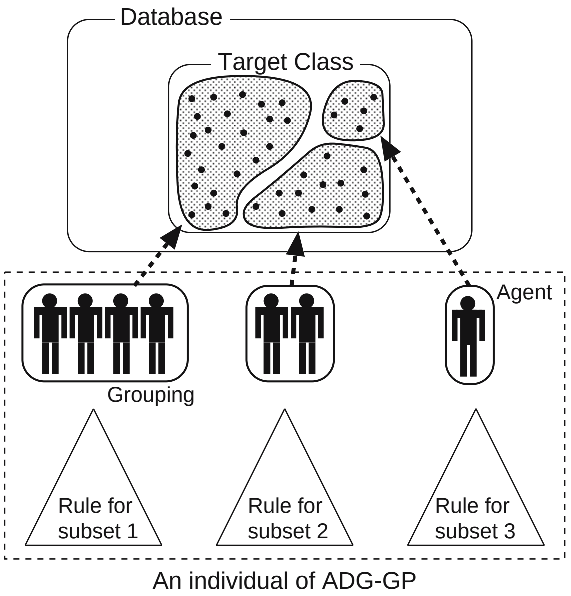


Fig. 2.3 Rule extraction using ADG

2.3 Knowledge Acquisition from Log Files by ADG

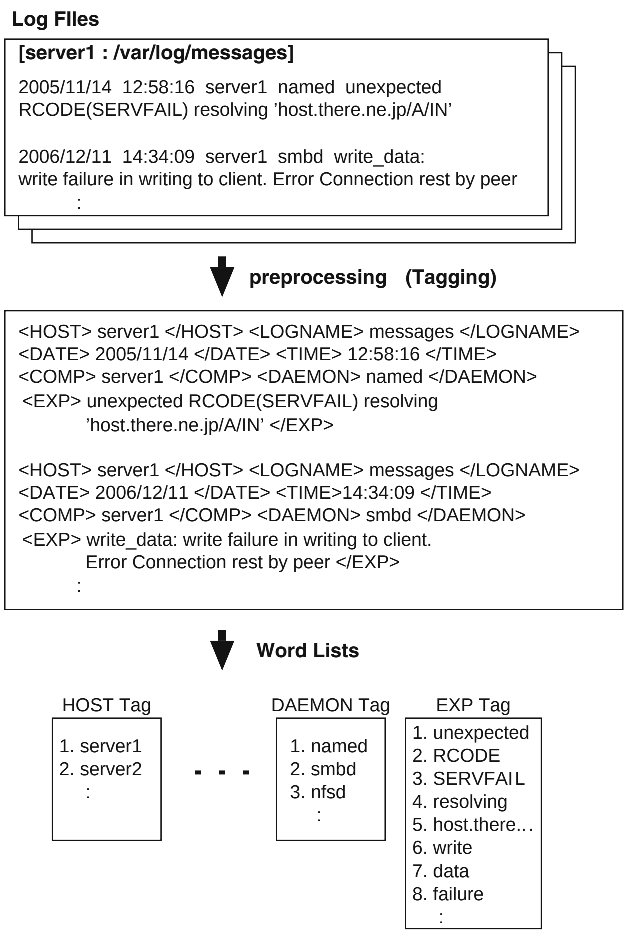
2.3.1 How to Extract Rules from Unclassified Log Messages

We apply the rule extraction method using ADG to detect trouble in computer systems from log files. In order to use the method described in the previous section, we need supervised information for its learning phase. In other words, we have to classify each message in the log files into two classes: normal message class and abnormal message class indicating system trouble. However, this is a difficult task because complete knowledge for computer administration is needed and log data are of enormous size. In order to classify log messages automatically into the appropriate class, we consider a state transition pattern of computer system operation. We focus on the following two different states and make use of the difference of the states as the supervised information.

1. Normal state. This is the state in the period of stable operation of the computer system. We assume that the administrators keep good conditions of various system configurations in this state. Therefore, frequently observed messages (e.g., "Successfully access," "File was opened," etc.) are not concerned with the error messages. Of course, some insignificant warning messages (e.g., "Short of paper in printer," etc.) may sometimes appear.
2. Abnormal state. This is the state in the period of unstable operation of the computer system. The transition to the abnormal state may happen due to hardware trouble such as hard disk drive errors, or by restarting service programs with new configurations in the current system. Moreover, some network security attacks may cause the unstable state. In this state, many error messages (e.g., "I/O error," "Access denied," "File not found," etc.) are included in the log files. Of course, the messages observed in the normal state also appear in the abnormal state.

The extraction of rules is performed by using log files in the respective states. First, we define the base period of the normal state, which seems to be stable, and define the testing period, which might be in the abnormal state. Then we prepare the two databases. One is composed of log messages in the normal state period, and the other is composed of log messages in the abnormal state period. By evolutionary computations, we can find rules, which respond to the messages appearing only in the abnormal state.

For knowledge representation to detect a remarkable problematic case, we use the logical expressions, which return true only to such problematic messages. The tagging procedure using regular expressions as described in [16] was used for the preprocessing to the log files and the representation of the rules. Figure 2.4 shows an illustration of the preprocessing. Each message in the log files is separated into several fields (e.g., daemon name field, host name field, comment field, etc.) by the preprocessing, and each field is tagged. Moreover, words that appear in the log messages are registered in the word lists for respective tags beforehand.



```
[server1 : /var/log/messages]
2005/11/14  12:58:16  server1  named  unexpected
RCODE(SERVFAIL) resolving 'host.there.ne.jp/A/IN'

2006/12/11  14:34:09  server1  smbd  write_data:
write failure in writing to client. Error Connection rest by peer
          :
```

```
<HOST> server1 </HOST> <LOGNAME> messages </LOGNAME>
<DATE> 2005/11/14 </DATE> <TIME> 12:58:16 </TIME>
<COMP> server1 </COMP> <DAEMON> named </DAEMON>
 <EXP> unexpected RCODE(SERVFAIL) resolving
           'host.there.ne.jp/A/IN' </EXP>

<HOST> server1 </HOST> <LOGNAME> messages </LOGNAME>
<DATE> 2006/12/11 </DATE> <TIME>14:34:09 </TIME>
<COMP> server1 </COMP> <DAEMON> smbd </DAEMON>
 <EXP> write_data: write failure in writing to client.
           Error Connection rest by peer </EXP>
          :
```

HOST Tag		DAEMON Tag	EXP Tag
1. server1	. . .	1. named	1. unexpected
2. server2		2. smbd	2. RCODE
:		3. nfsd	3. SERVFAIL
		:	4. resolving
			5. host.there...
			6. write
			7. data
			8. failure
			:

Fig. 2.4 Preprocessing to log files

The rule is made by the conjunction of multiple terms, each of which judges whether the selected word is included in the field of the selected tag. The following expression is an example of the rule.

```
(and (include <DAEMON> 3)(include <EXP> 8))
```

We assume that the word "nfsd" is registered third in the word list for the <DAEMON> tag, and the word "failure" is registered eighth in the word list for the <EXP> tag. For example, this rule returns true to the message including the following strings.

```
<DAEMON>nfsd</DAEMON> <EXP>Warning:access failure</EXP>
```

Multiple trees in an individual of ADG represent the respective logical expressions. Each message in the log files is input to all trees in the individual. Then, calculations are performed to determine whether the message satisfies each logical expression. The input message is regarded as the problematic case if one or more trees in the individual return true. In contrast, the input message is not regarded as the problematic case if all trees in the individual return false. Therefore, all the rules should return false to the messages that appear in both states.

The fitness is calculated based on the accuracy for error detection and load balancing among agents. High accuracy for error detection means that the rules detect as many messages as possible in the abnormal state and react to as few messages as possible in the normal state. The concept of each agent's load arises from the viewpoint of cooperative problem-solving by multiple agents. The load is calculated from the adopted frequency of each group's rule and the number of agents in each group. The adopted frequency of each rule is counted when the rule returns true to the messages in the abnormal state log. If multiple trees return true for a message, the frequency of the tree with more agents is counted. When the agent a belongs to the group g, the load of the agent w_a is defined as follows,

$$w_a = \frac{f_g}{n^g_{\mathrm{Agent}}}, \tag{2.1}$$

where n^g_{Agent} represents the number of agents that belong to group g, and f_g represents the adopted frequency of g. For balancing every agent's load, the variance of the loads V_w as shown in (2.2) should be minimized.

$$V_w = \tfrac{1}{N_{\mathrm{Agent}}} \left\{ \textstyle\sum_{i=1}^{N_{\mathrm{Agent}}} (\bar{w} - w_i)^2 \right\}, \tag{2.2}$$

$$\bar{w} = \tfrac{1}{N_{\mathrm{Agent}}} \textstyle\sum_{i=1}^{N_{\mathrm{Agent}}} w_i, \tag{2.3}$$

where N_{Agent} represents the number of agents in the individual. By load balancing, more agents are allotted to the group that has a greater frequency of adoption. On the other hand, the number of agents in the less-adopted group becomes small. Therefore, the number of agents of respective rules indicates how general each rule is for the detection of problematic messages. Moreover, when usual messages in the normal state are judged to be problematic messages through a mistake of a rule, it is considered that the number of agents who support the rule should be small. To satisfy the requirements mentioned above, we maximize the fitness f defined as follows.

$$\begin{aligned} f = {} & \frac{H_{\mathrm{Abn}}/N_{\mathrm{Abn}}}{H_{\mathrm{Norm}}/N_{\mathrm{Norm}}} \\ & - \beta \, \frac{\sum_{N_{\mathrm{Norm}}} \mathrm{fault\_agent}}{H_{\mathrm{Norm}} \times N_{\mathrm{Agent}}} - \delta \, V_w. \end{aligned} \tag{2.4}$$

In this equation, N_{Abn} and N_{Norm} represent the number of messages in the abnormal state and normal state, respectively. H_{Abn} and H_{Norm} represent the frequency that one or more trees in the individual return true for abnormal state logs and normal state logs, respectively. fault_agent represents the number of agents who support the wrong rule, when the rule returns true for messages in the normal state. Therefore, the second term represents the average rate of agents who support the wrong rules when misrecognition occurs. By this term, the allotment of agents to a rule with more misrecognition will be restrained. By the third term, load balancing of agents will be achieved. In addition, in order to inhibit the redundant division of groups, the fitness value is modified according to the number of groups, G $(G \geq 1)$, in the individual as follows,

$$f \leftarrow \gamma^{G-1} \times f \quad (0 < \gamma < 1), \tag{2.5}$$

where γ represents the discount rate for the fitness. This equation means that the fitness is penalized according to the increase of G.

By evolution, one of the multiple trees learns to return true for problematic messages that appear only in the abnormal state logs, and all trees learn to return false for normal messages that appear both in the normal and abnormal state logs. Moreover, agents are allotted to respective rules according to the adopted frequency and the low rate of misrecognition. Therefore, the rule with more agents is the typical and accurate error-detection rule, and the rule with less agents is a specialized rule for the rare case.

2.3.2 Preliminary Experiment

In order to examine the effectiveness of the rule extraction method, we apply the method to the log files in an actual computer environment. As the actual log files, the logs in the centralized control server for many client computers are used. The server can apply management operations such as boot or shutdown to client machines all at once. The numbers of messages included in log files, N_{Norm} and N_{Abn}, are 2992 and 728, respectively.

The parameter settings are as follows: population size is 300. The number of agents in each individual at initial population is 50. The respective weights in (2.4) and (2.5) are $\beta = 0.0001$, $\delta = 0.0001$, and $\gamma = 0.9999$. These parameter values were determined by preliminary trials.

As the result of a tagging procedure using regular expressions, six kinds of tags (HOST, LOGNAME, SORT, FUNC, EXP, and DATA) are attached to the messages in the log files. When we make word lists for respective tags, the largest word list size is 81 for the EXP tag. Figure 2.5 illustrates the generated word lists for respective tags.

Table 2.1 shows GP functions and terminals for these experiments. We impose constraints on the combination of these symbols, such as strongly typed genetic programming [17]. For example, terminal symbols do not enter directly in the arguments of the `and` function. Crossovers and mutations that break the constraints are not performed. In addition, the sizes of word lists for respective tags are different from one another. Therefore, the value in the second argument of the `include` function may exceed the size of the word list for the corresponding tag. In that case, the remainder in dividing the value by the word list size is used for the selection of a word.

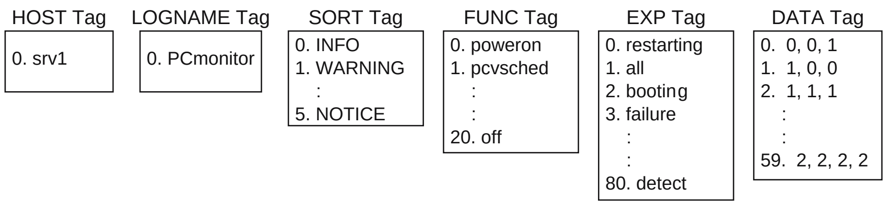


Fig. 2.5 Word lists for the centralized control server's log

Table 2.1 GP functions and terminals

Symbol	#Args	Functions
`and`	2	arg0 $\wedge$ arg1
`include`	2	If Tag arg0 includes arg1 (Word) then T else F
`<HOST>,<EXP>,...`	0	Tag name
`0,... ,N-1`	0	Select corresponding word from word list. N is the number of words in list.

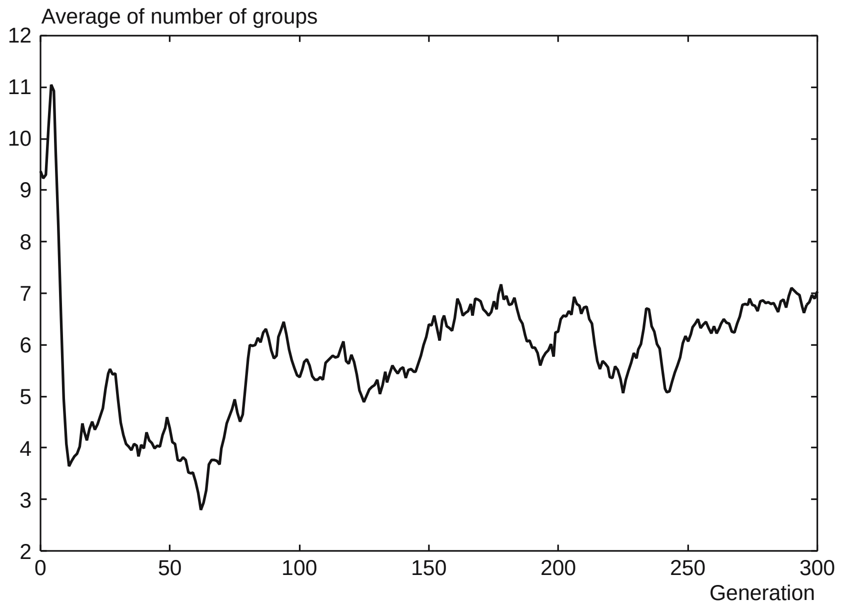


Fig. 2.6 Change of the average number of groups

As the result of a evolutionary optimization, multiple rules, which respond to the messages appearing only in the abnormal state, were acquired successfully. Figure 2.6 illustrates the change of the average number of groups. The number of groups corresponds to the number of extracted rules. As a result, 50 agents in the best individual were divided into eight groups. The best individual detected 157

```
Rule1 (42agents): (and (and (include <LOGNAME> PCmonitor)
             (include <HOST> srv1))(and (include <EXP> booting)
             (include <SORT> INFO)))

Rule2 ( 2agents): (include <DATA> 2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,
                   2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2)

Rule3 ( 1agent) : (include <EXP> Ftrans)

Rule4 ( 1agent) : (and (include <LOGNAME> PCmonitor)
            (include <EXP> dst=/usr/aw/maya7.0/scripts/startup))

Rule5 ( 1agent) : (include <DATA> 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                   2,2,2,2,2,2,2,2,2,2,0,2,0,2,2,2,2,2,2,2,2)

Rule6 ( 1agent) : (and (include <EXP> rexec)
                       (include <LOGNAME> PCmonitor))

Rule7 ( 1agent) : (include <EXP> Rexec)

Rule8 ( 1agent) : (include <DATA> 2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
                   2,2,2,2,2,2,2,2,2,2,0,2,0,2,0,2,0,2,0,2,0)
```

Fig. 2.7 Acquired rules for the centralized control server

```
<HOST>srv1</HOST> <LOGNAME>PCmonitor</LOGNAME>
<SORT>INFO[0]</SORT>
<FUNC>ftrans(470).boot_request_ftrans</FUNC>
<EXP>pc300 Already booting</EXP>

<HOST>srv1</HOST> <LOGNAME>PCmonitor</LOGNAME>
<SORT>INFO[0]</SORT><FUNC>poweron(31968).main</FUNC>
<EXP>322ALL</EXP> <DATA>2,0,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2</DATA>
```

Fig. 2.8 Examples of detected messages by the acquired rules

messages as problematic ones from the abnormal state log. On the other hand, the best individual detected no messages in the normal log.

Figure 2.7 shows the acquired rules in the best individuals. These rules are arranged according to the number of agents, and the second arguments of the `include` function are replaced by the corresponding words. Figure 2.8 shows examples of detected messages by the acquired rules. For example, the word "2, 0, 2, ..., 2" in the second rule in Fig. 2.7 represents a part of an output message, which is returned when performing the "power on" operation for all the client machines. According to an system administration expert, the values delimited by a comma correspond to the conditions of respective machines, and the value "0" means the occurrence of a failure of the operation. Thus, the proposed method can extract error-detection rules without expertise by utilizing the characteristic that such error messages do not appear in the normal state.

In addition, the first rule detects a large part of problematic messages (136 messages) from the abnormal state log. That is, the first rule can be considered as the typical error-detection rule for the environment. As shown in Fig. 2.7, the great mass of agents is assigned to the search of the rule. Thus, the number of agents in each group represents the important degree of the corresponding rule.

2.4 ADG with Variable Agent Size

2.4.1 Issue in Applying ADG to Large-Scale Logs

As mentioned in the previous section, we can extract problematic messages from actual log files by ADG without expertise on administration. As another experiment, we applied the method to large-scale log files. The log files used in the experiment were picked up by a network file server for many client computers. The numbers of messages included in log files N_{Norm} and N_{Abn} are 32,411 and 17,561, respectively. As a result, 44 rules were extracted by ADG using 50 agents as shown in Fig. 2.9. This result indicates an issue of this method. Only one agent was allotted for most rules because the number of prepared agents is insufficient. Even the rule with the maximum number of agents has only three agents. It was impossible to understand minutely the difference of the importance of the rules.

The above results show that the capability of evaluating the important degree of each rule is correlated with the number of agents in an individual. Each agent group extracts one rule. Therefore, the maximum of extractable rules is equal to the number of agents. We have to use as many agents as the supposed rules at least. Moreover, the rules' importance can be judged by the number of agents. Therefore, to evaluate rules in detail, more agents are needed so that the number of agents can exceed the extracted number of rules sufficiently. In short, the problem is that the number of agents is not sufficient to manage the rules. However, it is impossible to estimate the extracted number of rules because it is difficult to set a large enough number of agents beforehand.

2.4.2 ADG with Variable Agent Size

In order to solve the problem on the number of agents, we set that the number of agents dynamically increases to be more than multiples of the number of the acquired rules. The procedures for increasing agents are as follows.

In the best individual of each generation t, we find N^{t}_{Rules}, the number of rules that return true for problematic messages. When the number of agents in each individual

```
Rule1 ( 3agents): (include <DAEMON> smbd)
Rule2 ( 3agents): (include <EXP> race)
Rule3 ( 2agents): (include <EXP> nrpc)
Rule4 ( 1agent) : (include <EXP> NOTLIB)
 :
 :
Rule42( 1agent) : (include <DAEMON> gdm(pam_unix))
Rule43( 1agent) : (include <EXP Connection)
Rule44( 1agent) : (include <EXP> I/O)
```

Fig. 2.9 Acquired rules for a large-scale log files

at the generation t is N^t_{Agents}, the condition for increasing agents is expressed as follows,

$$N^t_{\text{Agents}} < kN^t_{\text{Rules}} \quad (k \geq 1.0), \tag{2.6}$$

where k is the parameter for controlling the agent size. When this condition is satisfied, the number of agents in each individual is incremented by one. The flow of the evolutionary process is shown below.

(a) Initialization of individuals.
(b) Fitness evaluation of each individual.
(c) Genetic operations
(Selection + Elitist Strategy, Crossover, Mutation).
(d) Operation for increasing agents.
(We find the number of rules and agents in the best individual. If the condition for increasing agents is satisfied, one agent is added to respective individuals.)
(e) If termination condition is not satisfied, return to (b).

When the number of extracted rules increases to N_{Rules}, the number of agents finally reaches kN_{Rules} by the above operations.

2.4.3 Experiments for Large-Scale Logs

2.4.3.1 Comparison Among Variable/Fixed Agent Size Methods

We apply the proposed method to rule extraction from large-scale log files, where the problem concerning the number of agents previously occurred as described in Sect. 2.4.1. We set the number of agents in each individual at initial population at 50, and set the parameter k in the condition for increasing agents at 3.0. For comparison with fixed large size of agents, we also perform another experiment using ADG with fixed 200 agents in each individual.

The parameter settings are as follows: population size is 300. The respective weights in (2.4) and (2.5) are $\beta = 0.001$, $\delta = 0.01$, and $\gamma = 0.9999$. These parameter values were determined by preliminary experiments.

As a result, agents in the best individual were divided into 72 groups by the proposed method. That is, we could get 72 rules. The number of agents was 216 at the last generation. Figure 2.10 shows the best fitness curves by the conventional ADG [16] using fixed 50 or 200 agents and the proposed ADG with variable agent size. We can see from this figure that the search of any method converged by 1000 generations, and the proposed method got better fitness value than the conventional fixed agent size methods. Figure 2.11 shows the change of the number of extracted rules and agents by the proposed method. We can see from this figure that 50 agents are enough for search upto 127 generations, but after the generation the number of agents increases according to the number of rules so as not to be short.

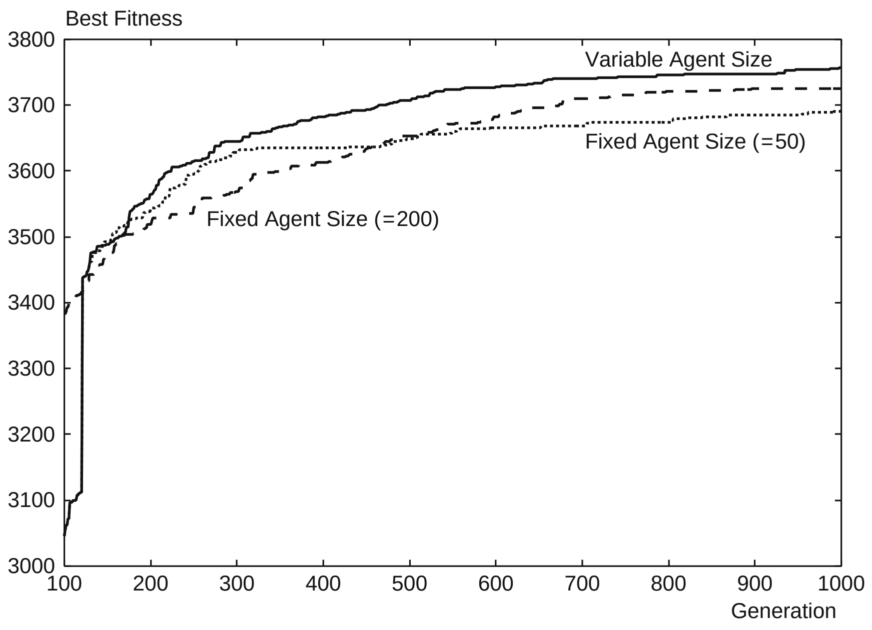


Fig. 2.10 Comparison of the best fitness curves

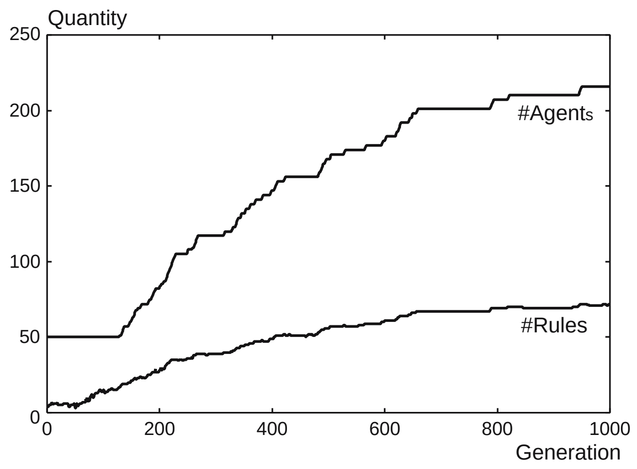


Fig. 2.11 The number of extracted rules and agents

When the number of the extracted rules converged at about the 950th generation, the number of agents also converged. Table 2.2 shows some of the acquired rules by the conventional method and proposed method. Respective rules correspond to the tree structural programs in the best individual. Table 2.2 also shows the number of agents of each rule. These rules are arranged according to the number of agents.

Table 2.2 Some acquired rules and the number of agents

ID	Rule	#Agents fixed (50)	#Agents variable
1	(Include DAEMON smbd)	3	15
2	(Include EXP race)	3	12
3	(Include EXP nrpc)	2	14
4	(Include EXP NOTLB)	1	9
⋮			
42	(Include DAEMON gdm(pam_unix))	1	2
43	(Include EXP Connection)	1	1
44	(Include EXP I/O)	1	1
⋮			
72	(Include EXP Journalled)	–	1

```
<HOST> fsv </HOST> <LOGNAME> messages </LOGNAME>
<DATE>2006/04/19</DATE> <TIME>14:15:37</TIME>
<COMP> fsv </COMP> <DAEMON> smbd </DAEMON>
<EXP> decode_pac_date: failed to verify PAC server
 signature </EXP>

<HOST> fsv </HOST> <LOGNAME> messages </LOGNAME>
<DATE>2006/04/19</DATE> <TIME>14:34:09</TIME>
<COMP> fsv </COMP> <DAEMON> smbd </DAEMON>
<EXP> write_data: write failure in writing to
client. Error Connection reset by peer </EXP>

<HOST> fsv </HOST> <LOGNAME> messages </LOGNAME>
<DATE>2006/04/19</DATE> <TIME>16:43:30</TIME>
<COMP> fsv </COMP> <DAEMON> kernel </DAEMON>
<EXP> I/O error: dev 08:f0, sector 0 </EXP>
```

Fig. 2.12 Log messages detected by the acquired rules

In the conventional method, the number of agents for each rule ranges from one to three, and most rules (rule ID 4, 5, ..., and 44) have only one agent. Therefore, we cannot list the rules in order of importance. In the proposed method, the number of agents for each rule ranges widely from 1 to 15 by the increase of agents. This result shows that the proposed method is useful for the minute evaluation of the importance of the respective rules.

Furthermore, the number of acquired rules by the proposed method is 72. On the other hand, the number of rules by the conventional method is 44. That is, we can get more rules by using a variable number of agents. This result indicates that the search performance of the proposed method becomes better with the increase of agents, and the new rule can be acquired. Therefore, the proposed method shows a higher fitness value than the conventional method as shown in Fig. 2.10. Examples of log messages detected by the acquired rules are shown in Fig. 2.12.

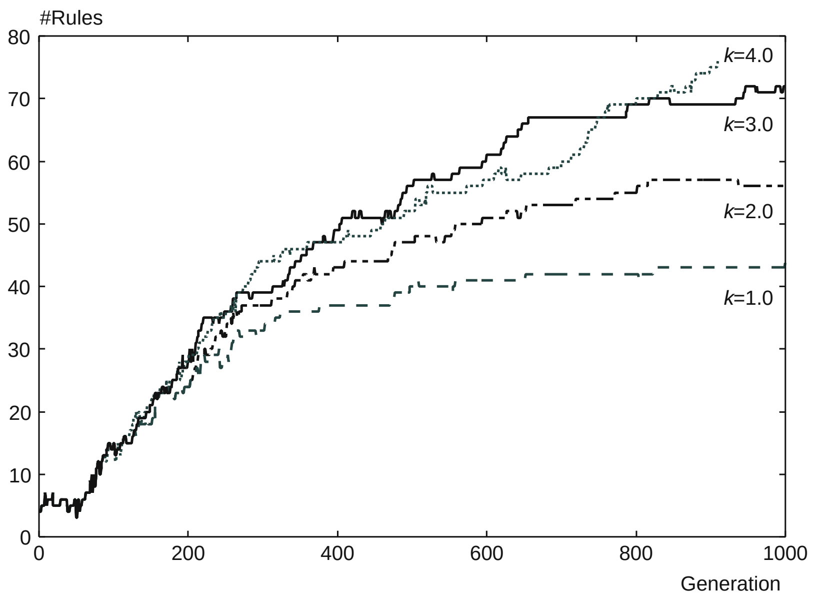


Fig. 2.13 The number of extracted rules in various k

2.4.3.2 Effect of Parameter for Agent Size

In order to examine the effect of the parameter k used in condition (2.6) for increasing agents, we perform experiments with various values of k. Figure 2.13 shows the number of extracted rules of four kinds of k ($k = 1.0, 2.0, 3.0$, and 4.0). We can see from this figure that 44 rules, 56 rules, 72 rules, and 75 rules were extracted, respectively. In the case of $k = 1.0$, the condition (2.6) for increasing agents had not been satisfied in the evolutionary process, and the number of agents remains 50 for the initial individual. As the value of k increases from 1.0 to 3.0, the number of rules also increases. That is, the more agents are used for evolutionary search, the more rules can be acquired. However, the number of acquired rules in $k = 4.0$ is almost the same as that in $k = 3.0$. To determine the adequate value of k adaptively is one of the challenges for improving the method.

2.5 Conclusions and Future Work

In this research, the mechanism where the number of agents increases in proportion to the number of discovered rules was introduced to the ADG method. As a result, two good effects were observed. One effect is that it becomes possible to evaluate the importance of respective rules in detail, and the other is that the number of extracted rules increases. In these experiments, we set that the number of agents should be more than three times the number of extracted rules. We have to examine an appropriate criterion for increasing the number of agents. Moreover, in the present fitness function, agents are allotted to respective rules from the viewpoints of load balancing of agents and from the viewpoint of the decrease of agents who support the wrong rules. As a result, the number of agents becomes an index of the importance for each rule, in which both the frequency of use and accuracy are considered. However, when log information is treated, not only the occurrence frequency but also the degree of the influence on computer systems becomes important. We have to investigate the way to introduce other viewpoints (e.g., risk or urgency level, etc.) into the fitness function, so that the number of agents can become a more profitable index.

References

1. SWATCH: The Simple WATCHer of Logfiles. (2007) http://swatch.sourceforge.net/
2. J.R. Koza (1992) *Genetic Programming – On the Programming of Computers by Means of Natural Selection*. The MIT Press
3. C.C. Bojarczuk, H.S. Lopes, A.A. Freitas (2000) Genetic programming for knowledge discovery in chest-pain diagnosis. *IEEE Engineering in Medicine and Biology*. Vol. 19, No. 4, pp. 38–44
4. C.C. Bojarczuk, H.S. Lopes, A.A. Freitas (2003) An innovative application of a constrained-syntax genetic programming system to the problem of predicting survival of patients. In: *Proceedings of Euro GP 2003*. pp. 11–21
5. L. Hirsch, M. Saeedi, R. Hirsch (2005) Evolving rules for document classification. In: *Proceedings of Euro GP 2005*. pp. 85–95
6. A. Hara, T. Nagao (1999) Emergence of cooperative behavior using ADG; Automatically defined groups. In: *Proceedings of the 1999 Genetic and Evolutionary Computation Conference*. pp. 1039–1046
7. A. Hara, T. Nagao (2002) Construction and analysis of stock market model using ADG; Automatically defined groups. *International Journal of Computational Intelligence and Applications (IJCIA)*. Vol. 2, No. 4, pp. 433–446
8. A. Hara, T. Ichimura, K. Yoshida (2005) Discovering multiple diagnostic rules from coronary heart disease database using automatically defined groups. *Journal of Intelligent Manufacturing*. Vol. 16, No. 6, pp. 645–661
9. A. Hara, T. Ichimura, T. Takahama, Y. Isomichi (2004) Discovery of cluster structure and the clustering rules from medical database using ADG; Automatically defined groups. In: T. Ichimura and K. Yoshida (eds) *Knowledge-Based Intelligent Systems for Healthcare*. pp. 51–86, CRC Press
10. T. Ichimura, S. Oeda, M. Suka, A. Hara, K.J. Mackin, K. Yoshida (2005) Knowledge discovery and data mining in medicine. In: N. Pal and L.C. Jain (eds) *Advanced Techniques in Knowledge Discovery and Data Mining*. pp. 177–210, Springer
11. A. Hara, T. Ichimura, T. Takahama, Y. Isomichi (2005) Extraction of risk factors by multi-agent voting model using automatically defined groups. In: *Proceedings of the Ninth Conference on Knowledge-Based Intelligent Information and Engineering Systems (KES'2005)*. Vol. 3, pp. 1218–1224
12. A. Hara, T. Ichimura, T. Takahama, Y. Isomichi (2003) Extraction of rules by heterogeneous agents using automatically defined groups. In: *Proceedings of the Seventh Conference on Knowledge-Based Intelligent Information and Engineering Systems (KES'2003)*. Vol. 2, pp. 1405–1411

13. S. Luke, L. Spector (1996) Evolving teamwork and coordination with genetic programming. In: *Genetic Programming 1996: Proceedings of the First Annual Conference*. pp. 150–156
14. H. Iba (1996) Emergent cooperation for multiple agents using genetic programming. In: *Parallel Problem Solving from Nature IV. Proceedings of the International Conference on Evolutionary Computation*. pp. 32–41
15. H. Iba (1997) Multiple-agent learning for a robot navigation task by genetic programming. In: *Genetic Programming 1997: Proceedings of the Second Annual Conference*. pp. 195–200
16. Y. Kurosawa, A. Hara, T. Ichimura, Y. Kawano (2006) Extraction of error detection rules without supervised information from log files using automatically defined groups. In: *Proceedings of The 2006 IEEE International Conference on System, Man and Cybernetics*. pp. 5314–5319
17. T. Haynes, R. Wainwright, S. Sen, D. Schoenefeld (1995) Strongly typed genetic programming in evolving cooperation strategies. In: *Genetic Algorithms: Proceedings of the Sixth International Conference (ICGA95)*. pp. 271–278