論 文

Genetic Network Programming with Automatic Program Generation for Agent Control

Bing Li — Graduate School of Information, Production and Systems, Waseda University
libing@asagi.waseda.jp

Shingo Mabu — Graduate School of Information, Production and Systems, Waseda University
mabu@aoni.waseda.jp

Kotaro Hirasawa — Graduate School of Information, Production and Systems, Waseda University
hirasawa@waseda.jp

keywords: Genetic Programming, Genetic Network Programming, program generation, tileworld, genotype-phenotype mapping

Summary

In this paper, a new Genetic Network Programming with Automatic Program Generation (GNP-APG) has been proposed and applied to the Tileworld problem. A kind of genotype-phenotype mapping process is introduced in GNP-APG to create programs. The procedure of the program generation based on evolution is demonstrated in this paper. The advantages of the proposed method are also described. Simulations use different Tileworlds between the training phase and testing phase for performance evaluations and the results shows that GNP-APG could have better performances than the conventional GNP method.

1. Introduction

Derived from Genetic Algorithm [Holland 75, Goldberg 89] and Genetic Programming (GP) [Koza 92, Koza 94, Koza 99], Genetic Network Programming (GNP) [Mabu 07, Eguchi 06] has been proposed as an evolutionary algorithm to efficiently solve the complex problems. GNP uses directed-graph structures as its gene structure. Based on the higher expression ability of graph structures, GNP has inherent features such as reusability of nodes and building block functions, which avoid the bloating problem and improves the performances of the algorithms [Mabu 07]. Up to now, GNP has been successfully applied to many fields such as elevator supervisory control systems [Hirasawa 08], stock market prediction [Chen 09], association rule mining [Shimada 06], and traffic prediction[Zhou 10].

On the other hand, automatic program generation, in other words, automatic programming or program induction is a way to obtain a program without explicitly programming it. Several evolutionary algorithms like GP, Gene Expression Programming (GEP) [Ferreira 01, Ferreira 02] and Grammatical Evolution (GE) [O'Neill 01, O'Neill 03] have been proposed with much success for this research field. In these methods, GP is the most well-known and widely used one. For example, in symbolic regression problems, GP builds up a function close to the target function by combining math operators called function set ('+', '-', 'sin', 'cos'...) and variables or constants called terminal set ('1', 'x', 'y', 'z'...)[Koza 92]; actually, in an artificial ant application, GP creates a program through function set ('if Food Ahead', 'prog2', ...) and terminal set ('move To Nest', 'pick Up Food', ...) to teach the artificial ants for searching the food and taking it to their nest [Koza 92].

Nowadays, some studies on GNP for automatic program generation (GNP-APG) has been conducted, and the simulation result shows good performances of it [Mabu 05, Mabu 09]. But, in these papers, only static problems are used to verify the performance of GNP-APG. So, the objective of this paper is to improve GNP-APG to deal with the time dependent environment problems like Tileworld and prove the proposed method works better than the conventional GNP. In addition, Tileworld is a famous test bed for agents with time dependent and uncertain characteristics, since the environment always changes and agents cannot get all information of the environment [Pollack 90].

The improved GNP-APG introduces a kind of genotype-phenotype mapping process to create programs [Banzhaf 94]. In GNP-APG, the genotype is the structure of GNP and the phenotype is programs. Through the transition from nodes to nodes (genotype), the programs (phenotype) are generated and stored in the outside memory. As

noted in [O'Neill 01] and [Banzhaf 94], a mapping process can separate the search space and solution space, which makes the search of the genotype unlimited, while still keeping the legality of the program. With the mapping process, genetic operations are not performed on the programs, but on GNP structure which works as a program generator. This is a key point on how the proposed method is different from GP.

Though GNP-APG extends from the conventional GNP, but there are many differences between them.

- An individual of GNP-APG is a solution generator, and it is only used for the mapping process to create the solutions for the problem. After evolution, a better solution generator can be obtained. But, the individual of the conventional GNP is a solution for the problem.
- GNP-APG only communicates with the outside memory, where its individuals get primary elements or subprograms from the memory and put subprograms to the memory. While the conventional GNP individuals directly probe the information from environments, then use this information to make a decision and tell agents what to do.
- GNP-APG has the outside memory for the generated programs, which can save more information explicitly. While the conventional GNP keeps the information in the network flow implicitly.

This paper is organized as follows. Next section provides the brief concept of GNP. In section 3, the proposed method of improved GNP-APG is described deeply. Section 4 shows the simulation environments and the performances of the conventional GNP and GNP-APG. Section 5 is devoted to conclusions.

2. The Conventional GNP

In this section, the conventional GNP (GNP for short) is reviewed briefly.

2·1 Basic structure of GNP

As mentioned before, GNP has a directed-graph structure which is different from strings in GA and trees in GP. The basic structure of GNP is shown in Figure 1. The structure of GNP consists of three kinds of primary nodes: start node, judgment node and processing node. These nodes are connected by directed links shown with arrows. The square stands for the start node which identifies the first node to perform. The hexagon represents the judgment nodes. The judgment nodes have several branches connected to the other judgment nodes or processing nodes. When the judgment node is executed, it probes the information from environments, then analyzes the current situation, and finally decides the next node to move depending on the result of judgments. The circle describes the processing node. Processing nodes only have one branch linked to the other node except the start node. Each processing node will make an agent take an action when the processing node is visited. After the action, the environment is changed. In practice, the number of branches of judgment nodes, the functions of judgment nodes and processing nodes are determined by designers according to the problem.

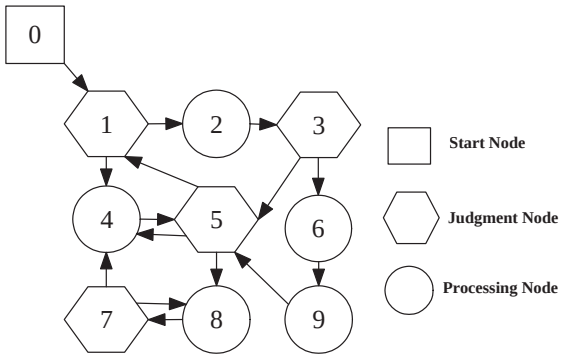

Fig. 1 Basic structure of the conventional GNP

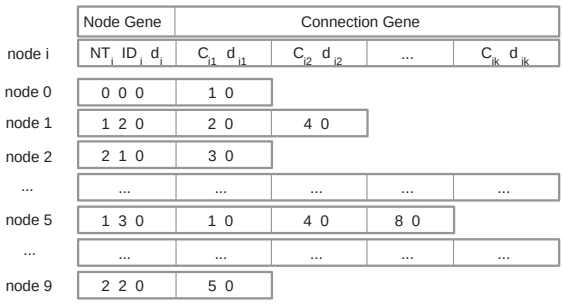

Fig. 2 Representation of GNP structure

Figure 2 shows the representation of GNP structure. An integer array is used to describe the gene of a node. The gene contains two part: node gene and connection gene. The node gene stores three kinds of data, which are NT_i, ID_i, and d_i. NT_i represents the type of node i. The three options 0, 1, and 2 means the start node, judgment node and processing node, respectively. ID_i means the identity of node i which shows the index of functions. d_i is used to describe the time delay for judgment and processing. The connection gene keeps the connection information from node i, where $C_{i1}, C_{i2}...$ represent the index of the connected nodes and $d_{i1}, d_{i2}...$ show the time delay for the transition of these connections.

2·2 Genetic operators of GNP

Like other evolutionary algorithms, GNP also introduces selection, crossover and mutation to evolve the GNP individuals.

GNP provides elite selection and tournament selection. Elite selection is simple, it picks up the best individual and move it to the next generation directly. Tournament selection chooses several individuals from the current population randomly, then runs several "tournaments". The winners of the individuals are selected for crossover and mutation.

Crossover is performed between two parents and generates two offspring. Two parents are selected through tournament selection. During crossover, the corresponding nodes have the probability of P_c to swap each other. After crossover, two new individuals are produced and moved to the next generation.

Mutation just needs one individual. All data in the gene except NT_i have the mutation rate of P_m to change randomly. After mutation, a new individual is created. There are two kinds of mutations in GNP: connection mutation and function mutation. Connection mutation changes the connections between nodes, in concrete, the values of C_{i1}, C_{i2}... are altered. Function mutation means the function of the individual is changed, which implies the value of ID_i is updated.

3. GNP-APG

In this section, the concepts of the improved GNP-APG for agent control are described in detail.

3·1 Basic structure of GNP-APG

It has been mentioned in section 1 that GNP-APG has the outside memory and exchanges the information with it, which makes the basic structure of GNP-APG a little different from GNP. Figure 3 shows the basic structure of GNP-APG. Compared with Figure 1, the outside memory is added. In Figure 3, although there are only *node 8* and *node 9* pointing to the memory, in fact, all the processing nodes can read from the memory like white arrows and write to the memory like black arrows. These other arrows are just omitted in order to keep the figure clear. The memory structure is different from previous GNP-APG. It consists of two parts. One is read only and shared by all individuals, which contains the basic actions of agents depending on the problem. In other words, the basic action set consists of two sets which are called as terminal set and function set like GP. The other is named subprogram pool which can be read and written. Each individual has its own subprogram pool used to store subprograms when the procedure of the program generation is carrying out. In the individual evaluation procedure, each subprogram in the pool is picked up as one program. Then, the programs are evaluated in order to calculate the fitness of the individual.

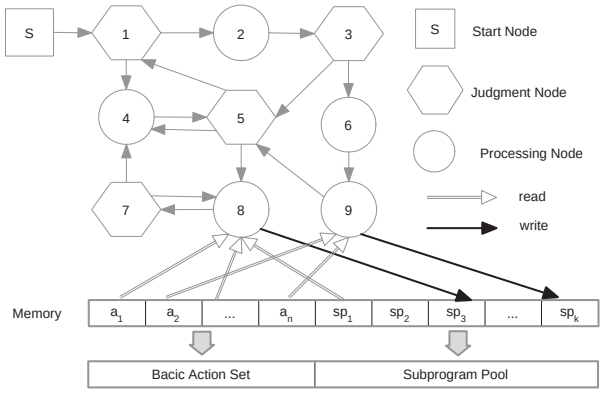

Fig. 3 Basic structure of GNP-APG

In addition, there are also three kinds of nodes in GNP-APG, but the roles of judgment nodes and processing nodes are not the same as GNP. In GNP-APG, the role of judgment nodes is to select the next node in turn. For example, suppose a judgment node has two branches, and when the judgment node is visited for the first time, it selects the node connected from the first branch, and selects the second branch for the second visit, then return to the first branch for the third visit, and so on. By this way, all branches of the judgment node can be selected, which make the sufficient search of the graph structure. In one individual, there are several kinds of judgment nodes, such as 2-branches, 3-branches and 4-branches in order to connect them to enough processing nodes and give different probabilities by which the next nodes are selected. The role of processing nodes is to create programs. A processing node gets the basic actions or subprograms from the memory, then combines them following some mapping rules to create a more complex subprogram and puts it to the subprogram pool.

Figure 4 shows the representation of GNP-APG. The chromosome of GNP-APG has two more segments compared to GNP. One segment contains $R_{i1}...R_{ip}$ which means the addresses of the reading memory of node i. If $p = 4$, the node reads four basic actions or subprograms from the memory. For example, *node 2* is this kind of processing node in the figure. The other segment contains $W_{i1}...W_{iq}$ which means the addresses of the writing memory of node i. Usually, the subprogram only needs to be stored once, so q always equals to 1. In the figure, the write address of

node 9 is 20.

	Node Gene	Connection Gene	Read Address	Write Address
node i	NT_i ID_i	C_{i1} C_{i2} ...C_{ik}	R_{i1} R_{i2} ...R_{ip}	W_i
node 0	0 0	1		
node 1	1 2	2 4		
node 2	2 1	3	3 4 12 10	16
...	...	...	...	...
node 5	1 3	1 4 8		
...	...	...	...	...
node 9	2 2	5	2 9	20

Fig. 4 Representation of GNP-APG

3·2 Procedure of program generation

Usually, a program consists of sequential, conditional and loop statements. Since the created program will repeat several times in the proposed method as a kind of loop, so only sequential and conditional statements are necessary. In the proposed method, two key words “SUB2” and “IF” are used to represent sequential and conditional statements, respectively. The procedure of the program generation of the proposed method is also different from previous GNP-APG.

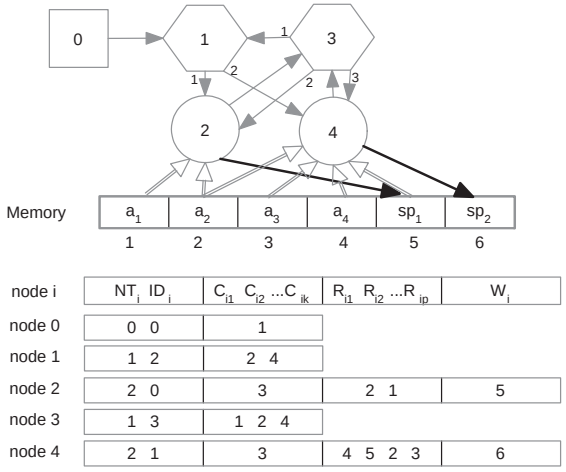

Fig. 5 Small but complete example of GNP-APG

Table 1 Functions of Processing Nodes

ID	Name	Function
0	SUB2	Create sequential statement
1	IF	Create conditional statement

Figure 5 is a small but complete example of GNP-APG. The number on the arrows means the index of branches. Table 1 shows the function of processing nodes, Table 2 describes the function of judgment nodes and Table 3 represents the argument number of actions. The detail of each

Table 2 Functions of Judgment Nodes

ID	Function
2	2-branches
3	3-branches

node is shown in the figure. For example, NT_2 and ID_2 of *node 2* are 2 and 0, respectively, so node 2 is a processing node which is used to create sequential statement. It connects to *node 3* according to the connection gene. Moreover, it reads from location 2 and 1 of the memory, then write to location 5 of the memory depending on the read addresses and write address, respectively.

Table 3 Argument numbers of actions

Action	Argument number
a_1	0
a_2	0
a_3	0
a_4	3

Table 4 Pseudocode of sp_2

```
Suppose a4 has three judgment results v1, v2 and v3
Let re be the return value of a4
if re == v1:
    a2;
    a1;
else if re == v2:
    a2;
else if re == v3:
    a3
```

Beginning from the start node, *node 1* is the first node to execute. *Node 1* is a judgment node, and its first branch connects to *node 2*, so the next node to visit is *node 2*. As the node type of *node 2* is 2 and its identity is 0, it is a processing node and the key word is “SUB2”. The key word “SUB2” is used to create the sequential statement. Suppose there are two actions “action1” and “action2” obtained from the memory, and the rule to create the statement is “SUB2(action1, action2)” which means that the agent take “action2” followed by “action1”. The read addresses of *node 2* are 2 and 1, so actions a_2 and a_1 are picked up. According to the rule, a subprogram “SUB2(a_2, a_1)” is generated. After that, the subprogram is written to location 5 of the memory, and sp_1 changes to “SUB2(a_2, a_1)”, since the write address of *node 2* is 5. Then, the next node becomes *node 3*. Like the same as *node 1*, *node 3* is a judgment node, and it selects the branch depending on the times of the visits. This is the first time for *node 3* to visit, so it chooses *node 1* as the next node to visit. At this time, *node 1* determines *node 4* to execute because this is the second visit to *node 1*. *Node*

4 is a processing node and used to create conditional statements. The key word of the conditional statement is "IF", and the rule is "IF(action1, action2, action3, ...)". In this statement, "action1" should be a judgment action which is like a function in the function set of GP. The argument number of "action1" determines the number of the other actions. In this case, *node 4* gets a_4 as a judgment action. From Table 3, a_4 needs three arguments, so sp_1, a_2 and a_3 are picked up depending on the read addresses of 5, 2, and 3, respectively. Then, a subprogram "IF(a_4, sp_1, a_2, a_3)" is created and stored at location 6 of the memory, because the write address of *node 4* is 6. By this way, sp_2 changes to "IF(a_4, sp_1, a_2, a_3)". For sp_1 has become "SUB2(a_2, a_1)", the entire representation of sp_2 is "IF(a_4, SUB2(a_2, a_1), a_2, a_3)". It can be seen as a program including sequential and conditional statements. The pseudocode of sp_2 is shown in Table 4. GNP-APG repeats this kind of procedure until the predefined number of transitions is reached. After finishing the procedure, the subprograms of the subprogram pool are picked up as the programs to solve the problem.

3·3 Flow chart of GNP-APG

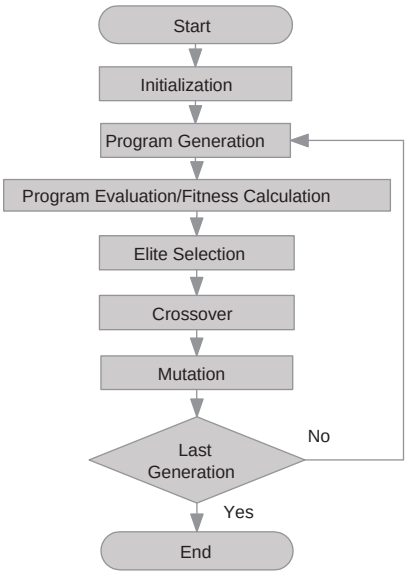

Fig. 6 Flowchart of GNP-APG

Figure 6 shows the flowchart of GNP-APG.

1) Parameters are set. Hundreds of GNP-APG individuals are generated randomly.
2) Each individual of the population creates programs by performing the above procedure.
3) Each individual evaluates the programs in its own subprogram pool. The best evaluation value among these programs becomes the fitness value of the individual.
4) The individual which has the highest fitness value is copied to the next generation directly.
5) Two individuals are selected by tournament selection as parents. These individuals exchange their nodes by the crossover rate. After the crossover, not only the connections are exchanged, but also the addresses of reading memory and writing memory are exchanged. So two new individuals are generated and moved to the next generation.
6) One individual is selected by tournament selection as a parent. The individual randomly changes its gene according to the mutation rate, i.e., the connections, the addresses of the reading memory and the writing memory randomly change their values. After mutation, one new individual is generated and moved to the next generation.
7) Determine whether it is the last generation. If the answer is yes, then the algorithm ends, otherwise, go to step 2.

3·4 Advantages of GNP-APG

GNP-APG uses genotype-phenotype mapping to create programs, which enables the genotype search without limitation, while still keeping the legality of the program. The following shows an example of program generated by the proposed method.

['SUB2', ['JF', ['JL', ['SUB2', ['HD', ['TL'], ['JF', ['TR'], ['JL', ['TR'], ['TL'], ['TL']], ['ST']], ['TD', ['JR', ['THD', ['JB', ['TL'], ['HD', ['MF'], ['TR'], ['TR'], ['TR'], ['ST']], ['TR']], ['TL'], ['ST'], ['TR'], ['ST']], ['ST'], ['TR']], ['TL'], ['MF'], ['JL', ['JB', ['TL'], ['HD', ['MF'], ['TR'], ['TR'], ['TR'], ['ST']], ['TR']], ['TL'], ['TR']], ['HD', ['MF'], ['TR'], ['TR'], ['TR'], ['ST']]], ['ST'], ['TR']], ['JL', ['TR'], ['TL'], ['TL']]], ['TL'], ['TR']], ['SUB2', ['JF', ['JL', ['JB', ['TL'], ['HD', ['MF'], ['TR'], ['TR'], ['TR'], ['ST']], ['TR']], ['TL'], ['TR']], ['HD', ['MF'], ['TR'], ['TR'], ['TR'], ['ST']], ['TL']], ['MF']], ['TL']], ['MF']]

GNP-APG has other important advantages. These advantages rely on the structure of the algorithm.

- More solution candidates. Since the individual of GNP-APG is a program generator, it can create several candidate programs stored in the memory. These programs are selected as solution candidates and evaluated. Then, the best one is picked up as the solution. So, GNP-APG can increase the probability to find better solutions.
- Sufficient use of the graph structure. As noted in [Eto 07], the conventional GNP cannot use the whole graph structure, because the judgment nodes usually select only several specific branches. But, GNP-APG selects the branches in turn, as a result, each branch has the same chance to select. By this way, GNP-APG can make full use of the graph structure.
- Keeping the diversity of the genotype. Figure 7 shows two different GNP-APG individuals. The memory is omitted. During program generation, the transition sequence of the left individual is *node 1⇒node 2⇒node 3⇒node 1⇒node 4⇒node 3⇒ node 2*, while the sequence of the right individual is *node 1⇒node 3⇒node 2⇒node 3⇒node 4⇒node 1⇒node 2*. Though the entire sequences between two individuals

are different, the sequences of processing nodes like *node 2⇒node 4⇒node 2* are the same. As long as the read addresses and write address of both individuals are the same, the two different individual can create the same program. Since the fitness value of individuals comes from the evaluation value of the program, the individuals have the same fitness value. Then, when the selection occurs, both individuals have the same probability to be selected as a parent. In this way, GNP-APG keeps the diversity of the genotype.

- Building the building blocks and subroutines. When the procedure of the program generation is carried on, the subprograms stored in the memory might be used many times. These subprograms work as building blocks and subroutines.

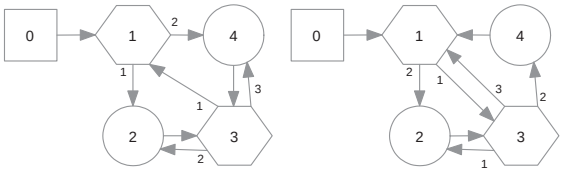

Fig. 7 Two different GNP-APG individuals with different connections

4. Simulations

In this section, the performances of the proposed method are evaluated and compared with the conventional GNP using Tileworld. In paper [Mabu 07], the performance of the conventional GNP has been compared with GP, GP-ADFs and EP, and proved better than the others. So, GNP is a reasonable method for solving the Tileworld problem.

4·1 Simulation environments

Tileworld is a famous agent-based test bed with time dependent and uncertain features, since the environment always changes and agents cannot get all the information of the environment [Pollack 90]. Tileworld consists of agents, floor, tiles, holes and obstacles. The agents need to move round the obstacles and to push all the tiles into the holes as soon as possible. Once a tile is pushed into a hole, the hole becomes the floor. A agent can only push one tile at a time.

In the training phase, 10 different Tileworlds are used to train the agents' behavior. Each world has 3 agents, 3 holes and 3 tiles. The position of obstacles, holes and agents are the same. However, the position of tiles are different from each other. Figure 8 shows the training environments.

Two kinds of changes are introduced in the testing phase. The first kind is to change the location of holes. The upper part of Figure 9 shows this kind of change. The difference is the position of the holes. The holes are moved a little farther away from the tiles compared with the training phase. The other kind is to change the location of agents which are shown in the lower part of Figure 9. The difference is the position of the agents. The agents are moved to the corner of the environments. These kinds of changes are applied to the training environments, so there are 10×2 testing environments.

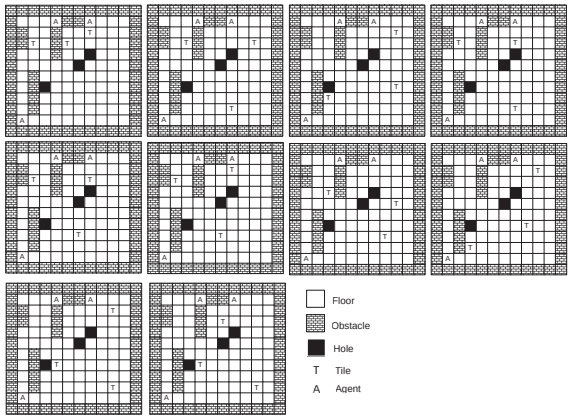

Fig. 8 Tileworlds for training phase

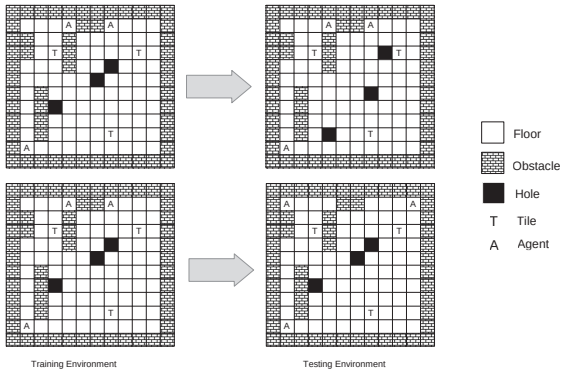

Fig. 9 Changes of Tileworld for testing phase

4·2 Simulation configuration

Table 5 Basic Action Set

Symbol	Action	Argument
JF	Judge Forward	5
JB	Judge Backward	5
JL	Judge Left	5
JR	Judge Right	5
JT	Judge the nearest Tile	5
JH	Judge the nearest Hole	5
JHT	Judge the nearest Hole from the nearest Tile	5
JST	Judge the second nearest Tile	5
MF	Move Forward	0
TR	Turn Right	0
TL	Turn Left	0
ST	Stay	0

The basic action set of GNP-APG is described in Table 5, the functions of processing nodes is the same as

Table 6 Functions of Judgment Nodes in Simulations

ID	Function
4	4-branches
6	6-branches
8	8-branches

Table 1 and the functions of judgment nodes are shown in Table 6. JF, JB, JL and JR return the floor, obstacle, tile, hole or agent; JT, JH, JHT and JST return the forward, backward, left, right or nothing. MF, TR, TL and ST do not have the return value. While JF, JB, JL, JR, JT, JH, JHT and JST are 8 kinds of judgment actions, while MF, TR, TL and ST are 4 kinds of processing actions in GNP.

Each individual of GNP-APG contains 60 nodes including 15 judgment nodes (5 for each kind) and 45 processing nodes (5 for "SUB2" processing node, 40 for "IF" processing node). Each individual of GNP has also 60 nodes (5 for each kind of nodes).

The parameters used in simulations is described in Table 7. The population size is 301, and during the evolution procedure, the best individual is copied to the next population. 180 individuals are generated through crossover, while 120 individuals are created through mutation. The crossover rate is 0.2 and mutation rate is 0.03. The program needs to iterate 500 generations. During the program generation, the maximum number of transitions is 60, and maximum length of programs is 6000 bytes in GNP-APG. The number of subprograms is 12.

Table 7 Parameters of simulations

Parameter Name	GNP-APG	GNP
Number of Individuals	301	301
Number of Elites	1	1
Crossover Size	120	120
Crossover Rate P_c	0.2	0.2
Mutation Size	180	180
Mutation Rate P_m	0.03	0.03
Number of Generations	500	500
Number of transitions	60	
Maximum length of program	6000 bytes	
Number of subprograms	12	

The fitness function of each Tileworld is defined by Eq.(1).

$$\begin{aligned} Fitness &= C_{tile} \times DroppedTile \\ &+ C_{dist} \times \sum_{t \in T} (InitDi(t) - FinDi(t)) \\ &+ C_{stp} \times (TotalStep - UsedStep) \end{aligned} \quad (1)$$

where, $DroppedTile$ is the number of tiles the agents have pushed into the holes. $InitDi(t)$ is the initial distance between t_{th} tile and the nearest hole, while $FinDi(t)$ represents the final distance between t_{th} tile and the nearest hole. T is the set of suffixes of tiles. $TotalStep$ is a predefined maximal number of actions all the agents can take, and $UsedStep$ represents the number of actions which the agents have taken. C_{tile}, C_{dist} and C_{stp} are rewards when an agent drops the tile into the hole, moves the tile near to the hole and takes less steps than the total steps when finishing the job. In the simulations, C_{tile}, C_{dist}, C_{stp} and $TotalStep$ are set at 100, 20, 1 and 180 (60 numbers of actions for each agent), respectively. In the simulations, the average of the fitness values over ten Tileworlds is calculated to show the performances of GNP-APG and GNP.

4·3 Simulation results

Figure 10 shows the average fitness value of the best training results of GNP-APG and the conventional GNP over 50 random seeds. In the 500th generation, the average of the best fitness values of GNP-APG and GNP are 319.71 and 297.30, respectively. At first, the fitness values of GNP-APG and the conventional GNP increase at the same speed. Then, after 50 generations, the evolving speed of both methods decrease. But, the evolving speed of GNP-APG is higher than the conventional GNP, since GNP-APG can use of the graph structure fully and keep the diversity of the genotype. At last, GNP-APG can gain a larger fitness value than the conventional GNP obviously.

Figure 11 shows the average length of the best programs of GNP-APG in the training phase. As noted in [Langdon 97], the growth of the length of the program is inherent in the proposed methods, since the length of the program is not fixed but varying. Figure 11 confirms this tendency, i.e., the average length over the best programs grows from 1954.1 to 4001.1. But, the length does not always increase, instead, it is fluctuated during the evolution. It is found from Figure 11 that the growth of the length is far below the maximal allowable value in the last generation. From this point of view, the proposed method does not suffer from the bloating problem.

Figure 12 and Figure 13 show the average fitness value of each Tileworld of GNP-APG and the conventional GNP in the test cases, where the best 50 individuals from the training phase are used to test these new environments. When changing the location of holes, the average fitness values are 58.2 in GNP-APG and 25.2 in the conventional GNP, respectively. It can be seen from Figure 12 that GNP-APG performed a little worse than the conventional GNP only in world 1. While GNP-APG is much better than the conventional GNP in world 9 and 10. When changing the location of agents, the average fitness values are 170.7 in GNP-APG and 84.1 in the conventional GNP, respectively. It is also found from Figure 13 that

GNP-APG can get more rewards in each world, especially in world 2, 6, 7 and 10, where the fitness values are twice than the conventional GNP. So, the proposed method has more generalization ability than the conventional GNP to deal with time dependent environment problems, which means the robustness of the proposed method is better than the conventional GNP.

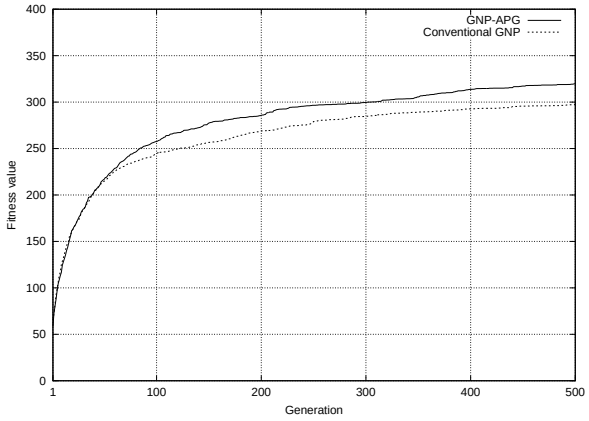

Fig. 10 Simulation result of training phase

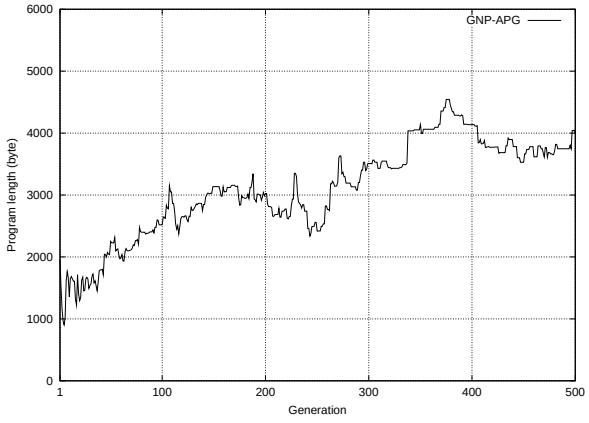

Fig. 11 Program length of training phase

In addition, another set of Tileworld with different obstacle configurations are used to verify the performance on GNP-APG and the conventional GNP. The ten different types of obstacles are shown in Figure 14. In each type, there are ten worlds with different configurations on holes or tiles. So, there are totally 100 worlds. Figure 15 shows the average of the best fitness values over 10 random seeds and 100 worlds. It is found from Figure 15 that GNP-APG can also have better performance than the conventional GNP in many kinds of worlds.

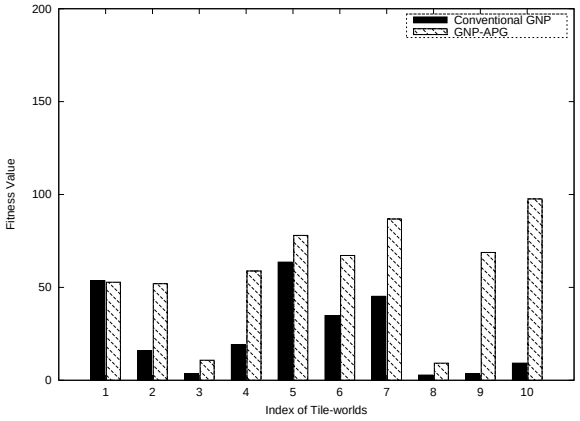

Fig. 12 Simulation result of testing phase when the location of holes is changed

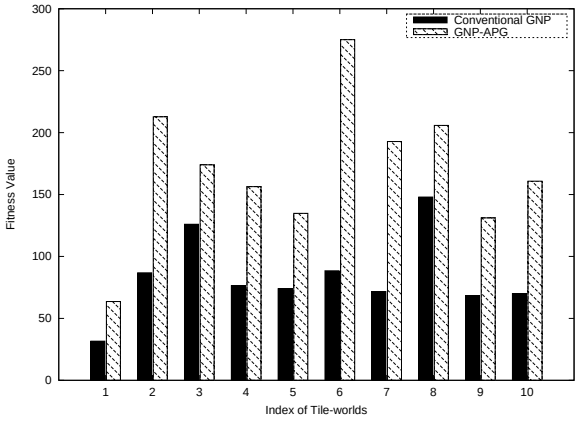

Fig. 13 Simulation result of testing phase when the location of agents is changed

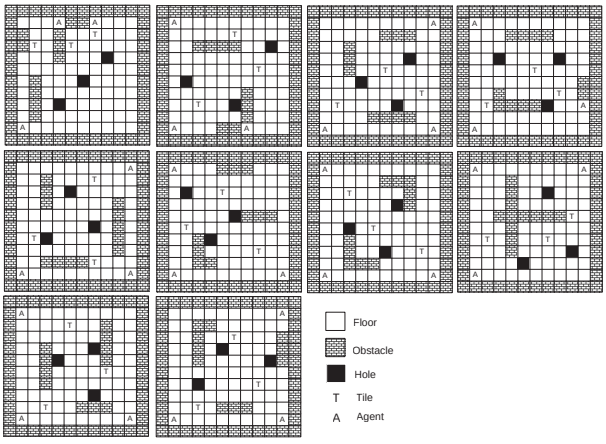

Fig. 14 Tileworld with different obstacle configurations

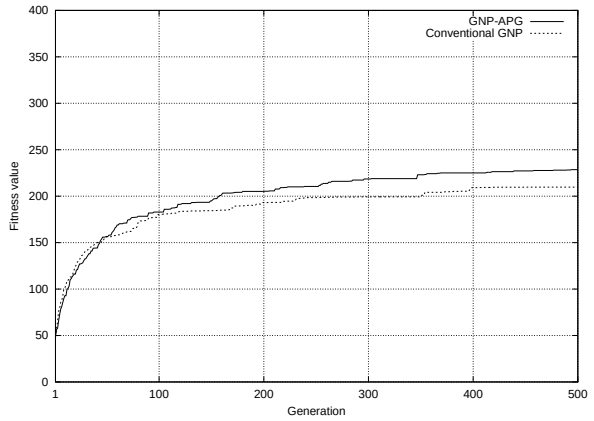

Fig. 15 Simulation result of Tileworld with different obstacle configurations

4·4 Simulation Analysis

From the training results, GNP-APG could get higher fitness values than the conventional GNP. Because the individual of GNP-APG is a program generator, it can create several candidate programs stored in the memory, and evaluate them to pick up the best one, which increases the probability to find better solutions. Besides, GNP-APG can sufficiently use the graph structure since the branches of judgment nodes in GNP-APG have the same probability to select, which also helps to improve the performance of the proposed method. Table 8 shows the mean, standard deviation and p-value of the training fitness results. The p-value of t-test of the mean fitness values is much smaller than 0.05 which means the means of two groups are statistically different from each other. The conclusions derived from the above results are convincing.

Table 8 Statistical fitness values of training phase

	GNP-APG	the conventional GNP
Mean	3197.12	2973.04
Standard deviation	385.65	402.36
p-valve (t-test)	5.89E-06	

Moreover, it is found from the testing results that GNP-APG is much better than the conventional GNP. Because there are 8 kinds of judgment nodes and 4 kinds of processing nodes in the conventional GNP, the proportion of judgment nodes and processing nodes is 2:1, which means the average number of judgment nodes needed for processing is 2. In other words, the agent will judge two kinds of situations, then take an action. But, many kinds of Tileworlds are to be dealt with in many cases, so two judgments are not enough. While there are 40 "IF" processing nodes in GNP-APG, then the program generated by GNP-APG will contain many judgments. So, GNP-APG works better than the conventional GNP in the testing phase. Table 9 and Table 10 describe the statistical values of the fitness in the testing phase, where the location of the holes and agents are changed, respectively. In each testing case, the p-value of t-test of the mean fitness values is much smaller than 0.05, which implies two groups are significantly different.

Table 9 Statistical fitness values of testing phase when the location of holes is changed

	GNP-APG	the conventional GNP
Mean	582.00	251.60
Standard deviation	421.12	289.37
p-valve (t-test)	0.0064	

Table 10 Statistic fitness values of testing phase when the location of agents is changed

	GNP-APG	the conventional GNP
Mean	1707.18	841.22
Standard deviation	609.17	497.97
p-valve (t-test)	0.0008	

4·5 Parameters Discussion

GNP-APG has more parameters than GNP, i.e., the number of transitions, maximum length of program and the number of the subprograms. These parameters influence the length and the fitness value of the program. Simulations on Figure 8 are used to study the effect of different parameters.

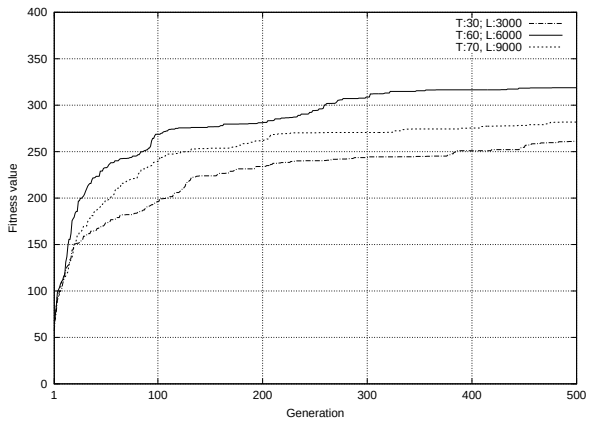

Fig. 16 Fitness value of programs with different number of transitions and different maximum length of programs

The number of transitions and the maximum length of the program are used to work together to control the size of the program. If the number of transitions and the maximum length of the program are small, the length of program is small, and GNP-APG cannot generate a complicate program to deal with complicated environments, so the fitness becomes low; otherwise, if they are large, the

search space increases rapidly, then it is also very hard to find a good solution, so the fitness value becomes also low. Figure 16 and Figure 17 show the average fitness values and lengths of programs by different parameter settings over 10 random seeds. T and L mean the number of transitions and the maximum length of programs, respectively. It is found from Figure 16 and Figure 17 that when T equals 60 and L equals 6000, the algorithm can get the highest fitness value and proper size of the programs, which confirms the previous description.

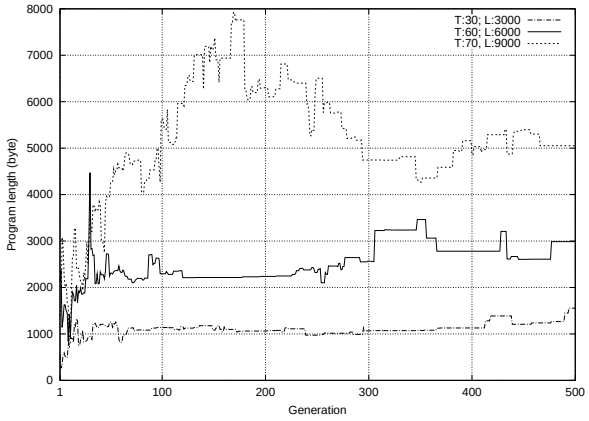

Fig. 17 Length of programs with different number of transitions and different maximum length of programs

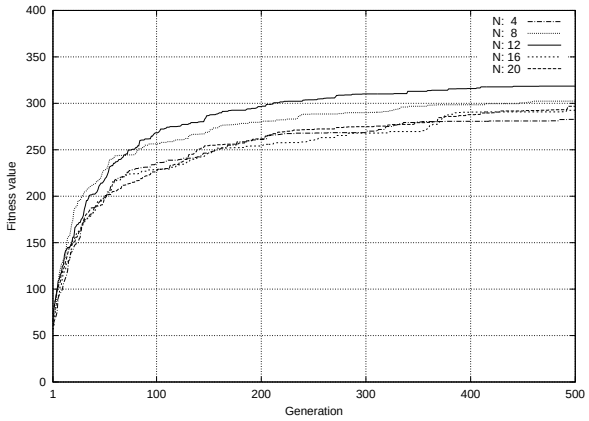

Fig. 18 Fitness value of programs with different number of subprograms

The number of subprograms is an important parameter for the fitness values and the length of the programs. If the number of subprograms is small, the old subprograms are mostly replaced by the new one, and some useful subprograms may be lost, so the fitness becomes low, besides the processing nodes always read the subprogram from the same memory, as a result, the length of the program will increase quickly. On the other hand, if the number of subprograms is large, some subprograms may not be selected to generate statements, by which the fitness values are also affected. But, as the processing nodes read different subprograms from different memories, the growth rate of the length of the program will decrease. Figure 18 and Figure 19 describe the average fitness values and lengths of programs by different number of subprograms over 10 random seeds. N means the number of subprograms. It is found from Figure 18 that when N equals 12, the algorithms can get the highest performance compared with other settings. Figure 19 confirms the tendency described before, i.e., when N is small, the length becomes large, while when N is large, the length becomes small.

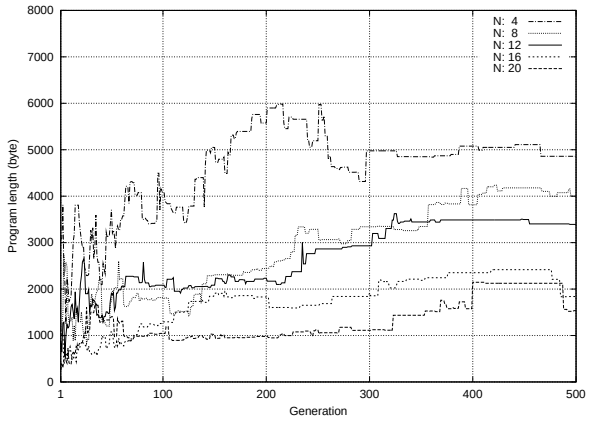

Fig. 19 Length of programs with different number of subprograms

5. Conclusion

In this paper, automatic program generation with Genetic Network Programming has been proposed and applied to the Tileworld problem for agent control. The proposed method introduces two functions ”IF” and ”SUB2” to create conditional statements and sequential statements which are two basic statements in a program. The proposed method introduces a genotype-phenotype mapping technology to generate legal programs. Through the transition of nodes, it does not only create simple statements, but also create some complex programs to deal with the problem. Since the proposed method has advantages of using graph structures fully, keeping the diversity of the genotype and using the building blocks and subroutines, it can find better solutions than the conventional GNP. The simulations confirms that the proposed method is more robust than the conventional GNP. It is also found that although the proposed method improves the robustness, the fitness value of the testing phase is still much lower than the training phase, which means the proposed method still suffers from the overfitting problem.

In the future, we will introduce subroutines into GNP-

APG to improve the performances of the proposed method and find a way to solve the overfitting problem.

♢ References ♢

[Holland 75] J. H. Holland, *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, 1975.

[Goldberg 89] D. E. Goldberg, *Genetic Algorithm in Search Optimization and Machine Learning*, Reading, MA: Addison-Wesley, 1989.

[Koza 92] J. R. Koza, *Genetic Programming, on the Programming of Computers by Means of Natural Selection*, Cambrige, MA: MIT Press, 1992.

[Koza 94] J. R. Koza, *Genetic Programming II, Automatic Discovery of Reusable Programs*, Cambrige, MA: MIT Press, 1994.

[Koza 99] J. R. Koza, *Genetic Programming III, Darwinian Invention and Problem Solving*, San Mateo, CA: Morgan Kaufmann, 1999.

[Mabu 07] S. Mabu, K. Hirasawa and J. Hu, "A Graph-Based Evolutionary Algorithm: Genetic Network Programming (GNP) and Its extension Using Reinforcement Learning", *Evolutionary Computation*, Vol.15, No.3, pp.369–398, 2007.

[Eguchi 06] T. Eguchi, K. Hirasawa, J. Hu and N. Ota, "A study of Evolutionary Multiagent Models Based on Symbiosis", *IEEE Trans. on Systems, Man and Cybernetics, Part B*, Vol.35, No.1, pp.179–193, 2006.

[Hirasawa 08] K. Hirasawa, T. Eguchi, J. Zhou, L. Yu, J. Hu and S. Markon, "A Double-Deck Elevator Group Supervisory Control System Using Genetic Network Programming", *IEEE Transactions on Systems, Man and Cybernetics, Part C*, Vol.38, No.4, pp.535-550, July 2008.

[Chen 09] Y. Chen, S. Mabu, K. Shimada and K. Hirasawa, "A genetic network programming with learning approach for enhanced stock trading model", *Expert System with Applications*, Vol.36, No.10, pp.12537-12546, 2009.

[Shimada 06] K. Shimada, K. Hirasawa and J. Hu, "Genetic Network Programming with Acquisition Mechanisms of Association Rules", *Journal of Advanced Computational Intelligence and Intelligent Informatics*, Vol.10, No.1 pp.102-111, 2006.

[Zhou 10] H. Zhou, S. Mabu, W. Wei, K. Shimada and K. Hirasawa, "Time Related Class Association Rule Mining and Its Application to Traffic Prediction", *IEEJ Transactions on Electronics, Information and Systems*, Vol.130, No.2, pp.289-301, 2010.

[Ferreira 01] C. Ferreira, "Gene Expression Programming: a New Adaptive Algorithm for Solving Problems", *Complex Systems*, Vol.13, No.2, pp. 87-129, 2001.

[Ferreira 02] C. Ferreira, *Expression Programming:Mathematical Modeling by an Artificial Intelligence*, Angra do Heroismo, Portugal, 2002.

[O'Neill 01] M. O'Neill and C. Ryan, "Grammatical evolution", *IEEE Transactions on Evolutionary Computation*, Vol.5, No.4, pp.349-358, 2001.

[O'Neill 03] M. O'Neill and C. Ryan, *Grammatical Evolution. Evolutionary Automatic Programming in an Arbitrary Language*, Kluwer Academic Publishers, 2003.

[Mabu 05] S. Mabu, K. Hirasawa, Y. Matsuya and J. Hu, "Genetic Network Programming for Automatic Program Generation", *Journal of Advanced Computational Intelligence and Intelligent Informatics*, Vol.9, No.4, pp.430-436, 2005.

[Mabu 09] S. Mabu and K. Hirasawa, "Evolving plural programs by genetic network programming with multi-start nodes", In Proc. of the *IEEE International Conference on Systems, Man and Cybernetics*, pp.1382-1387, San Antonio, TX, 2009.

[Pollack 90] M. E. Pollack and M. Ringuette, "Introducing the Tileworld: Experimentally evaluating agent architectures", In Proc. of the *Eighth National Conference on Artificial Intelligence (AAAI-90)*, pp.183-189, Boston, MA, 1990.

[Banzhaf 94] W. Banzhaf, "Genotype-phenotype-mapping and neutral variation – A case study in Genetic Programming", *Lecture Notes in Computer Science*, Vol.866, pp.322-332, 1994.

[Eto 07] S. Eto, S. Mabu, K. Hirasawa and T. Huruzuki, "Genetic Network Programming with control nodes", In Proc. of the *IEEE Congress on Evolutionary Computation, 2007, CEC 2007*, pp.1023-1028, Sept. 2007.

[Langdon 97] W. B. Langdon and R. Poli, "Fitness causes bloat", *Technical Report CSPR-97-09*, University of Birmingham, Birmingham, UK, 1997.

〔担当委員：半田 久志〕

Received August 5, 2010.

Author's Profile

Bing Li

He received B.E. degree from the Institute of Software, Nanjing University, China, in 2008. He received the M.E. degree from the Graduate School of Information, Production and Systems, Waseda University, Japan in 2010. Since September of 2010, he has been a doctor candidate of Graduate School of Information, Production and Systems, Waseda University. He is a student member of IEEE and the Japanese Society for Evolutionary Computation.

Shingo Mabu (Member)

He received the B.E. and M.E. degree in Electrical Engineering from Kyushu University, Japan in 2001 and 2003, respectively. He received Ph.D. degree from Waseda University, Japan in 2006. From 2006 to 2007, he was a visiting lecturer at Waseda University. Since 2007, he is an assistant professor at Graduate School of Information, Production and Systems, Waseda University.

Kotaro Hirasawa

He received the B.E. and M.E. degrees from Kyushu University, Japan in 1964 and 1966, respectively. From 1966 to 1992, he worked at Hitachi Ltd. as Vice President of Hitachi Research Laboratory. From December of 1992 to August of 2002, he was a professor at Graduate School of Information Science and Electrical Engineering of Kyushu University. Since September 2002, he is a professor at Graduate School of Information, Production and Systems, Waseda University. Dr. Hirasawa is a member of the Society of Instrument and Control Engineers, the Institute of Electrical Engineers of Japan and IEEE.