A Variable Size Mechanism of Distributed Graph Programs and Its Performance Evaluation in Agent Control Problems

Shingo Mabu [a,*] Kotaro Hirasawa [b] Masanao Obayashi [a]
Takashi Kuremoto [a]

[a] *Graduate School of Science and Engineering, Yamaguchi University, Tokiwadai 2-16-1, Ube, Yamaguchi, 755-8611, Japan*

[b] *Information, Production and Systems Research Center, Waseda University, Hibikino 2-2, Kitakyushu, Fukuoka, 808-0135, Japan*

Abstract

Genetic Algorithm (GA) and Genetic Programming (GP) are typical evolutionary algorithms using string and tree structures, respectively, and there have been many studies on the extension of GA and GP. How to represent solutions, e.g., strings, trees, graphs, etc., is one of the important research topics and Genetic Network Programming (GNP) has been proposed as one of the graph-based evolutionary algorithms. GNP represents its solutions using directed graph structures and has been applied to many applications. However, when GNP is applied to complex real world systems, large size of the programs is needed to represent various kinds of control rules. In this case, the efficiency of evolution and the performance of the systems may decrease due to its huge structures. Therefore, we have been studied distributed GNP based on the idea of divide and conquer, where the programs are divided into several subprograms and they cooperatively control whole tasks. However, because the previous work divided a program into some subprograms with the same size, it cannot adjust the sizes of the subprograms depending on the problems. Therefore, in this paper, an efficient evolutionary algorithm of variable size distributed GNP is proposed and its performance is evaluated by the tileworld problem that is one of the benchmark problems of multiagent systems in dynamic environments. The simulation results show that the proposed method obtains better fitness and generalization abilities than the method without variable size mechanism.

Key words: evolutionary computation, directed graph, distributed structure, variable size, reinforcement learning, decision making

* Corresponding author. Tel.: +81-836-85-9519; fax: +81-836-85-9519.
Email address: `mabu@yamaguchi-u.ac.jp` (Shingo Mabu).

1 Introduction

Genetic Algorithm (GA) [1] and Genetic Programming (GP) [2,3] are typical evolutionary algorithms that have been widely studied. A large number of real world applications have been also studied such as robot programming [4], financial problems [5–7] and network security systems [8,9].

In order to create reliable systems using evolutionary algorithms, the program structures (phenotype and genotype representations) and how to efficiently evolve them are important issues. Therefore, as an extended algorithm of GA and GP, Genetic Network Programming (GNP) and its extension using reinforcement learning (GNP-RL) [10,11] have been proposed and applied to many applications [12,13]. The program of GNP is represented by directed graph structures and evolved by crossover and mutation. Originally, GNP was proposed because graph structures may have better representation abilities than strings and trees. In addition, human brain also has a graph (network) structure, so some inherent abilities may be involved in the graph structure. In fact, the graph structure has some advantages such as 1) reusability of nodes and 2) applicability to dynamic environments. Actually, GNP has the following features. 1) The directed graph structure automatically generates some repetitive processes like subroutines, and reuses nodes repeatedly during the node transition, which contributes to creating programs with compact structures. 2) Once GNP starts its node transition from the start node, GNP executes judgment nodes (if-then functions) and processing nodes (action functions) according to the connections between nodes without any terminal nodes. Therefore, the node transition implicitly memorizes the history of judgment and actions, which contributes to the decision making in dynamic environments because GNP can make decisions based not only on the current, but also the past information.

Evolutionary Programming (EP) [14,15] is a graph-based evolutionary algorithm to create finite state machines (FSMs) automatically, but the characteristics of EP and GNP are different. Generally, FSM must define the state transition rules for all the combinations of states and possible inputs, thus the FSM program will become large and complex when the number of states and inputs is large. On the other hand, the evolution of GNP selects only the necessary nodes by changing connections between nodes, which means that GNP can judge only the essential inputs for making decisions at the current situations. As a result, GNP does not have to consider all the combinations of the inputs and actions, which makes the compact program structures.

When GNP is applied to complicated real world systems, large size of graph structures are needed to represent various kinds of rules for adapting to various kinds of situations. However, huge structures may decrease the efficiency of evolution, as a result, decrease the performance of the systems. Therefore, we have been studied distributed GNP [16] based on the idea of divide and conquer which is used to create the complicated systems using relatively small size of the programs [17]. In [16], the programs are divided into several subprograms which cooperatively control whole tasks, and this method is applied to a stock trading model, which shows better performances than the method without distributed structures. However, because the previous work divided a program into some subprograms with the same size, it cannot adjust the sizes of the subprograms depending on the problems. The best size of the structure is different depending on the difficulties of the task, thus the fixed size of the structure limits the representation ability of the solutions. In order to solve this problem, an efficient evolutionary algorithm of variable size distributed GNP (VS-DGNP) is proposed and its performance is evaluated by the tileworld problem [18] that is one of the benchmark problems of multiagent systems in dynamic environments. The features of the proposed method are as follows.

The complicated structure can be divided into several substructures with different sizes. Genetic operations can be executed inside each substructure and between substructures, respectively, as a result, the efficient optimization can be done. Migration of nodes from a certain substructure to another substructure is executed to optimize the sizes of the substructures appropriately.

The rest of this paper is organized as follows. In section 2, the basic structure of GNP and GNP with reinforcement learning (GNP-RL) is reviewed. In section 3, the structure of distributed GNP and how to realize variable size structure are explained. In section 4, after simulation environments and conditions are explained, the results and analysis are described. Section 5 is devoted to conclusions.

2 Review of Genetic Network Programming with Reinforcement Learning

Because the proposed Variable Size Distributed GNP (VS-DGNP) is based on GNP with Reinforcement Learning (GNP-RL), the structure of GNP-RL and its learning and evolution mechanisms are firstly reviewed. An individual of GNP-RL is represented by a directed graph structure, evolved by crossover and mutation, and the node transition rules are learned by reinforcement learning.

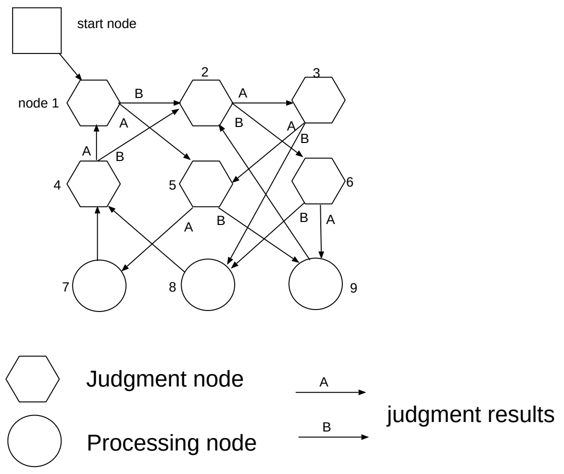


Fig. 1. Basic structure of GNP

2.1 Basic structure of original GNP

Before explaining the detailed structure of GNP-RL, the basic structure of original GNP is introduced (Fig. 1). It consists of a number of judgment nodes and processing nodes and one start node. The number of nodes are determined in advance depending on the complexity of the problems. Each judgment node has a if-then branch decision function and each processing node has an action function. For example, a judgment node examines a sensor input of agents, and a processing node determines agent actions, e.g., go forward, turn left, turn right, etc. The role of the start node is to determine the first node to be executed. Therefore, the node transition starts from the start node, and a sequence of judgments and processing is created by the connections between nodes and judgment (if-then) results.

2.2 Additional components in GNP-RL: Subnodes

The difference between the original GNP and GNP-RL is as follows. GNP-RL has subnodes in each judgment and processing node as shown in Fig. 2. In the original GNP, one function is assigned to each node and executed when the node is visited. In GNP-RL, several functions (two functions in Fig. 2) are assigned as subnodes and one of them is selected and executed by a reinforcement learning algorithm. Therefore, reinforcement learning selects better function/subnode for each node and the route of the node transition is optimized. For example, subnode 1 in Fig. 2 (a) has a judgment function of "judge forward", and subnode 2 has that of "judge right". Subnode 1 in Fig. 2 (b) has a processing function of "go forward", and subnode 2 has that of "turn left".

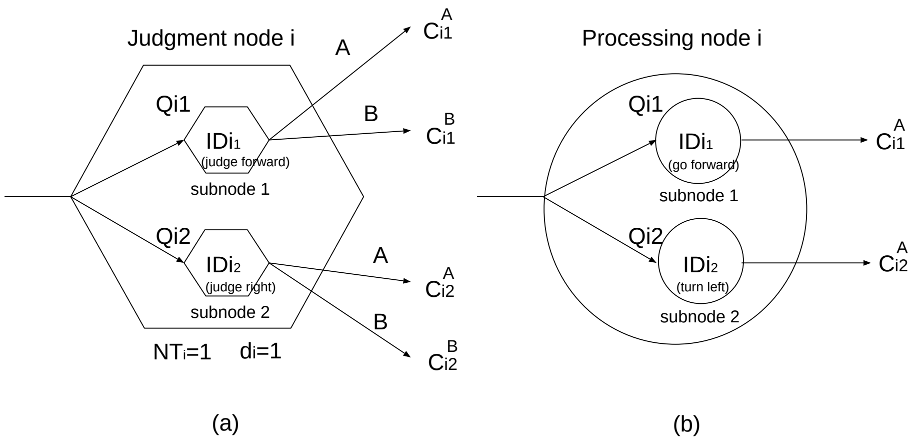


Fig. 2. Judgment node and Processing node structures

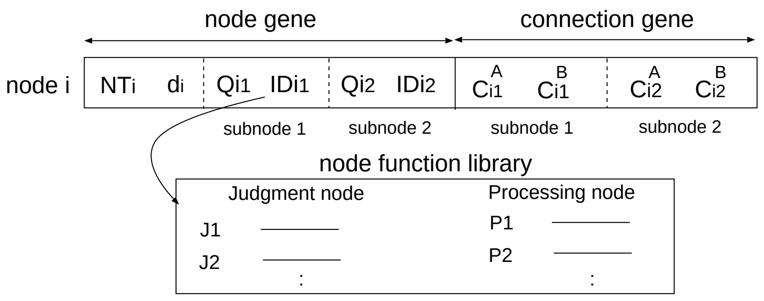


Fig. 3. Genotype representation of GNP

2.3 Genotype representation

The graph structure is realized by the combination of gene structures shown in Fig. 3. NT_i shows the node type of node i. $NT_i = 0$ means start node, $NT_i = 1$ means judgment node, and $NT_i = 2$ means processing node. d_i is a time delay which shows the time spent on the execution of node i. In this paper, d_i of judgment node is set at 1 and that of processing node is set at 5. The time delay is useful to fix the maximum number of nodes to be executed in each action step. In this paper, one action step is defined as a certain time units that an agent can use for executing judgments and processing. If five time units (time delay) are assigned to one action step, the action step ends when the time units used by the node transition become five or exceed five. That is, GNP can execute “less than five judgments nodes and one processing node” in one action step to determine an action. If “five judgment nodes” are else executed before visiting a processing node, one action step ends without taking any actions. The explanation on the time delay is explained in [10] in more detail.

$Q_{i1}, Q_{i2}, \ldots$ are Q values [19] assigned to the subnodes in node i. $ID_{i1}, ID_{i2}, \ldots$ are the node functions of the subnodes. Q value estimates the sum of the discounted rewards obtained in the future. The contents of the functions are described in the node function library. For example, $NT_i = 1$ and $ID_{i2} = 2$ show that the function of subnode 2 in node i is J_2. Node functions used in the simulations of this paper are shown in Table 1 in section 4.1.

$C_{i1}^A, C_{i1}^B, \ldots$ and $C_{i2}^A, C_{i2}^B, \ldots$ show the next node numbers connected from node i. For example, subnode 1 in node i is connected to $C_{i1}^A, C_{i1}^B, \ldots$. The superscripts A and B correspond to the judgment results, i.e., if the judgment result is A at subnode 1, the next node becomes C_{i1}^A.

2.4 Node transition and its learning algorithm

In this paper, Sarsa [19] which is one of the reinforcement learning algorithms is used for the learning of GNP-RL. The reason why Sarsa is selected is that on-policy algorithm (Sarsa) is effective to optimize state transitions considering the effect of $\varepsilon-$ greedy policy used in this paper. The node transition of GNP-RL is carried out as follows.

When the current node is a judgment node, one of the subnodes is selected according to $\varepsilon-$greedy policy, i.e., the subnode with the largest Q value is selected with the probability of $1 - \varepsilon$, otherwise one subnode is randomly selected. After executing the function of the selected subnode, the current node is transferred to the next node according to the judgment result and connection. In Fig. 2 (a), suppose subnode 1 is selected, then one of the connections (A or B) is selected according to the judgment result. When the current node is a processing node, one of the subnodes is selected by the same way as a judgment node. After executing the action of the selected subnode, the next node is determined by the connection from the subnode.

The above procedure corresponds to the basic reinforcement learning framework, i.e., each node corresponds to "state" and the selection of a subnode corresponds to "action", and the optimal state transition is learned by Sarsa.

Every time the node transition is done, Q value is updated by the following equation.

$$Q_{ip} \leftarrow Q_{ip} + \alpha \left(r + \gamma Q_{jq} - Q_{ip} \right), \tag{1}$$

where Q_{ip} shows the Q value of the selected subnode p at node i, Q_{jq} shows the Q value of the selected subnode q at the next node j and r is a reward obtained after executing subnode p of node i. $\alpha(\in (0, 1.0])$ is a learning rate, $\gamma(\in [0, 1.0])$ is a discount rate.

2.5 Genetic operation

The genetic operation is based on elite selection, tournament selection, crossover and mutation. Here, the procedure of crossover and mutation [10] is described.

- Crossover: Two individuals are selected as parents by tournament selection. Two new offspring are generated from the parents by exchanging the genetic codes. In detail, each node i is selected as a crossover node with the probability of P_c, then the genes of the crossover nodes are swapped between the parents.
- Mutation: One individual is selected by tournament selection and a new individual is generated by the following operations.

(1) Connection mutation: each connection gene $(C_{i1}^A, C_{i1}^B, \dots, C_{i2}^A, C_{i2}^B, \dots)$ is selected with the probability of P_m and changed randomly.
(2) Function mutation: each node function ID_i is selected with the probability of P_m and changed randomly.
(3) Mutation of the number of subnodes: the number of subnodes in node i is changed to another number (selected from 1 to M[1]). If the selected number is larger than the original number, one or more new subnodes with randomly assigned judgment/processing functions are added to the node depending on the selected number. If the selected number is smaller, one or more subnodes are removed from the node.

3 Proposed method: Variable size mechanism of Distributed GNP-RL

In section 2, the basic structure of GNP-RL is explained, where one individual consists of one graph structure. In this section, the distributed graph structure is firstly introduced, then a variable size mechanism of the distributed structures is proposed.

3.1 Basic structure of variable size distributed GNP-RL

Fig. 4 is a basic structure of variable size distributed GNP-RL (Hereafter, variable size distributed GNP-RL is called "VS-DGNP" for simplicity). VS-DGNP has several features: 1) assignment of nodes to subprograms and 2) normal and transferring nodes [16]. Branches from normal nodes are connected

[1] M shows the maximum number of subnodes in each node. In this paper, M is set at 4.

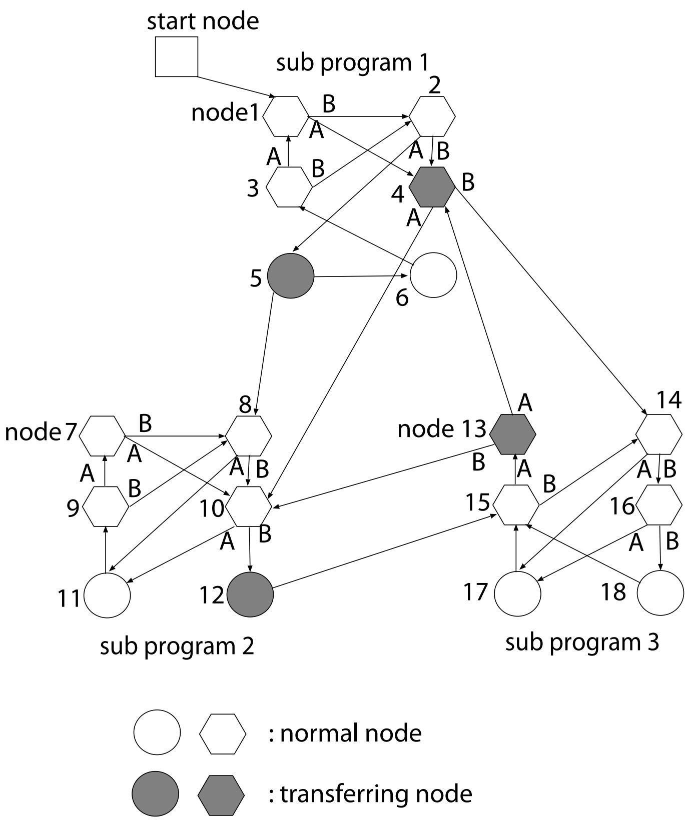


Fig. 4. Basic structure of distributed GNP-RL

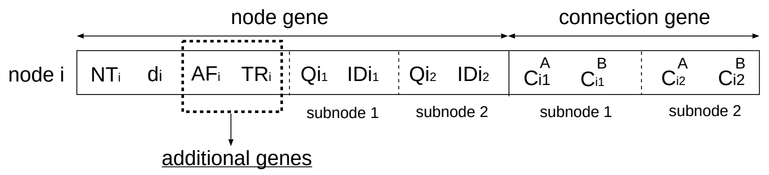


Fig. 5. Genotype representation of VS-DGNP

to other nodes in the same subprogram (internal connections), and those from transferring nodes are connected to the nodes in other subprograms (external connections). In order to represent these features, each node has two additional genes (AF_i and TR_i, where i is a node number) shown in Fig. 5. AF_i shows to which subprogram node i is assigned, and TR_i shows that node i is normal or transferring node. If a node is normal node, $TR_i = 0$, and if transferring node, $TR_i = 1$. For example, in Fig. 4, node 1 is a normal node and belongs to subprogram 1, thus $TR_i = 0$ and $AF_i = 1$. Node 12 is a transferring node and belongs to subprogram 2, thus $TR_i = 1$ and $AF_i = 2$.

3.2 Initialization of the population

Before starting the first generation, an initial population is produced according to the following rules.

(1) Node types (NT_i): The numbers of judgment and processing nodes are determined, respectively.
(2) Node affiliation (AF_i): The same number of judgment and processing nodes are assigned to each subprogram. For example, at the first generation, 1/3 of the nodes are assigned to subprogram 1, other 1/3 are assigned to subprogram 2, and other 1/3 are assigned to subprogram 3, if the number of subprograms is three. However, in the evolutionary process, AF_i is changed to optimize the sizes of the subprograms.
(3) Normal/Transferring nodes (TR_i): Each node becomes transferring node with the probability of P_t, otherwise normal node, where P_t shows a transferring node ratio which is set at 0.1 in this paper.
(4) The number of subnodes in each node is determined separately node by node in the range permitted, i.e., $1, \ldots, 4$.
(5) Node functions (ID_i): A randomly selected judgment/processing function is assigned to each subnode.
(6) Connections ($C_i^A, C_i^B, \ldots$): If node i is a normal node, the next node(s) are selected randomly from the same subprogram. If node i is a transferring node, the next node(s) are selected randomly from the other subprograms.
(7) Q values ($Q_{ip}^A, Q_{ip}^B, \ldots$): All the Q values are set at zero.

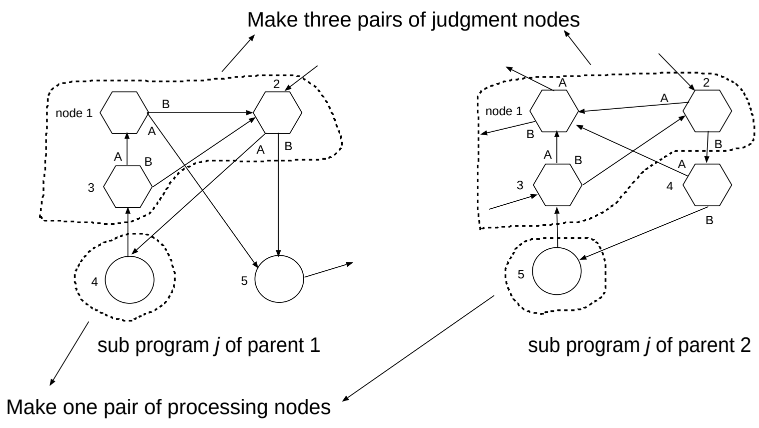
3.3 Genetic operations for optimizing the sizes of subprograms

In this subsection, crossover and mutation operators for VS-DGNP are explained.

3.3.1 Crossover

Since each subprogram has the different number of nodes, new crossover operator should be introduced.

First, two parent individuals (parent 1 and 2) are selected by tournament selection, which is the same as the crossover introduced in section 2.5. However, it should be noted that there are two important conditions in the crossover of VS-DGNP. 1) The crossover of VS-DGNP exchanges genes between subprograms with the same number, i.e., between subprogram 1 of parent 1 and that of parent 2, between subprogram 2 of parent 1 and that of parent 2, and



Make three pairs of judgment nodes
node 1
2
3
4
5
A
B
sub program j of parent 1
sub program j of parent 2
Make one pair of processing nodes

Fig. 6. An example of making node pairs between subprograms of two parents

so on. 2) The crossover exchanges the genes between the same type of nodes (NT_i), i.e., between processing & processing, or judgment & judgment, but does not between judgment and processing nodes in order to avoid too large changes by the crossover.

The crossover is carried out between two subprograms j ($j \in \{1, 2, \ldots\}$ is a subprogram number) as follows. First, the crossover of judgment nodes is executed, then that of processing nodes is executed, thus the following procedure is executed two times; one is for judgment nodes and the other is for processing nodes.

(1) Determine the maximum number of judgment (processing) nodes exchanged between two subprograms as $N = min\{N_{1j}, N_{2j}\}$, where N_{1j} is the number of judgment (processing) nodes contained in subprogram j of parent 1, and N_{2j} is that of parent 2.
(2) One judgment (processing) node is selected from parent 1 and another judgment (processing) node is selected form parent 2, then one pair of judgment (processing) nodes is made. Repeating the above procedure N times, N pairs of judgment (processing) nodes are made. Note that the nodes for crossover are selected in ascending order of the node number, and the same nodes are not selected more than once to make pairs.
(3) All the genes of each pair are exchanged with the probability of P_C

Fig. 6 shows an example of making pairs between subprograms of two parents, where three judgment nodes and one processing node in each subprogram are selected as crossover nodes.

The above procedure is repeated until the crossover of all the subprograms is completed.

When two individuals are selected as parents for crossover, the reason why the crossover is executed between subprograms with the same subprogram number is to avoid the destructive effect of the crossover. In fact, in our preliminary experiments, the crossover without considering subprograms, i.e., the crossover can be executed between nodes belonging to any subprograms, did not show better fitness than the proposed method. Because the advantage of the graph structure is re-usability of nodes, some nodes may be shared by some different node transitions, therefore even a small change may have a strong impact on the whole structure. The proposed method can separately evolve each subprogram, thus 1) localized function structures can be created, and 2) the changes in a certain subprogram do not affect the structures of the other subprograms.

3.3.2 Mutation

There are two mutation operators in VS-DGNP 1) to change normal (transferring) node to transferring (normal) node, and 2) to change subprograms that each node belongs to. These two operators are added to the original mutation introduced in section 2.5.

The mutation operator of "change normal/transferring nodes" is as follows. Each gene TR_i is selected with the probability of P_m^{tr}, then reset at 1 (= transferring) with the probability of P_t, otherwise reset at 0 (= normal).

The mutation operator of "change normal/transferring nodes" can determine the good relations between the subprograms. For example, if the environmental situation is not changed during the task, normal nodes should be used to stay at the current subprogram, and if the situation is changed, transferring nodes should be used to move to other subprograms in order to adapt to the different situation. Note that this operator can be also used in Distributed GNP without variable size mechanism (DGNP), therefore DGNP used in the simulations also uses this operator.

The mutation operator of "change subprogram numbers (change affiliation numbers)" is as follows. Each subprogram number (AF_i) is selected with the probability of P_m^{aff} and changed to a random subprogram number. If it is changed to a different subprogram number from the original one, all the branches connected from node i are initialized to satisfy the condition of normal/transferring nodes. That is, if node i is a normal node, all the branches have to be connected to the nodes of the same subprogram, and if it is a transferring node, they have to be connected to the nodes of other subprograms.

In the above change affiliation operation, two kinds of P_m^{aff} are used to control the degree of the structural changes. One is for the nodes that are used in the task execution (P_m^{aff1}), and the other is for the nodes that are not used at all (P_m^{aff2}). In the simulations, P_m^{aff1} is set at 0.002 and P_m^{aff2} is set at 0.01, which keeps the balance between exploitation and exploration of good structures. The change affiliation operator can flexibly change the size of the structure in order to create good action rules, which is one of the advantages of the proposed method, however, the good node transitions might be destroyed by the changes of the structure because some nodes are shared and reused by many different action rules (transitions). Too large changes of the structure might become the disadvantage in terms of the efficient evolution. Therefore, the mutation rates have to be carefully set at the appropriate values. Although they are determined through our experiences, the automatic adjustment of the rates is one of the issues to be studied in the future.

At the initialization, each subprogram has the same number of nodes, but as the generation goes on, the number of nodes in each subprogram is changed by the mutation of change affiliation and crossover operations. By repeating this process, VS-DGNP can create appropriate sizes of the subprograms which are suitable to solve the given tasks.

4 Simulations

To confirm the effectiveness of the proposed method, the simulations for determining agents' behavior using the Tileworld problem [18] are described in this section.

4.1 Tileworld

Tileworld is a benchmark problem for evaluating agent behaviors. Fig. 7 shows examples of tileworld environments, each of which is a 2D grid world including multi-agents, obstacles, tiles, holes and floors. Agents can move to a contiguous cell in one step. Moreover, agents can push a tile to the next cell except when an obstacle or other agent exists in the cell. When a tile is dropped into a hole, the hole and tile vanish, i.e., the hole is filled with the tile. Agents have some sensors and action abilities, and their aim is to drop many tiles into holes as fast as possible. Therefore, agents are required to use sensors and take actions properly according to their situations. Since the given sensors and simple actions are not enough to achieve tasks, agents must make clever combinations of judgments and processing.

The nodes used by the agents are shown in Table 1. The judgment nodes { JF, JB, JL, JR } return { tile, hole, obstacle, floor, agent }, and { TD, HD, THD, STD } return { forward, backward, left, right, nothing } as judgment results, like $A, B, \ldots$ in Fig. 2.

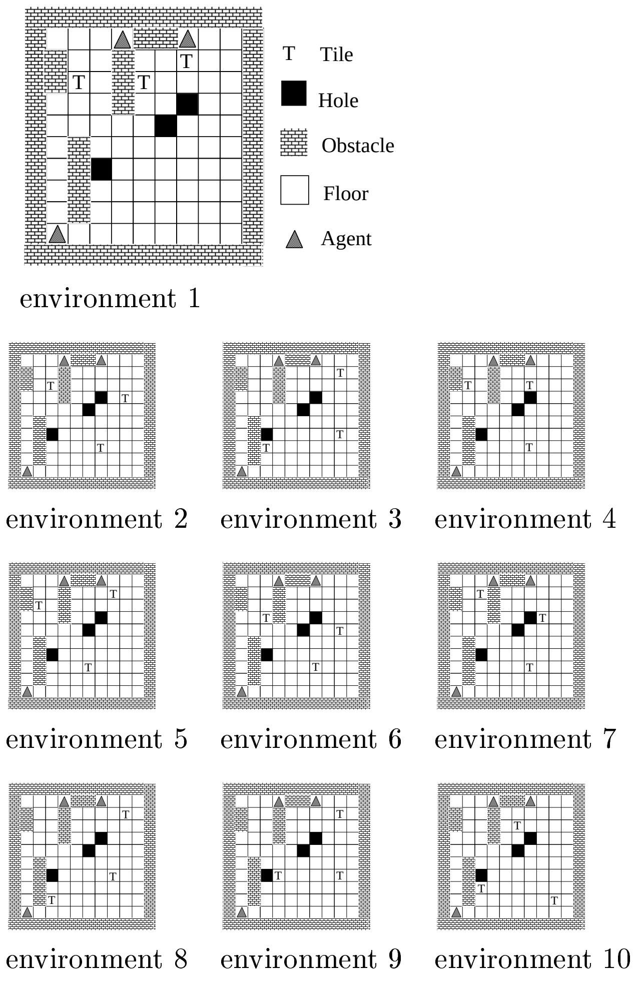


Fig. 7. Tileworld

A trial ends when the action step reaches the preassigned step, then fitness is calculated. (60 action steps are assigned to each agent in the simulations.) The following "Fitness" is used in the evolution phase and "Reward" is used in the learning phase.

$$Fitness = \sum_{env=1}^{ENV} [100 \times D_{tile} + 20 \times D_{distance} + T_{remain}] , \tag{2}$$

$$Reward = 1 \text{ (when an agent drops a tile into a hole)}, \tag{3}$$

where, env is an environment number, ENV is the total number of environments, D_{tile} is the number of dropped tiles, $D_{distance}$ shows how many cells the agents moved tiles close to the nearest holes, and T_{remain} is the remaining time when the agents dropped all the tiles into holes. The coefficients of each

Table 1
Function Set

Judgment node

J	symbol	content
J_1	JF	judge FORWARD
J_2	JB	judge BACKWARD
J_3	JL	judge LEFT side
J_4	JR	judge RIGHT side
J_5	TD	direction of the nearest TILE from the agent
J_6	HD	direction of the nearest HOLE from the agent
J_7	THD	direction of the nearest HOLE from the nearest TILE
J_8	STD	direction of the second nearest TILE from the agent

Processing node

P	symbol	content
P_1	MF	move FORWARD
P_2	TR	turn RIGHT
P_3	TL	turn LEFT
P_4	ST	stay

term (100 and 20) are the weights on the behavior of dropping a tile into a hole and moving the tile to the good direction, respectively. The remaining time is added only when the agents can drop all the tiles into holes in each environment.

In the training simulations, 10 different tileworld environments in Fig. 7 are used, and the agents have to obtain action rules which maximize the sum of the fitness values given by the 10 tileworld environments. These 10 environments have different initial positions of tiles, therefore different sequences of actions should be obtained in order to drop all the tiles into holes as fast as possible.

4.2 Conditions of DGNP and VS-DGNP

The aim of the simulations is to confirm the effectiveness of distributed GNP with variable size mechanism comparing to that without variable size mechanism. The superiority of GNP and GNP-RL has been shown in [10] comparing to Genetic Programming (GP), GP with automatic defined functions (GP with ADFs) and evolutionary programming (EP).

Table 2
Simulation conditions

	VS-DGNP	DGNP
the number of individuals	300	
the number of crossover	120	
the number of mutation	175	
the number of elite	5	
the number of subprograms	3	
the total number of nodes	120 (judgment: 80, processing: 40)	
crossover rate P_c	0.1	
mutation rate P_m	0.01	
mutation rate P_m^{tr}	0.01	
mutation rate P_m^{aff1}	0.002	—
mutation rate P_m^{aff2}	0.01	—

The simulation conditions are shown in Table 2. The number of individuals is 300. Although all the individuals are initialized by the procedure introduced in section 3.2, 120 new individuals are generated by crossover, 175 new individuals are generated by mutation, and five elite individuals are kept in each generation. The number of subprograms is set at three, and the total number of judgment nodes is 80 and that of processing nodes is 40. Crossover rate P_c, Mutation rates P_m and P_m^{tr} are determined appropriately through our experiments, which keeps the variation of the population, but does not change the programs too large. The settings of the parameters used in the learning phase are as follows. The learning rate α is set at 0.9 in order to find solutions quickly and the discount rate γ is set at 0.9 in order to sufficiently consider the future rewards. ε is set at 0.1 experimentally considering the balance between exploitation and exploration. In fact, the programs with lower ε fall into local minima with higher probability, and those with higher epsilon take too much random actions. As described before, mutation rate P_m^{aff1} and P_m^{aff2} is set at 0.002 and 0.01, respectively, based on our preliminary experiments.

4.3 Training results

Fig. 8 shows the fitness curves obtained by VS-DGNP and DGNP. Each curve shows the fitness of the best individual at each generation averaged over 30 independent simulations. From Fig. 8, both methods can obtain better fitness as

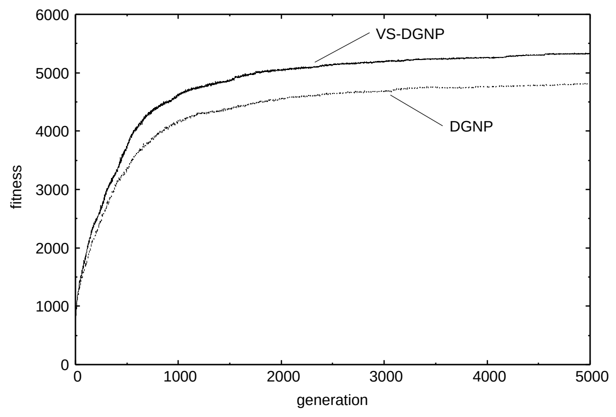


Fig. 8. Fitness curves

Table 3
t-test on the average fitness in the training environments

	VS-DGNP	DGNP
average fitness	5,333	4,818
standard deviation	592	905
p-value [%]	0.597	

the generation goes on. However, VS-GNP obtains better fitness than DGNP throughout the generations. Table 3 shows the average fitness, standard deviation and p-value calculated by t-test on the average fitness obtained in the last generation. It can be seen from Table 3 that the p-value is 0.597[%], which shows that there is a significant difference between VS-DGNP and DGNP.

Fig. 9 shows the average fitness obtained in each environment in the last generation, where VS-DGNP show better fitness than DGNP in all the environments. According to the definition of fitness (Eq. 2), the fitness of more than 500 can be obtained when all the tiles are dropped into holes in each environment.[2] It can be seen from Fig. 9 that VS-DGNP obtains more than fitness 500 in eight environments out of ten, while DGNP obtains more than 500 in three environments. Even in these three environments where DGNP obtains more than 500, VS-DGNP shows better fitness because the remaining time of VS-DGNP is larger than DGNP, which means that VS-DGNP can find the shorter and more efficient routes than DGNP to complete the tasks. As a result, it is concluded that VS-DGNP can create the program adaptable to various situations in the 10 environments by optimizing the size of the

[2] Each environment has three tiles, thus fitness 300 can be obtained by the first term of Eq. 2. Next, the sum of the initial distances between each tile and the nearest hole is 10, thus agents can move the tiles by 10 cells to the nearest hole direction. Then, fitness 200 can be obtained by the second term of Eq. 2. Therefore, the total fitness is at least 500 when all the tiles are dropped into holes.

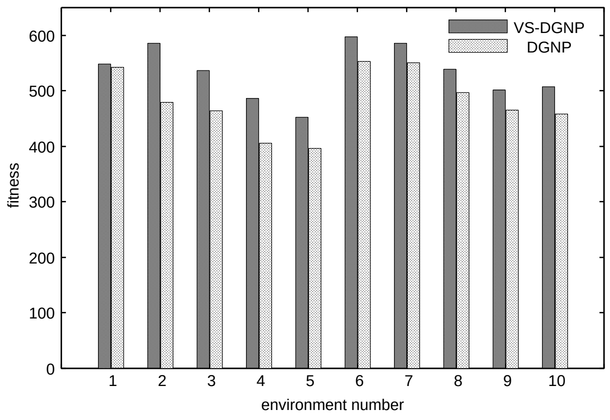


Fig. 9. Average fitness obtained in each training environment

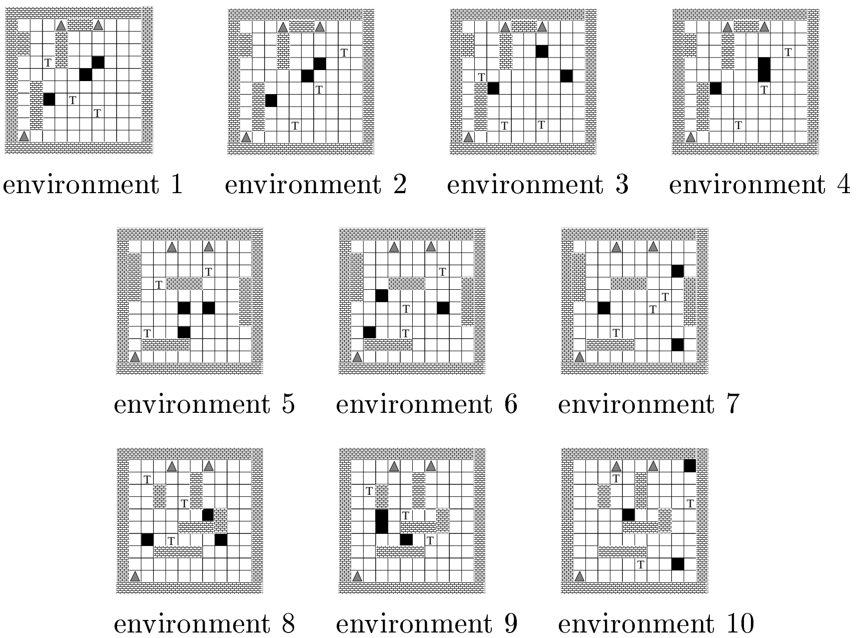


Fig. 10. Testing environments

substructures.

4.4 *Testing results*

In the testing simulations, the programs evolved by VS-DGNP and DGNP are executed in unknown testing environments that are shown in Fig. 10. The difference between the testing and training environments are as follows. The positions of tiles are different in environments 1 and 2, the positions of tiles and holes are different in environments 3 and 4, and the positions of tiles, holes and obstacles are different in environments 5–10 where environments 5–7 and 8–10 have the same obstacle positions, respectively. Therefore, the evolved program are required to have a general knowledge for achieving the

Table 4
t-test on the average fitness in the testing environments

	VS-DGNP	DGNP
average fitness	2,010	1,424
standard deviation	1,037	975.6
p-value [%]	1.40	

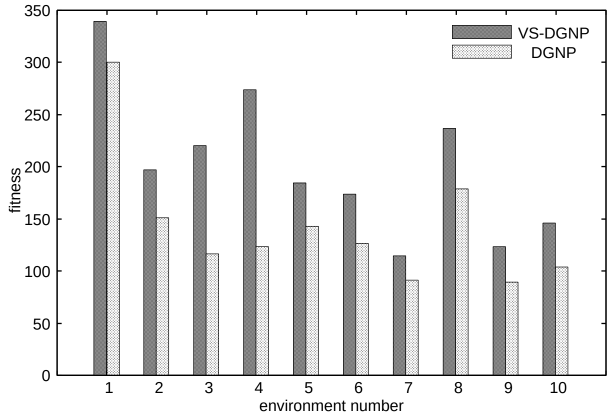


Fig. 11. Average fitness obtained in each testing environment

task in unknown environments. The assigned action steps are 100.

Table 4 shows the average fitness, standard deviation and p-value calculated by t-test on the fitness values obtained in the testing simulations. The average fitness is calculated in such a way that 30 elite individuals independently evolved in the training environments are executed in the testing environments and the average fitness is calculated. It can be seen from Table 4 that the p-value is 1.40[%], which shows that there is a significant difference between VS-DGNP and DGNP in the testing simulations.

Fig. 11 shows the average fitness obtained in each testing environment, where VS-DGNP show better fitness than DGNP in all the environments. Since the positions of tiles, holes and/or obstacles are different from the training environments, it is difficult for both VS-DGNP and DGNP to complete the tasks; however, VS-DGNP effectively makes use of the evolved structure to solve the unexperienced tasks better than DGNP. It means that the variable size structure and its evolution mechanism can acquire general action rules in the graph structure for adapting to various situations. The subprograms of VS-DGNP can give and take nodes to/from another subprogram to optimize the complexity of the subprograms depending on the problems, thus rules can be created efficiently, while DGNP has the fixed number of nodes in each subprogram and has to make action rules within the given size of the structure. However, as mentioned in section 3.3.2, the flexibility of the program size might cause too large changes of the structure and destroy the good building blocks, therefore, the parameters on the evolution have to be determined carefully and the learning or evolution algorithm of the parameters is a remaining issue to be studied.

5 Conclusions

In this paper, a variable size mechanism of distributed GNP is proposed. In order to realize this mechanism, the new crossover and mutation operators considering the distributed structures are introduced. From the simulation results in the Tileworld problem, the proposed method shows better results than distributed GNP without using variable size mechanism in both training and testing environments. In the future work, the contribution of the distributed structure and variable size mechanism is analyzed in detail. For example, the action rules generated in each subprogram should be analyzed in order to find the functions contained in each subprogram generated by evolution. In addition, the applicability of the variable size distributed GNP to many problems such as data mining, robot programming and stock trading models will be studied.

References

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

[2] J. R. Koza: *Genetic Programming, on the programming of computers by means of natural selection*, Cambridge, Mass.: MIT Press, (1992).

[3] J. R. Koza: *Genetic Programming II, Automatic Discovery of Reusable Programs*, Cambridge, Mass.: MIT Press, (1994).

[4] S. Kamio and H. Iba: Adaptation technique for integrating genetic programming and reinforcement learning for real robots, *IEEE Trans. Energy Convers.*, vol. 9, no. 3, pp. 318–333, (2005).

[5] R. Ruiz-Torrubiano and A. Suárez: Hybrid approaches and dimensionality reduction for portfolio selection with cardinality constraints, *Comp. Intell. Mag.*, vol. 5, pp. 92–107, (2010).

[6] E. Alfaro-Cid, P. A. Castillo, A. Esparcia, K. Sharman, J. Merelo, A. Prieto, A. Mora, and J. Laredo: Comparing multiobjective evolutionary ensembles for minimizing type i and ii errors for bankruptcy prediction, in *Proc. of the IEEE World Congress on Computational Intelligence*, pp. 2902–2908, (2008).

[7] H. Iba and T. Sasaki: Using genetic programming to predict financial data, in *Proc. of the Congress on Evolutionary Computation 99*, pp. 244–251, (1999).

[8] Z. Banković, D. Stepanović, S. Bojanić, and O. Nieto-Taladriz: Improving network security using genetic algorithm approach, *Computers and Electrical Engineering*, vol. 33, pp. 438–451, (2007).

[9] G. Folino, C. Pizzuti, and G. Spezzano: GP ensemble for distributed intrusion detection systems, in *ICAPR 2005, Lecture Notes in Computer Science (LNCS) 3686.* Springer-Verlag Berlin Heidelberg, pp. 54–62, (2005).

[10] 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).

[11] K. Hirasawa, M. Okubo, H. Katagiri, J. Hu, and J. Murata: Comparison between genetic network programming (GNP) and genetic programming (GP), in *Proc. of the Congress on Evolutionary Computation*, pp. 1276–1282, (2001).

[12] S. Mabu, C. Chen, N. Lu, K. Shimada, and K. Hirasawa: An intrusion-detection model based on fuzzy class-association-rule mining using genetic network programming, *IEEE Trans. on Syst., Man, Cybern. C*, vol. 41, no. 1, pp. 130–139, (2011).

[13] K. Hirasawa, T. Eguchi, J. Zhou, L. Yu, and S. Markon: A double-deck elevator group supervisory control system using genetic network programming, *IEEE Trans. on Syst., Man, Cybern. C*, vol. 38, no. 4, pp. 535–550, (2008).

[14] L. J. Fogel, A. J. Owens, and M. J. Walsh: *Artificial Intelligence through simulated Evolution*, John Wiley & Sons, (1966).

[15] D. B. Fogel: An introduction to simulated evolutionary optimization, *IEEE Trans. Neural Netw.*, vol. 5, no. 1, pp. 3–14, (1994).

[16] Y. Yang, X. He, S. Mabu, and K. Hirasawa: A cooperative coevolutionary stock trading model using genetic network programming-sarsa, *Journal of Advanced Computational Intelligence and Intelligent Informatics*, vol. 16, pp. 581–590, (2012).

[17] R. Li, T.-P. Tian, and S. Sclaroff: Divide, conquer and coordinate: Globally coordinated switching linear dynamical system, *IEEE Trans. on Pattern Analysis and Machine Intelligence*, vol. 34, pp. 654–669, (2012).

[18] M. E. Pollack and M. Ringuette: Introducing the tile-world: Experimentally evaluating agent architectures, in *Proc. of the conference of the American Association for Artificial Intelligence*, pp. 183–189, (1990).

[19] R. S. Sutton and A. G. Barto: *Reinforcement Learning - An Introduction*, Cambridge, Massachusetts, London, England: MIT Press, (1998).