Waseda University Doctoral Dissertation

Study on Genetic Network Programming with Variable Size Structure and Genotype/Phenotype Mapping Mechanism

Bing LI

Graduate School of Information, Production and Systems

Waseda University

June 2013

I would like to dedicate this thesis to my loving parents and wife.

Acknowledgements

I would like to express my gratitude to all those who helped me during the writing of this thesis.

My deepest gratitude goes first and foremost to my supervisor Professor Hirasawa, who has provided me with valuable guidance in every stage of the writing of this thesis. It is my honor to study in his lab. He is a good professor, and give me many advices on my research. When I got confused, he can always show me the path to right way. He taught me how to analyze a problem, describe my points and give a good presentation. Without his enlightening instruction, impressive kindness and patience, I could not have completed my thesis. His keen and vigorous academic observation enlightens me not only in this thesis but also in my future study.

I would like to thank Professor Takayuki Furuzuki, Professor Osamu Yoshie and Professor Shigeru Fujimura for their some valuable advice on improving the quality of this thesis.

I would like to appreciate Dr. Mabu for giving me a lot of supports on my research. Having discussion with him is very useful for his abundant experiences on evolutionary computation. His invaluable suggestions and kindness are very important for the completion of my doctor courses.

I would like to thank Waseda University and JSPS for supporting me to do the research in Japan.

To all my friends and classmates, my life in Japan will not be so happy without you.

Thank you. And thank you very much for what you have done for me. Thank you to you all.

Abstract

In the research field of Artificial Intelligence, Evolutionary Algorithms (EAs) are subset of important optimization technologies, which are inspired by the Darwin's theory of evolution. EAs are kinds of effective algorithms for solving very large search space problems with less prior knowledge and human intervention. Starting from the 1950s, a lot of EAs are developed, such as Evolution Strategies (ES), Genetic Algorithm (GA), Genetic Programming (GP) and Evolution Programming (EP). They have been successfully applied to many fields such as engineering, biology, economics, marketing, robotics, physics, chemistry, education and so on.

After investigating the benefits and shortcoming of GA and GP, Genetic Network Programming (GNP) was proposed around 2000. The directed graph structure of GNP extends the chromosome representation of strings in GA and trees in GP, which makes it have high expression ability with relevant small size of individuals, and consequently GNP has the better performance than other evolutionary algorithms. Nowadays, GNP is not only used to solve benchmark problems but also applied to many real world applications such as elevator supervisory control systems, stock market prediction, data mining and traffic prediction.

Since GNP was proposed, many methods have been developed to improve the performance of GNP such as combining GNP with reinforcement learning, introducing symbiotic learning in GNP, upgrading the structure of GNP by defining macro node and rule accumulation. Although these methods have been proved to improve the performance of GNP by combing some other machine learning methods, some useful prior knowledge of biology: variable length of gene, evolution by

gene duplication and genotype-phenotype mapping, are not well considered. Therefore, in this research, two kinds of methods and their extensions have been proposed to improve the performance including the expression and generalization ability of GNP by upgrading the structure of GNP using the above theories, and to solve two problems of GNP, i.e., "the node size of GNP is fixed" and "an individual is a solution".

One of the methods is Variable Size Genetic Network Program (GNPvs) and its extension GNPvs with Replacement (GNPvs-R), which simulates the variable length of gene and gene duplication, and solve the problem that the node size of GNP is fixed. The other is Genetic Network Programming for Automatic Program Generation with Mapping Mechanism (GNP-APGm) and its extension Subroutine embedded GNP-APGm (GNPsr-APGm), which implements the genotype-phenotype mapping process, and solve the problem that an individual is a solution.

The above methods are verified on the tileworld benchmark problem. The simulation results shows these proposed methods increase the performance of GNP exactly.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

1.1.1 Evolutionary Algorithms

Darwin's theory of evolution firstly gives a convincing explanation of the origin of creatures, and shows the fantastic power of nature selection. Inspired by the theory, Evolutionary Algorithms (EAs) are developed by computer scientists, which is a kind of meta-heuristic optimization algorithm for solving the problem with very large search space by improving the candidate solution iteratively. The essential advantage of EAs is to solve the particular problem with less prior knowledge and human intervention.

Starting from the 1950s, the study of computer simulations of evolution began [1, 2, 3, 4], but this kind of research was not widely noticed. Until the 1960s and early 1970s, the work of Ingo Rechenberg and Hans-Paul Schwefel on solving complex engineering problems by Evolution Strategies (ES) [5] and Fogel on predicting environments through Evolutionary Programming [6] made evolutionary algorithm well recognized. After that, the most popular algorithm in particular for solving optimization problems – Genetic Algorithm (GA) was proposed by Holland on 1975 [7]. Moreover, Holland and his successors attempted to explain the working principle of EA by proposing the building block hypothesis [8]. This hypothesis supposes that the "building blocks", i.e., low order, low defining-length schemata with more than the average fitness is the key of candidate solution im-

provement. By implicitly and efficiently identifying and recombining "building blocks", EA continuously upgrades the candidate solution. Through this original work, the behaviors and the effectiveness of EA became more understandable.

Besides, as another branch of EA, Genetic Programming (GP)[9, 10] was developed around 1990s by Koza. The major challenge of GP is to find proper computer programs according to the user-defined task in order to make computers automatically solve problems. 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'...) [9]; 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 to search food and take the food to their nest [9]. GP has been proved to be completive with human performance on some problems like cellular automata, circuit design, controller design, etc [11, 12]. Inspired by GP, more and more evolutionary algorithms for automatic programming have been proposed. Some successful paradigms are list as follows.

- Cartesian Genetic Programming (CGP) - This algorithm is directed graph structure and optimized to design circuit [13, 14].
- Gene Expression Programming (GEP) - The essential point of the algorithm is to use linear chromosome to generate tree-structure programs [15, 16]
- Grammatical Evolution (GE) - This method combines the grammar of programming languages and bit strings of GA to produce programs [17, 18].

1.1.2 Related biology knowledge

Since EAs are loosely based on nature precedent [19], some prior knowledge of biology could be used to improve the performance of EAs. Firstly, it is a common sense that the length of gene in spices is evolved from short to long over long period, i.e., there are about 2 million DNA base pairs in some bacterial, in contrast to about 3 billion in human. Obviously, human are much more complex and

functional than bacterial, which means that longer gene could have more expression ability to generate more functions for dealing with problems in nature. On the other words, it is possible to improve the performance of EAs by increasing the length of chromosome. Secondly, the theory of *Evolution by gene duplication* [20, 21] describes that the gene duplication is a major driving force of evolvability. In this theory, it can make the individual survive under the selection pressure and eventually might accumulate mutations on duplicated gene, which produces new features of individuals for adapting to the new environments by copying a part of gene. By this point of view, the gene duplication operation could be included in EAs to increase the generalization ability of the algorithms. Finally, in biological systems, there is a genotype-phenotype mapping mechanism, which explains the procedure of protein generation from the sequence of gene. With the working principle of this mechanism, alleles [22] will be translated to the same amino acids, and consequently build the same protein, which makes two individual with different chromosome have the same phenotype. Therefore, EAs can keep the diversity of individuals for enlarging the search space by introducing this mechanism.

1.1.3 Genetic Network Programming

Genetic Network Programming (GNP) [23, 24] is a kind of evolutionary algorithm with directed graph structure, which is an extension of GA and GP. The novel structure of GNP makes it have high expression ability with relevant small size of individuals, and consequently has the better performance than other evolutionary algorithms [23, 25, 26]. It has following advantages [23].

1. Partially observable systems. GNP can realize the partially observable systems by setting any flexible simple judgment and processing nodes and combining them by evolution.

2. Building block function. When GNP is executed, some parts of the structure will repeat several times, which means these parts are memorized as some useful functions. These functions work like the Automatic Defined Functions (ADFs) in GP, which improves the performance of the algorithm [10].

3. Fixed number of nodes and reusability of nodes. The number of nodes in a GNP individual is fixed. And, during the transitions, the nodes in GNP structure can be visited many times, which makes the GNP structure compact. Therefore, GNP can avoid the bloating problem [23, 27, 28] while still keeping high expression ability.

4. Easy to implement. Although the structure of GNP seems complicated, it can be easily represented by a list of nodes. Moreover, genetic operations like crossover and mutation on the list of nodes are implemented simply.

Therefore, GNP has been successfully applied to many real world applications like elevator supervisory control systems [29], stock market prediction [30], association rule mining [31], and traffic prediction [32].

1.1.3.1 Basic structure of GNP

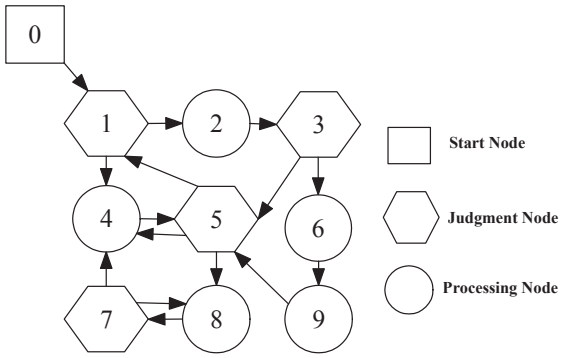


Figure 1.1: Basic structure of conventional GNP

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 Fig.1.1. The structure of GNP contains three kinds of primary nodes: start node, judgment node and processing node. These nodes are connected by directed links shown with arrows. The square represents the start node which just points the first node to perform. The hexagon stands for judgment nodes. Judgment nodes have several branches connected to the other judgment nodes or processing nodes. When the judgment node is executed, it collects the information from

environments, then analyzes the current situation, 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. Each processing node will make an agent take an action when the processing node is visited. After the action, the environment might be 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.

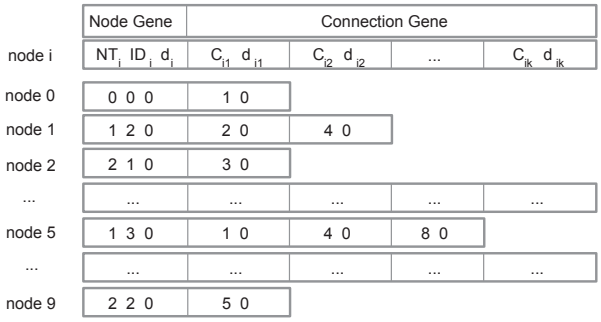


Figure 1.2: Representation of GNP structure

Fig.1.2 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, and they 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 judgments and processings. Time delay describes the time cost when GNP executes the judgment or 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. In this thesis, time delay is always 0.

1.1.3.2 Genetic operators of GNP

Like other evolutionary algorithms, GNP uses 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 individuals and move them to the next generation directly. Tournament selection chooses several individuals from the current population randomly, then runs several "tournaments". The winners of all 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 P_c to swap each other. After crossover, two new individuals are produced and moved to next generation. Fig.1.3 shows a simple example of crossover. In the example, *node 3, node 4* and *node 9* marked with grey color are decided to exchange. The black arrows describe the connections of these nodes to the other nodes. After crossover, these connections are swaped, and the structures of offspring become different from their parents.

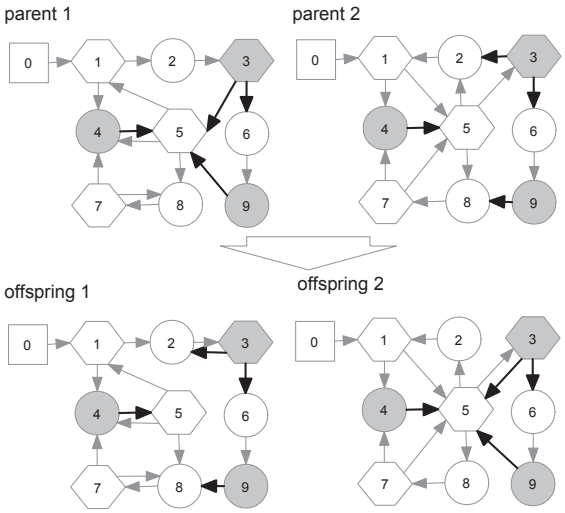


Figure 1.3: Crossover of GNP

Mutation occurs on one individual. All data in the gene except NT_i have the mutation rate 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.

The left part of Fig.1.4 shows the connection mutation and the right part shows the function mutation, where the black arrows are changed to point the other node in the connection mutation, while the *ID* of *node 4* turns to 3 from 2 in the function mutation.

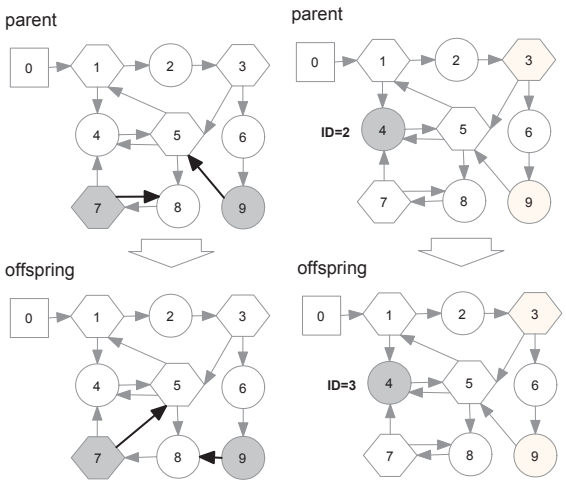


Figure 1.4: Mutation of GNP

1.2 Research objective

After GNP was proposed, many methods have been developed to improve the performance of GNP such as combining GNP with reinforcement learning [23], introducing symbiotic learning in GNP [24], upgrading the structure of GNP by defining macro node [33] and rule accumulation [34]. Although these methods have been proved to improve the performance of GNP and are applied to many real world applications successfully, two problems of GNP have not been solved, i.e., "the node size of GNP is fixed" and "an individual is a solution". The first problem indicates that the predefined number of nodes is used, and it is difficult to set the optimal size of an individual in the beginning. The second problem means that it is hard to design and evolve a complicated individual as the sophisticated solution for solving a difficult problem.

Therefore, the objective of this research is to solve both problems by enhancing the structure of GNP using the above theories.

1.3 Organization of the thesis

Besides the introduction, the rest of thesis is organized as follows.

Chapter 2 introduces Variable Size Genetic Network Programming (GNPvs), which changes the size of the individuals and obtain the optimal size of them during evolution. The proposed method will select the number of nodes to move from one parent GNP to the other parent GNP in crossover to implement the new feature of GNP. The probability of selecting the number of nodes to move satisfies the binomial distribution. The proposed method can keep the effectiveness of crossover, improve the performance of GNP and find the optimal size of the individuals.

Chapter 3 describes the improvement method of GNPvs - GNPvs with Replacement (GNPvs-R), in which a kind of replacement mechanism is firstly proposed in GNPvs. Inspired by the theory of *Evolution by Gene Duplication*, in the proposed method, the non-frequently used nodes are replaced with the frequently used nodes, on the other words, a part of nodes are copied, which makes the algorithm has higher generalization ability than former algorithms.

Chapter 4 presents GNP for Automatic Program Generation with Mapping Mechanism (GNP-APGm), in which a kind of genotype-phenotype mapping process is introduced in GNP to create programs. In this method, an individual of GNP 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.

Chapter 5 shows Subroutine embedded GNP for Automatic Program Generation with Mapping Mechanism (GNPsr-APGm), which improves the performance of GNP with Mapping Mechanism. The proposed method automatically defines the main function and usage of the potential useful subroutines during evolution. By using subroutines, a complex program can be decomposed to several simple programs which are obtained more easily. Moreover, these subroutines are called many times from the main program, which results in reducing the size of the program significantly.

Chapter 5 is devoted to conclusions.

Chapter 2

Variable Size Genetic Network Programming (GNPvs)

2.1 Introduction

As described in the introduction, GNP uses the directed graph chromosome as its basic structure, which is extended from the strings in GA and trees in GP. With the help of high expression ability of graph structures, GNP has some valuable features like partially observable systems, building block functions, fixed number of nodes and reusability of nodes [23], which improves the performances of the algorithms and avoid the bloating problem.

Since the feature of the fixed number of nodes for avoiding the bloating problem [23], most GNP and GNP based methods do not change the size of the individuals during evolution. Fixed number of nodes means that a predefined number of nodes is used, which needs prior knowledge of the problems or needs to try different sizes of GNP. However, when the problem is complicated, it is impossible to get enough prior knowledge or try many different sizes of GNP. Moreover, if the size is defined too small, the individual will be lack of the expression ability, while if the size is too large, it needs a large search space and the individual will be overfitting easily. Therefore, a new type of GNP is needed, in which the size of the individuals is variable. Besides, more nodes mean much higher expression ability which is necessary for solving complicated problems,

while too large number of nodes lack the generalization ability. For these reasons, Variable Size Genetic Network Programming (GNPvs) is proposed, which changes the size of the individuals and obtain the optimal size of them during evolution. The proposed method defines a new type of crossover to implement the new feature of GNP. The new crossover will select the number of nodes to move from one parent GNP to the other parent GNP. The probability of selecting the number of nodes to move satisfies the binomial distribution. Besides, the new crossover can prevent the bloating problem which may occur in GNPvs, since the difference between the number of nodes selected from two individuals is small, i.e., the size of offsprings will not increase dramatically..

On the other hand, there are some research have been conducted on variable reforestation of chromosome. Messy Genetic Algorithm is one of them. At first, most research on GA uses fixed-length strings, however variable-length strings are more close to nature, i.e., the genotypes of lives evolved from simple to complex in nature [19]. Therefore, in [19], a messy genetic algorithm (mGA) was proposed, which defines messy strings, and its operators permit the algorithm to exploit and form higher-performance building blocks than simple GA with fixed length. Since the chromosome in mGA is variable, simple crossover no longer works, two simple operators: cut and splice are used to replace crossover. Fig.2.1 shows the schema of cut and splice coordination in mGA. The cut operator divides the individuals into two parts and the splice operator joins the different parts from parents to generate new individuals. By combining both operators, the parents can exchange the gene like crossover. As noted in [19], mGA works better than simple GA with fixed coding. The crossover in the proposed method is inspired from this method, that is, different numbers of nodes in GNP from both individuals are swapped by crossover. It is the first time in this thesis to introduce this kind of crossover to the graph-based evolutionary algorithm, since other graph-based evolutionary algorithms like Evolution Strategy and Cartesian Genetic Programming only use mutation to generate new individuals [35, 36].

Obviously, GP is developed to evolve programs, therefore, GP has variable gene structures. As described in [12], "GP now routinely delivers high-return human-competitive machine intelligence". However, GP suffers from the bloating problem which will affect the performance of the algorithm [27, 28, 37]. Derived

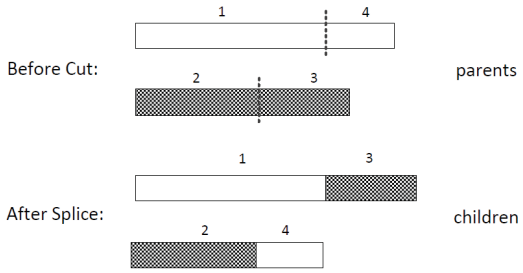


Figure 2.1: Schema of cut and splice coordination in mGA

from GP, Gene Expression Programming (GEP) [15, 16] and Grammatical Evolution (GE) [17, 18] are proposed to generate programs automatically. Different from GP, both methods introduce genotype-phenotype mapping mechanism [38] to generate programs, by which fixed length genes can be translated to variable length programs. Moreover, both methods can alleviate the bloating phenomenon [39, 40] and get better performances than GP.

Besides, the paper [41] is the first attempt to develop a variable size GNP. In this paper, the contribution of each kind of nodes for the fitness value is calculated, then the algorithm determines whether one particular kind of node is to be added to all the individuals or is to be deleted from them depending on the contribution of it to the fitness value. Although the algorithm implements the variable size GNP and has proved its good performances, all the individuals in the population still have the same number of nodes, which limits the ability of keeping the diversity of chromosomes and enhance the exploration ability of the algorithm by containing different sizes of individuals in the population.

2.2 Variable Size Genetic Network Programming with Binomial Distribution

Different from the algorithm in [41], Variable Size Genetic Network Programming (GNPvs) with Binomial Distribution implements the feature of changing the size of the individual and obtaining the optimal size of it during evolution in a more natural way.

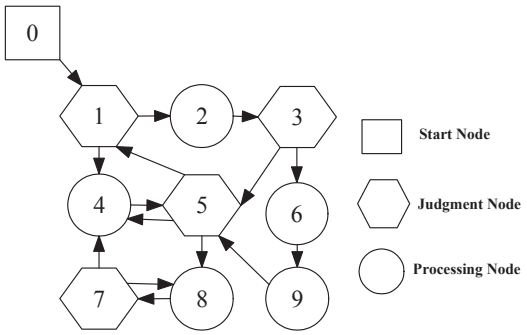


Figure 2.2: Basic structure of GNPvs

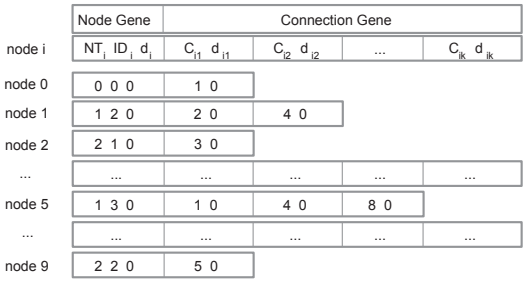


Figure 2.3: Representation of GNPvs structure

2.2.1 Structure of GNPvs

Although GNPvs is a new type of GNP, the phenotype and genotype of GNPvs is the same as GNP, which are shown in Fig.2.2 and Fig.2.3. Therefore, the individual of GNPvs also contains three kinds of primary nodes: start node, judgment node and processing node, and each node has its node ID, node function, and connections, which are described in the introduction.

2.2.2 Foundation of GNPvs

The main difference between GNPvs and GNP is that the size of the individuals in GNPvs is variable and GNPvs allows the individuals to change the size during evolution. In order to implement the above features, a new type of crossover is introduced in GNPvs to replace the normal uniform crossover in GNP. The crossover of GNPvs is an extension of uniform crossover. In uniform crossover, the corresponding nodes have the probability, i.e., crossover rate of P_c to swap

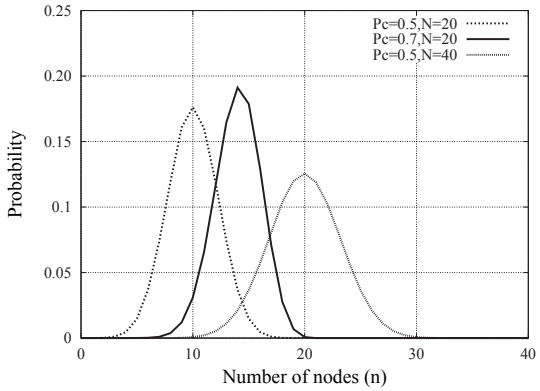


Figure 2.4: Binomial distribution of the number of nodes selected

each other. Since uniform crossover randomly picks up the nodes from the individual at crossover rate of P_c, the probability $f(n; N, P_c)$ of choosing n nodes from the total N nodes equals $C_N^n P_c^n (1 - P_c)^{N-n}$, which satisfies the following binomial distribution Eq.(2.1). However, in the crossover of GNPvs, each parent has its own crossover rate, for example, parents $GNPvs_A$ and $GNPvs_B$ have crossover rate of P_A and P_B, respectively. P_A and P_B could be the same or different. Besides, each parent selects the nodes to move independently. Therefore, $GNPvs_A$ and $GNPvs_B$ have their own binomial distributions like Fig.2.4. When performing crossover, two GNPvs individuals may choose different number and different kind of nodes to move, even when the crossover rate is the same.

$$
\begin{aligned}
& f(n; N, P_c) = C_N^n P_c^n (1 - P_c)^{N-n} \\
& \qquad n = 0, 1, 2, ..., N, \quad where, \\
& C_N^n = \tfrac{N!}{n!(N-n)!} \\
& n : the\ number\ of\ nodes\ selected\ to\ move \\
& N : total\ number\ of\ nodes\ in\ an\ individual \\
& P_c : crossover\ rate\ of\ the\ individuals
\end{aligned}
\tag{2.1}
$$

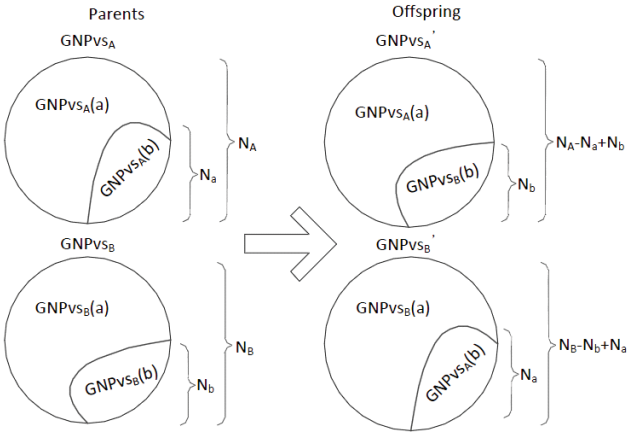


Figure 2.5: Schema of crossover in GNPvs

2.2.3 Crossover of GNPvs

As described above, crossover is the most important genetic operator to realize the proposed method. Fig.2.5 shows the schema of crossover in GNPvs. In Fig.2.5, two offspring $GNPvs_A{}'$ and $GNPvs_B{}'$ are generated by parents $GNPvs_A$ and $GNPvs_B$.

where,

$GNPvs_A(a)$: the remaining part of $GNPvs_A$ during crossover;

$GNPvs_B(a)$: the remaining part of $GNPvs_B$ during crossover;

$GNPvs_A(b)$: the moving part of $GNPvs_A$ during crossover;

$GNPvs_B(b)$: the moving part of $GNPvs_B$ during crossover;

N_A: the number of nodes of $GNPvs_A$;

N_a: the number of nodes of $GNPvs_A(b)$;

N_B: the number of nodes of $GNPvs_B$;

N_b: the number of nodes of $GNPvs_B(b)$;

$GNPvs_A = GNPvs_A(a) + GNPvs_A(b)$;

$GNPvs_B = GNPvs_B(a) + GNPvs_B(b)$;

$GNPvs_A{}' = GNPvs_A(a) + GNPvs_B(b)$;

$GNPvs_B{}' = GNPvs_B(a) + GNPvs_A(b)$.

In Fig.2.5, $GNPvs_A(b)$ and $GNPvs_B(b)$ are selected to move, then new individual $GNPvs_A{}'$ is generated by combining the remaining part of $GNPvs_A(a)$ and moving part of $GNPvs_B(b)$, and $GNPvs_B{}'$ is obtained by combining the

remaining part of $GNPvs_B(a)$ and moving part of $GNPvs_A(b)$, respectively. As described above, the probability of selecting the number of nodes to move, i.e., N_a and N_b follows the binomial distribution.

Based on the basic concept of crossover in Fig.2.5, the algorithm of crossover is shown as follows. The detail of the algorithm is described in the next subsection by a simple example.

Procedure 1: Algorithm of crossover

Input: Two individual $GNPvs_A$ and $GNPvs_B$
P_A, P_B: crossover rate of $GNPvs_A$ and $GNPvs_B$
N_A, N_B: size of $GNPvs_A$ and $GNPvs_B$
N_a, N_b: size of $GNPvs_A(b)$ and $GNPvs_B(b)$
$S(A)$, $S(B)$: sets of moving nodes
Output: Two new individual $GNPvs_A{}'$ and $GNPvs_B{}'$
#Determine the number of nodes to move
$N_a \leftarrow 0$, $N_b \leftarrow 0$
for $i = 1$ to N_A **do**
  if *random value* $< P_A$ **then**
    $N_a \leftarrow N_a + 1$
  end if
end for
for $i = 1$ to N_B **do**
  if *random value* $< P_B$ **then**
    $N_b \leftarrow N_b + 1$
  end if
end for
#Select nodes to move
$S(A) \leftarrow \emptyset$, $S(B) \leftarrow \emptyset$
for $i = 1$ to N_a **do**
  if $S(A) = \emptyset$ **then**
    Randomly choose a node from $GNPvs_A$ and set the chosen node as $node_i$
  else
    Select the node connected from the last node in $S(A)$ and set the selected node as $node_i$
  end if

$S(A) \leftarrow S(A) \cup node_i$, delete $node_i$ from $GNPvs_A$
end for
for $i = 1$ to N_b **do**
  if $S(B) = \emptyset$ **then**
    Randomly choose a node from $GNPvs_B$ and set the chosen node as $node_i$
  else
    Select the node connected from the last node in $S(B)$ and set the selected node as $node_i$
  end if
  $S(B) \leftarrow S(B) \cup node_i$, delete $node_i$ from $GNPvs_B$
end for
#Move nodes
$GNPvs_A{}' \leftarrow S(B) \cup GNPvs_A$
$GNPvs_B{}' \leftarrow S(A) \cup GNPvs_B$
#Update connections of $GNPvs_A{}'$ and $GNPvs_B{}'$
for $i = 1$ to $N_A - N_a + N_b$ **do**
  Update connections of $node_i$
end for
for $i = 1$ to $N_B - N_b + N_a$ **do**
  Update connections of $node_i$
end for
return $GNPvs_A{}', GNPvs_B{}'$

2.2.4 Example of crossover in GNPvs

Two simple individuals of GNPvs are used as an example in Fig.2.6, in order to show the procedure of crossover and explain Procedure 1. $GNPvs_A$ and $GNPvs_B$ are individuals consisted of 5 and 6 nodes, respectively. The upper part of Fig.2.6 is the genotype of two individuals, and lower part is the representation of genotypes.

As shown in Procedure 1, the first step is to determine the number of nodes to move. In this example, the algorithm decides 1 node and 2 nodes from $GNPvs_A$ and $GNPvs_B$ to move, respectively. Fig.2.7 shows the next two steps for selecting and moving nodes described in the above procedure. Since $GNPvs_A$ only need to choose one node to move, $node4_A$ is picked up randomly. While, $GNPvs_B$ is

required to select two nodes. Therefore, it firstly chooses $node1_B$ randomly, then picks up $node4_B$ connecting from $node1_B$. The rules to select the next node to move as $S(A)$ or $S(B)$ is shown as follows.

- If the current node is a processing node, then just select the node connected from the current node.
- If the current node is a judgment node, then choose one branch from the current node to select the node randomly.
- If the next node is already in the moving set, then pick up another node randomly.

The moving nodes are copied to set $S(A)$ and $S(B)$. After that, $node4_A$ is inserted into $index1$ of $GNPvs_B$ to generate $GNPvs_B{}'$, and $node1_B$ and $node4_B$ are put into $index4$ and $index5$ of $GNPvs_A$, respectively, to create $GNPvs_A{}'$.

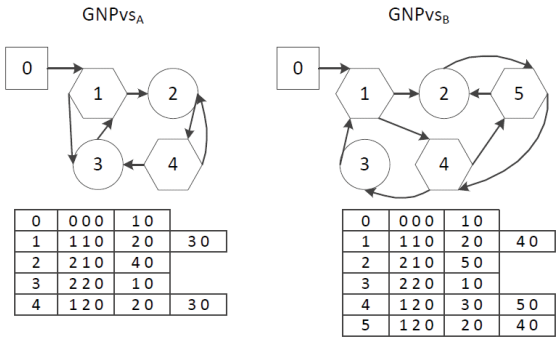


Figure 2.6: Parents before crossover in GNPvs

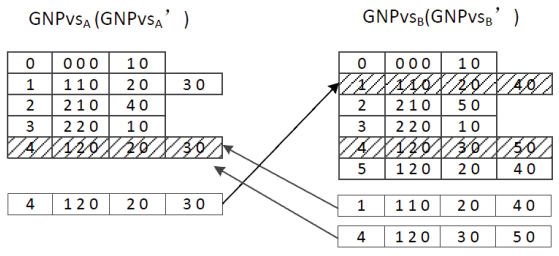


Figure 2.7: Node selection and movement in crossover of GNPvs

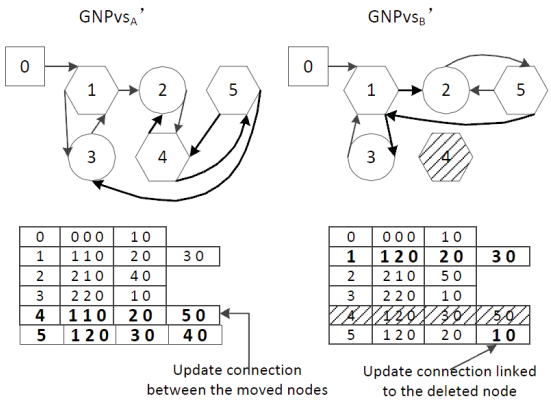


Figure 2.8: Offspring after crossover in GNPvs

The final step is to update connections. Two kinds of connections need to be updated. One is the connections of the moved nodes, the other is the connections of all the nodes in the individual, which links to the deleted nodes. For the moved nodes, since the indexes of nodes are changed, the connections of these nodes should be changed in order to keep the building blocks of these nodes. For example, in Fig.2.7, $node1_B$ connects to $node4_B$, and after crossover, $node1_B$ and $node4_B$ become $node4'_A$ and $node5'_A$, respectively, therefore the connection of $node4'_A$ should be changed to $node5'_A$, which is shown in Fig.2.8. Because the nodes will be deleted from parents GNPvs, the connections of the nodes linked to the deleted nodes need to be updated. For example, in Fig.2.7, $node5_B$ connects to $node4_B$, while in $GNPvs_B{}'$, $node4_B$ is deleted, therefore, the connection of $node5_B$ is changed from $node4_B$ to $node1_B$ as described in Fig.2.8. When the updating is finished, two new individuals $GNPvs_A{}'$ and $GNPvs_B{}'$ are obtained, which is represented in Fig.2.8.

2.2.5 Flowchart of GNPvs

Fig.2.9 shows the following steps of the flowchart of GNPvs.

Step 1: Parameters are set, especially crossover rate, mutation rate and so on. And hundreds of individuals of GNPvs are generated randomly.

Step 2: Evaluate each individual using the current environment to get the fitness of the individual.

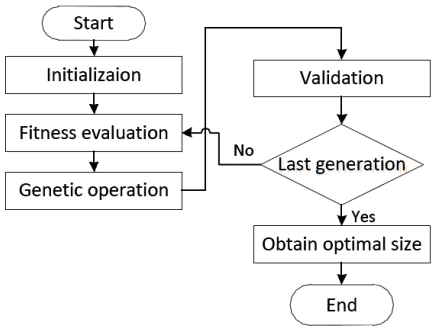


Figure 2.9: Flowchart of GNPvs

Step 3: Execute genetic operations, i.e., elite selection, crossover and mutation. The new crossover is deployed in this step.

Step 4: A number of top individuals are picked up to validate the generalization ability of individuals, since the proposed method should find the optimal size of the individual.

Step 5: Determine whether it is the last generation or not. If the answer is yes, then the algorithm ends, otherwise, go to step 2.

Step 6: Check the performance by validation data and find the optimal size of the individual.

2.3 Simulations

In this section, the performances of the proposed method are evaluated and compared with the conventional GNP using Tileworld [42].

2.3.1 Simulation environments

Tileworld is a famous agent-based test bed with time dependent and uncertain features, since the current action taken by the agent depends on its previous actions, and different agents cannot communicate with each other, which means that they cannot predict the actions taken by the other agents [42, 43, 44, 45]. Tileworld consists of agents, floor, tiles, holes and obstacles. The agents need

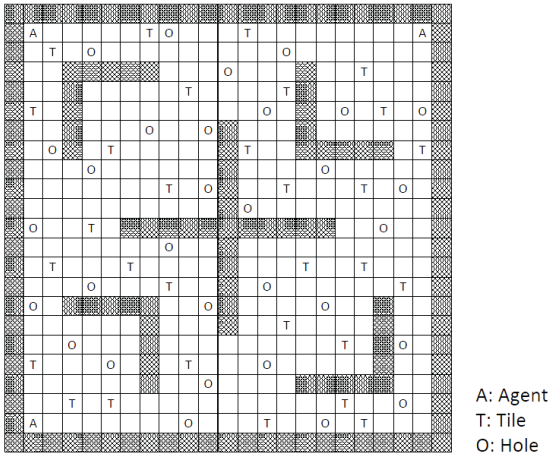


Figure 2.10: Example of tileworld used in simulations

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. An agent can only push one tile at a time. Fig.2.10 shows an example of the tileworld used in the simulations. In each tileworld, there are three agents, 30 holes and 30 tiles.

2.3.2 Simulation I

As noted in [37], the effectiveness of crossover will decrease during evolution. Therefore, the effectiveness of crossover in GNPvs and GNP is studied in this simulation by removing mutation from genetic operators.

The functions of judgment nodes and processing nodes are described in Table 2.1. JF, JB, JL, JR, JT, JH, JHT and JST are 8 kinds of judgment nodes, while MF, TR, TL and ST are 4 kinds of processing nodes in both algorithms. Moreover 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. Thus, each judgement node has 5 branches. Besides, there are totally 60 nodes (12 kinds of nodes $\times$ 5 for each kind of node) in each individual at first. In order to avoid the bloating problem, the maximal number of nodes is set at 180.

The parameters used in the simulations are described in Table 2.2. The population size is 300, and during reproduction, the top 10 individuals are copied to the next population. Since there is no mutation, all the other 290 individuals

Table 2.1: Functions of judgment nodes and processing nodes

NT	ID	Symbol	Function
1	1	JF	Judge Forward
1	2	JB	Judge Backward
1	3	JL	Judge Left
1	4	JR	Judge Right
1	5	JT	Judge the nearest Tile
1	6	JH	Judge the nearest Hole
1	7	JHT	Judge the nearest Hole from the nearest Tile
1	8	JST	Judge the second nearest Tile
2	1	MF	Move Forward
2	2	TR	Turn Right
2	3	TL	Turn Left
2	4	ST	Stay

are generated through crossover. In this simulation, the crossover rate of two individuals in GNPvs is the same, i.e., P_A=P_B. The algorithms need to iterate 1000 generations. The fitness function is $f = \sum_{w=1}^{N} DroppedTile(w) + \frac{N_J}{N_T}$, where N is the number of tileworlds, $DroppedTile(w)$ is the number of dropped tiles in tileworld w, N_J is the number of judgment nodes and N_T is the total number of nodes, respectively. The fitness function calculates the total number of dropped tiles in the tileworlds and the ratio of the judgment nodes and processing nodes. In this simulation, $N = 1$. For each agent, there are 200 time steps. Each processing node and judgment node takes 1 time step and 0.2 time step, respectively.

Fig.2.11 shows the average number of dropped tiles of the best individuals over 30 random trials in each generation of GNPvs and GNP. It is found from Fig.2.11 that although the increasing speed of the fitness of GNPvs is slower than GNP at first, GNPvs still gradually improves the structure of individuals, while GNP is trapped into the local optimum around 150th generation. Therefore, GNPvs gets the number of dropped tiles of 12.4 and 14.5 at the last generation when the crossover rates are 0.1 and 0.2, respectively, while GNP only gets 9.5 and 11.4.

Table 2.2: Parameters of simulation I

Parameter Name	GNPvs	GNP
Initial Size	60	60
Number of Individuals	300	300
Number of Elite	10	10
Crossover Size	290	290
Crossover Rate	Case 1: $P_A = P_B = 0.1$	0.1
	Case 2: $P_A = P_B = 0.2$	0.2
Number of Generations	1000	1000

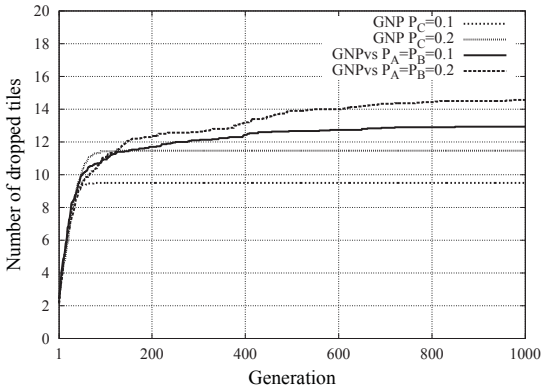


Figure 2.11: Curves of the number of dropped tiles in simulation I

It is found from Fig.2.11 that GNPvs works better than GNP with no mutation, which shows the effectiveness of crossover in GNPvs. Although the crossover in GNPvs can change the structure of individuals, i.e., change the number of nodes and the proportion of judgment nodes and processing nodes, the proposed method might be trapped into local optima, however it has the strong ability of generating new individuals by exchanging different number of nodes. That's why the crossover in GNPvs is better than uniform crossover in GNP.

2.3.3 Simulation II

In this simulation, three different tileworlds are used to train GNPvs and GNP and to validate the generalization ability of the algorithm, that is, 2 tileworlds

Table 2.3: Parameters of simulation II

Parameter Name	GNPvs	GNP
Initial Size	60	60
Number of Individuals	300	300
Number of Elites	10	10
Crossover Size	170	170
Crossover Rate	Case 1: $P_A = P_B = 0.1$	0.1
	Case 2: $P_A = P_B = 0.2$	0.2
	Case 3: $P_A = P_B = 0.3$	0.3
	Case 4: $P_A = P_B = 0.4$	0.4
	Case 5: $P_A = P_B = 0.5$	0.5
	Case 6: $P_A = 0.1$, $P_B = 0.3$	
	Case 7: $P_A = 0.2$, $P_B = 0.4$	
	Case 8: $P_A = 0.4$, $P_B = 0.5$	
Mutation Size	120	120
Mutation Rate	0.05	0.05
Number of Generations	1000	1000

for training and 1 tileworld for validating. From the validation data, the optimal size of GNPvs under particular parameters will be obtained.

The functions of judgement nodes and processing nodes, the initial number of nodes, the fitness function and the total time steps are the same as those of simulation I, while the number of tileworld is 2 in the training phase and 1 in the validation phase. The other parameters are shown in Table 2.3. The population size is 300, and during reproduction, the top 10 individuals are copied to the next population and these 10 individuals are picked up to validate in another tileworld to check the generalization ability. 170 individuals are generated by crossover. In this simulation, 8 different pairs of crossover rate in GNPvs are used to show the performances of GNPvs. Since the crossover rates of the last three cases are different, there are no simulation results of GNP in these cases. Besides, 120 individuals are created through mutation. The mutation rate is 0.05.

Fig.2.12 and Fig.2.13 shows the average number of dropped tiles and the average size of the best individuals over 30 random trials in the last generation of GNPvs and GNP in the training phase. It is found from Fig.2.12 that the

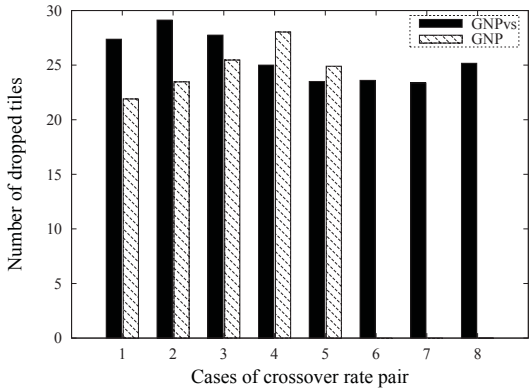


Figure 2.12: Average number of dropped tiles over the best individuals in the training phase

Table 2.4: Average number of dropped tiles in the training phase

Case	GNP	GNPvs
1	21.9	**27.4**
2	23.4	**29.1**
3	25.5	**27.8**
4	**28.0**	25.0
5	**24.9**	23.5
Average	24.7	**26.6**

crossover rates of the best GNPvs and GNP are different. It is because when the crossover rate is small, the exploration ability of GNP is low, which means that GNP cannot find more candidate solutions in the search space, while if the crossover rate is large, GNPvs individuals may exchanges too many nodes for individuals, which makes it hard to keep useful building blocks of individuals. Although the performance of GNPvs and GNP depends on the crossover rate, GNPvs has the highest performance when the crossover rate is 0.2, and GNPvs can increase the average number of dropped tiles by 7.7% over 5 cases, which is shown in Table 2.4. Therefore, it is confirmed that GNPvs has better performance than GNP, when the number of the training tileworlds is 2.

Besides, the growth of the size of individuals is a basic feature of GNPvs [37], however it is found from Fig.2.13 that the size of the individuals does not increase

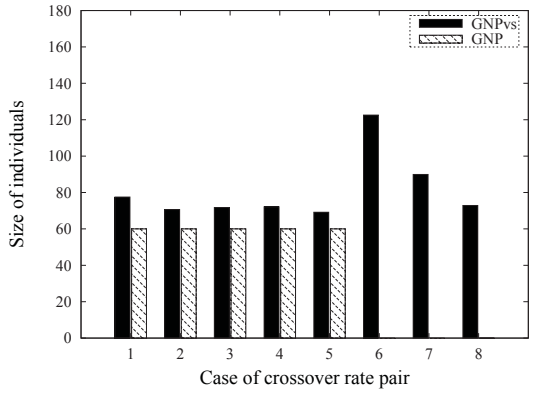


Figure 2.13: Average size of the best individuals

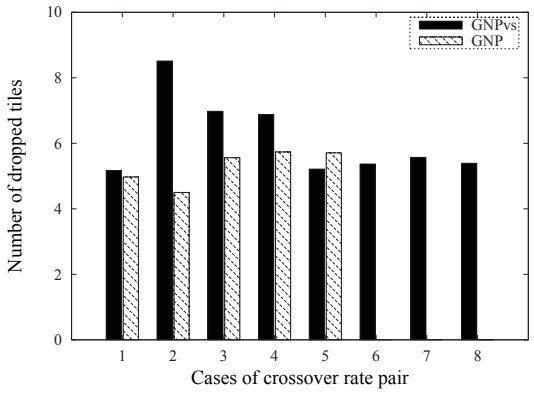


Figure 2.14: Average number of dropped tiles over the best individuals in the validating phase

rapidly from case 1 to case 5. From this point of view, the proposed method can alleviate the bloating phenomenon.

Fig.2.14 and Table 2.5 show the average number of dropped tiles of the best individuals in the last generation over 30 random trials in the validating phase. It is found from Fig.2.14 and Table 2.5 that GNPvs can push more tiles than GNP in most cases, and the average number of dropped tiles of both methods are 5.3 and 6.6, respectively, which means GNPvs can improve the performance by 24.5%. Therefore GNPvs has more generalization ability than GNP, because GNPvs can optimize the proportion between judgment nodes and processing nodes during crossover.

Moreover, the validation data is used to determine the optimal size of the

Table 2.5: Average number of dropped tiles in the validating phase

Case	GNP	GNPvs
1	5.0	**5.2**
2	4.5	**8.5**
3	5.6	**7.0**
4	5.7	**6.9**
5	**5.7**	5.2
Average	5.3	**6.6**

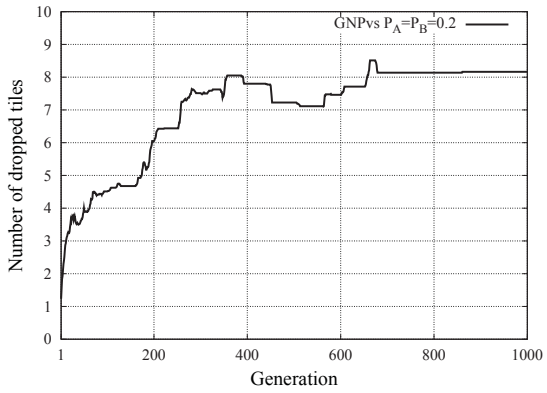


Figure 2.15: Average number of dropped tiles of Case 2 in the validating phase

individuals by checking the peak of the curve of the number of dropped tiles, since the individual has the highest generalization ability around this point. For example, Fig.2.15 and Fig.2.16 describe the curves of the average number of dropped tiles and the average size of individuals of Case 2 in the validating phase, respectively. By analyzing both figures, it is firstly found from Fig.2.15 that the best performance occurs around 660th generation, then the optimal size of the individual is obtained from Fig.2.16, that is, the number of nodes in the individual is 71.

The proposed method should be compared with other methods in order to show the effectiveness of it, on the other hand, GNP and other methods have been compared fully in the paper [23]. Since the simulation environments and parameters in the paper [23] are different from those of our paper, it is difficult to compare the performances between the proposed method and other methods

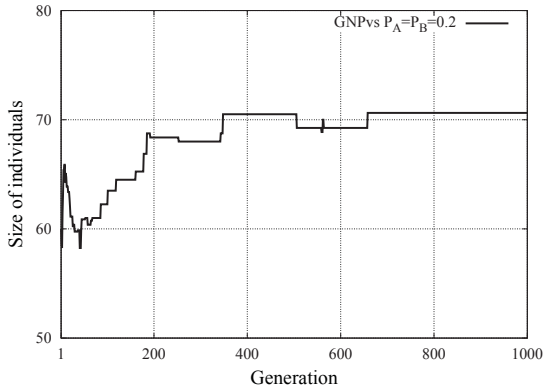


Figure 2.16: Average size of individuals of Case 2 in the validating phase

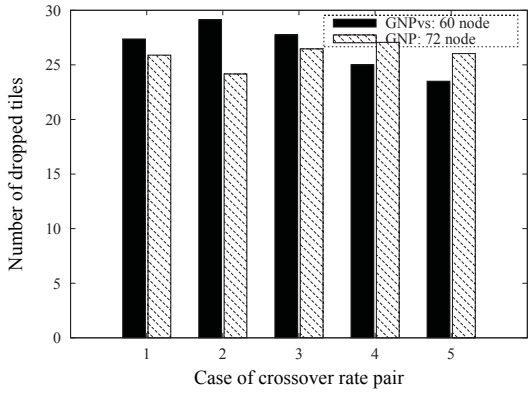


Figure 2.17: Average number of dropped tiles over the best individuals in the training phase

directly. But, roughly speaking, in the paper [23], GNP can increase the number of dropped tiles by 14.3%, 22.2% and 9.4% compared to GP-ADFs, GP and EP, respectively. While the proposed method can improve the performances compared to GNP by 7.7% and 24.5% in the training phase and validating phase, respectively, in this simulation. Therefore, the proposed method can improve the performance effectively.

2.3.4 Simulation III

Since the proposed method will increase the size of individuals and a good number of nodes means the higher expression ability of individuals, the different initial

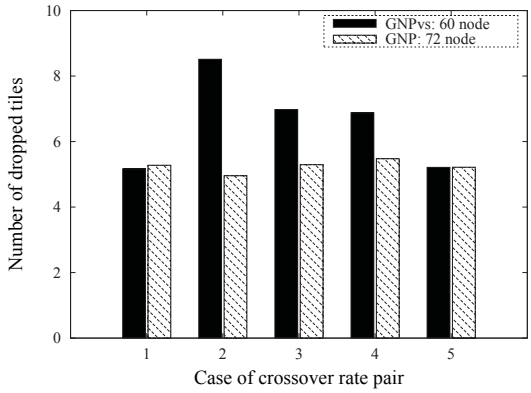


Figure 2.18: Average number of dropped tiles over the best individuals in the validating phase

number of nodes of GNPvs and GNP should be tested for confirming the effectiveness of the proposed method further.

From the former simulations, when the initial number of nodes of GNPvs is 60, the optimal size approaches to 71, therefore the initial number of nodes of GNP is set at 72 in this simulation, which means there are 6 nodes for each kind of nodes. 5 cases are studied in which the crossover rate changes from 0.1 to 0.5 like simulation II.

Fig.2.17 and Fig.2.18 show the average number of dropped tiles of the best individuals over 30 random trails in the training and validating phase . It is found from both figures that GNPvs still has better performances than GNP, because GNPvs can change the ratio of judgment nodes and processing nodes, even if the initial number of nodes of GNP increases. Therefore, it is confirmed that the proposed method can obtain the optimal size of the individuals.

2.3.5 Simulation IV

For further study, different initial number of nodes and the change of the ratio of judgment nodes and processing nodes are studied. In this simulation, 5 cases of the initial size are studied, which are 36, 48, 60, 72 and 84 nodes, respectively. For each case, the average number of dropped tiles are calculated over 5 different crossover rates, i.e. 0.1, 0.2, 0.3, 0.4 and 0.5, where each crossover rate has 30 random trials. The results are shown in Table 2.6 and Table 2.7. It is found from

Table 2.6: Average number of dropped tiles in the training phase

Initial size	GNP	GNPvs
36	23.1	**25.0**
48	24.3	**26.9**
60	24.8	**26.6**
72	**25.9**	25.8
84	23.3	**25.3**

Table 2.7: Average number of dropped tiles in the validating phase

Initial size	GNP	GNPvs
36	5.0	**6.3**
48	5.4	**6.3**
60	5.3	**6.5**
72	5.2	**6.3**
84	4.9	**6.2**

both tables that GNPvs has better performance than GNP on different initial sizes. Therefore, the proposed method can stably improve the effectiveness and efficiency of GNP.

Moreover, the evolution of the ratio of judgment nodes and processing nodes for these cases are shown in Fig.2.19, where the ratio is $\frac{N_J}{N_P}$, N_J is the number of judgment nodes and N_P is the number of processing nodes. It is found from Fig.2.19 that the ratio of judgment nodes and processing nodes decreases for all cases. Because the total steps of agents are fixed and the actions taken by agents are followed the processing nodes, which means that more processing nodes in the individual, more actions the agent may take to push tiles into hole, consequently the individual will have high fitness value. Therefore, the ratio of judgment nodes and processing nodes decrease. However, since the simulation environment is dynamic, the agent needs to analyze the information collected from its surrounding for taking proper actions, which means that a number of judgment nodes is necessary. Therefore, when the simulations are terminated, the ratios stay between

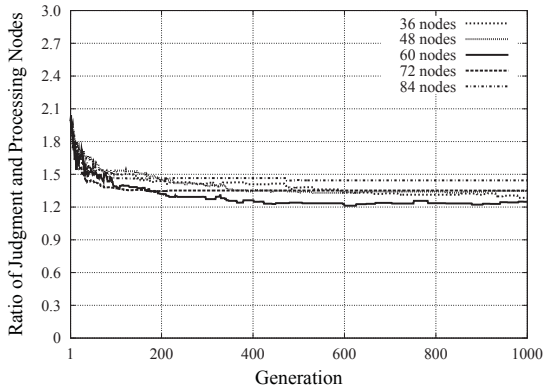


Figure 2.19: Ratio of judgment and processing nodes

Table 2.8: Parameters of verifying the optimal ratio

Case	Initial Size	Ratio
1	56	1.33
2	72	1.25
3	84	1.33

1.2 and 1.5.

In order to verify the validity of the obtained optimal ratio, three other cases of GNP are tested, where the initial size of GNP and the ratio of judgment nodes and processing nodes are described in Table 2.8. Since the number of nodes must be integer, the ratio in Table 2.8 cannot be exactly the same as Fig.2.19. However, the ratio is in the same range as Fig.2.19 from 1.2 to 1.5, which keeps the fair comparison. Table 2.9 shows the average number of dropped tiles in both the training phase and validating phase. Comparing Table 2.9 with Table 2.6 and Table 2.7, it is found that the number of dropped tiles by GNP increases, which means that the performance of GNP can be improved by using the optimal ratio. Therefore, it is confirmed from the simulations that the proposed method can obtain the optimal ratio, which is also helpful to the evolution of GNP.

Table 2.9: Average number of dropped tiles for verifying the optimal ratio

Case	Training phase	Validating phase
1	26.6	5.9
2	26.1	5.1
3	26.5	5.6

2.4 Conclusions

In this chapter, a new type of Genetic Network Programming (GNP)– Variable Size Genetic Network Programming (GNPvs) with Binomial Distribution is proposed. The size of the individuals in GNPvs is variable, as a result, GNPvs allows the individual to change its size during evolution. For implementing this feature, a new crossover is developed to replace the uniform crossover in GNP. The new crossover makes some nodes to move from one parent GNP to another parent GNP following binomial probability distribution. It is found from simulations that the proposed method can enhance the effectiveness of crossover during evolution in terms of obtaining the better performance than the conventional GNP and obtaining the optimal size of the individuals by introducing the validation mechanism. The performance of the proposed method is proved to be fairly good on agent-based problems.

Chapter 3

Variable Size Genetic Network Programming with Replacement (GNPvs-R)

3.1 Introduction

After GNP was proposed around 2000, many methods have been developed to improve the performance of GNP such as combining GNP with reinforcement learning [23], introducing symbiotic learning in GNP[24], upgrading the structure of GNP by defining macro node [33] and rule accumulation [34]. Recently, another improvement method named Variable Size Genetic Network Programming (GNPvs) [46] has been proposed. Since GNPvs is an extension of GNP, the basic structure of GNPvs is the same as GNP, i.e., a directed-graph representation with plural nodes of different functions. The main difference between GNPvs and GNP is that the size of individual is changeable in GNPvs, while it is fixed in GNP. Therefore, the size of each individual in GNPvs can be different and updated during evolution. In order to implement this feature, a new type of crossover is developed in GNPvs, which selects a number of nodes to move from one parent individual to another parent individual, then generate two new offspring. The advantage of GNPvs is that it can keep the effectiveness of crossover, which eventually improve the performance of GNP [46].

On the other hand, generalization ability is one of important criteria to measure the effectiveness of artificial learning systems, especially for supervised learning [47]. An algorithm of high generalization ability means that it can adapt to new environments as desired after training on a number of environments. In GNP, since the processing node is not compulsorily transferred to the start node and the transitions of the nodes follow the flow of network, it is possible that only a small part of individual will be well evolved, which might causes the reduction of generalization ability [48]. For solving this problem, GNP with Control Node and Multi-start nodes GNP are proposed in [48] and [49], respectively. The idea of both methods is to artificially separate the whole structure into several parts according to the number of additional nodes (control nodes in [48] and plural start node in [49]) and activate one part by one of these node under specific conditions.

Although the above methods certainly improve the performance of GNP, it might not be applied to GNPvs successfully. Since the performance of both methods depends on the cooperation of the partition of individuals and the connection of nodes in each partition, it is reasonable to find out the best combination of partitions and connections, considering that the number of nodes and the ratio of judgment nodes and processing are fixed in GNP. Otherwise, the number of nodes and function of nodes may rapidly change in GNPvs, which makes it difficult to evolve a good combination of partitions and connections. Therefore, a new method that is insensitive to the changeable number of nodes should be developed for improving the effectiveness of GNPvs.

Inspired by the theory of *Evolution by Gene Duplication* [20, 21], a kind of replacement mechanism is firstly proposed in GNPvs. In the proposed method, the non-frequently used nodes are replaced with the frequently used nodes in good individuals, i.e., building blocks, which can make the individual survive under the selection pressure and eventually might accumulate mutations that produce new features of individuals for adapting to the new environments.

3.2 GNPvs with Replacement

The basic idea of the proposed method is to replace the non-frequently used nodes with frequently used nodes in good individuals, thus the proposed method

is named GNPvs with Replacement (GNPvs-R). This method is proposed based on two points of view: building block hypothesis [8] and evolution by gene duplication [20, 21].

1. In GNP, since the behavior of the agent is controlled through the transitions of nodes, the performance of the agent is mainly depended on the blocks of the frequently used nodes, i.e., building blocks, while the non-frequently used nodes are barely contributed to the fitness of individuals. Therefore, the replacement can speed up the search by combing building blocks explicitly.

2. By the theory of evolution by gene duplication, the replacement is also a kind of duplication and deletion, i.e., copy the frequently used nodes and delete non-frequently used nodes, therefore, it can keep the competitiveness of the individual, and make the individual have the potential to accumulate mutations for generating new features to adapt new environments.

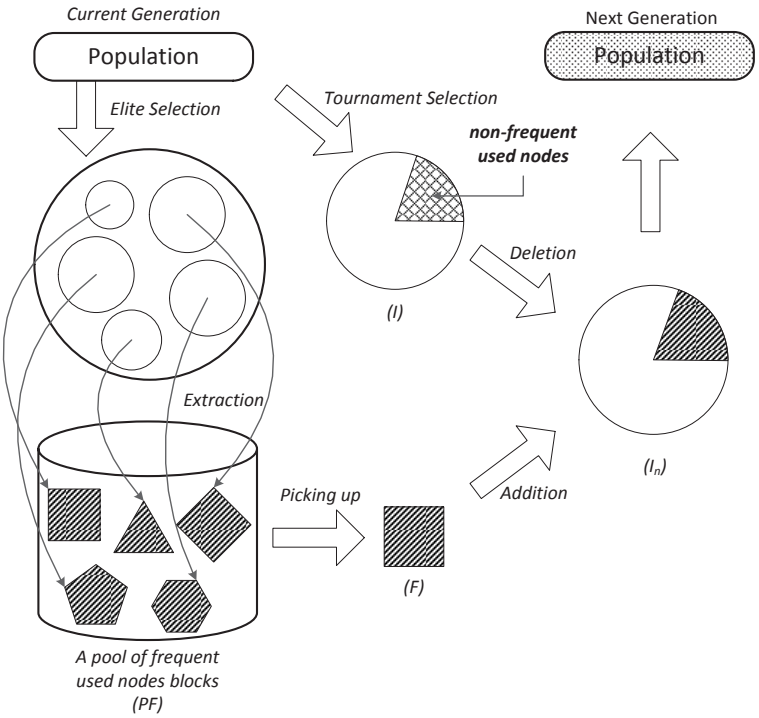


Figure 3.1: Outline of replacement

3.2.1 Outline of replacement

Fig.3.1 shows the outline of the replacement mechanism, in which the circle represents the individual of GNPvs-R and the circle with larger radius means the larger size of individuals, and the hatching in the pool of frequently used nodes blocks shows the set of frequently used nodes and different types of hatching mean different combinations of nodes.

3.2.2 Procedure of replacement

The arrow flow of Fig.3.1 explains the following procedure of replacement.

Step 1: From current population, N elite individuals are selected for extracting blocks of frequently used nodes.

Step 2: For each selected individual, a set of frequently used nodes is extracted by Procedure 2, and put to a pool of frequently used nodes blocks PF.

Step 3: From current population, an individual I is selected by tournament selection for replacement

Step 4: Randomly pick up a set of frequently used nodes F, i.e., building block, from PF.

Step 5: Combine I and F by Procedure 3 to generate a new individual I_n.

Step 6: Put the new individual I_n to next population.

Step 7: Repeat Step 3 to Step 5 until a predefined number of individuals are generated by replacement.

After replacement, some new individuals are generated, in which the non-frequently used nodes are replaced with frequently used nodes from elite individuals, in other words, the most useful blocks of nodes are contributed to evolution, which increases the search speed. Moreover, the extraction of blocks of frequently used nodes is executed every generation, which means that pool PF will update the information of individuals generation by generation for avoiding the premature.

Procedure 2 Algorithm of frequently used nodes extraction

Input: individual I
VT: threshold of visiting times
Output: A set of frequently used nodes F
1: $F \leftarrow \emptyset$
2: **for** $node_i$ in I **do**
3: **if** visiting time of $node_i > VT$ **then**
4: $F \leftarrow F \cup node_i$
5: **end if**
6: **end for**
7: **return** F

Procedure 3 Algorithm of new individual generation

Input: individual I
F: a set of frequently used node
Output: A new individual I_n
1: Rank the nodes in I by the visiting time in ascending order
2: Remove the worst $|F|$ nodes from I to create a new individual I_n
3: **for** $node_i$ in F **do**
4: $I_n \leftarrow I_n \cup node_i$
5: **end for**
6: Update the connections of I_n
7: **return** I_n

3.3 Simulations

In this section, the performances of the proposed method are also evaluated and compared with the GNP and GNPvs using Tileworld.

3.3.1 Simulation environments

Since the purpose of the proposed method is to increase the generalization ability of GNPvs, the simulation environment is a little different from that in chapter 2. In the training phase, the same type of the tileworld is prepared as chapter 2 like Fig.3.2. However a new pair of hole and tile will be randomly generated when the agent pushes a tile into a hole in the simulations which is different from the fixed number of tiles and holes in chapter 2. Besides, in the testing phase, randomly

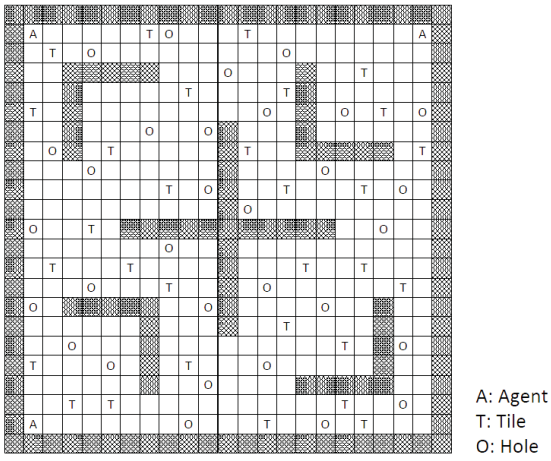


Figure 3.2: Example of tileworld used in training phase

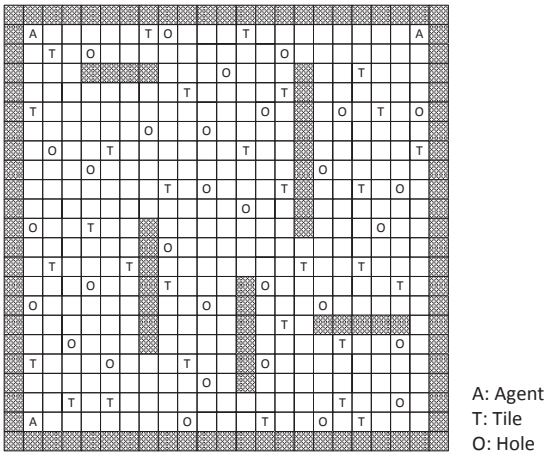


Figure 3.3: Example of tileworld used in testing phase

generated tileworld like Fig.3.3 is introduced, in which the locations of obstacles, holes and tiles are totally randomly created, therefore the tileworld may have big difference from the one in the training phase.

On the other hand, the functions of judgment nodes and processing nodes are the same as in chapter 1, which are described in Table 3.1 for convenience. JF, JB, JL, JR, JT, JH, JHT and JST are 8 kinds of judgment nodes, while MF, TR, TL and ST are 4 kinds of processing nodes in both algorithms. Moreover 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. Thus, each judgement node has 5 branches. Besides, there are totally 60 nodes (12 kinds of nodes $\times$ 5 for each kind of node) in each individual at first.

Table 3.1: Functions of judgment nodes and processing nodes

NT	ID	Symbol	Function
1	1	JF	Judge Forward
1	2	JB	Judge Backward
1	3	JL	Judge Left
1	4	JR	Judge Right
1	5	JT	Judge the nearest Tile
1	6	JH	Judge the nearest Hole
1	7	JHT	Judge the nearest Hole from the nearest Tile
1	8	JST	Judge the second nearest Tile
2	1	MF	Move Forward
2	2	TR	Turn Right
2	3	TL	Turn Left
2	4	ST	Stay

3.3.2 Simulation I

In this simulation, the performances of GNP, GNPvs and GNPvs-R are compared using 2 training tileworlds and 8 testing tileworlds. The parameters used in the simulations are described in Table 3.2. The population size is 300, and during reproduction, the top 10 individuals are directly copied to the next gen-

Table 3.2: Parameters of simulation I

Parameter Name	GNPvs-R	GNPvs	GNP
Initial Size	60	60	60
Number of Individuals	300	300	300
Number of Elite	10	10	10
Crossover Size	120	170	170
Crossover Rate	0.1	0.1	0.1
Replacement Size	50		
Mutation Size	120	120	120
Mutation Rate	0.05	0.05	0.05
Number of Generations	500	500	500

eration, on the other hand, these 10 individuals are used for frequently used nodes extraction. Besides, for GNP and GNPvs, the number of individuals generated by crossover is 170, while its number is 120 for GNPvs-R, since there are 50 individuals are produced by replacement. The mutation number for these methods is 120. In this simulation, the crossover rate is 0.1 and mutation rate is 0.05. The algorithms need to iterate 500 generations. The fitness function is $f = \sum_{w=1}^{N} DroppedTile(w)$, where N is the number of tileworlds and $DroppedTile(w)$ is the number of dropped tiles in tileworld w, respectively, i.e., the fitness function calculates the total number of dropped tiles in the tileworlds. In this simulation, $N = 2$ for the training phase and $N = 1$ for each tileworld in the testing phase. For each agent, there are 200 time steps. Each processing node and judgment node takes 1 time step and 0.2 time step, respectively.

Fig.3.4 shows the average fitness of the best individuals in the training phase over 30 random experiments. Since a pair of hole and tile will be created after pushing one tile into the hole by the agent, the fitness curve is fluctuated. From Fig.3.4, it is found that the proposed method GNPvs-R has the fastest search speed and get the highest fitness value among three methods. Finally, the agents controlled by GNPvs-R push around 29 tiles, while GNPvs and GNP push 26 and 23, respectively. As a result, GNPvs-R can improve the performance by about 11.5% and 26% comparing GNPvs and GNP, respectively. Therefore, the proposed method can improve the performance by increasing the search speed

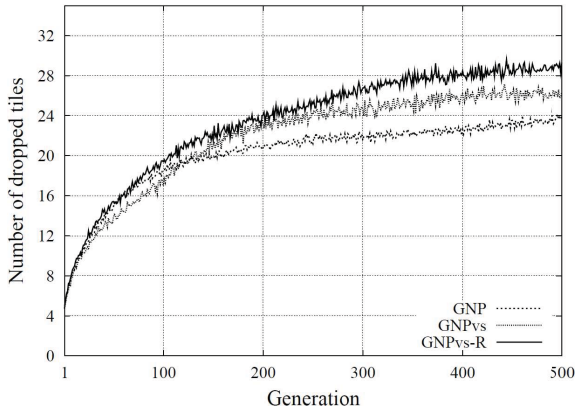


Figure 3.4: Average fitness of the best individuals in training phase of simulation I

Table 3.3: Standard deviations of node visiting times

	1st generation	last generation	Reduction
GNPvs-R	93.7	38	59.4%
GNPvs	123.9	57.2	53.8%
GNP	120.5	79.9	33.8%

when the useful blocks of nodes are extracted and combined with other individuals explicitly.

Moreover, Table 3.3 shows the standard deviations of node visiting times of three methods. The larger value in Table 3.3 means that the smaller part of nodes in the individual is used. Therefore, it is found from Table 3.3 that the standard deviations of all methods are reduced, which means that more nodes become used during evolution. More used nodes of the individuals could mean that the individual has more potential to evolve new features for dealing with new environments. In order to confirm the above, the generalization ability of these methods are tested using 8 different tileworlds.

Fig.3.5 to Fig.3.12 shows the average fitness value of the best evolved individuals generation by generation in the training phase using 8 different tileworlds when they are evolved in training phase. From these figures, it is found that GNPvs-R obtains the highest fitness value among three method in all 8 cases.

Therefore, GNPvs-R increases the generalization ability compared with GNPvs and GNP by duplicating the frequently used nodes, which can keep the competitiveness of the individual making it have the potential to accumulate mutations for generating new features to adapt new environments.

3.3.3 Simulation II

In this simulation, it is studied how the number of individuals produced by replacement affects the results using the same tileworlds as simulation I, i.e., 2 training worlds and 8 testing worlds. Therefore, the simulation parameters in this simulation is also the same as simulation I, except the replacement size R and crossover size C, i.e., $R = 30, C = 140$; $R = 70, C = 100$ and $R = 90, C = 80$, respectively.

Fig.3.4 shows the average fitness of the best individuals in the training phase over 30 random experiments, and from Fig.3.5 to Fig.3.12 shows the average fitness value of the best evolved individuals generation by generation in the training phase using 8 different tileworlds. It is found from these figures that the performance among three different parameter settings are not significantly different. In other words, the proposed method is not sensitive to the replacement number. Therefore, the performance of GNPvs-R can improve the performance of GNPvs even if there are not so many individuals generated by replacement.

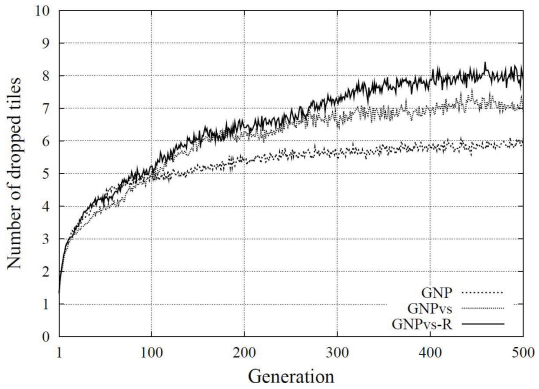


Figure 3.5: Average fitness of the best individuals in testing phase of simulation I for world 1

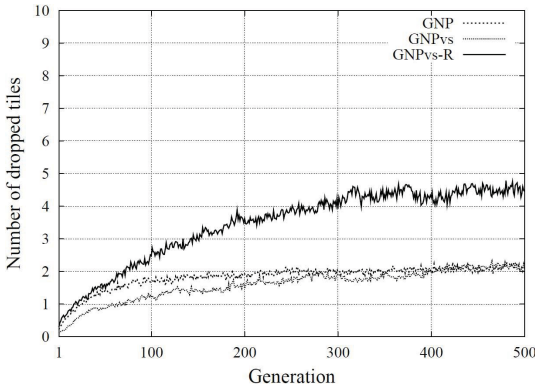


Figure 3.6: Average fitness of the best individuals in testing phase of simulation I for world 2

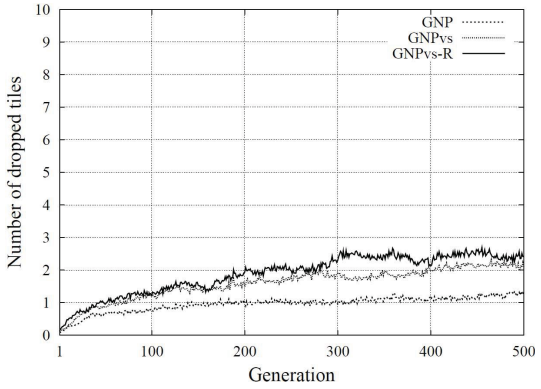


Figure 3.7: Average fitness of the best individuals in testing phase of simulation I for world 3

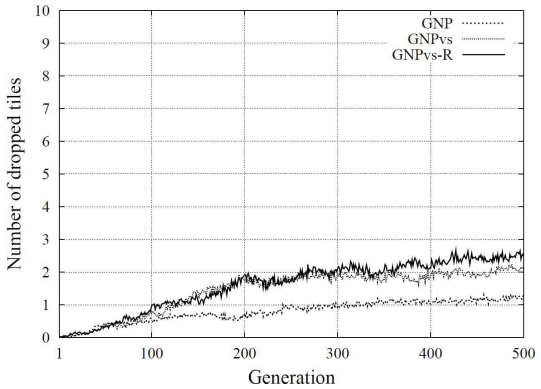


Figure 3.8: Average fitness of the best individuals in testing phase of simulation I for world 4

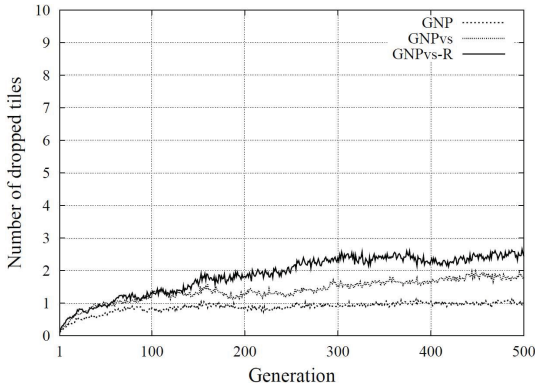


Figure 3.9: Average fitness of the best individuals in testing phase of simulation I for world 5

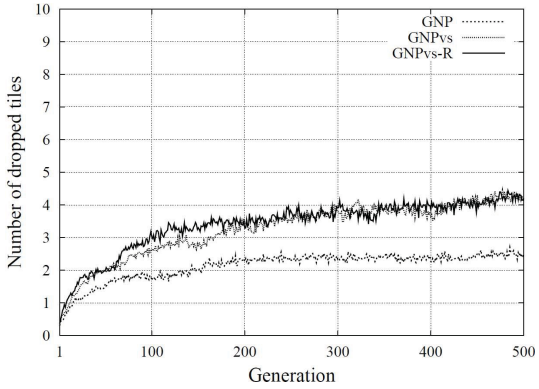


Figure 3.10: Average fitness of the best individuals in testing phase of simulation I for world 6

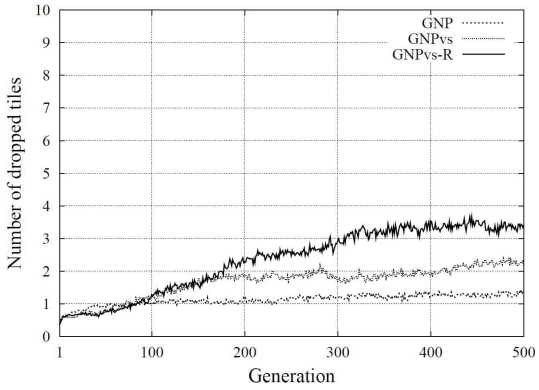


Figure 3.11: Average fitness of the best individuals in testing phase of simulation I for world 7

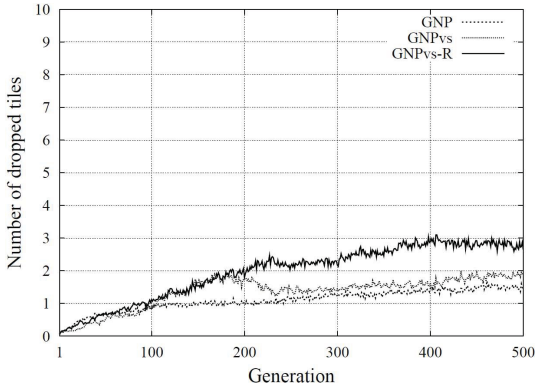


Figure 3.12: Average fitness of the best individuals in testing phase of simulation I for world 8

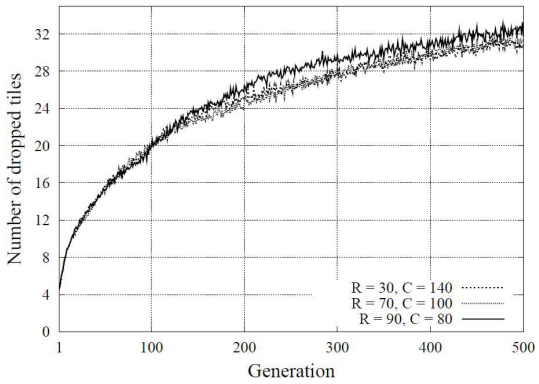


Figure 3.13: Average fitness of the best individuals in training phase of simulation II

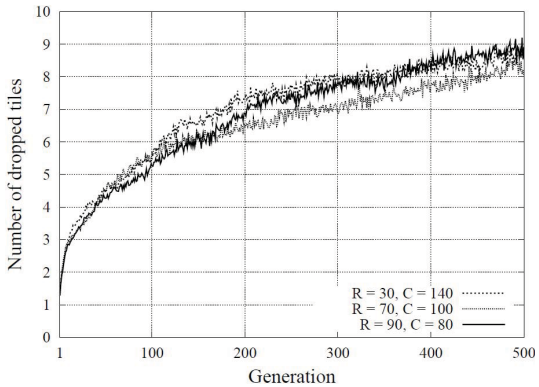


Figure 3.14: Average fitness of the best individuals in testing phase of simulation II for world 1

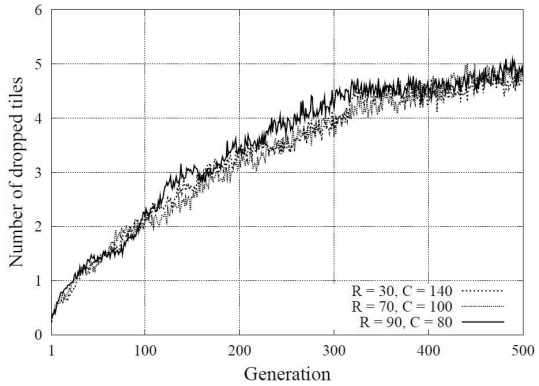


Figure 3.15: Average fitness of the best individuals in testing phase of simulation II for world 2

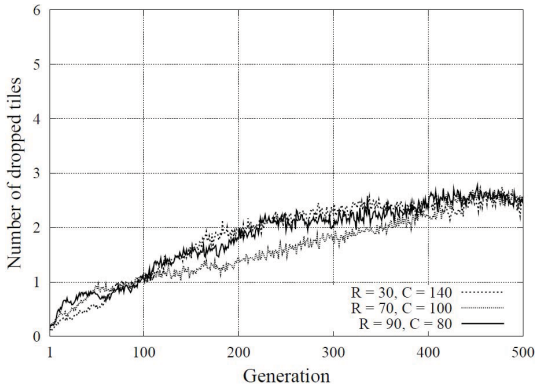


Figure 3.16: Average fitness of the best individuals in testing phase of simulation II for world 3

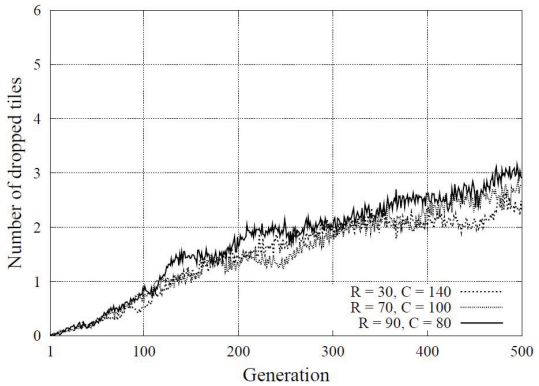


Figure 3.17: Average fitness of the best individuals in testing phase of simulation II for world 4

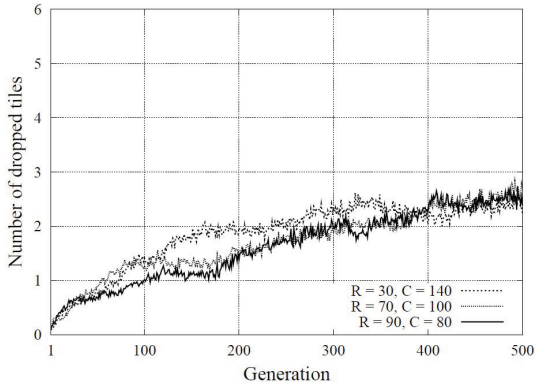


Figure 3.18: Average fitness of the best individuals in testing phase of simulation II for world 5

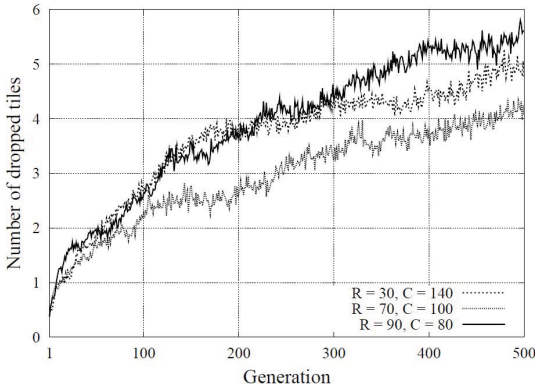


Figure 3.19: Average fitness of the best individuals in testing phase of simulation II for world 6

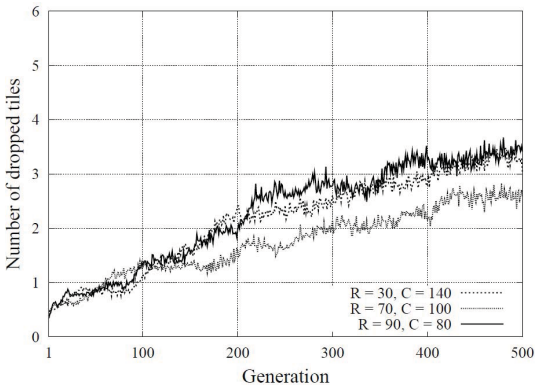


Figure 3.20: Average fitness of the best individuals in testing phase of simulation II for world 7

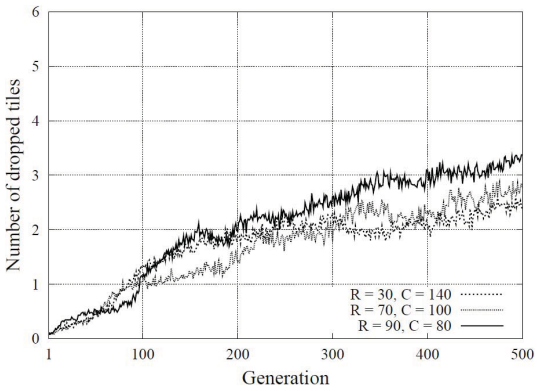


Figure 3.21: Average fitness of the best individuals in testing phase of simulation II for world 8

3.4 Conclusions

In this chapter, the replacement mechanism is introduced in the Variable Size Genetic Network Programming (GNPvs) in order to improve the generalization ability of GNPvs. The proposed method is named GNPvs with Replacement (GNPvs-R), in which the sets of frequently used nodes are extracted from elite individuals and these sets are used to replace the non-frequently used nodes of individuals. By this mechanism, the whole structure of the individual is evolved and the most valuable information from elite individuals contributes to the evolution of the individuals. The effectiveness of the proposed method is verified on the dynamic environments of tileworlds, and it is proved to increase the generalization ability of GNPvs exactly.

Chapter 4

Genetic Network Programming for Automatic Program Generation with Mapping Mechanism (GNP-APGm)

4.1 Introduction

The center to Artificial Intelligence is to make computers automatically solve problems. Therefore, there could be methods to generate computer programs automatically corresponding to problems i.e. Automatic Program Generation (APG). Automatic program generation, in other words, automatic programming or program induction is a way to obtain a program without explicitly programming it. Then, the program becomes the solution to cope with specific problem. Several evolutionary algorithms like Genetic Programming (GP) [9, 10, 11], Cartesian Genetic Programming (CGP) [13, 14], Gene Expression Programming (GEP) [15, 16] and Grammatical Evolution (GE) [17, 18] have been proposed with much success in 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 termi-

nal set (‘1’, ‘x’, ‘y’, ‘z’...) [9]; 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 to search food and take the food to their nest [9].

Nowadays, some studies on GNP for automatic program generation (GNP-APG) has been conducted, and the simulation result shows good performances of it [50, 51]. But, in these papers, only static problems are used to verify the performance of GNP-APG. Therefore, the objective of this chapter is to improve GNP-APG to deal with the time dependent environment problems like Tileworld and to solve the problem that an individual is a solution.

The improved GNP-APG algorithm is named GNP-APGm, in which a kind of genotype-phenotype mapping process is introduced to create programs [38] . Fig.4.1 describes the outline of the mapping process and comparison between GNP-APGm and biological systems. During the procedure of the program generation, the graph structure of GNP-APGm (genotype) is firstly transcribed into a sequence of processing nodes. Then, this sequence is translated into program fragments by applying mapping rules. Finally, these fragments are assembled together as an executable program (phenotype). As noted in [17] and [38], 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-APGm extends from the conventional GNP, but there are many differences between them.

- An individual of GNP-APGm 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-APGm 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 di-

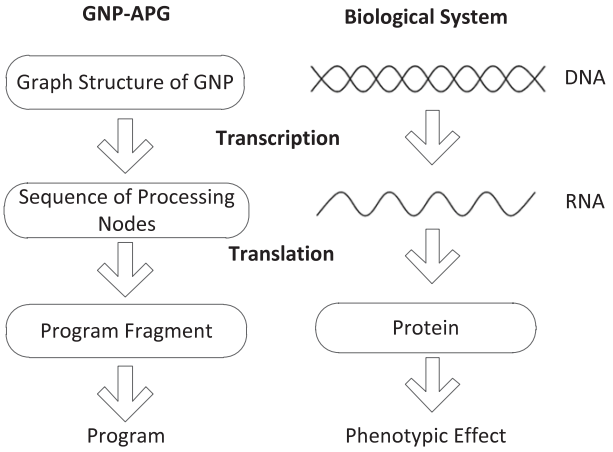


Figure 4.1: Genotype-phenotype mapping of GNP-APGm

rectly probe the information from environments, then use this information to make a decision and tell agents what to do.

- GNP-APGm 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.

4.2 GNP-APGm

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

4.2.1 Basic structure of GNP-APGm

It has been mentioned in section 4.1 that GNP-APGm has the outside memory and exchanges the information with it, which makes the basic structure of GNP-APGm a little different from GNP. Fig.4.2 shows the basic structure of GNP-APGm. Compared with Fig.1.1, the outside memory is added. In Fig.4.2, 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 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.

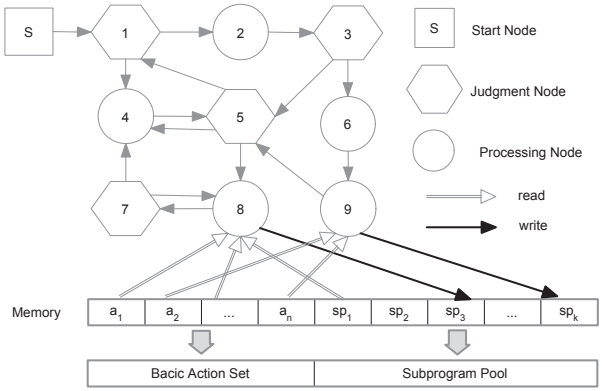


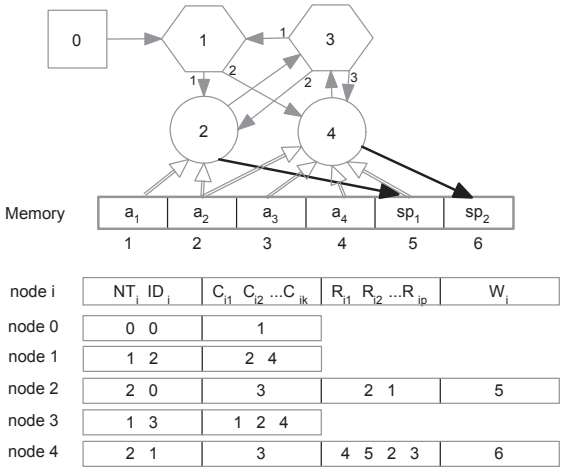
Figure 4.2: Basic structure of GNP-APGm

In addition, there are also three kinds of nodes in GNP-APGm, but the roles of judgment nodes and processing nodes are not the same as GNP. In GNP-APGm, 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.

Fig.4.3 shows the representation of GNP-APGm. The chromosome of GNP-APGm 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 the fourth basic actions or subprograms from the memory. For example, *node 2* is this kind of processing node in Fig.4.3. 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 Fig.4.3, 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

Figure 4.3: Representation of GNP-APGm



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 0	3	2 1	5
node 3	1 3	1 2 4		
node 4	2 1	3	4 5 2 3	6

Figure 4.4: Small but complete example of GNP-APGm

4.2.2 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 "ACT" 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.

Table 4.1: Functions of processing nodes

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

Table 4.2: Functions of judgment nodes

ID	Function
2	2-branches
3	3-branches

Fig.4.4 is a small but complete example of GNP-APGm. The number on the arrows means the index of branches. Table 4.1 shows the function of processing nodes, Table 4.2 describes the function of judgment nodes and Table 4.3 represents the argument number of actions. The detail of each node is shown in Fig.4.4. For example, NT_2 and ID_2 of *node 2* are 2 and 0, respectively, therefore 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.

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

Table 4.3: Argument numbers of actions

Action	Argument number
a_1	0
a_2	0
a_3	0
a_4	3

and the function is "ACT". The function "ACT" 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 "ACT(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 "ACT(a_2, a_1)" is generated. After that, the subprogram is written to location 5 of the memory, and sp_1 in the memory changes to "ACT(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 function 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 actions. In this case, *node 4* gets a_4 as a judgment action. From Table 4.3, 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 "ACT(a_2, a_1)", the entire representation of sp_2 is "IF(a_4, ACT(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 4.4. GNP-APGm 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.

Table 4.4: Pseudocode of sp_2

Suppose a_4 has three judgment results v_1, v_2 and v_3 Let re be the return value of a_4

if $re == v_1$:
  a_2;
  a_1;
else if $re == v_2$:
  a_2;
else if $re == v_3$:
  a_3

4.2.3 Flowchart of GNP-APGm

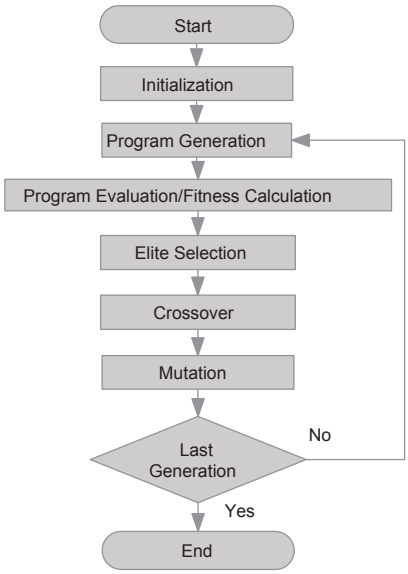


Figure 4.5: Flowchart of GNP-APGm

Fig.4.5 shows the flowchart of GNP-APGm.

1) Parameters are set. Hundreds of GNP-APGm 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. Therefore 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.

4.2.4 Advantages of GNP-APGm

GNP-APGm 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.

['ACT', ['JF', ['JL', ['ACT', ['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']], ['ACT', ['JF', ['JL', ['JB', ['TL'], ['HD', ['MF'], ['TR'], ['TR'], ['TR'], ['ST']], ['TR']], ['TL'], ['TR']], ['HD', ['MF'], ['TR'], ['TR'], ['TR'], ['ST']], ['TL']], ['MF']], ['TL']], ['MF']]

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

- More solution candidates. Since the individual of GNP-APGm 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. Therefore, GNP-APGm can increase the probability to find better solutions.

- Sufficient use of the graph structure. As noted in [48], the conventional GNP cannot use the whole graph structure, because the judgment nodes usually select only several specific branches. But, GNP-APGm selects the branches in turn, as a result, each branch has the same chance to select. By this way, GNP-APGm can make full use of the graph structure.

- Keeping the diversity of the genotype. Fig.4.6 shows two different GNP-APGm 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*. Although the entire sequences between two individuals are different, the sequences of processing nodes are the same as *node 2⇒node 4⇒node 2*. 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-APGm keeps the diversity of the genotype.

- Making 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.

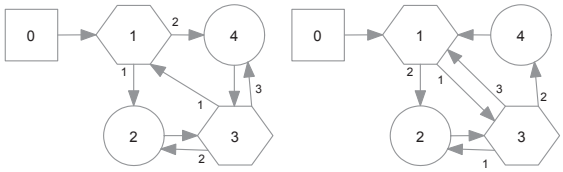


Figure 4.6: Two different GNP-APGm individuals with different connections

4.3 Simulations

In this section, the performances of the proposed method are evaluated and compared with the conventional GNP using Tileworld.

4.3.1 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 environments [42]. 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.

Since the purpose of the proposed method is very different from the previous chapter, i.e., to propose a new method on automatic program generation, the simulation environment is also a little different from before, in order to focus on a new method. 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. Fig.4.7 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 Fig.4.8 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 Fig.4.8. 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.

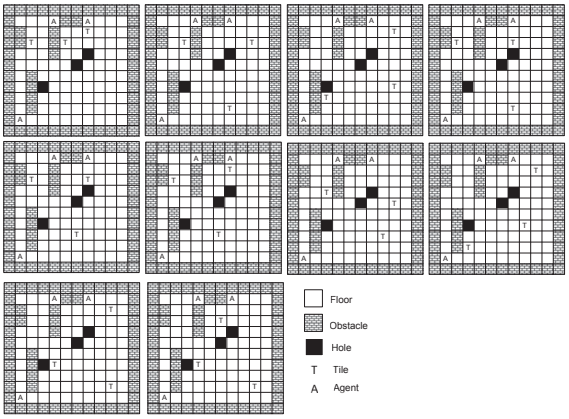


Figure 4.7: Tileworlds for training phase

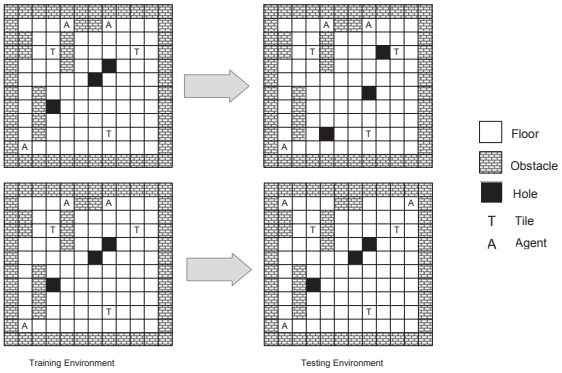


Figure 4.8: Changes of Tileworld for testing phase

4.3.2 Simulation configurations

The basic action set of GNP-APGm is described in Table 4.5, the functions of processing nodes is the same as Table 4.1 and the functions of judgment nodes are shown in Table 4.6. 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.

Table 4.5: 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

Table 4.6: Functions of judgment nodes in simulations

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

Each individual of GNP-APGm contains 60 nodes including 15 judgment nodes (3 kinds of nodes × 5 for each kind) and 45 processing nodes (5 for "ACT" processing node, 40 for "IF" processing node). Each individual of GNP has also 60 nodes (12 kinds of nodes × 5 for each kind of node).

The parameters used in simulations is described in Table 4.7. The population size is 301, and during the evolution procedure, the best individual is copied to the next population. 120 individuals are generated through crossover, while 180 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, i.e., starting from the start node, 60 nodes are visited, and the maximum length of programs is 6000 bytes in GNP-APGm. The number of subprograms is 12.

Table 4.7: Parameters of simulations

Parameter Name	GNP-APGm	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. 4.1.

$$\begin{aligned} &Fitness = C_{tile} \times DroppedTile \\ &+C_{dist} \times \textstyle\sum_{t \in T}(InitDi(t) - FinDi(t)) \\ &+C_{stp} \times (TotalStep - UsedStep) \end{aligned} \tag{4.1}$$

where, $DroppedTile$ is the number of tiles the agents 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 its nearest hole. T is the set of suffixes of tiles. $TotalStep$ is a predefined maximal number of time

steps all the agents can take, and $UsedStep$ represents the number of time steps 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 time steps for each agent), respectively. In the simulations, the average of the fitness values over ten Tileworlds is calculated to show the performances of GNP-APGm and GNP.

4.3.3 Simulation results

Fig.4.9 shows the average fitness value of the best training results of GNP-APGm and the conventional GNP over 50 random seeds. In the 500th generation, the average of the best fitness values of GNP-APGm and GNP are 319.71 and 297.30, respectively. At first, the fitness values of GNP-APGm 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-APGm is higher than the conventional GNP, since GNP-APGm can use of the graph structure fully and keep the diversity of the genotype. At last, GNP-APGm can gain a larger fitness value obviously than the conventional GNP.

Fig.4.10 shows the average length of the best programs of GNP-APGm in the training phase. As noted in [52], the growth of the length of the program is inherent in the proposed methods, since the length of the program is not fixed but varies. Fig.4.10 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 Fig.4.10 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.

Fig.4.11 and Fig.4.12 show the average fitness value of each Tileworld of GNP-APGm 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-APGm and 25.2 in the conventional GNP, respectively. It can be seen from Fig.4.11 that

GNP-APGm performed a little worse than the conventional GNP only in world 1. While GNP-APGm 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-APGm and 84.1 in the conventional GNP, respectively. It is also found from Fig.4.12 that GNP-APGm 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. Therefore, 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.

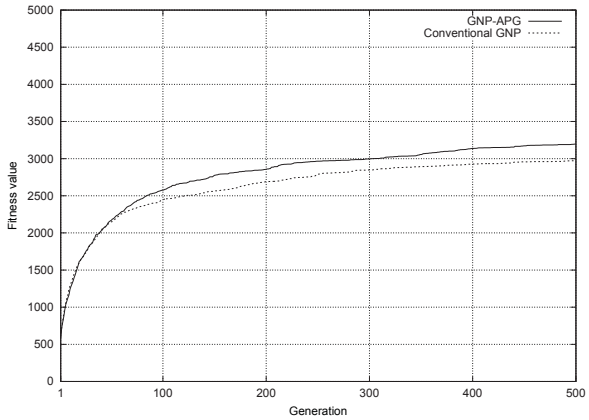


Figure 4.9: Simulation result of training phase

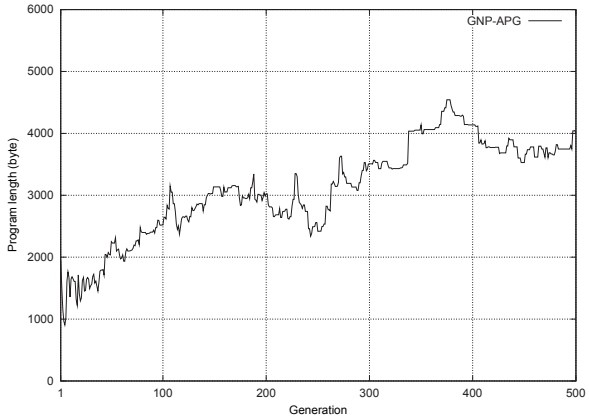


Figure 4.10: Program length of training phase

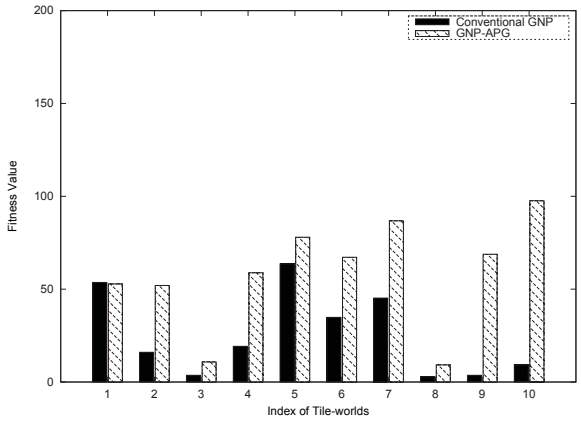


Figure 4.11: Simulation result of testing phase when the location of holes is changed

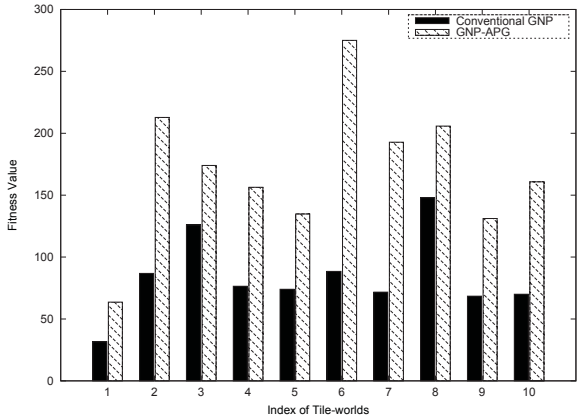


Figure 4.12: Simulation result of testing phase when the location of agents is changed

In addition, another set of Tileworld with different obstacle configurations are used to verify the performance on GNP-APGm and the conventional GNP. The ten different types of obstacles are shown in Fig.4.13. In each type, there are ten worlds with different configurations on holes or tiles. Therefore, there are totally 100 worlds. Fig.4.14 shows the average of the best fitness values over 10 random seeds and 100 worlds (totally 1000 cases simulations). It is found from Fig.4.14 that GNP-APGm can also have better performance than the conventional GNP in many kinds of worlds.

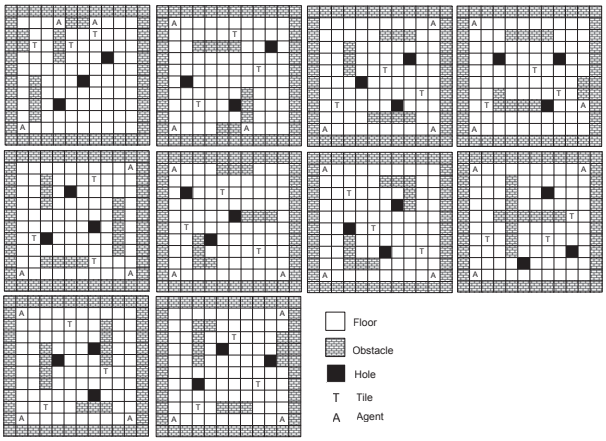


Figure 4.13: Tileworld with different obstacle configurations

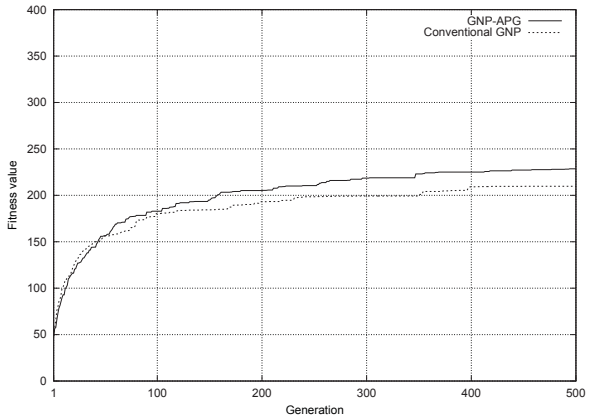


Figure 4.14: Simulation result of Tileworld with different obstacle configurations

4.3.4 Simulation analysis

It is found from the training results that GNP-APGm could get higher fitness values than the conventional GNP. In addition, because the individual of GNP-APGm is a program generator, it can create several candidate programs stored in the memory, and they are evaluated to pick up the best one, which increases the probability to find better solutions. Besides, GNP-APGm can sufficiently use the graph structure since the branches of judgment nodes in GNP-APGm have the same probability to select, which also helps to improve the performance of the proposed method. Table 4.8 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 GNP-APGm and GNP are statistically different from each other. The conclusions derived from the above results are convincing.

Table 4.8: Statistical fitness values of training phase

	GNP-APGm	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-APGm is much better than the conventional GNP. There are 8 kinds of judgment nodes and 4 kinds of processing nodes in the conventional GNP, as a result, 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 judges two kinds of situations, then take an action. But, many kinds of Tileworlds are to be dealt with in many cases, thus two judgments are not enough. While there are 40 "IF" processing nodes in GNP-APGm, then the program generated by GNP-APGm will contain many judgments. Therefore, GNP-APGm works better than the conventional GNP in the testing phase. Table 4.9 and Table 4.10 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 the performance of GNP-APGm and GNP are significantly different.

Table 4.9: Statistical fitness values of testing phase when the location of holes is changed

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

Table 4.10: Statistic fitness values of testing phase when the location of agents is changed

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

4.3.5 Parameters discussion

GNP-APGm 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 Fig.4.7 are used to study the effect of different parameters.

The number of transitions and the maximum length of the program are used to study controlling 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-APGm cannot generate a complex program to deal with complicated environments, therefore the fitness becomes low; on the other hand, 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. Fig.4.15 and Fig.4.16 show the average fitness values and lengths of programs by different parameter settings over

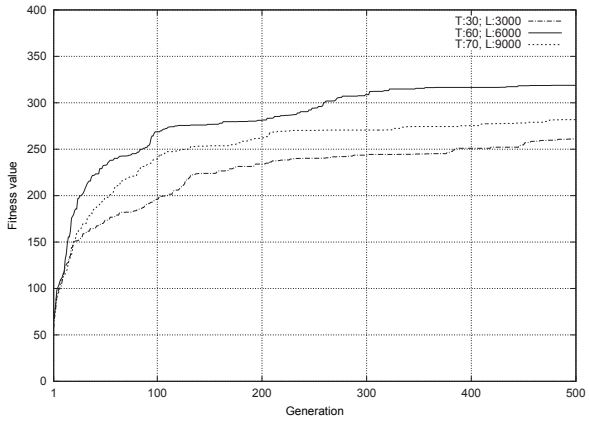


Figure 4.15: Fitness value of programs with different number of transitions and different maximum length of programs

10 random seeds. T and L mean the number of transitions and the maximum length of programs, respectively. It is found from Fig.4.15 and Fig.4.16 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.

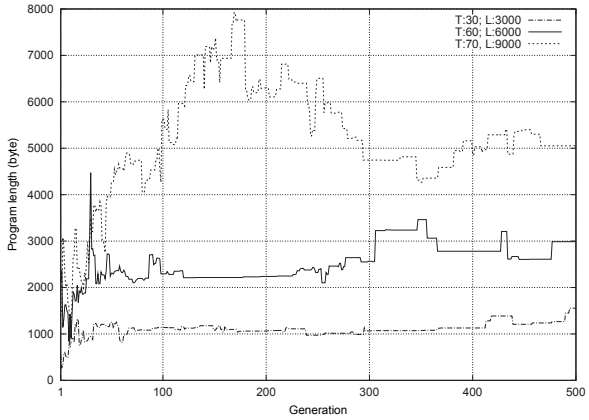


Figure 4.16: Length of programs with different number of transitions and different maximum length of programs

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

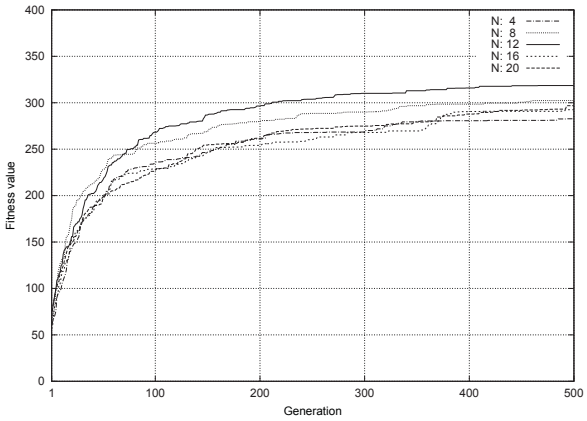


Figure 4.17: Fitness value of programs with different number of subprograms

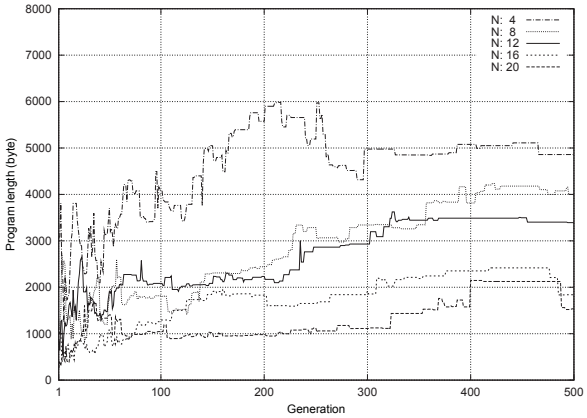


Figure 4.18: Length of programs with different number of subprograms

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. Fig.4.17 and Fig.4.18 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 Fig.4.17 that when N equals 12, the algorithms can get the highest performance compared with other settings. Fig.4.18 confirms the tendency described before, i.e., when N is small, the length becomes large, while when N is large, the length becomes small.

4.4 Conclusions

In this chapter, automatic program generation with Genetic Network Programming using mapping mechanism has been proposed and applied to the Tileworld problem for agent control. The proposed method introduces two functions "IF" and "ACT" 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 the advantages of using graph structures fully, keeping the diversity of the genotype and using the building blocks, it can find better solutions than the conventional GNP. The simulations confirms that the proposed method is more robust than the conventional GNP.

Chapter 5

Subroutine embedded Genetic Network Programming for Automatic Program Generation with Mapping Mechanism (GNPsr-APGm)

5.1 Introduction

In the real world, a three-step method is often employed to solve complex problems, which first tries to break the given problem into several subproblems, then finds solutions to cope with each subproblem, finally seeks a way to assemble the solutions of the subproblems into a solution for the original problem. Likewise, the programming task is usually divided into several steps to discover a main function and some meaningful subroutines for finding a useful complicated program effectively to deal with the overall problem.

The advantages of decomposing a program into subroutines are:

- Breaking a complex programming task into some simpler steps. In other words, a high dimensional problem is divided into some low dimensional problems which can be solved more easily.

- Reusability of the existing codes. Sometimes, different problems might have an identical part. If a subroutine is obtained for the identical part, it can be invoked by other problems, which increases the efficiency of finding a solution to the overall problems.
- Decreasing the size of the program. By calling subroutines, some duplicate codes are not necessary in the program, so the size of the program is reduced significantly.

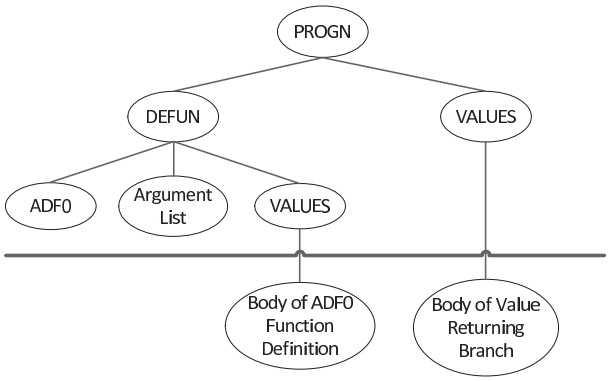


Figure 5.1: Overall structure of the program generated by GP with ADFs

GP also implements the subroutine mechanism by Automatic Defined Functions (ADFs) [10]. Fig.5.1 shows an example of the program generated by GP with ADFs, which contains one function-defining branch and one result-producing branch.The left part of the figure is the definition of a subroutine and the right part is the main function. Table 5.1 shows the description of the symbols used in Fig.5.1, which are the keywords of programming language Lisp. As noted in [10], GP with ADFs can significantly improve the performance of the algorithm.

Besides, Gene Expression Programming (GEP) [15, 16, 53] and Grammatical Evolution (GE) [17, 18, 54, 55] have been also proposed as new evolutionary algorithms to evolve computer programs. GEP encodes a program into a string, and translates it into expression trees, while GE defines some grammars, and uses a binary string to generate the program by matching the grammars. Moreover, there are subroutine mechanisms in both methods, which are Automatically Defined Functions in GEP [53], Dynamically Defined Functions [54] and Grammar based function definition [55] in GE.

Table 5.1: Description of symbols in GP

Symbol	Description
PROGN	A special form that makes each of its arguments to be evaluated in sequence and then returns the value of the last one
DEFUN	Define a function
ADF0	The name of the function
VALUES	Return values after evaluating the body of the function

Based on these advantages, some research on Genetic Network Programming with Automatic Program Generation (GNP-APG) have been also conducted. In [50, 51], the symbolic regression problems are studied and in [56, 57], the dynamic environment problem like Tile-world [42] is discussed. GNP-APGm introduces a kind of genotype-phenotype mapping process to create legal programs [17, 38]. Since the environment in Tile-world always changes and the agents should work together to finish the mission, the search space of the problem is large. In addition, there are several repeated procedures used by agents during the execution in the Tile-world problem like how to move to the nearest tile. These procedures could be considered as subroutines. From this point of view, GNP-APGm should be improved. Therefore, subroutines are introduced to enhance the performances of GNP-APGm in this chapter and a new method named Subroutine embedded Genetic Network Programming for Automatic Program Generation with Mapping Mechanism (GNPsr-APGm) is proposed.

5.2 Subroutine embedded GNP for APG with Mapping Mechanism

Subroutine embedded Genetic Network Programming for Automatic Program Generation with Mapping Mechanism (GNPsr-APGm) is an extension of the algorithm of GNP-APGm, which implements the subroutine mechanism. GNPsr-

APGm can generate a program containing a main function and some subroutines.

5.2.1 Basic structure of Subroutine embedded GNP for APG with Mapping Mechanism

In order to generate a main function and several subroutines, more than one directed graph structure (GNP structure) are needed in GNPsr-APGm. Fig.5.2 describes the basic structure of GNPsr-APGm. The circle represents the directed graph structure, which has been shown in Fig.4.2. The left part of Fig.5.2 is the main function part, which is designed to create the main function of a program. The main function contains the control logic of an agent, and it might also call subroutines to finish specific tasks. The right part of Fig.5.2 is the subroutines part, which is used to create subroutines. In this part, there are several independent GNP structures, where each structure works in the same way as the individual of GNP-APGm (Fig.4.2) to generate its own subroutine. Right now, the subroutines cannot call each other. These subroutines would be invoked by the main function.

In addition, the structure of memory of GNPsr-APGm has been also changed compared with Fig.4.2. The memory consists of three parts: basic action set, subroutine set and subprogram pools. Firstly, there is only one basic action set, which is shared by the main function part and subroutines part. This set contains the basic actions of agents like "Move Forward", "Turn Right", etc. It is immutable, which means the elements in this part are only read. Secondly, the subroutines set is also exclusive and immutable. It stores the name of subroutines like "SR0" and "SR1", i.e., subroutine0 and subroutine1. Only the main part has a chance to read these elements. During the procedure of the program generation, the way to deal with these elements is the same as the one in the basic action part. The third part is the subprogram pools which are mutable. Each GNP structure has their own subprogram pool to store subprograms generated during the procedure of the program generation. Generally, the main function part reads elements from the basic action set, subroutine set or its own subprogram pool, then writes the generated subprogram to the subprogram pool, meanwhile, each GNP structure in subroutines part reads elements from the basic action set or

its own subprogram pool, then write the created subroutines to the subprogram pool. When the main function invokes the subroutines, the subroutines stored in the subprogram pools of the subroutines part are picked up and executed.

Moreover, the arrows in Fig.5.2 explains the relationship between GNP structure and the memory as follows.

1. Arrow with line - GNP structure can read contents from memory.
2. Arrow with dash line - GNP structure can read contents from the memory and write generated subprograms to the memory.
3. White arrow - Subroutine set can call the subprograms in the subprogram pool.

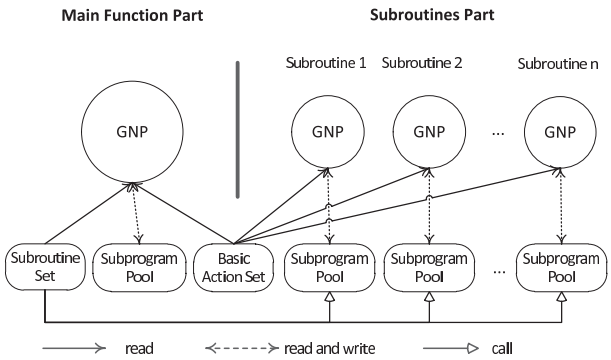


Figure 5.2: Basic structure of Subroutine embedded GNP for APG with Mapping Mechanism

5.2.2 Procedure of program generation

Since GNPsr-APGm is derived from GNP-APGm, it also uses the genotype-phenotype mapping to generate programs, which is mentioned in chapter 3.

Fig.5.3 is an example of the main function part of GNPsr-APGm. The upper part of the figure shows the structure of the main function part, while the lower part is the genotype. The number on the arrows means the index of branches. Table 5.2 represents the meaning of NT. Table 5.3 shows the function of processing nodes, Table 5.4 describes the function of judgment nodes and Table 5.5

represents the argument number of actions. The detail of each node is shown in Fig.5.3. For example, NT_4 and ID_4 of *node 4* are 2 and 1, respectively, so node 4 is a processing node which is used to create conditional statement. It connects to *node 3* according to the connection gene. Moreover, it reads from location 2, 3, 1 and 6 of the memory, then write to location 4 of the memory depending on the read addresses and write address, respectively. Besides, as mentioned above, the memory is divided into three part. The left part is basic action set, which contains three basic actions of agent, i.e., a_1, a_2 and a_3. The middle part is subprogram pool, which stores subprograms generated by the proposed method. In the example, there are two subprograms obtained during the procedure of program generation, i.e., sp_1 and sp_2. The right part is subroutine set. The length of subroutine set is 2, which means that the individual of GNPsr-APGm has two subroutine GNP structure. The symbols $SR0$ and $SR1$ are the labels of subroutines, by which GNPsr-APGm can call subroutines.

According to Fig.4.1, the first step of the program generation is transcription, i.e., converting the graph structure of GNPsr-APGm to a sequence of processing nodes. 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, it is a processing node, so the first node in the sequence of processing nodes is *node 2*. Then, the next node becomes *node 3*, because the connection gene of *node 2* is 3. In the same way as *node 1*, *node 3* is a judgment node, and it selects the branch depending on the times of its visits. This is the first time for *node 3* to be visited, 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 also a processing node and it is appended to the sequence. Now, the sequence becomes *node 2*⇒*node 4*. The transcription continues until the predefined maximal number of transitions is reached. Suppose the transcription stops when *node 4* is visited, the sequence of *node 2*⇒*node 4* is generated for the next step.

After the transcription is finished, the sequence of the processing nodes should be translated into program fragments. The first node in the sequence is *node 2*. Its identity is 0, so the function is "ACT". The function "ACT" is used to create the sequential statement. Suppose that two actions "action1" and "ac-

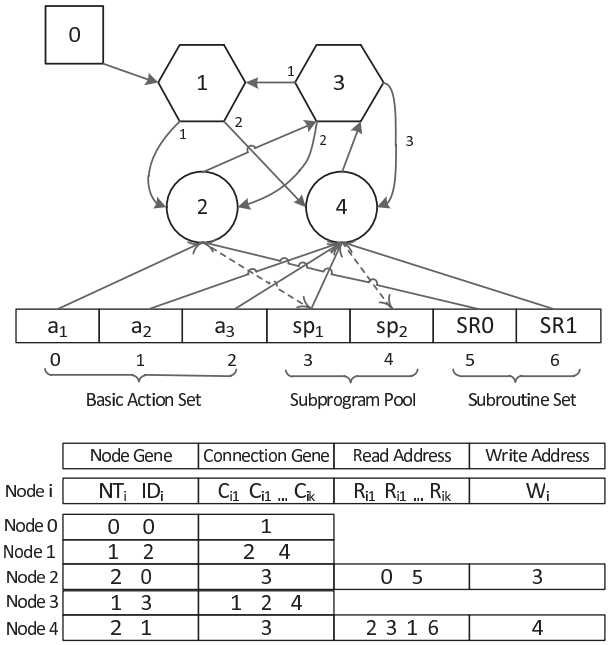


Figure 5.3: An example of the main function part in Subroutine embedded GNP for APG with Mapping Mechanism

Table 5.2: Node type of GNPsr-APGm

NT	Node Type
0	Start node
1	Judgment node
2	Processing node

Table 5.3: Functions of processing nodes

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

tion2" are obtained from the memory, and the rule to create the statement is "ACT(action1|subprogram1|subroutine1, action2|subprogram2|subroutine2)" which means that the agent takes "action2", "subprogram2" or "subroutine2" followed by "action1", "subprogram1" or "subroutine1". The read addresses of *node 2* are 0 and 5, so actions a_1 and $SR0$ are picked up. According to the rule, a subprogram "ACT(a_1, $SR0$)" is generated. After that, the subprogram is written to location 3 of the memory, and sp_1 changes to "ACT(a_1, $SR0$)", since the write address of *node 2* is 3. The next node in the sequence is *node 4*, which is used to create conditional statement. The function of the conditional statement is "IF", and the rule is "IF(action1, action2|subprogram1|subroutine1, action3|subprogram2|subroutine2, ...)". 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 other actions. In this case, *node 4* gets a_3 as a judgment action. From Table 5.5, a_3 needs three arguments, which means that a_3 has three return values x_1, x_2 and x_3, so sp_1, a_2 and $SR1$ are picked up depending on the read addresses of 3, 1 and 6, respectively. Then, a subprogram "IF(a_3, sp_1, a_2, $SR1$)" is created and stored at location 4 of the memory, because the write address of *node 4* is 4. By this way, sp_2 changes to "IF(a_3, sp_1, a_2, $SR1$)". As sp_1 has become "ACT(a_1, $SR0$)", the entire representation of sp_2 is "IF(a_3, ACT(a_1, $SR0$), a_2, $SR1$)". *Node 4* is the last node in the sequence, so the procedure of the translation of the main function part is completed. sp_2 is a main function containing sequential and conditional statements and invokes two subroutines such as SR0 and SR1.

Table 5.4: Functions of judgment nodes

ID	Function
2	2-branches
3	3-branches

The procedure of the subroutine generation is the same as the main function, the only difference is that the GNP structure cannot read the elements from subroutine set, but only communicates with its own subprogram pool. After the

Table 5.5: Argument number of actions

Action	Argument number
a_1	0
a_2	0
a_3	3

Table 5.6: Pseudocode of the program

Suppose a_3 has three judgment results x_1, x_2 and x_3 Let re be the return value of a_3
Main function

if $re == x_1$:
  a_1;
  SR0;
else if $re == x_2$:
  a_2;
else if $re == x_3$:
  SR1;

SR0

if $re == x_1$:
  a_2;
else if $re == x_2$:
  a_1;
  a_2;
else if $re == x_3$;
  a_1;

SR1

a_2;
  a_2;
  a_1;

procedure of the subroutine generation is accomplished, suppose subroutine0 is "IF(a_3, a_2, ACT(a_1, a_2), a_1)", and subroutine1 is "ACT(a_2, ACT(a_2, a_1)))". The pseudocode of the program is shown in Table 5.6.

5.2.3 Genetic operator of Subroutine embedded GNP for APG with Mapping Mechanism

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

GNPsr-APGm provides elite selection and tournament selection. Elite selection is simple, and 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. Each part of the individual exchanges its nodes independently, which means the main function part swap nodes with the main function part of the other individual, while the subroutines part exchanges nodes in the subroutines part of the other individual. After crossover, two new individuals are produced and moved to the next generation.

Mutation just needs one individual which is picked up through tournament selection. All data in the gene except NT_i and ID_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 GNPsr-APGm: connection mutation and address mutation. Connection mutation changes the connections between nodes, in concrete, the values of C_{i1}, C_{i2}... are changed. But, the value should be in the range from 0 to (the total number of nodes - 1). Address mutation means the read address or write address of the individual is changed, which implies the value of R_{i1}, R_{i2} ... , or W_i is updated. The values of the read address R_{i1}, R_{i2} ... should be from 0 to (the length of memory - 1), i.e., from 0 to 6 in Fig.5.3. The number of write address W_i should be in the range of the index of the subprogram pool, i.e., from

3 to 4 in Fig.5.3.

5.2.4 Flowchart of Subroutine embedded GNP for APG with Mapping Mechanism

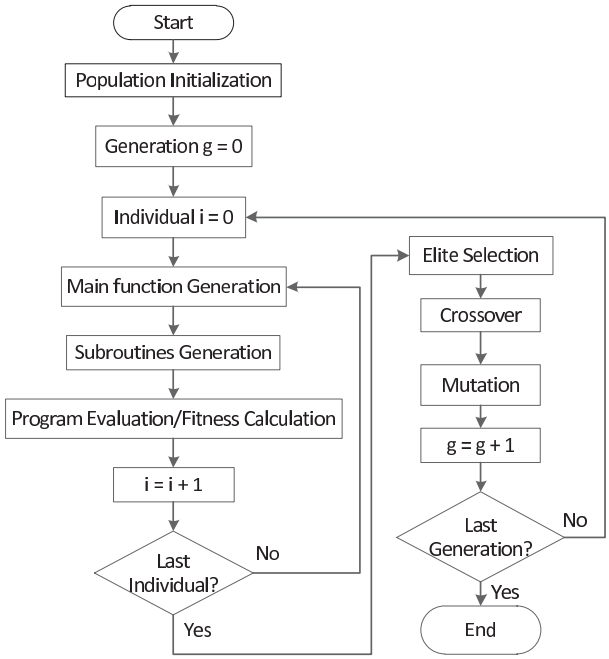


Figure 5.4: Flowchart of Subroutine embedded GNP for APG with Mapping Mechanism

Fig.5.4 shows the flowchart of GNPsr-APGm.

Step 1: Parameters like crossover rate and mutation rate are set. Hundreds of individuals of GNPsr-APGm are generated randomly. One action from the basic action set is randomly assigned to all cells of the subprogram pools. Any useful subroutine is not used from the start, and all the subroutines are automatically generated by the proposed method, which make the comparison fair.

Step 2: Let generation $g = 0$.

Step 3: Let individual $i = 0$.

Step 4: Individual i generates a main function based on the rules mentioned in the previous subsection.

Step 5: Individual i generates several subroutines based on the rules mentioned in the previous subsection. After finishing step 4 and 5, a program containing a main function and some subroutines are obtained.

Step 6: The programs generated in step 4 and 5 are evaluated in the simulation environment. Since one individual can create several programs, the best evaluation value among these programs becomes the fitness value of the individual.

Step 7: Increase the index of the individual, $i = i + 1$

Step 8: Judge whether the last individual has been evaluated. If the answer is yes, go to the next step, otherwise, go to step 4.

Step 9: The individual which has the highest fitness value is copied to the next generation directly.

Step 10: Two individuals are selected by tournament selection as parents. These individuals exchange their nodes under the crossover rate. Two new individuals are generated and moved to the next generation.

Step 11: One individual is selected by tournament selection as a parent. The individual randomly changes its gene according to the mutation rate. One new individual is generated and moved to the next generation.

Step 12: Increase the index of the generation, $g = g + 1$

Step 13: Determine whether it is the last generation. If the answer is yes, then the algorithm ends, otherwise, go to step 3.

5.2.5 Advantages of Subroutine embedded GNP for APG with Mapping Mechanism

The advantages of Subroutine embedded GNP for APG with Mapping Mechanism are described as follows.

- The total number of nodes of GNPsr-APGm and GNP-APGm is the same. Since the individual of the GNPsr-APGm is divided into two parts, the number of nodes in each part becomes smaller, which reduces the size of the search space and improves the efficiency of the global search .
- Because the subroutine part evolves independently, one subroutine can focus on a specific subproblem, which makes it possible to obtain the solution of the subproblem more easily.
- The repeatable parts of the program are picked up as subroutines, which reduces the size of the program. It can also alleviate the bloating problem.

5.3 Simulations

5.3.1 Simulation on artificial ant problem

In order to show the effect of unique features of GNPsr-APGm, an artificial ant problem is used firstly.

5.3.1.1 Simulation environments

The environment for the problem is the San Mateo trail [10], which consists of nine parts, each made up of a square of 13-by-13 grid containing different discontinuities in the sequence of food (black and grey grids represent food and gap, respectively) like Fig.5.5. The goal of the problem is to find a program for controlling the movement of an artificial ant to find and eat all the foods in all nine parts of the San Mateo trail. The ant can only sense the food which is located in the square in front of it. And the actions of the ant are limited to moving forward, turning left and turning right.

5.3.1.2 Simulation configurations

The basic action set of GNPsr-APGm and GNP-APGm is described in Table 5.7. The functions of processing nodes is the same as Table 5.3 and the functions of judgment nodes are shown in Table 5.8. FA returns whether there is food in front

of the ant, i.e., "True" and "False", while MV, TR and TL do not have the return value.

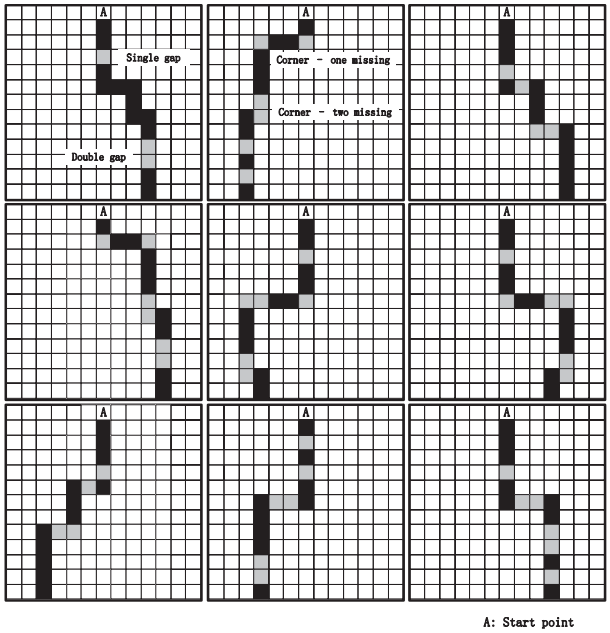


Figure 5.5: The nine parts of the San Mateo trail for the artificial ant problem

Table 5.7: Basic action set of artificial ant problem

Symbol	Action	Argument
FA	Judge whether there is food ahead	2
TR	Turn Right	0
TL	Turn Left	0
MV	Move forward and pickup food	0

In order to make sure that the comparison is fair, the number of nodes of two algorithms of GNPsr-APGm and GNP-APGm should be the same. Therefore, each individual of GNP-APGm contains 35 nodes including 15 judgment nodes (3 kinds of nodes $\times$ 5 for each kind) and 20 processing nodes (10 for "ACT" processing node, 10 for "IF" processing node). On the other hand, because the subroutine part of GNPsr-APGm contains only one GNP structure "SR0" in this

Table 5.8: Functions of judgment nodes of artificial ant problem

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

simulations, 14 nodes (6 judgment nodes and 8 processing nodes) are distributed to the main function part and 21 nodes (9 judgment nodes and 12 processing nodes) are distributed to the GNP structure of the subroutine part in order for both algorithms to have the identical number of nodes.

Table 5.9: Parameters of artificial ant problem

Parameter Name	GNPsr-APGm	GNP-APGm
Number of Individuals	2001	2001
Number of Elite	1	1
Crossover Size	1000	1000
Crossover Rate P_c	0.2	0.2
Mutation Size	1000	1000
Mutation Rate P_m	0.03	0.03
Number of Generations	100	100
Number of Transitions in Main Function	50	50
Number of Transitions in Subroutine		20
Length of Subprogram Pool in Main Function	12	12
Length of Subprogram Pool in Subroutine		4
Maximal Length of Program	3000 bytes	3000 bytes

The parameters used in the simulations are described in Table 5.9. The population size is 2001, and during reproduction, the best individual is copied to the next population, 1000 individuals are generated through crossover, while 1000 individuals are created through mutation. The crossover rate is 0.2 and mutation rate is 0.03. The algorithms need to iterate 100 generations. During the program generation, the maximum number of transitions in GNP-APGm and the

main function part of GNPsr-APGm is 50, while the number of transitions of the subroutine part is 20. The maximum length of programs is 3000 bytes in both GNP-APGm and GNPsr-APGm (1000 bytes for the main function and 2000 bytes for the subroutine). The length of the subprogram pool of GNP-APGm and the main function part of GNPsr-APGm is 12, while the length of the subprogram pool of the subroutine part is 4. In each case of the San Mateo Trail, the ant can take a total of 120 time steps, i.e., each action of "TR", "TL" or "MV" costs one time step. The fitness value equals the total number of food left in nine parts of the San Mateo Trail, therefore when the value becomes 0, the solution is found.

5.3.1.3 Simulation result and analysis

Fig.5.6 describes the success rate of GNPsr-APGm and GNP-APGm of the San Mateo Trail based on 25 runs. It is found from Fig.5.6 that GNPsr-APGm and GNP-APGm have the success rate of 96% and 80%, respectively in the last generation.

The following is a 1310 byte program generated by GNP-APGm, which makes the ant eat all the food.

['ACT', ['IF', 'FA', ['IF', 'FA', ['ACT', ['IF', 'FA', ['MV'], ['TL']], ['MV']], ['TR']], ['MV']], ['ACT', ['IF', 'FA', ['IF', 'FA', ['ACT', ['IF', 'FA', ['MV'], ['TL']], ['MV']], ['TR']], ['MV']], ['ACT', ['IF', 'FA', ['IF', 'FA', ['ACT', ['IF', 'FA', ['MV'], ['TL']], ['MV']], ['TR']], ['MV']], ['ACT', ['IF', 'FA', ['IF', 'FA', ['ACT', ['IF', 'FA', ['MV'], ['TL']], ['MV']], ['TR']], ['MV']], ['ACT', ['IF', 'FA', ['IF', 'FA', ['ACT', ['IF', 'FA', ['MV'], ['TL']], ['MV']], ['TR']], ['MV']], ['ACT', ['IF', 'FA', ['IF', 'FA', ['ACT', ['IF', 'FA', ['MV'], ['TL']], ['MV']], ['TR']], ['MV']], ['ACT', ['IF', 'FA', ['IF', 'FA', ['ACT', ['IF', 'FA', ['MV'], ['TL']], ['MV']], ['TR']], ['MV']], ['ACT', ['IF', 'FA', ['IF', 'FA', ['ACT', ['IF', 'FA', ['MV'], ['TL']], ['MV']], ['TR']], ['MV']], ['ACT', ['IF', 'FA', ['IF', 'FA', ['ACT', ['IF', 'FA', ['MV'], ['TL']], ['MV']], ['TR']], ['MV']], ['ACT', ['IF', 'FA', ['IF', 'FA', ['ACT', ['IF', 'FA', ['MV'], ['TL']], ['MV']], ['TR']], ['MV']], ['ACT', ['IF', 'FA', ['IF', 'FA', ['MV'], ['TL']], ['ACT', ['TR'], ['MV']]], ['TL']]]]]]]]]]]]]

The following is a main function and a subroutine obtained by GNPsr-APGm, which is a 326 byte program and accomplishes the target

Main function

['IF', 'FA', ['MV'], ['ACT', ['SR0'], ['IF', 'FA', ['MV'], ['ACT', ['SR0'], ['IF', 'FA', ['MV'], ['ACT', ['ACT', ['ACT', ['TL'], ['TL']], ['MV']], ['ACT', ['MV'], ['TR']]]]]]]

Subroutine

['ACT', ['MV'], ['IF', 'FA', ['MV'], ['ACT', ['TL'], ['IF', 'FA', ['MV'], ['ACT', ['ACT', ['TR'], ['TR']], ['IF', 'FA', ['MV'], ['TL']]]]]]

Each square bracket pair of the program embraces a statement, i.e., basic actions, subprograms or subroutines. Since the two basic statements of the proposed method are 'IF' and 'ACT', the program consists of the statements like

['IF','FA',[...],[...]] and ['ACT', [...],[...]]. The subroutine will be decoded in Table 5.10. This subroutine makes the ant detect the squares around it. When the food exists, the ant will move to the food and eat it, otherwise it will keep stay. By calling this subroutine, the main function can make the ant check the food located two squares ahead of it.

Table 5.10: Code of subroutine

```
MV
IF FA
   MV
ELSE
   TL
   IF FA
      MV
   ELSE
      TR
      TR
      IF FA
         MV
      ELSE
         TL
```

By comparing both programs, it is found that GNPsr-APGm can generate much shorter programs than GNP-APGm for the same problems. Moreover, it is found from the trajectories of the artificial ant for the third and seventh trails by GNPsr-APGm in Fig.5.7 that GNPsr-APGm actually discovers a useful subroutine for solving the artificial ant problem.

5.3.2 Simulation on the tileworld problem

For further verifying the effects of introducing subroutines to GNP-APGm, the performances of the proposed method are evaluated and compared with GNP-APGm and conventional GNP using the Tile-world.

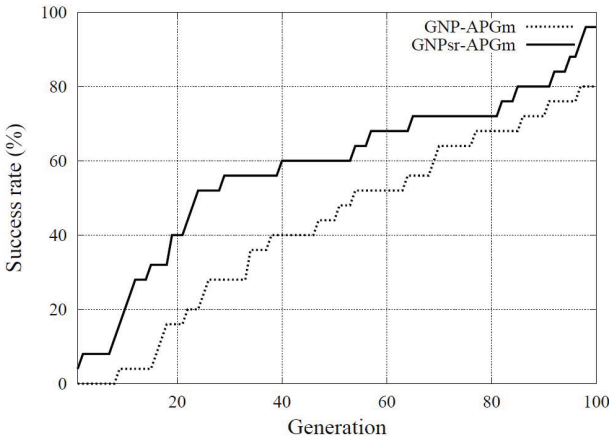


Figure 5.6: Success rate of GNPsr-APGm and GNP-APGm

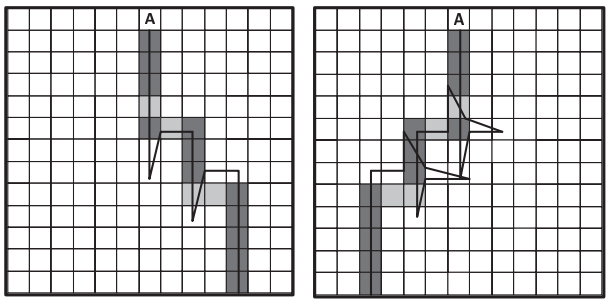


Figure 5.7: Trajectories of the artificial ant for the third and seventh trails

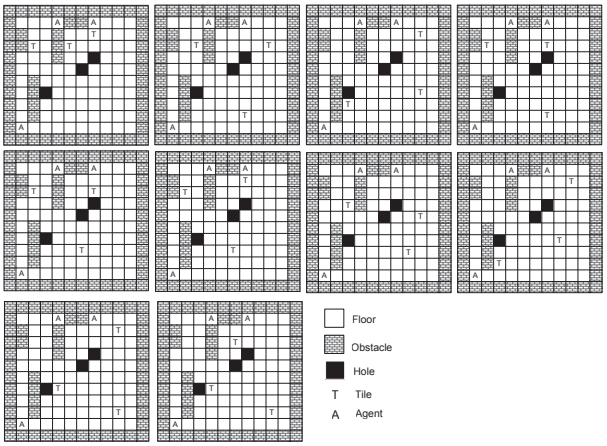


Figure 5.8: Tileworld in training phase

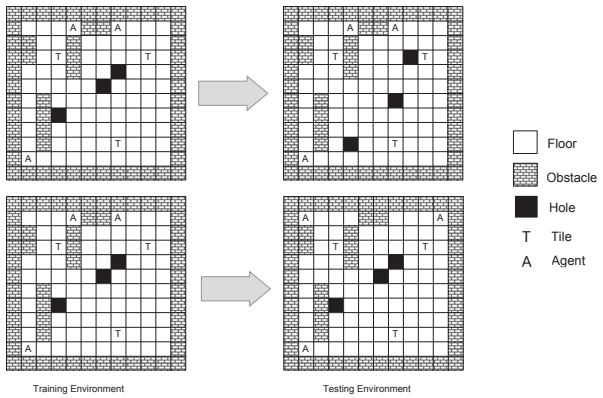


Figure 5.9: Changes of Tileworld in testing phase

5.3.2.1 Simulation environments

The simulation environment of tileworld in this chapter is the same as the one in chapter 3. For convenience, it is explained as follows once more.

In the training phase, 10 different tile-worlds are used to train the agents' behaviors. Each world has 3 agents, 3 holes and 3 tiles. The position of obstacles, holes and agents is the same. However, the position of tiles is different from each other. Fig.5.8 shows the training environments.

Two kinds of changes are introduced in the testing phase. The first kind is to change the location of the holes. The upper part of Fig.5.9 shows this kind of change. The distances between holes are larger compared with the training phase. The other kind is to change the location of the agents which is shown in the lower part of Fig.5.9. The agents are moved to the corner of the environments. Both kinds of changes are applied to the training environments, so there are 10×2 testing environments.

5.3.2.2 Simulation configurations

The basic action set of GNPsr-APGm and GNP-APGm is described in Table 5.11. The functions of processing nodes is the same as Table 5.3 and the functions of judgment nodes are shown in Table 5.12. 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.

In order to make sure that the comparison is fair, the number of nodes of two algorithms of GNPsr-APGm and GNP-APGm should be the same. Therefore, each individual of GNP-APGm contains 60 nodes including 15 judgment nodes (3 kinds of nodes $\times$ 5 for each kind) and 45 processing nodes (5 for "ACT" processing node, 40 for "IF" processing node). Because the subroutine part of GNPsr-APGm contains two GNP structures in this simulations, 36 nodes (9 judgment nodes and 27 processing nodes) are distributed to the main function part and 12 nodes (3 judgment nodes and 9 processing nodes) are distributed to each GNP structure of the subroutine part. Besides, the number of the subroutines is two, i.e., "SR0" and "SR1". In addition, each individual of GNP has also 60 nodes (12 kinds of nodes $\times$ 5 for each kind of node).

Table 5.11: Basic action set of tileworld problem

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 parameters used in simulations are described in Table 5.13. The population size is 301, and during reproduction, the best individual is copied to the next population. 120 individuals are generated through crossover, while 180 individuals are created through mutation. The crossover rate is 0.2 and mutation rate is 0.03. The algorithms need to iterate 500 generations. During the program generation, the maximum number of transitions in GNP-APGm and the main function part of GNPsr-APGm is 50, while the number of transitions of the

Table 5.12: Functions of judgment nodes of tileworld problem

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

subroutine part is 20. The maximum length of programs are 6000 bytes in both GNP-APGm and GNPsr-APGm (4000 bytes for the main function and 1000 bytes for each subroutine). The length of the subprogram pool of GNP-APGm and the main function part of GNPsr-APGm is 12, while the length of the subprogram pool of the subroutine part is 4.

Table 5.13: Parameters of simulations of tileworld problem

Parameter Name	GNPsr-APGm	GNP-APGm	GNP
Number of Individuals	301	301	301
Number of Elite	1	1	1
Crossover Size	120	120	120
Crossover Rate P_c	0.2	0.2	0.2
Mutation Size	180	180	180
Mutation Rate P_m	0.03	0.03	0.03
Number of Generations	500	500	500
Number of Transitions in Main Function	50	50	
Number of Transitions in Subroutines		20	
Length of Subprogram Pool in Main Function	12	12	
Length of Subprogram Pool in Subroutines	4		
Maximal Length of Program	6000 bytes	6000 bytes	

The fitness function of each tile-world is defined by Eq.(5.1).

$$\begin{aligned} Fitness = {} & C_{tile} \times DroppedTile \\ & + C_{dist} \times \textstyle\sum_{t \in T}(InitialDist(t) - FinalDist(t)) \\ & + C_{stp} \times (TotalStep - UsedStep), \end{aligned} \tag{5.1}$$

where, $DroppedTile$ is the number of tiles the agents have pushed into the holes. $InitialDist(t)$ is the initial distance between the t_{th} tile and the its hole, while $FinalDist(t)$ represents the final distance between the t_{th} tile and its nearest hole. T is the set of suffixes of tiles. $TotalStep$ is the predefined maximal time steps all the agents can move, and $UsedStep$ represents the used time steps by all agents. 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 time steps than the total time 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 time steps for each agent), respectively. In the simulations, the sum of the fitness values of ten tile-worlds is calculated to show the performances of three algorithms of GNPsr-APGm, GNP-APGm and GNP.

5.3.2.3 Simulation results and analysis

Fig.5.10 shows the average fitness value of the best individuals over 30 random trials in the training phase in each generation of GNPsr-APGm, GNP-APGm and GNP. From Fig.5.10, GNPsr-APGm and GNP-APGm works better than GNP. In the 500th generation, the average of the best fitness values of GNPsr-APGm, and GNP-APGm are 3334.9 and 3122.2, respectively. It is found from Fig.5.10 that the average fitness value of GNP-APGm increases faster than GNPsr-APGm at first, but the average fitness value of GNPsr-APGm surpasses that of GNP-APGm around the 250th generation. Obviously, GNPsr-APGm gets higher fitness value than GNP-APGm at the last generation. The reason why this phenomenon appears is that GNPsr-APGm decomposes a complex problem into some simpler problems, then finds a main function and subroutines to cope with each subproblem, finally seeks a way to assemble the main function and subroutines of the subproblems into a program for the original problem. In other words, GNPsr-APGm needs not only to find a good main function and some excellent subroutines, but also needs to combine them with a proper way, which is a more complicated work than just finding a single program. Therefore, the speed of GNPsr-APGm is slower than GNP-APGm at the beginning. But, GNPsr-APGm could improve the main function and subroutines until it finds an appropriate combination during evolution. At that time, GNPsr-APGm can generate better programs, which

gain higher fitness values. Fig.5.11 verifies this conjecture. During early generations of evolution, the usage time of subroutines changes rapidly. Then, after the 100th generation, the curve of the usage time becomes more smooth. Fig.5.11 matches the tendency of the fitness curve in Fig.5.10, where the speed of improving the average fitness value in GNPsr-APGm increases around the 100th generation compared with GNP-APGm. Besides, it is found from Fig.5.12 that GNPsr-APGm could decide which subroutine to use and how many times to use, which is also an advantage of GNPsr-APGm over GNP-APGm.

Fig.5.13 shows the average length of the best programs of GNPsr-APGm and GNP-APGm over ten tileworlds. It is found from Fig.5.13 that GNPsr-APGm significantly reduce the size of the program compared with GNP-APGm, i.e., from 4001.1 bytes to 2792.7 bytes. As noted in [52], the growth of the length of the program is inherent in the proposed method, since the length of the program is not fixed but varied. Fig.5.13 confirms this tendency, i.e., the average length of the programs of GNPsr-APGm grows from 2341.1 bytes to 2792.7 bytes and GNP-APGm increases from to 1954.1 bytes to 4001.1 bytes. But, the length does not always increase, instead, it is fluctuated during evolution. It is found from Fig.5.13 that the growth of the length is far below the maximal allowable values, i.e., 6000 bytes, in the last generation. From this point of view, both methods could alleviate the bloating problem. Moreover, the main function part and subroutine part of GNPsr-APGm are evolved independently, therefore all these parts have the tendency of increasing their program length. That is why the length of the program of GNPsr-APGm is larger than GNP-APGm at first. But, GNPsr-APGm has the ability to reduce the size of the program as long as it finds good subroutines and reuses them. Therefore, GNPsr-APGm can avoid the dramatic increase of its program, and obtains smaller program than GNP-APGm at last.

Fig.5.14 and Fig.5.15 show the average fitness value of each test tile-world of GNPsr-APGm and GNP-APGm in the test cases, where the best 30 individuals from the training phase are used to test these new environments. When changing the location of holes, the average fitness value is 68.5 in GNPsr-APGm and 58.2 in GNP-APGm, which shows the increase of 17%. It can be seen from Fig.5.14 that GNPsr-APGm performed a little worse than GNP-APGm only in world 3

and 9. While GNPsr-APGm is much better than GNP-APGm in world 4, 5 and 10, i.e., more than 30% increase. When changing the location of agents, the average fitness value is 194.9 in GNPsr-APGm and 173.7 in GNP-APGm, which shows the increase of 12%. It is also found from Fig.5.15 that GNPsr-APGm can get more rewards in every world except world 4. Especially, in world 3, 5, 7 and 9, the average fitness value increases by more than 15%.

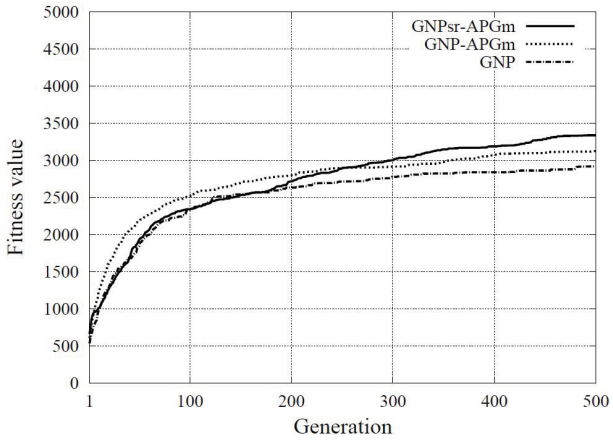


Figure 5.10: Average fitness value in the training phase

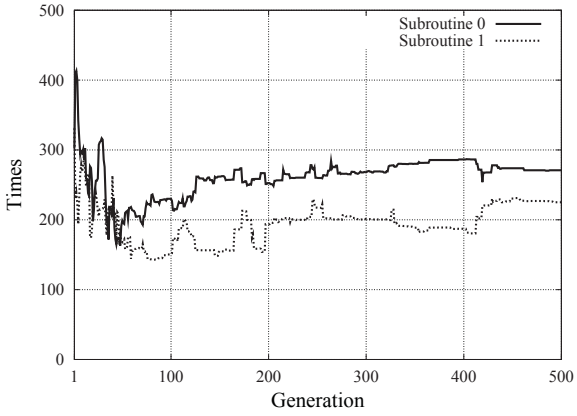


Figure 5.11: Average usage time of subroutines over 30 trials

Therefore, the proposed method has more generalization ability than GNP-APGm to deal with dynamical environments, which means the robustness of the proposed method is better than GNP-APGm. Because the subroutines generated by GNPsr-APGm focus on specific subproblems which are the common to the

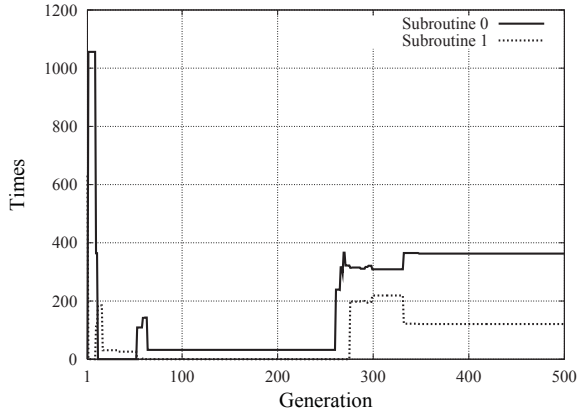


Figure 5.12: Usage time of subroutines in one trial

Tileworld, they are suitable for many kinds of tileworld environments. Therefore, the GNPsr-APGm can create more robust programs than GNP-APGm.

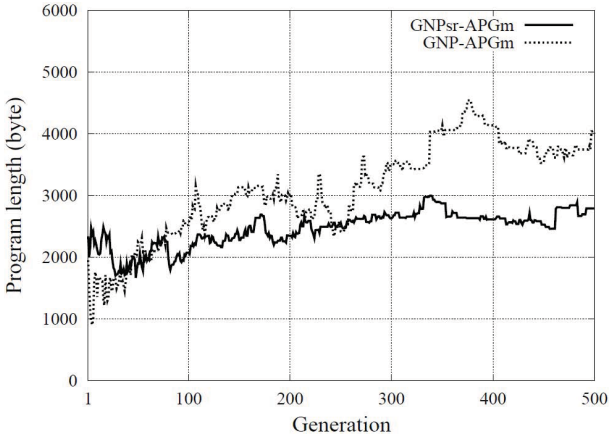


Figure 5.13: Average length of programs

5.3.2.4 Parameters discussion

Parameters are very important in evolutionary algorithms. Therefore, more simulations on Fig.5.8 are studied to show the parameter sensitivity of the proposed method.

Fig.5.16 shows the simulation results for different number of subroutines of GNPsr-APGm compared with GNP-APGm. It is clear from Fig.5.16 that GNPsr-APGm has better performance than GNP-APGm, even the number of subroutines

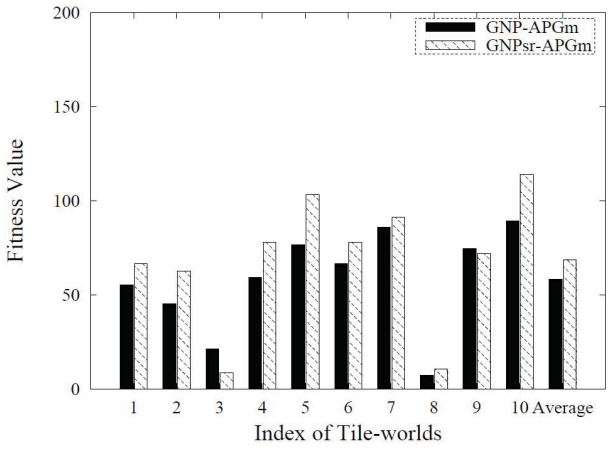


Figure 5.14: Testing result for changing the location of holes

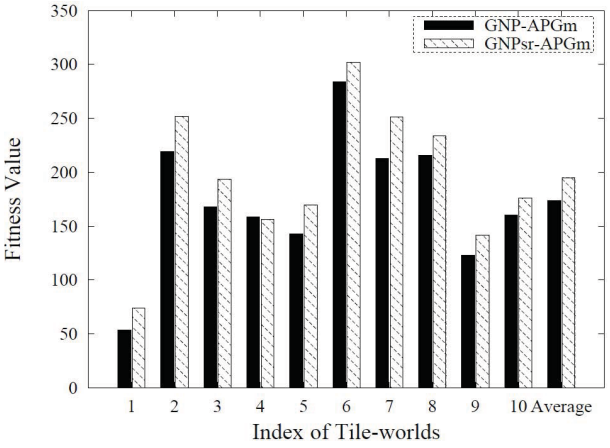


Figure 5.15: Testing result for changing the location of agents

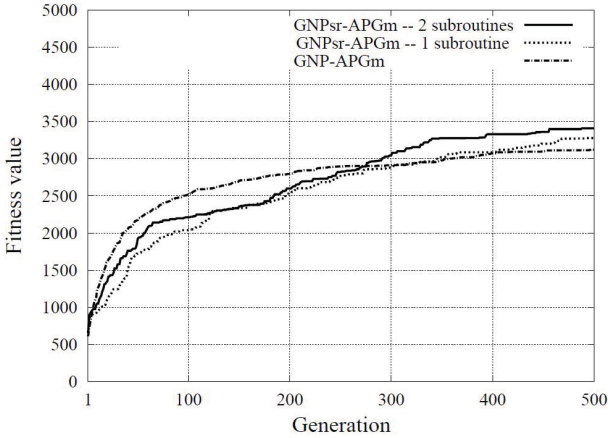


Figure 5.16: Simulation results for different number of subroutines

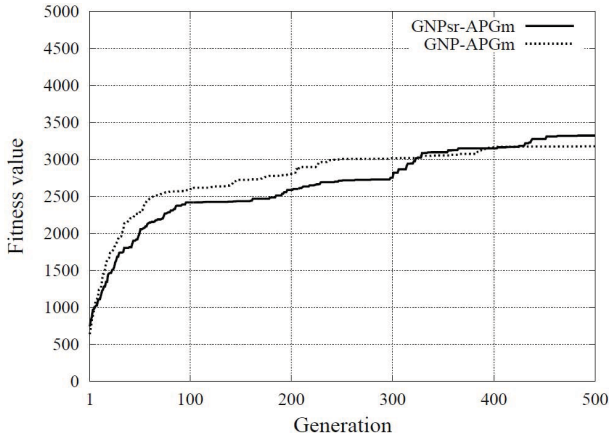


Figure 5.17: Simulation results for crossover rate 0.1 and mutation rate 0.01

is changed. Moreover, GNPsr-APGm with two subroutines works better than that with one subroutine, because the Tileworld problem is a difficult problem, where more than one repeated procedures can be found.

Fig.5.17, Fig.5.18 and Fig.5.19 show the performances of GNPsr-APGm and GNP-APGm by using different crossover rates and mutation rates. In each figure, GNPsr-APGm are better than GNP-APGm. From the results, it can also be found that the changes of the crossover and mutation rate do not affect the fitness values of GNPsr-APGm so much. Therefore, GNPsr-APGm is not so sensitive to the crossover and mutation rate.

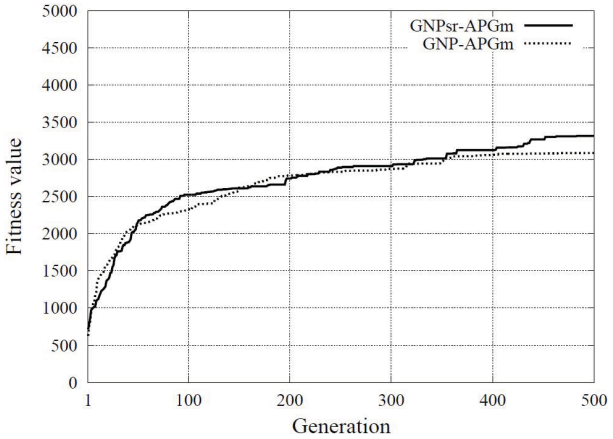


Figure 5.18: Simulation results for crossover rate 0.3 and mutation rate 0.03

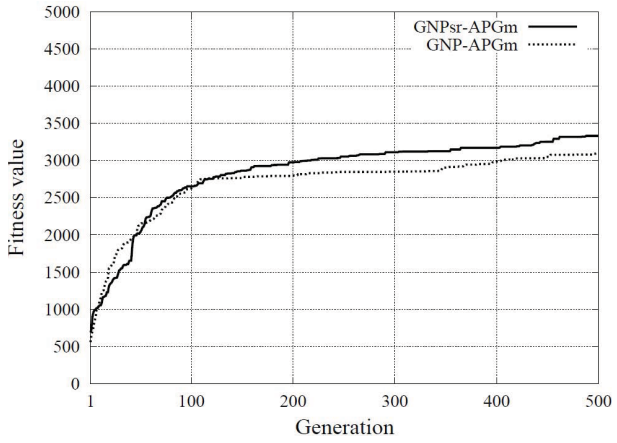


Figure 5.19: Simulation results for crossover rate 0.5 and mutation rate 0.05

5.4 Conclusions

In this chapter, Subroutine embedded Genetic Network Programming for Automatic Program Generation with Mapping Mechanism (GNPsr-APGm) has been proposed. The proposed method tries to decompose a complex problem into some simpler problems, then finds a main function and subroutines to cope with each subproblem, finally seeks a way to assemble the main function and subroutines of the subproblems into a program for the original problem. Since the program is decomposed into a main function and subroutines, each part of the program can deal with one specific subproblem, which makes the proposed method find a better solution more effectively and efficiently. Moreover, the subroutines obtained by the proposed method are called many times in the program, which results in decreasing the size of the program.

In addition, it is found from the simulation results that the proposed method can find useful subroutines and get higher fitness values both in the training phase and testing phase, and can decrease the size of the program significantly. Besides, the changes of the crossover and mutation rate do not affect the fitness values of the proposed method so much. Therefore, it is found the proposed method is not so sensitive to the crossover and mutation rate.

Chapter 6

Conclusions

In this research, the structure of Genetic Network Programming (GNP) is studied by making some changes to both the phenotype and genotype of GNP. Two kinds of methods and their extensions are proposed guided by some prior knowledge on biology systems and problem solving technologies. One of the methods focuses on improving the effectiveness and the generalization ability of GNP, the other one is to extend the application field of GNP to automatic program generation. All these methods changes the structure of GNP to make it have better search ability.

The first method is a new type of GNP – Variable Size Genetic Network Programming (GNPvs), which inspired by the increasing length of gene in species. In this method, the size of the individuals in GNPvs could be changed by evolution by a new crossover developed to replace the uniform crossover in GNP. The new crossover makes some nodes to move from one parent GNP to another parent GNP following binomial probability distribution. At last, the new method can obtain the optimal size and optimal ratio of judgment nodes and processing nodes of individuals, which results in improving the effectiveness of GNP.

Moreover, in order to improve the generalization ability of GNPvs, a kind of replacement mechanism is developed by learning the concept of build block hypothesis and evolution by gene duplication. The extension method of GNPvs is named GNPvs with Replacement (GNPvs-R), in which the blocks of frequently used nodes are extracted from elite individuals and these blocks are used to replace the non-frequently used nodes of individuals. With the help of this

mechanism, the whole structure of the individual will be evolved and the most valuable information from elite individuals will be contributed to all individuals to population. Finally, this method increases the generalization ability of GNPvs.

On the other hand, the structure of GNP is modified using the adapting genotype-phenotype mapping mechanism to cope with automatic program generation task. The proposed method called GNP-APGm develops two functions "IF" and "ACT" to create two basic statement in a program, i.e., conditional statements and sequential statements. Through the transition of nodes and the mapping process, 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 GNP.

Besides, the three steps problem solving methodology, i.e., decompose a complex problem into some simpler problems, then finds solution to cope with each subproblem, finally seeks a way to assemble these solution of the subproblems into one solution for the original problem, is introduced in GNP-APGm. Since the program is decomposed into a main function and subroutines, each part of the program can deal with one specific subproblem, which makes the proposed method find a better solution more effectively and efficiently. Moreover, the size of the program is decreased by reusing the subroutines obtained by the proposed method.

In the future, these methods can be combined together to further enhance the structure of GNP and improve the effectiveness and efficiency of GNP.

References

[1] R. M. Friedberg, "A learning machine: Part i," *IBM Journal of Research and Development*, vol. 2, no. 1, pp. 2–13, 1958. 1

[2] R. M. Friedberg, B. Dunham, and J. H. North, "A learning machine: Part ii," *IBM Journal of Research and Development*, vol. 3, no. 7, pp. 282–287, 1959. 1

[3] N. Barricelli, "Esempi numerici di processi di evoluzione," *Methodos*, pp. 45–68, 1954. 1

[4] ——, "Symbiogenetic evolution processes realized by artificial methods," *Methodos*, pp. 143–182, 1957. 1

[5] I. Rechenberg, *Evolutionsstrategie*. Fromman-Holzboog, 1973. 1

[6] L. J. Fogel, A. J. Owens, and M. J. Walsh, *Artificial Intelligence through Simulated Evolution*. John Wiley, 1966. 1

[7] J. Holland, *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975. 1

[8] D. E. Goldberg, *Genetic Algorithm in Search Optimization and Machine Learning*. Addison-Wesley, 1989. 1, 34

[9] J. R. Koza, *Genetic Programming, on the Programming of Computers by Means of Natural Selection*. MIT Press, 1992. 2, 48, 49

[10] ——, *Genetic Programming II, Automatic Discovery of Reusable Programs*. MIT Press, 1994. 2, 3, 48, 72, 83

[11] ——, *Genetic Programming III, Darwinian Invention and Problem Solving.* Morgan Kaufmann, 1999. 2, 48

[12] J. R. Koza, M. A. Keane, M. J. Streeter, W. Mydlowec, and J. Yu, *Genetic programming IV: Routine human-competitive machine intelligence.* Springer, 2005. 2, 10

[13] J. F. Miller and P. Thomson, "Cartesian genetic programming," in *Proc. of the 3rd European Conference on Genetic Programming*, 2000, pp. 121–132. 2, 48

[14] J. A. Walker and J. F. Miller, "The automatic acquisition, evolution and reuse of modules in cartesian genetic programming," *Evolutionary Computation, IEEE Transactions on*, vol. 12, no. 4, pp. 397–417, Aug. 2008. 2, 48

[15] C. Ferreira, "Gene expression programming: a new adaptive algorithm for solving problems," *Complex Systems*, vol. 13, no. 2, pp. 87–129, 2001. 2, 11, 48, 72

[16] ——, *Expression Programming:Mathematical Modeling by an Artificial Intelligence.* Angra do Heroismo, 2002. 2, 11, 48, 72

[17] M. O'Neill and C. Ryan, "Grammatical evolution," *IEEE Transactions on Evolutionary Computation*, vol. 5, no. 4, pp. 349–358, 2001. 2, 11, 48, 49, 72, 73

[18] ——, *Grammatical Evolution. Evolutionary Automatic Programming in an Arbitrary Language.* Kluwer Academic Publishers, 2003. 2, 11, 48, 72

[19] D. E. Goldberg, B. Korb, and K. Deb, "Messy genetic algorithms: Motivation, analysis, and first results," *Complex systems*, vol. 3, pp. 493–530, 1989. 2, 10

[20] S. Ohno, *Evolution by Gene Duplication.* New York: Springer-Verlag, 1970. 3, 33, 34

[21] J. Zhang, "Evolution by gene duplication:update," *Trends in Ecology and Evolution*, vol. 18, no. 6, pp. 292–298, 2003. 3, 33, 34

[22] W. G. Feero, A. E. Guttmacher, and F. S. Collins, "Genomic medicine – an updated primer," *New England Journal of Medicine*, vol. 362, no. 21, pp. 2001–2011, 2010. 3

[23] 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, 2001. 3, 4, 7, 9, 26, 27, 32

[24] 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. 3, 7, 32

[25] H. Katagiri, K. Hirasawa, and J. H, "Genetic network programming - application to intelligent agents," in *Proc. of the IEEE International Conference on Systems, Man and Cybernetics*, Nashville, TN, USA, 2000, pp. 3829–3824. 3

[26] 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 IEEE International Congress on Evolutionary Computation*, Seoul, South Korea, 2001, pp. 1276–1282. 3

[27] W. Tackett, "Genetic programming for feature discovery and image discrimination," in *Proc. of the 5th Int. Conf. Genet. Algorithms*, Evanston, IL: Morgan Kaufmann, 1993, pp. 303–309. 4, 10

[28] W. B. Langdon, "Evolving data structures using genetic programming," in *Proc. of the 6th Int. Conf. Genetic Algorithms*, Pittsburgh, PA: Morgan Kaufmann, 1995, pp. 295–302. 4, 10

[29] K. Hirasawa, T. Eguchi, J. Zhou, J. H. L. Yu, and S. Markon, "A double-deck elevator group supervisory control system using genetic network pro-

gramming," *IEEE Transactions on Systems, Man and Cybernetics, Part C*, vol. 38, no. 4, pp. 535–550, 2008. 4

[30] Y. Chen, S. Mabu, K. Shimada, and K. Hirasawa, "Trading rules on stock markets using genetic network programming with sarsa learning," *Journal of Advanced Computational Intelligence and Intelligent Informatics*, vol. 12, no. 4, pp. 383–392, 2008. 4

[31] 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. 4

[32] 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. 4

[33] H. Nakagoe, K. Hirasawa, and J. Hu, "Genetic network programming with automatically generated variable size macro nodes," in *Proc. of the IEEE International Congress on Evolutionary Computation*, San Diego, CA, USA, 2004, pp. 713–719. 7, 32

[34] S. Mabu, F. Ye, S. Eto, X. Fan, and K. Hirasawa, "Genetic network programming with rule accumulation and its application to tile-world problem," *Journal of Advanced Computational Intelligence and Intelligent Informatics*, vol. 13, no. 5, pp. 551–572, 2009. 7, 32

[35] H. G. Beyer and H. P. Schwefel, "Evolution strategies: A comprehensive introduction," *Journal Natural Computing*, vol. 1, no. 1, pp. 3–52, 2002. 10

[36] J. F. Miller, "Cartesian genetic programming," *Natural Computing Series*, pp. 17–34, 2011. 10

[37] W. B. Langdon and R. Poli, "Fitness causes bloat," *Technical Report CSPR-97-09*, 1997. 10, 20, 24

[38] 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. 11, 49, 73

[39] C. Zhou, W. Xiao, T. M. Tirpak, and P. C. Nelson, "Evolving accurate and compact classification rules with gene expression programming," *IEEE Transactions on Evolutionary Computation*, vol. 7, no. 6, pp. 519– 531, 2003. 11

[40] M. O'Neill, C. Ryan, M. Keijzer, and M. Cattolico, "Crossover in grammatical evolution," *Genetic Programming and Evolvable Machines*, vol. 4, no. 1, pp. 67–93, 2003. 11

[41] H. Katagiri, K. Hirasawa, J. Hu, and M. Junichi, "Variable size genetic network programming," *IEEJ Trans. EIS*, vol. 123, no. 1, 2003. 11

[42] M. E. Pollack and M. Ringuette, "Introducing the tileworld: Experimentally evaluating agent architectures," in *Proc. of the Eighth National Conference on Artificial Intelligence*, 1990, pp. 183–189. 19, 58, 73

[43] C. V. Goldman and J. S. Rosenshein, "Emergent coordination through the use of cooperative state-changing rules," in *Proc. of the 12th National Conference on Artificial Intelligence*, 1994. 19

[44] S. Hanks, M. E. Pollack, and P. R. Cohen, "Benchmarks, test beds, controlled experimentation, and the design of agent architectures," *AI Magazine*, vol. 14, no. 4, 1993. 19

[45] H. Iba, "Emergent cooperation for multiple agents using genetic programming," *Lecture Notes in Computer Science*, pp. 32–41, 1996. 19

[46] B. Li, X. Li, S. Mabu, and K. Hirasawa, "Variable size genetic network programming with binomial distribution," in *Proc. of the IEEE International Congress on Evolutionary Computation*, New Orleans, LA, USA, 2011, pp. 973–980. 32

[47] T. Mitchell, *Machine Learning*. McGraw Hill, New York, 1996. 33

[48] 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, pp. 1023–1028. 33, 57

[49] S. Mabu and K. Hirasawa, "Efficient program generation by evolving graph structures with multi-start nodes," *Applied Soft Computing*, vol. 11, no. 4, pp. 3618–2624, 2011. 33

[50] 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. 49, 73

[51] 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*, 2009, pp. 1382–1387. 49, 73

[52] W. Langdon and R. Poli, "Fitness causes bloat," 1997, technical Report CSPR-97-09,University of Birmingham. 62, 93

[53] C. Ferreira, N. Nedjah, L. de M. Mourelle, and A. Abraham, "Automatically defined functions in gene expression programming," *Genetic Systems Programming: Theory and Experiences, Studies in Computational Intelligence*, vol. 13, pp. 21–56, 2006. 72

[54] R. Harper and A. Blair, "Dynamicall defined functions in grammatical evolution," *IEEE Transactions on Evolutionary Computation*, pp. 2638 – 2645, 2006. 72

[55] M. O'Neil and C. Ryan, "Grammar based function definition in grammatical evolution," in *Proc. of the GECCO*, 2000, pp. 485–490. 72

[56] B. Li, S. Mabu, and K. Hirasawa, "Genetic network programming with automatic program generation for agent control," *Transaction of the Japanese Society for Evolutionary Computation*, vol. 1, no. 1, pp. 43–53, 2010. 73

[57] ——, "Automatic program generation with genetic network programming using subroutines," in *Proc. of the Society of Instrumentation and Control Engineering International Annual Conference*, Taipei, 2010, pp. 3089–3094. 73

Research Achievements

Journals

1. **B. Li**, S. Mabu and K. Hirasawa, "Evolving Graph-based Chromosome by means of Variable Size Genetic Network Programming", *IEEJ Transactions on Electrical and Electronic Engineering*, 2013 (accepted).

2. **B. Li**, S. Mabu and K. Hirasawa, "Genetic Network Programming with Subroutines for Automatic Program Generation", *IEEJ Transactions on Electrical and Electronic Engineering*, Vol. 7, No. 2, pp. 197-207, 2012/3.

3. X. Li, S. Mabu, **B. Li** and K. Hirasawa, "Probabilistic Model Building Genetic Network Programming using Reinforcement Learning", *Transaction of the Japanese Society for Evolutionary Computation*, Vol. 2, No.1, pp. 29-40, 2011.

4. **B. Li**, S. Mabu and K. Hirasawa, "Genetic Network Programming with Automatic Program Generation for Agent Control", *Transaction of the Japanese Society for Evolutionary Computation*, Vol. 1, No.1, pp.43-53, 2010.

International Conference papers

1. **B. Li**, X. Li, S. Mabu, and K. Hirasawa, "Towards automatic discovery and reuse of subroutines in variable size genetic network programming", in *Proc. of the IEEE Congress on Evolutionary Computation (CEC 2012)*, pp. 485-492, Brisbane, Australia, 2012/6.

2. X. Li, **B. Li**, S. Mabu, and K. Hirasawa, "A continuous estimation of distribution algorithm by evolving graph structures using reinforcement learning", in *Proc. of the IEEE Congress on Evolutionary Computation (CEC 2012)*, pp. 2097-2104, Brisbane, Australia, 2012/6.

3. **B. Li**, X. Li, S. Mabu and K. Hirasawa, "Analysis of Crossover Rate in Variable Size Genetic Network Programming with Binomial Distribution", in *Proc. of the SICE International Annual Conference 2011*, pp. 155-160, Tokyo, Japan, 2011/9.

4. **B. Li**, X. Li, S. Mabu and K. Hirasawa, "Variable Size Genetic Network Programming with Binomial Distribution", in *Proc. of the IEEE Congress on Evolutionary Computation (CEC2011)*, pp. 973-980, New Orleans, USA, 2011/6/6.

5. X. Li, **B. Li**, S. Mabu and K. Hirasawa, "A Novel Estimation of Distribution Algorithm Using Graph-based Chromosome Representation and Reinforcement Learning", in *Proc. of the IEEE Congress on Evolutionary Computation (CEC2011)*, pp. 37-44, New Orleans, USA, 2011/6.

6. **B. Li**, S. Mabu and K. Hirasawa, "Tile-world - A case study of Genetic Network Programming with Automatic Program Generation", in *Proc. of*

the IEEE International Conference on Systems, Man, and Cybernetics, pp. 2708-2715, Istanbul, Turkey, 2010/10.

7. **B. Li**, S. Mabu and K. Hirasawa, "Automatic Program Generation with Genetic Network Programming using Subroutines", in *Proc. of the SICE International Annual Conference 2010*, pp.3089-3094, Taipei, Taiwan, 2010/8.