Graph Structured Program Evolution with Automatically Defined Nodes

Shinichi Shirakawa
Graduate School of Environment and Information Sciences
Yokohama National University
79-7, Tokiwadai, Hodogaya-ku, Yokohama
Kanagawa, 240-8501, Japan
shirakawa@nlab.sogo1.ynu.ac.jp

Tomoharu Nagao
Graduate School of Environment and Information Sciences
Yokohama National University
79-7, Tokiwadai, Hodogaya-ku, Yokohama
Kanagawa, 240-8501, Japan
nagao@ynu.ac.jp

ABSTRACT

Currently, various automatic programming techniques have been proposed and applied in various fields. Graph Structured Program Evolution (GRAPE) is a recent automatic programming technique with graph structure. This technique can generate complex programs automatically. In this paper, we introduce the concept of automatically defined functions, called automatically defined nodes (ADN), in GRAPE. The proposed GRAPE program has a main program and several subprograms. We verified the effectiveness of ADN through several program evolution experiments, and report the results of evolution of recursive programs using GRAPE modified with ADN.

Categories and Subject Descriptors

I.2.2 [**Artificial Intelligence**]: Automatic Programming—*program synthesis*; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search

General Terms

Algorithms, Experimentation, Performance

Keywords

Automatic Programming, Genetic Programming, Automatically Defined Function, Graph-based Genetic Programming, Graph Structured Program Evolution, Evolutionary Algorithm, Genetic Algorithm, Recursive Program

1. INTRODUCTION

Automatic programming is the research field of generating computer programs automatically. Genetic programming (GP) [6, 7], which was originally introduced by Koza, is a typical example of automatic programming. GP evolves computer programs, which usually follow a tree structure, and searches for the desired program using a genetic algorithm (GA).

Thus far, various representations have been proposed for GP, including graph representation. Parallel algorithm discovery and orchestration (PADO) [18], which has been applied to object recognition problems, is a graph-based GP that does not have a tree structure. PADO uses stack and index memories, and has action and branch-decision nodes. PADO execution is carried out from the start node to the end node in the network. Another graph based GP is the parallel distributed genetic programming (PDGP) [12]. In this approach, the tree is represented as a graph with functions and terminals nodes located over a grid, which enables execution of several nodes concurrently. Cartesian genetic programming (CGP) [8, 9] was developed from a representation used for evolution of digital circuits, and represents a program as a graph. In certain respects, it is similar to the graph-based PDGP technique. However, PDGP evolved without the use of genotype-phenotype mapping, and various sophisticated crossover operators were defined. In CGP, the genotype is an integer string, which denotes a list of node connections and functions. This string is mapped onto a phenotype of an index graph. Genetic network programming (GNP) [5], which has a directed graph structure, was proposed. GNP mainly applied to make the behavior sequences of agents. In addition, the ADATE system for automatic programming were proposed and solved algorithmic problems (e.g. list reversing, list sorting, list splitting, and so on) [11].

The graph structured program evolution (GRAPE) [14, 15] is a recent automatic programming technique, which can generate complex programs (e.g., factorial, exponentiation, and list sorting) automatically. The work described in this paper is based on this GRAPE technique. The representation used in GRAPE includes graph structure; therefore, it can represent complex programs (e.g., branches and loops). The genotype of GRAPE is in the form of a linear string of integers. Although this genotype is a fixed length representation, the number of nodes in the phenotype can vary (not all nodes encoded in the genotype have to be connected).

The evolution of functions or modules has been examined by various researchers in the GP community. Automatically defined functions (ADF) [7], module acquisition [3], and automatically defined macros (ADM) [16] were tested to integrate modularity into the GP paradigm. Embedded CGP [20, 21] is a method in which CGP is implemented as ADFs by automatically acquiring and evolving modules. GNP also has an ADF-like mechanism, called automatically generated macro nodes (AGM) [10]. These AGMs have network structures and are evolved like the main GNP.

In this paper, we introduce the concept of automatically defined functions, called automatically defined nodes (ADN), in GRAPE. The proposed GRAPE program includes a main program and several subprograms. We verified the effectiveness of ADN through program evolution experiments. We report the results of the evolution of recursive programs using GRAPE modified with ADN.

An overview of GRAPE is presented in the next section. In Section 3, we describe our automatically defined node approach in GRAPE. In Section 4, we apply the proposed method to the problem of automatic program generation and show several experimental results. In section 5, we modify the proposed method and generate recursive programs. Finally, in Section 6, we present our conclusions and possible future work.

2. GRAPH STRUCTURED PROGRAM EVOLUTION (GRAPE)

2.1 Overview

GRAPE [15, 14] constructs graph structured programs automatically. The features of GRAPE are as follows:

- Arbitrary directed graph structures
- Ability to handle multiple data types using a data set
- Genotype of an integer string

GRAPE has different representation from PDGP, CGP and Linear-Graph GP. These methods have some restriction of connections (e.g. restrict loops and allow only feedforward connectivity). The representation of GRAPE is arbitrary directed graph of nodes. PADO is one of the similar methods to GRAPE. PADO has stack memory and index memory, and the execution of PADO is carried out from the start node to the end node in the network. GRAPE is different from PADO in the fact that GRAPE handles multiple data types using the data set for each type and adopts genotype-phenotype mapping.

GRAPE has been shown to successfully generate a variety of programs, such as factorial, Fibonacci sequence, exponentiation, list reversing, and list sorting.

2.2 Structure of GRAPE

The graph structured programs are composed of an arbitrary directed graph of nodes and a data set (index memory). The data set flows through the directed graph and is processed at each node. Figure 1 illustrates an example of the structure of GRAPE. In GRAPE, each node has two parts –one for processing and the other for branching. The processing component executes several types of processing using the data set, e.g., arithmetic and Boolean calculations. After processing is complete, the next node is selected. The branching component decides the next node according to the data set. Examples of nodes in GRAPE are shown in Figure 2. No.1 adds data[0] to data[1], substitutes it for data[0] using an integer data type, and selects No.2 as the next node.

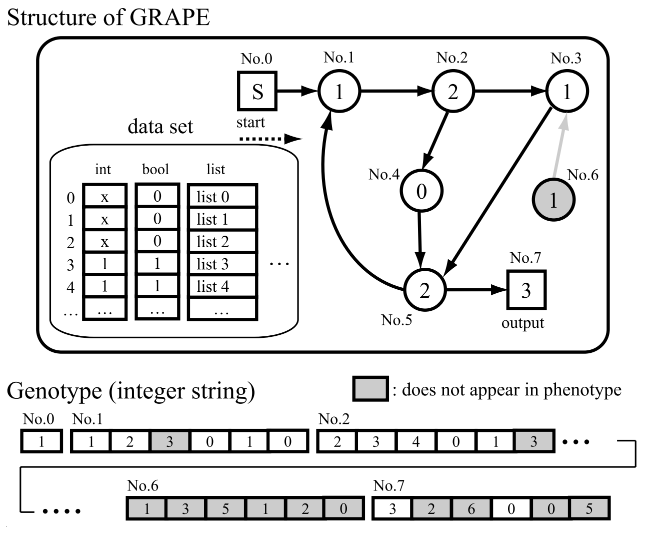


Figure 1: Structure of GRAPE (phenotype) and the genotype, which denotes a list of node types, connections, and arguments. No.6 is the inactive node.

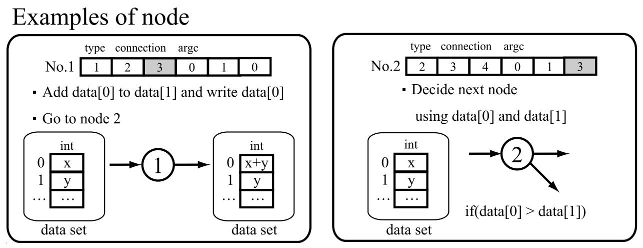


Figure 2: Examples of processing (left) and branching nodes (right).

No.2 decides the next node using integer data[0] and data[1]; if data[0] is greater than data[1], connection 1 is chosen, else connection 2 is chosen. There are special nodes shown in Figure 1. No.0 is the start node, which is the equivalent of root node of GP. It is the first node to be executed when a GRAPE program runs. No.7 is the output node. When this node is reached, the GRAPE program outputs data and the program terminates. In Figure 1, No.7 outputs integer data[0]. Although the GRAPE program has only one start node, it may have several output nodes.

Since GRAPE is represented by a graph structure, it can represent complex programs (e.g., branches and loops). There are several data types in GRAPE programs, e.g., integer, Boolean, and list. The GRAPE program handles multiple data types using the data set for each type.

To adopt an evolutionary method, GRAPE uses genotype-phenotype mapping. This genotype-phenotype mapping method is similar to CGP. The GRAPE program is encoded in the form of a linear string of integers. The genotype is an integer string, which denotes a list of node types, connections, and arguments. The node connections are arbitrary, which is different from CGP. The length of the genotype is fixed and equals $N*(n_c+n_a+1)+1$, where N is the maximum number of nodes, n_c is the maximum number of connections, and n_a

is the maximum number of arguments. Although the genotype in GRAPE is a fixed-length representation, the number of nodes in the phenotype can vary but is bounded (not all nodes encoded in the genotype have to be connected). This allows the existence of inactive nodes. In Figure 1, No.6 is inactive and the other nodes are active.

2.3 Genetic Operators and Generation Alternation Model in GRAPE

To obtain the optimum structure of GRAPE, an evolutionary method is adopted. The genotype of GRAPE is a linear string of integers. Therefore, GRAPE is able to use a typical GA. GRAPE uses uniform crossover and mutation for a string of integers as genetic operators.

It uses the Minimal Generation Gap (MGG) model [4, 13, 19] as a generation alternation model. It shows a higher performance with the MGG model than just with simple GA in a variety of program evolutions [15]. The MGG model is a steady-state model proposed by Satoh et al., having a desirable convergence property of being able to maintain the diversity of the population, and shows higher performance than other conventional models in a wide range of applications (especially real parameter optimization). The MGG model is summarized as follows:

1. Set the generation counter $t = 0$. Generate N individuals randomly as the initial population $P(t)$.
2. Select a set of two parents M by random sampling from the population $P(t)$.
3. Generate a set of children C by applying the crossover and the mutation operation to M.
4. Select two individuals from set $M + C$. One is the elitist individual and the other is the individual from roulette-wheel selection. Then replace M with the two individuals in population $P(t)$ to get population $P(t + 1)$.
5. Stop if a certain specified condition is satisfied; otherwise, set $t = t + 1$ and go to step 2.

The MGG model localizes its selection pressure, not to the whole population as simple GA or steady state does, but only to the family (children and parents).

3. AUTOMATICALLY DEFINED NODES (ADN)

In this paper, we introduce the concept of automatic function definitions to GRAPE, called automatically defined nodes (ADN). In the proposed method, the whole program consists of a main program and several sub-programs, similar to the ADFs in a GP. Figure 3 illustrates an example of the GRAPE structure with ADN. The main program considers an ADN as one of the nodes. Since ADN has arbitrary connections, it can represent loop structures. When the current node reaches the ADN in the main program, the transition starts at the start node in the selected ADN, and the processing in it terminates when the output node is reached. The same data set is processed or used at the nodes in the main program and ADN. In other words, the data set is global.

In the proposed method, the presence of one ADN in another is allowed. However, this type of a hierarchical or

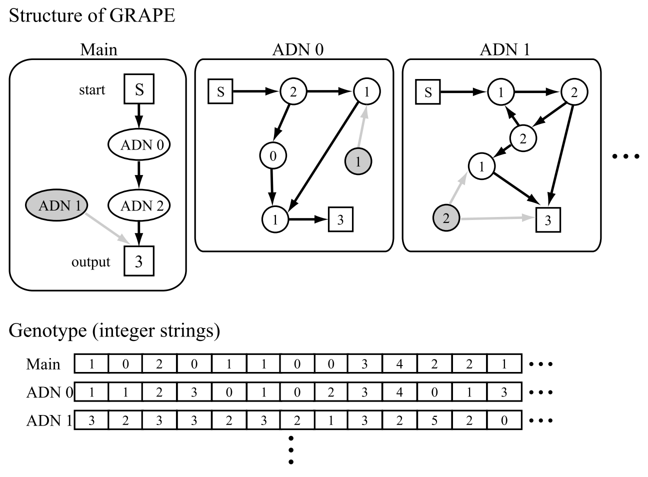

Structure of GRAPE
Main
ADN 0
ADN 1
start
S
ADN 0
ADN 1
ADN 2
output
3
Genotype (integer strings)
Main 1 0 2 0 1 1 0 0 3 4 2 2 1 ...
ADN 0 1 1 2 3 0 1 0 2 3 4 0 1 3 ...
ADN 1 3 2 3 3 2 3 2 1 3 2 5 2 0 ...

Figure 3: Structure of GRAPE with ADN.

recursive structure is restricted to reduce non-terminating programs in the experiments in section 4. Moreover, the connections in the main program are feedforward structures in the experiments in section 4, because GRAPE can solve problems without using ADN, if loop structures can be represented in the main program. To realize the feedforward structure in the main program, the nth node can only connect to (n+1) and higher nodes. These connection restrictions are performed at the genotype level.

The graph structures of the main program and the ADN are encoded in the form of linear integer strings. The genotype is integer strings, which denote a list of node types, connections, and arguments of the main program and the ADN. The genetic operators affect these strings. In the proposed method, we use uniform crossover and mutation for integer strings as the genetic operators similar to conventional GRAPE. The uniform crossover operator affects two individuals as follows:

- Select several genes randomly according to the uniform crossover rate P_c for each gene.
- The selected genes are swapped between two parents and off-springs are generated.

The mutation operator affects one individual as follows:

- Select several genes randomly according to the mutation rate P_m for each gene.
- The selected genes are randomly changed.

4. EXPERIMENTS AND RESULTS

In this section, we evolve factorial ($n!$), exponentiation (a^b), and list sorting programs using GRAPE and GRAPE with ADN and compare their performance. Evolution of these programs is difficult for standard GP, which must develop iterative or recursive mechanisms to solve these problems.

4.1 Experimental Settings

Table 1 gives the GRAPE parameters for the experiments. There are a maximum of 30 nodes in the main program and

Table 2: GRAPE node functions for the experiments.

Name	# Connections	# Args.	Argument(s)	Description
$+$	1	3	x, y, z	Use integer data type. Add data[x] to data[y] and substitute for data[z].
$-$	1	3	x, y, z	Use integer data type. Subtract data[x] from data[y] and substitute for data[z].
$*$	1	3	x, y, z	Use integer data type. Multiply data[x] by data[y] and substitute for data[z].
/	1	3	x, y, z	Use integer data type. Divide data[x] by data[y] and substitute for data[z].
$=$	2	2	x, y	Use integer data type. If data[x] is equal to data[y], choose connection 1; else, choose connection 2.
$>$	2	2	x, y	Use integer data type. If data[x] is greater than data[y], choose connection 1; else, choose connection 2.
$<$	2	2	x, y	Use integer data type. If data[x] is less than data[y], choose connection 1; else, choose connection 2.
SwapList	1	2	x, y	Use integer type and a list data. Swap list[data[x]] for list[data[y]].
EqualList	2	2	x, y	Use integer type and a list data. If list[data[x]] equals list[data[y]], choose connection 1; else, choose connection 2.
GreaterList	2	2	x, y	Use integer type and a list data. If list[data[x]] is greater than list[data[y]], choose connection 1; else, choose connection 2.
LessList	2	2	x, y	Use integer type and a list data. If list[data[x]] is less than list[data[y]], choose connection 1; else, choose connection 2.
OutputInt	0	1	x	Output data[x] and terminate the program.
OutputList	0	0	-	Output a list data and terminate the program.

Table 1: Parameters used in the experiments.

Parameter	*Value*
Number of evaluations	5000000
Population size	500
Children size (for MGG)	50
Uniform crossover rate P_c	0.1
Crossover rate	0.7
Mutation rate P_m	0.02
Maximum number of nodes	30 (main)
	30 (ADN)
Number of ADNs	5
Execution step limits	500 (factorial)
	1000 (exponentiation)
	3000 (list sorting)

the ADN. The number of ADN is five. In our experiment, the maximum number of nodes in conventional GRAPE are set to 30 and 180, respectively.

To avoid problems caused by non-terminating structures, we limited the execution steps to 500 (factorial), 1000 (exponentiation), and 3000 (list sorting). When a program reaches the execution limit, the individual is assigned a fitness value of 0.0. These execution step limits are sufficient to solve the target problem. We prepared a data set size sufficient to compute the problems. Initially, we set input values and constant values on the data set. Therefore, GRAPE handles or creates constants within its programs.

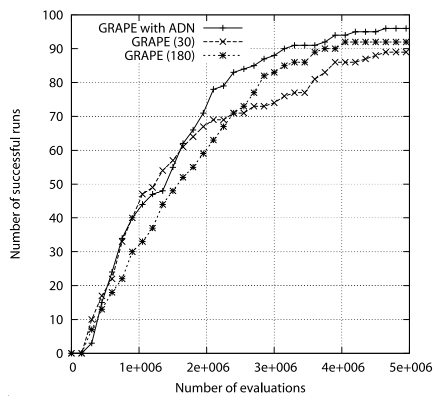
Table 2 shows the GRAPE node functions used in the experiments. We prepared simple node functions, arithmetic functions, and functions to swap and compare the elements of the list. We did not prepare special node functions such as iteration functions.

4.2 Factorial

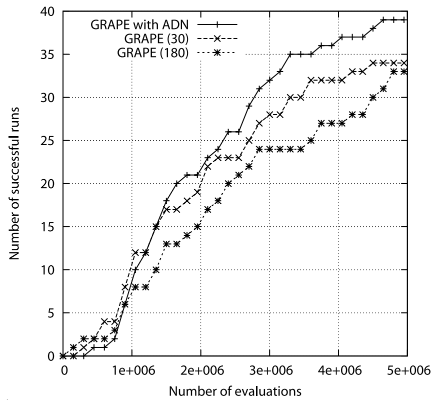
In this problem, we seek to evolve an implementation of the factorial function. We used the integers from 0 to 5 as the training set with the following input/output pairs (a, b): (0, 1), (1, 1), (2, 2), (3, 6), (4, 24), and (5, 120). We used the normalized absolute mean error of the training set as a fitness. The fitness function F used in the experiments is:

$$F = 1 - \frac{\sum_{i=1}^{n} \frac{|Correct_i - Out_i|}{|Correct_i| + |Correct_i - Out_i|}}{n} \quad (1)$$

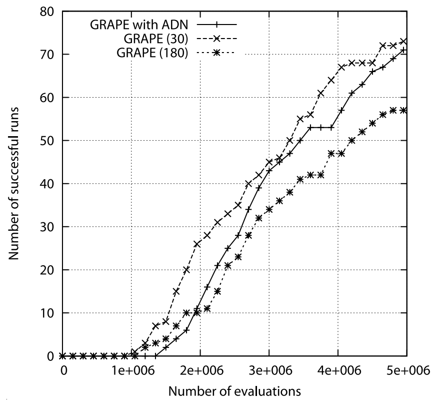
where $Correct_i$ is the correct value for the training data i, Out_i is the value returned by the generated program for the training data i, and n is the size of the training set. The range of this fitness function was [0.0, 1.0]. A higher numerical value indicated better performance. If this fitness value is equal to 1.0, the program gives the perfect solution



(a) Factorial

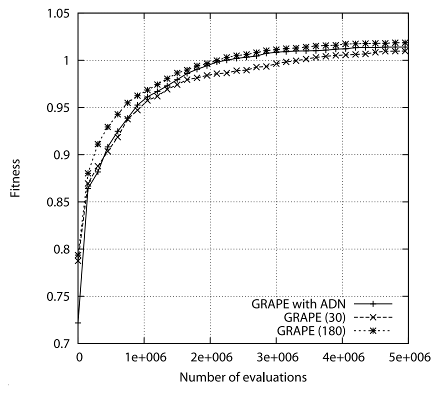


(b) Exponentiation

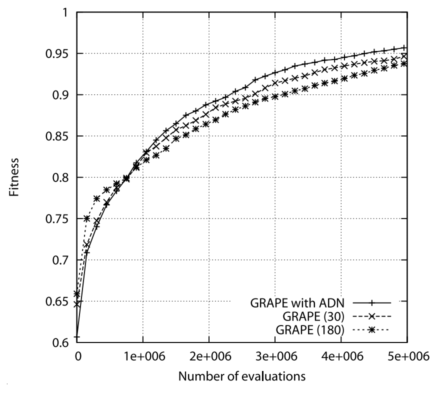


(c) List sorting

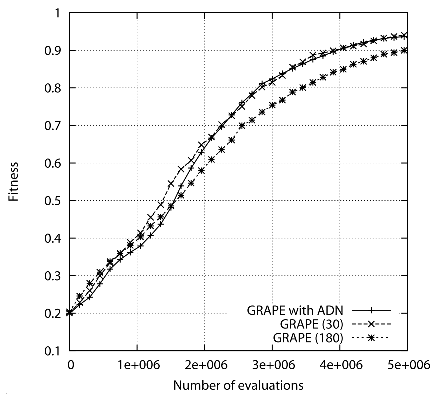
Figure 4: Transitions of the number of successful runs over 100 runs. (a) Factorial, (b) Exponentiation, (c) List sorting.



(a) Factorial



(b) Exponentiation



(c) List sorting

Figure 5: Transitions of the fitness value of over 100 runs. (a) Factorial, (b) Exponentiation, (c) List sorting.

for the training set. If the fitness value in Equation 1 reaches 1.0, the fitness is calculated as follows:

$$F = 1.0 + \frac{1}{S_{exe}} \qquad (2)$$

where S_{exe} is the total number of execution steps of the generated program. This fitness function indicates that fewer execution steps provide a better solution.

In the factorial experiment, integer data was used, and the size of integer data in GRAPE was 10. Initially, we set the input value a for data[0] to data[4] and a constant value of 1 for data[5] to data[9]. The node functions used in this experiment are { $+$, $-$, $*$, $=$, $>$, $<$, OutputInt } in Table 2.

Results are given for 100 different runs with the same parameter set. Figure 4 (a) shows the evolution of the number of successful runs for the number of fitness evaluations. When an individual whose fitness value is equal or greater than 1.0 is found, the run is counted as a successful run. Figure 5 (a) shows the transitions of the fitness value for the number of fitness evaluations.

4.3 Exponentiation

In this problem, we seek to evolve an implementation of the integer exponential. There are two inputs in this problem. The training set (a, b, c) used in this experiment is (2, 0, 1), (2, 1, 2), (2, 2, 4), (3, 3, 27), (3, 4, 81), (3, 5, 243), (4, 6, 4096), (4, 7, 16384), and (4, 8, 65536). We also used the fitness functions in Equations 1 and 2 on the training set.

In the experiment of exponentiation, integer data type is used, and the size of integer data in GRAPE is 15. Initially, we set the input value a for data[0] to data[4], input value b for data[5] to data[9], and the constant value of 1 for data[10] to data[14]. The node functions used in this experiment are { $+$, $-$, $*$, $=$, $>$, $<$, OutputInt } in Table 2.

Results are given for 100 different runs with the same parameter set. Figure 4 (b) shows the evolution of the number of successful runs for the number of fitness evaluations. Figure 5 (b) shows the transitions of the fitness value for the number of fitness evaluations.

4.4 List sorting

In this problem, we seek to evolve an implementation of the sorting algorithm. We provide a list of integers as input. A correct program returns a sorted input list of any length (e.g., input: (2, 1, 7, 5, 1), output: (1, 1, 2, 5, 7)). The training data set is 30 random lists, whose lengths are between 10 and 20. List elements are randomly chosen from the range of [0, 255].

The fitness function F used in this experiment is given in Equation 3. The range of this fitness function is [0.0, 1.0]. Higher numerical value indicates better performance. If this fitness value is equal to 1.0, the program gives the perfect solution for the training set.

$$F = 1.0 - \frac{\sum_{i=1}^{n} \frac{\sum_{j=0}^{l_i} (1 - \frac{1}{2^{d_{ij}}})}{l_i}}{n} \tag{3}$$

where d_{ij} is the distance between the correct position and the return value position for the training data i for element j. l_i is the list length for the training data i and n is the size of the training data set. If the fitness value in Equation 3 reaches 1.0, the fitness value is calculated using Equation 2.

In this experiment, a list of integers and an integer data type were used, and the size of integer data in GRAPE is 15. Initially, we set the size of the input list (the list length) for data[0] to data[4], a constant value of 0 for data[5] to data[9], and a constant value of 1 for data[10] to data[14]. The node functions used in this experiment are { +, −, ∗, /, =, >, <, SwapList, EqualList, GreaterList, LessList, OutputList } from Table 2.

Results are given for 100 different runs with the same parameter set. Figure 4 (c) shows the evolution of the number of successful runs for the number of fitness evaluations. Figure 5 (c) shows the transitions of the fitness value for the number of fitness evaluations.

4.5 Results and Discussion

In Figure 4, GRAPE with ADN shows a more stable number of successful runs compared with conventional GRAPE. Figure 5 shows that the transition of the fitness value doesn't have a big difference.

Table 3 compares the number of successful runs between GRAPE with ADN and GRAPE at the final generation. These are the numbers of runs it takes to generate programs that return correct outputs for the training data. The number of successful runs improved a little in GRAPE with ADN. The number of successful runs for list sorting decreased when the maximum number of nodes was 180 in conventional GRAPE. However, GRAPE with ADN achieved a high number of successful runs for the list sorting problem. The advantages of GRAPE with ADN are in the reusing of ADN or the representation of compact structures in a program.

We apply the elitist individual generated by GRAPE to the test data set for each run. For the problem of factorial, the integers from 6 to 12 are used as the test set inputs. For the problem of exponentiation, the test set inputs (a, b) are (5, 9) (5, 10) (5, 11) (4, 12) (4, 13) (4, 14) (3, 15) (3, 16) (3, 17) (2, 18) (2, 19) (2, 20). For the problem of list sorting, the test set is 100 random lists. We use the length of the list between 10 and 50 as the test set. The number of correct programs for the test set appear in Table 4. The result of GRAPE with ADN of list sorting is the worst. From this result, GRAPE with ADN tends to construct the specialized sorting algorithm for the training set. However, the result of GRAPE with ADN of exponentiation is the best.

5. AUTOMATIC GENERATION OF RECURSIVE PROGRAMS

In this section, programs with recursive structures are evolved by GRAPE with ADN. Conventional GRAPE does not have a mechanism of recursive structures. This is the first attempt to generate recursive programs automatically using GRAPE. To evolve a recursive structure, we modified the structure of GRAPE with ADN. We interdicted the feedback structure (loop structure) in ADN to evolve programs without loop structures. The presence of an ADN within another is permitted; therefore, GRAPE with ADN can represent recursive structures.

The experimental setup is the same as used in section 4.1. Results are given for 100 different runs with the same parameter set for each problem. The numbers of successful runs for training data were 16 (factorial), 9 (exponentiation), and 13 (list sorting). There were fewer successful runs than in the previous experiments; however, we confirmed that recursive programs were generated. And the numbers of successful runs for test data were 9 (factorial), 9 (exponentiation), and 1 (list sorting).

The examples of the recursive programs produced for factorial and exponentiation are shown in Figure 6 and 7, respectively. These programs return factorial or exponentiation for all inputs, respectively. These programs call ADN in ADN (recursive structure), demonstrating that GRAPE with ADN can evolve recursive programs, although they are simple and trivial recursive programs.

Previously, two automatic programming techniques, PushGP [17] and object oriented genetic programming (OOGP) [2, 1], tackled the problems of generating recursive programs (e.g. factorial, Fibonacci sequence, exponentiation, list sorting, and so on) and obtained these programs automatically. PushGP evolves programs using a Push language proposed by Spector, et al. Push is a stack-based programming language. OOGP evolves object oriented programs instead of the form of LISP parse tree. Although it is difficult to simply compare with these methods, the success rates of these methods for generating recursive programs are superior than GRAPE with ADN.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced the concept of ADFs to GRAPE, which evolves graph structured programs. We applied the proposed method to evolve programs such as factorial, exponentiation, and list sorting, and confirmed that ideal programs for each problem were obtained by the proposed method. The experimental results showed that the proposed method performed better than conventional GRAPE. Moreover, we confirmed that the modified proposed method obtained recursive programs.

In future, we will develop a more efficient method of evolving programs. Further, we will propose a method for evolving more complex programs, like algorithms designed by humans.

Table 3: Comparison of the number of successful runs between GRAPE with ADN and GRAPE for training data. Parenthetic numbers are the maximum number of nodes in conventional GRAPE.

	GRAPE with ADN	GRAPE (30)	GRAPE (180)
Factorial	96	89	92
Exponentiation	39	34	34
List sorting	72	74	57

Table 4: Comparison of the number of successful runs between GRAPE with ADN and GRAPE for test data. Parenthetic numbers are the maximum number of nodes in conventional GRAPE.

	GRAPE with ADN	GRAPE (30)	GRAPE (180)
Factorial	60	69	52
Exponentiation	39	34	32
List sorting	20	33	22

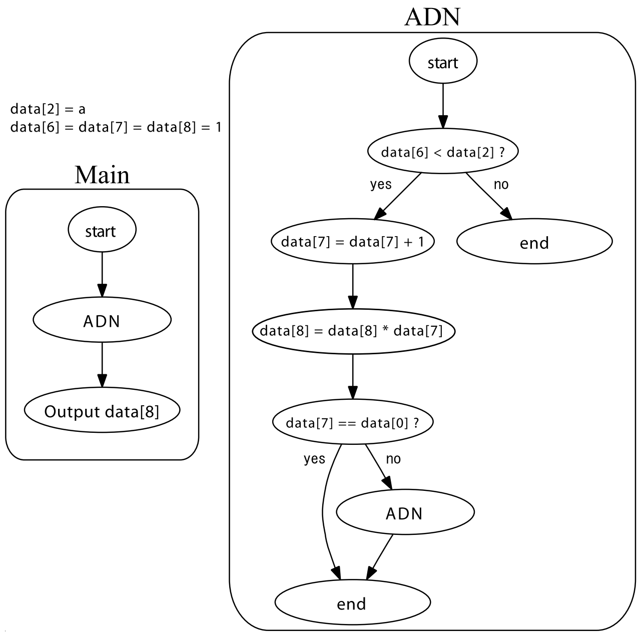


Figure 6: Example of structure obtained by GRAPE with ADN (recursive program for factorial).

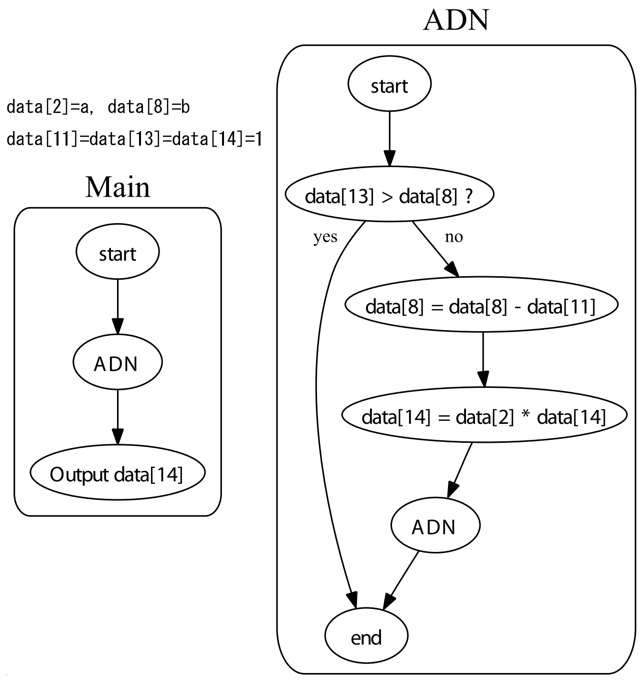


Figure 7: Example of structure obtained by GRAPE with ADN (recursive program for exponentiation).

7. REFERENCES

[1] A. Agapitos and S. M. Lucas. Evolving efficient recursive sorting algorithms. In *Proceedings of the 2006 IEEE Congress on Evolutionary Computation (CEC '06)*, pages 9227–9234, Vancouver, 6-21 July 2006. IEEE Press.

[2] A. Agapitos and S. M. Lucas. Learning recursive functions with object oriented genetic programming. In P. Collet, M. Tomassini, M. Ebner, S. Gustafson, and A. Ekárt, editors, *Proceedings of the 9th European Conference on Genetic Programming*, volume 3905 of *Lecture Notes in Computer Science*, pages 166–177, Budapest, Hungary, 10 - 12 Apr. 2006. Springer.

[3] P. J. Angeline and J. B. Pollack. Evolutionary module acquisition. In D. Fogel and W. Atmar, editors, *Proceedings of the Second Annual Conference on Evolutionary Programming*, pages 154–163, La Jolla, CA, USA, 25-26 1993.

[4] K. Deb, A. Anand, and D. Joshi. A computationally efficient evolutionary algorithm for real-parameter optimization. *Evolutionary Computation*, 10(4):371–395, 2002.

[5] H. Katagiri, K. Hirasawa, J. Hu, and J. Murata. Network structure oriented evolutionary model-genetic network programming-and its comparison with genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2001) Late Breaking Papers*, pages 219–226, San Francisco, California, USA, 9-11 July 2001.

[6] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

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

[8] J. F. Miller and S. L. Smith. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 10(2):167–174, 2006.

[9] J. F. Miller and P. Thomson. Cartesian genetic programming. In *Genetic Programming, Proceedings of EuroGP 2000*, volume 1802 of *LNCS*, pages 121–132, Edinburgh, 15-16 Apr. 2000. Springer-Verlag.

[10] H. Nakagoe, K. Hirasawa, and J. Hu. Genetic network programming with automatically generated variable size macro nodes. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation (CEC 2004)*, pages 713–719, Portland, Oregon, 20-23 June 2004. IEEE Press.

[11] R. Olsson. Inductive functional programming using incremental program transformation. *Artificial Intelligence*, 74(1):55–81, Mar. 1995.

[12] R. Poli. Evolution of graph-like programs with parallel distributed genetic programming. In *Proceedings of the Seventh International Conference on Genetic Programming*, pages 346–353, Michigan State University, East Lansing, MI, USA, 19-23 July 1997. Morgan Kaufmann.

[13] H. Satoh, M. Yamamura, and S. Kobayashi. Minimal generation gap model for considering both exploration and exploitations. In *Proceedings of the IIZUKA'96*, pages 494–497, 1996.

[14] S. Shirakawa and T. Nagao. Evolution of sorting algorithm using graph structured program evolution. In *Proceedings of the 2007 IEEE International Conference on Systems, Man and Cybernetics (SMC 2007)*, pages 1256–1261, Montreal, Canada, 7-10 Oct. 2007.

[15] S. Shirakawa, S. Ogino, and T. Nagao. Graph Structured Program Evolution. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2007)*, volume 2, pages 1686–1693, London, 7-11 July 2007. ACM Press.

[16] L. Spector. Simultaneous evolution of programs and their control structures. In P. J. Angeline and K. E. Kinnear, Jr., editors, *Advances in Genetic Programming 2*, pages 137–154. MIT Press, Cambridge, MA, USA, 1996.

[17] L. Spector and A. Robinson. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines*, 3(1):7–40, Mar. 2002.

[18] A. Teller and M. Veloso. PADO: A new learning architecture for object recognition. In K. Ikeuchi and M. Veloso, editors, *Symbolic Visual Learning*, pages 81–116. Oxford University Press, 1996.

[19] S. Tsutsui, M. Yamamura, and T. Higuchi. Multi-parent re-combination with simplex crossover in real coded genetic algorithms. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 1999)*, pages 657–664, 1999.

[20] J. A. Walker and J. F. Miller. Evolution and acquisition of modules in cartesian genetic programming. In *Genetic Programming, Proceedings of EuroGP 2004*, volume 3003 of *LNCS*, pages 187–197. Springer-Verlag, 2004.

[21] J. A. Walker and J. F. Miller. The automatic acquisition, evolution and reuse of modules in cartesian genetic programming. *IEEE Transactions on Evolutionary Computation*, 12(4):397–417, Aug. 2008.