Graph Structured Program Evolution

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

Shintaro Ogino
Venture Business Laboratory
Yokohama National University
79-7, Tokiwadai, Hodogaya-ku, Yokohama
Kanagawa, 240-8501, Japan
ogino@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

In recent years a lot of Automatic Programming techniques have developed. A typical example of Automatic Programming is Genetic Programming (GP), and various extensions and representations for GP have been proposed so far. However, it seems that more improvements are necessary to obtain complex programs automatically. In this paper we proposed a new method called Graph Structured Program Evolution (GRAPE). The representation of GRAPE is graph structure, therefore it can represent complex programs (e.g. branches and loops) using its graph structure. Each program is constructed as an arbitrary directed graph of nodes and *data set*. The GRAPE program handles multiple data types using the *data set* for each type, and the genotype of GRAPE is the form of a linear string of integers. We apply GRAPE to four test problems, factorial, Fibonacci sequence, exponentiation and reversing a list, and demonstrate that the optimum solution in each problem is obtained by the GRAPE system.

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, Graph-based Genetic Programming, Genetic Algorithm, factorial, Fibonacci sequence, exponentiation, reversing a list

GECCO'07, July 7–11, 2007, London, England, United Kingdom.

1. INTRODUCTION

This paper introduces a new method for Automatic Programming. This new method, named **GRA**ph structured **P**rogram **E**volution (GRAPE), uses graph structure as a representation of programs.

In standard Genetic Programming (GP), programs are represented as trees containing terminal and non-terminal nodes. Complex programs and hand written programs, however, may contain several branches and loops. We think that graph representation is the nearest representation of hand written programs. Therefore, we adopt the graph structure as a representation of programs. In GRAPE programs each program is constructed as an arbitrary directed graph of nodes and *data set*. The GRAPE program handles multiple data types using the *data set* for each type, and the genotype of GRAPE is the form of a linear string of integers.

The next section of this paper is an overview of several related works. In section 3, we describe our proposed method, Graph Structured Program Evolution (GRAPE). Several experiments are shown in section 4. Section 5 provides some discussion of the results. Finally, in section 6, we describe conclusions and future works.

2. RELATED WORKS

Automatic Programming is the method of generating computer programs automatically. Genetic Programming (GP) [9, 10] is a typical example of Automatic Programming, which was proposed by Koza. GP evolves computer programs, which are usually tree structure, and searches a desired program using Genetic Algorithm (GA). A lot of extensions and improvements of GP were introduced. Automatically Defined Function (ADFs)[10], Module Acquisition[2] and Automatically Defined Macros[19] were attempted to integrate modularity into the GP paradigm. Montana developed a strategy for incorporating multiple data types called Strongly Typed Genetic Programming[14]. In Strongly Typed Genetic Programming the user is required to specify the types of all values, function inputs, and function outputs, and the program generation, mutation and crossover algorithms are modified to obey these type restrictions. It affects the shape of the program search space (e.g. by restricting crossover points).

Various representations for GP have been proposed so far. GP with index memory [23, 24] was introduced by Teller and was proven that the system is Turing complete. This

means that, in theory, GP with indexed memory can be used to evolve any algorithm. Linear Genetic Programming (LGP) [3] uses a specific linear representation of computer programs. Instead of the tree-based GP expressions of a functional programming language (like LISP), programs of an imperative language (like C) are evolved. A LGP individual is represented by a variable-length sequence of simple C language instructions. Instructions operate on one or two indexed variables (registers) r or on constants c from predefined sets. The result is assigned to a destination register, e.g. $r_i = r_j * c$. Grammatical Evolution (GE) [15, 16] is an evolutionary algorithm that can evolve computer programs in any language, and can be considered a form of grammar-based genetic programming. GE uses a chromosome of numbers encoded using eight bits to indicate which rule from the BNF (Backus Naur Form) grammar to apply at each state of the derivation sequence, starting from a defined start symbol.

Recently two interesting Automatic Programming techniques were proposed, PushGP [20, 21, 22] and Object Oriented Genetic Programming (OOGP) [11, 1]. 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. The both method tackled the problems of generating recursive programs (e.g. factorial, Fibonacci sequence, exponentiation, sorting a list and so on) and obtained these programs automatically.

There are various representations using a graph. Parallel Algorithm Discovery and Orchestration (PADO) [25, 26]is one of the graph based GPs instead of the tree structure. PADO uses stack memory and index memory, and there are *action* and *branch-decision* nodes. The execution of PADO is carried out from the start node to the end node in the network. PADO was applied to the object recognition problems. Another graph based GP is the Parallel Distributed Genetic Programming (PDGP)[17]. In this approach the tree is represented as a graph with functions and terminals nodes located over a grid. In this way it is possible straightforward to execute several nodes concurrently. Cartesian Genetic Programming (CGP)[12, 13] was developed from a representation that was used for the evolution of digital circuits and represents a programs as a graph. In certain respects, it is similar to the graph-based technique PDGP. However, PDGP were evolved without the use of a 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 into phenotype of an index graph. Linear-Graph GP [6] is the extension of Linear GP and Linear-Tree GP [5]. In Linear-Graph GP each program is represented as a graph. Each node in the graph has two parts, a linear program and a branching node. Recently, Genetic Network Programming (GNP) [4, 7] which has a directed graph structure is proposed. GNP is applied to make the behavior sequences of agents and shows better performances compared with GP.

3. GRAPH STRUCTURED PROGRAM EVOLUTION (GRAPE)

3.1 Overview

Various extensions and representations for GP have been proposed so far. However, it seems that more improvements is necessary to obtain more complex programs automatically. Graph Structured Program Evolution (GRAPE) constructs graph structured programs automatically. The graph structured programs is composed of arbitrary directed graph of nodes and *data set*.

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 feed-forward 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.

The features of GRAPE are summarized as follows:

- Arbitrary directed graph structures.
- Handle multiple data types using the *data set*.
- Genotype of integer string.

3.2 Structure of GRAPE

The representation of GRAPE is graph structure. Each program is constructed as an arbitrary directed graph of nodes and *data set*. The *data set* flows the directed graph and is processed at each node. Figure 1 illustrates an example of structure of GRAPE. Each node in GRAPE program has two parts, a *processing* and *branching*. The *processing* executes several kind of processing using the *data set*, for instance, arithmetic calculation and boolean calculation. After the *processing* is executed, a next node is selected. The *branching* decides the next node according to the *data set*.

Examples of node in GRAPE are shown in Figure 2. "No.1 node" add $data[0]$ to $data[1]$ and substitute for $data[0]$ using integer data type, and select next node "No.2". "No.2 node" decides next node using integer $data[0]$ and $data[1]$, if $data[0]$ is grater than $data[1]$, connection 1 is chosen, else connection 2 is chosen. There are special nodes shown in Figure 1. "No.0 node" 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 node" is the *output node*. When this node is reached, the GRAPE program outputs data and then the program halts. In Figure 1 "No.7 node" outputs integer $data[0]$. Although the GRAPE program has only one start node, it has several *output nodes*.

The representation of GRAPE is graph structure, therefore it can represent complex programs (e.g. branches and loops) using its graph structure. There are several data types in GRAPE program, integer data type, boolean data type, list data type and so on. The GRAPE program handles multiple data types using the *data set* for each type.

To adopt evolutionary method, genotype-phenotype mapping is used in GRAPE system. This genotype-phenotype

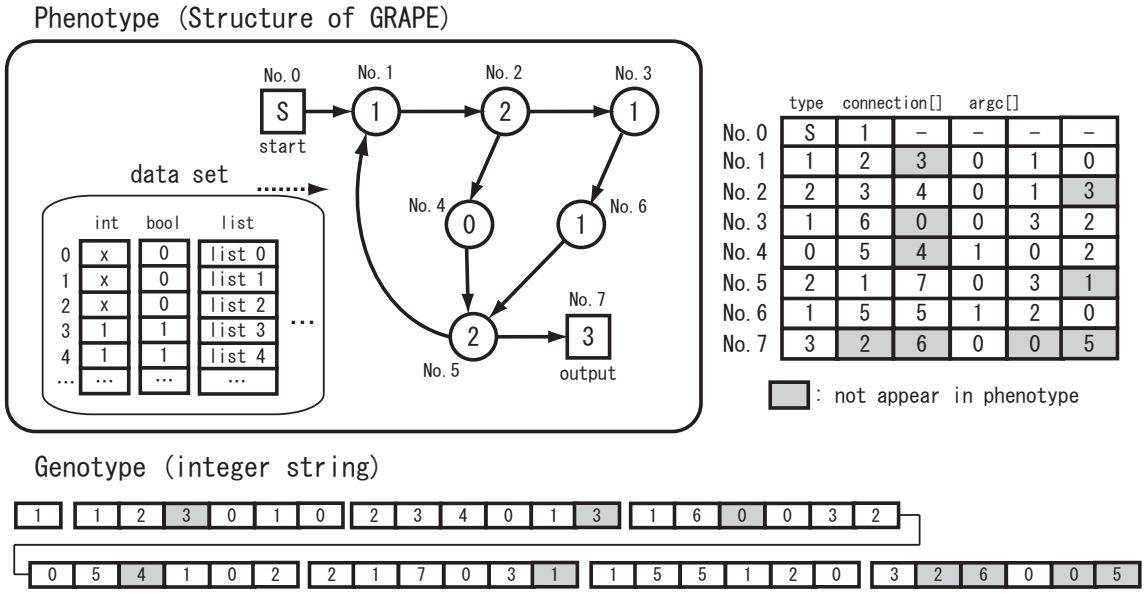


Figure 1: Structure of GRAPE (phenotype) and the genotype which denotes a list of node types, connections and arguments.

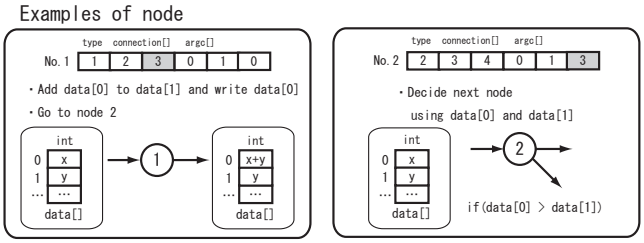


Figure 2: Examples of node in GRAPE.

mapping method is similar to Cartesian Genetic Programming (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 connections of nodes are arbitrary, that is different from CGP. The length of the genotype is fixed and equals to $N * (n_c + n_a + 1) + 1$, where N is the number of nodes, n_c is the maximum number of connections and n_a is the maximum number of arguments.

3.3 Genetic Operators of 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 usual Genetic Algorithm (GA). In this paper we use uniform crossover and mutation as the genetic operators. The uniform crossover operator effects two individuals, as follows:

- Select several genes randomly according to the crossover rate P_c for each gene.
- The selected genes are swapped between two parents, and generate offspring.

The mutation operator effects one individual, as follows:

Table 1: The parameters of GRAPE algorithm.

Parameter	Value
The number of evaluations	2500000
Population size	500
Crossover rate P_c	0.9
Mutation rate P_m	0.02
The number of nodes	10, 30, 50
Execution step limits	500

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

4. EXPERIMENTS AND RESULTS

Several different problems have been tackled in order to verify the effectiveness of GRAPE. The problems include the computations of factorial, Fibonacci sequence, exponentiation and reversing a list. Evolution of these programs is difficult for standard GP. It needs to prepare iteration or recursion mechanisms.

The parameters of GRAPE are given in Table 1, and the node functions are shown in Table 2. We prepare sufficient *data set* size 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. We used three different search strategies which are Simple Genetic Algorithm (SGA), Minimal Generation Gap (MGG) and Random Search (RS), using the number of nodes of 10, 30, and 50. Tournament selection (a tournament size of 2) along with elitist strategy (an elite size of 5) was used as the selection mechanism in SGA. We use tournament size of 2 to prevent the early convergence and maintain the diversity of the population. The MGG model [18, 8, 27] is a steady

state model proposed by Satoh et al. The MGG model has a desirable convergence property maintaining the diversity of the population, and shows higher performance than the other conventional models in a wide range of applications. We used the MGG model in these experiments as follows:

1. Set 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 m offspring 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 by the 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.

In these experiments we used $m = 50$. All individuals generate randomly in RS (i.e. no selection pressure and the genetic operators).

In order to avoid the problem caused by non-terminating structures we limited the execution step to 500. When a program reaches the execution limit, the individual is assigned the fitness 0.0.

4.1 Factorial

In this problem we seek to evolve a implementation of the factorial function. We used 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) (5, 120). We used the normalized absolute mean error on the training set as a fitness function. The fitness function used in these experiments is:

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

where $Correct_i$ is correct value for the training data i, $Estimate_i$ 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 is [0.0, 1.0]. The higher the numerical value indicates the better performance. If the fitness in Equation 1 is reached 1.0, the fitness function is calculated as follows:

$$fitness = 1.0 + \frac{1}{S_{exe}} \quad (2)$$

where S_{exe} is the total number of execution steps of the generated program. This fitness function means the less execution step is the better solution.

In this experiment integer data type is used, and the size of integer data in GRAPE is 10. Initially, we set input value a on the $data$[0] to $data$[4] and constant value 1 on the $data$[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 3 (a) shows transition of success rate. The success rate is computed as:

$$\text{Success rate} = \frac{\text{Number of successful runs}}{\text{Total number of runs}}. \quad (3)$$

We apply the elitist individual generated by GRAPE to the test data set for each run. The integers from 6 to 12 are used as the test set inputs. The success rate for the test set appear in Table 3. The "MGG node 50" shows best performance (training set:69% , test set:59%).

Figure 4 (a) is an example of obtained structure for factorial. This GRAPE program has loop structure, and calculates completely factorial.

4.2 Fibonacci Sequence

We used the first 12 elements of the sequence as the training set, with the following input/output pairs (a, b): (1, 1) (2, 1) (3, 2) (4, 3) (5, 5) (6, 8) (7, 13) (8, 21) (9, 34) (10, 55) (11, 89) (12, 144). We also used the fitness function in Equation 1 and 2 on the training set.

In this experiment integer data type is used, and the size of integer data in GRAPE is 10. Initially, we set input value a on the $data$[0] to $data$[4] and constant value 1 on the $data$[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 3 (b) shows transition of success rate.

We apply the elitist individual generated by GRAPE to the test data set for each run. The integers from 13 to 30 are used as the test set inputs. The success rate for the test set appear in Table 3. The "MGG node 30" shows best performance (training set:8% , test set:6%).

Figure 4 (b) is an example of obtained structure for Fibonacci sequence. This GRAPE program also has loop structure, and calculates completely Fibonacci sequence.

4.3 Exponentiation

In this problem we seek to evolve a implementation of the integer exponential a^b. There are two inputs in this problem. The training set (a, b, c) used in this experiment are (2, 0, 1) (2, 1, 2) (2, 2, 4) (3, 3, 9) (3, 4, 27) (3, 5, 81) (4, 6, 4096) (4, 7, 16384) (4, 8, 65536). We also used the fitness function in Equation 1 and 2 on the training set.

In this experiment integer data type is used, and the size of integer data in GRAPE is 9. Initially, we set input value a on the $data$[0] to $data$[2], input value b on the $data$[3] to $data$[5] and constant value 1 on the $data$[6] to $data$[8]. 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 3 (c) shows transition of success rate.

We apply the elitist individual generated by GRAPE to the test data set for each run. 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). The success rate for the test set appear in Table 3. The "MGG node 30" shows best performance (training set:45% , test set:44%).

Figure 4 (c) is an example of obtained structure for exponentiation. This GRAPE program also calculates completely exponentiation.

4.4 Reversing a List

For this problem we provide a list of integers as input. A correct program returns a reverse input list, of any length (e.g. input: (1 2 3 4), output: (4 3 2 1)). We use the length of list between 5 and 10 as the training set. The fitness function used in this experiment is given in Equation 4. The range of this fitness function is [0.0, 1.0]. The higher

Table 2: GRAPE node functions for each experiment.

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	If data[x] is equal data[y] connection 1 is chosen else connection 2 is chosen.
>	2	2	x, y	If data[x] is greater than data[y] connection 1 is chosen else connection 2 is chosen.
<	2	2	x, y	If data[x] is less than data[y] connection 1 is chosen else connection 2 is chosen.
SwapList	1	2	x, y	Use integer type and a list data. Swap list[data[x]] for list[data[y]].
OutputInt	0	1	x	Output integer data[x] and then the program halts.
OutputList	0	0	-	Output a list data and then the program halts.

the numerical value indicates the better performance.

$$fitness = 1 - \frac{\sum_{i=1}^{n} \frac{\sum_{j=0}^{l}(1 - \frac{1}{2^{d_{ij}}})}{l_i}}{n} \quad (4)$$

where d_{ij} is the distance between the correct position and the return value position for the training data i for the element j. l_i is the length of the list for the training data i and n is the size of the training set. If the fitness in Equation 4 is reached 1.0, the fitness function is calculated using Equation 2.

In this experiment a list of integers and integer data type are used, and the size of integer data in GRAPE is 9. Initially, we set the size of input list (the list length) on the $data[0]$ to $data[2]$, constant value 0 on the $data[3]$ to $data[5]$ and constant value 1 on the $data[6]$ to $data[8]$. The node functions used in this experiment are $\{ +, -, *, /, =, >, <, SwapList, OutputList \}$ in Table 2.

Results are given for 100 different runs with the same parameter set. Figure 3 (d) shows transition of success rate.

We apply the elitist individual generated by GRAPE to the test data set for each run. The length of list between 11 and 15 are used as the test set. The success rate for the test set appear in Table 3. The "MGG node 50" shows best performance (training set:71% , test set:65%).

Figure 4 (d) is an example of obtained structure for reversing a list. This GRAPE program also calculates completely reversing a list. The obtained graph structured program handles multiple data types, integer and list data types.

5. DISCUSSION

GRAPE successfully generates solution to four problems automatically, and obtained structure is unique and solves each problem completely. GRAPE generates easily the programs including the branches and loops using its graph representation and handles multiple data types through *data set*. The evolution of the GRAPE programs is efficient without *bloat* through the genotype of fixed integer string.

Table 3 provides a summary and comparison of the performance of GRAPE on each of the problem domains tackled. In all problems MGG model outperforms the other strategies, while random search is the worst performance. Therefore, it shows that the evolutionary method is functionally effective. As a result of the experiments, the number of nodes 30 or 50 shows the better performance than the number of node 10. Therefore, we should prepare sufficient nodes to representing the programs. The almost GRAPE programs which succeed the training set are also solve the test set.

The programs of these four problems can represent simply using recursion. Although we have not prepared recursion function in this paper, the GRAPE system has constructed the optimum programs using the branches, the loops and the multiple data types. If we introduce recursion functions or modularity mechanisms (like ADFs) to GRAPE, the performance of GRAPE may improve.

6. CONCLUSIONS AND FUTURE WORKS

In this paper we propose a new method for Automatic Programming, Graph Structured Program Evolution (GRAPE). The representation of GRAPE is graph structure. Each program is constructed as an arbitrary directed graph of nodes and *data set*. The data set flows the directed graph and is processed at each node. GRAPE adopts the genotype-phenotype mapping. The genotype is an integer string which denotes a list of node types, connections and arguments.

We applied GRAPE to four different problems, factorial, Fibonacci sequence, exponentiation and reversing a list, and confirmed that the optimum solution in each problem was obtained by the GRAPE system. We used several search strategies, Minimal Generation Gap (MGG), Simple GA (SGA) and Random Search (RS), and compared the performance of these methods. As a result we showed that the evolutionary method is functionally effective.

In future works we will introduce recursion functions or

modularity mechanisms to solve more complex problems. Moreover, we will plan to apply GRAPE to the problems which are more large scale and require more complex structure, for example, sorting a list, signal processing and so on.

7. REFERENCES

[1] 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.

[2] 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.

[3] M. Brameier and W. Banzhaf. A comparison of linear genetic programming and neural networks in medical data mining. *IEEE Transactions on Evolutionary Computation*, 5(1):17–26, Feb. 2001.

[4] T. Eguchi, K. Hirasawa, J. Hu, and N. Ota. A study of evolutionary multiagent models based on symbiosis. *IEEE Transactions on Systems, Man and Cybernetics Part B*, 36(1):179–193, 2006.

[5] W. Kantschik and W. Banzhaf. Linear-tree GP and its comparison with other GP structures. In J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, editors, *Genetic Programming, Proceedings of EuroGP'2001*, volume 2038 of *LNCS*, pages 302–312, Lake Como, Italy, 18-20 Apr. 2001. Springer-Verlag.

[6] W. Kantschik and W. Banzhaf. Linear-graph GP—A new GP structure. In J. A. Foster, E. Lutton, J. Miller, C. Ryan, and A. G. B. Tettamanzi, editors, *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 83–92, Kinsale, Ireland, 3-5 Apr. 2002. Springer-Verlag.

[7] H. Katagiri, K. Hirasawa, J. Hu, and J. Murata. Network structure oriented evolutionary model-genetic network programming-and its comparison with genetic programming. In E. D. Goodman, editor, *2001 Genetic and Evolutionary Computation Conference Late Breaking Papers*, pages 219–226, San Francisco, California, USA, 9-11 July 2001.

[8] H. Kita, I. Ono, and S. Kobayashi. Multi-parental extension of the unimodal normal distribution crossover for real-coded genetic algorithms. In *Proceedings of the 1999 Congress on Evolutionary Computation (CEC99)*, volume 2, pages 1581–1587, 1999.

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

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

[11] S. Lucas. Exploiting reflection in object oriented genetic programming. In M. Keijzer, U.-M. O'Reilly, S. M. Lucas, E. Costa, and T. Soule, editors, *Genetic Programming 7th European Conference, EuroGP 2004, Proceedings*, volume 3003 of *LNCS*, pages 369–378, Coimbra, Portugal, 5-7 Apr. 2004. Springer-Verlag.

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

[13] J. F. Miller and P. Thomson. Cartesian genetic programming. In R. Poli, W. Banzhaf, W. B. Langdon, J. F. Miller, P. Nordin, and T. C. Fogarty, editors, *Genetic Programming, Proceedings of EuroGP'2000*, volume 1802 of *LNCS*, pages 121–132, Edinburgh, 15-16 Apr. 2000. Springer-Verlag.

[14] D. J. Montana. Strongly typed genetic programming. *Evolutionary Computation*, 3(2):199–230, 1995.

[15] M. O'Neill and C. Ryan. Grammatical evolution. *IEEE Transactions on Evolutionary Computation*, 5(4):349–358, Aug. 2001.

[16] M. O'Neill and C. Ryan. *Grammatical Evolution: Evolutionary Automatic Programming in a Arbitrary Language*, volume 4 of *Genetic programming*. Kluwer Academic Publishers, 2003.

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

[18] 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.

[19] 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.

[20] L. Spector. Autoconstructive evolution: Push, pushGP, and pushpop. In L. Spector, E. D. Goodman, A. Wu, W. B. Langdon, H.-M. Voigt, M. Gen, S. Sen, M. Dorigo, S. Pezeshk, M. H. Garzon, and E. Burke, editors, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)*, pages 137–146, San Francisco, California, USA, 7-11 July 2001. Morgan Kaufmann.

[21] L. Spector, J. Klein, and M. Keijzer. The push3 execution stack and the evolution of control. In H.-G. Beyer, U.-M. O'Reilly, D. V. Arnold, W. Banzhaf, C. Blum, E. W. Bonabeau, E. Cantu-Paz, D. Dasgupta, K. Deb, J. A. Foster, E. D. de Jong, H. Lipson, X. Llora, S. Mancoridis, M. Pelikan, G. R. Raidl, T. Soule, A. M. Tyrrell, J.-P. Watson, and E. Zitzler, editors, *GECCO 2005: Proceedings of the 2005 conference on Genetic and evolutionary computation*, volume 2, pages 1689–1696, Washington DC, USA, 25-29 June 2005. ACM Press.

[22] 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.

[23] A. Teller. Learning mental models. In *Proceedings of the Fifth Workshop on Neural Networks: An International Conference on Computational Intelligence: Neural Networks, Fuzzy Systems, Evolutionary Programming, and Virtual Reality*, 1993.

[24] A. Teller. Turing completeness in the language of genetic programming with indexed memory. In *Proceedings of the 1994 IEEE World Congress on Computational Intelligence*, volume 1, pages 136–141, Orlando, Florida, USA, 27-29 June 1994. IEEE Press.

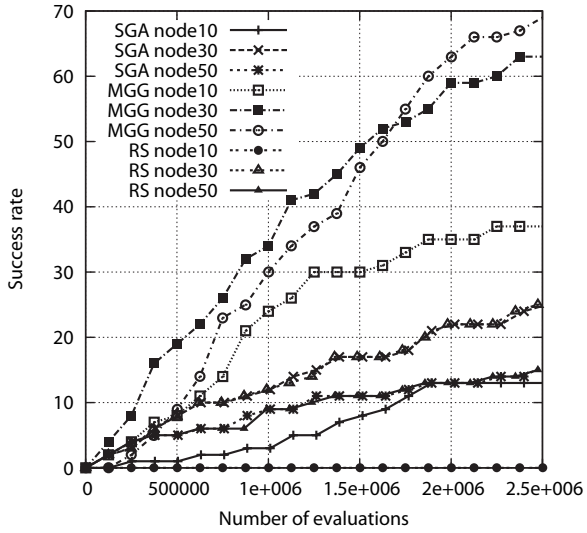
[25] A. Teller and M. Veloso. Program evolution for data mining. *The International Journal of Expert Systems*, 8(3):216–236, 1995.

[26] 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.

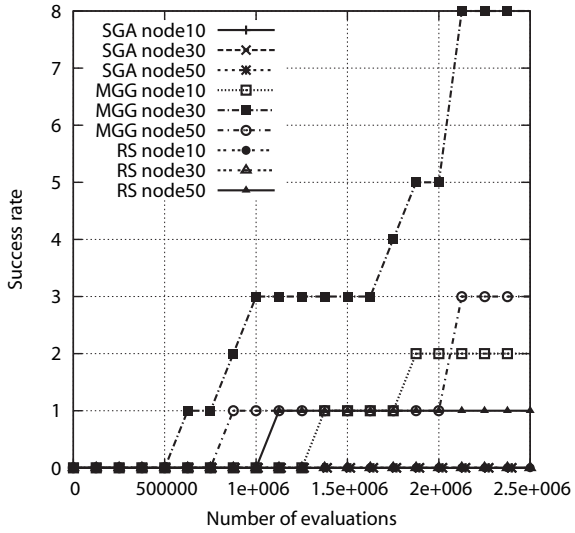
[27] S. Tsutsui, M. Yamamura, and T. Higuchi. Multi-parent re-combination with simplex crossover in real coded genetic algorithms. In *In Proceedings of Genetic and Evolutionary Computation Conference (GECCO'99)*, pages 657–664, 1999.

Table 3: The success rate on each of the problem domains tackled.

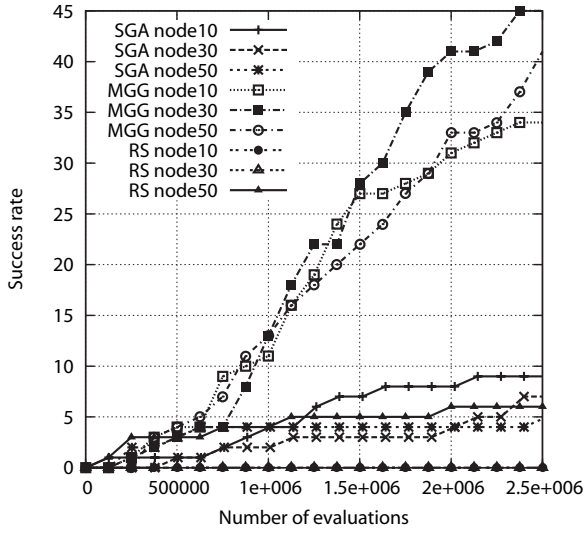
	Factorial		Fibonacci Sequence		Exponentiation		Reversing a List	
	Training set	Test set	Training set	Test set	Training set	Test set	Training set	Test set
SGA node10	13%	12%	0%	0%	9%	9%	22%	20%
SGA node30	25%	23%	0%	0%	7%	7%	41%	30%
SGA node50	16%	16%	0%	0%	5%	5%	37%	34%
MGG node10	37%	37%	2%	2%	34%	34%	22%	21%
MGG node30	63%	57%	**8%**	**6%**	**45%**	**44%**	63%	56%
MGG node50	**69%**	**59%**	3%	2%	41%	40%	**71%**	**65%**
RS node10	0%	0%	0%	0%	0%	0%	0%	0%
RS node30	1%	1%	0%	0%	0%	0%	0%	0%
RS node50	15%	13%	0%	0%	6%	1%	0%	0%



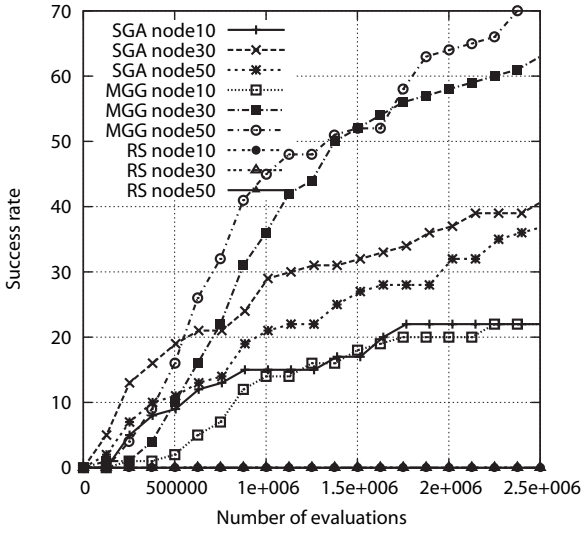
(a) Factorial.



(b) Fibonacci Sequence.

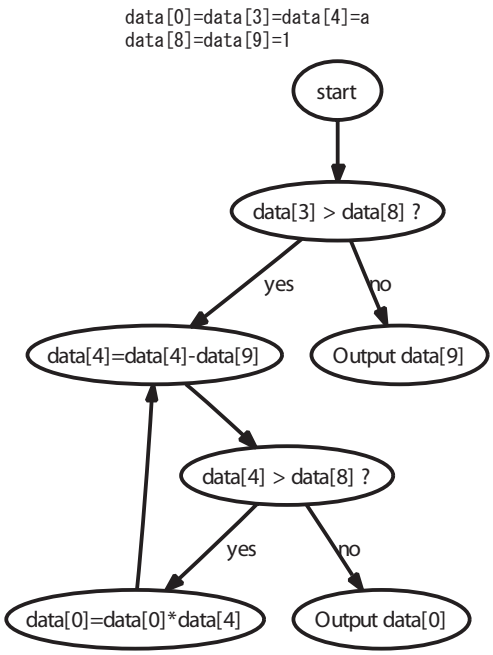


(c) Exponentiation.

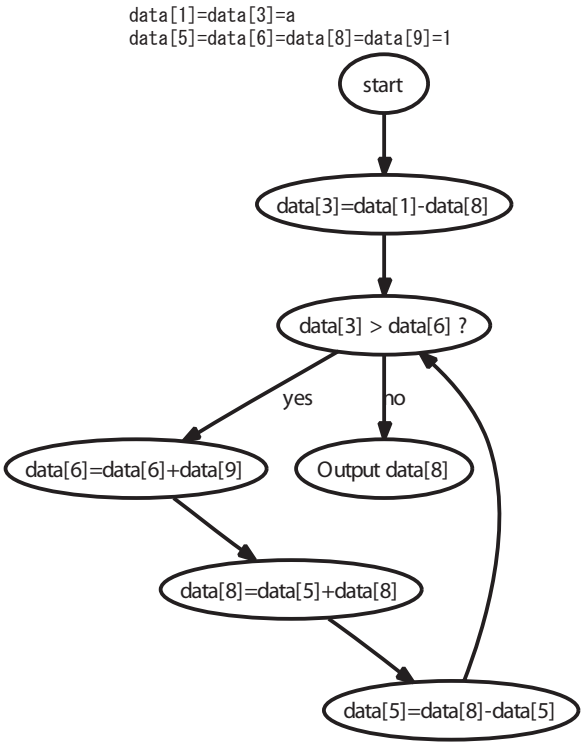


(d) Reversing a list.

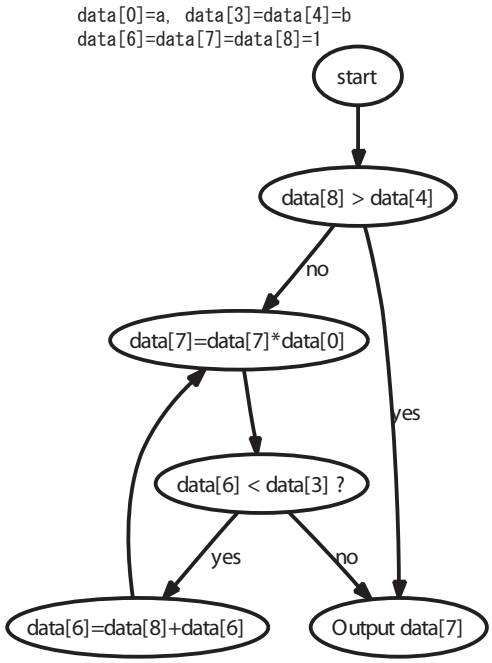
Figure 3: This graphs show the comparison of the success rate of the various runs.



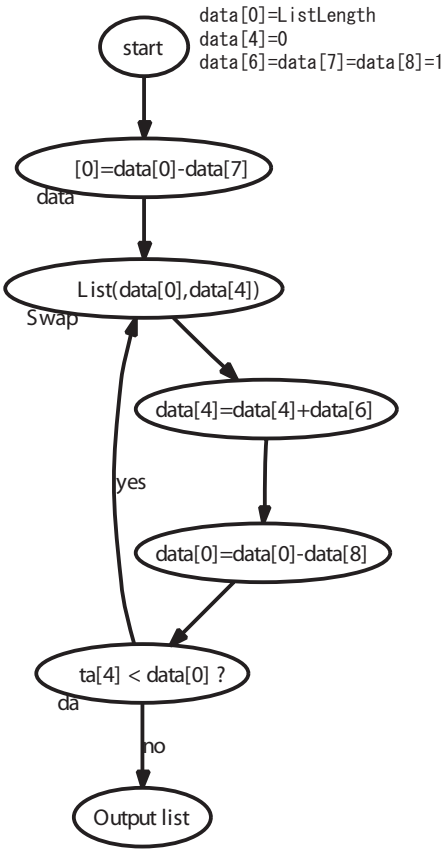
(a) Factorial.



(b) Fibonacci Sequence.



(c) Exponentiation.



(d) Reversing a list.

Figure 4: Examples of obtained structure of GRAPE.