Evolution of Sorting Algorithm
using Graph Structured Program Evolution

Shinichi Shirakawa and Tomoharu Nagao

Abstract—In this paper, we apply Graph Structured Program Evolution (GRAPE) to evolution of general sorting algorithm. GRAPE is a new Automatic Programming technique. The representation of GRAPE is graph structure, therefore it can express 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. GRAPE handles multiple data types using data set for each type, and the genotype of GRAPE is the form of a linear string of integers. The aim of this work is to evolve a program which correctly sort any sequence of numbers. We demonstrate that GRAPE constructs general sorting algorithm automatically.

I. INTRODUCTION

In recent years a lot of Automatic Programming techniques have developed and applied to a variety of problem domains. A typical example of Automatic Programming is Genetic Programming (GP), and various extensions and representations for GP have been proposed so far. In standard 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 believe that graph representation is the nearest representation of hand written programs. Therefore, we adopt the graph structure as a representation of programs. Our method, named GRAph Structured Program Evolution (GRAPE) [1], uses graph structure as a representation of programs. In GRAPE, each program is constructed as an arbitrary directed graph of nodes and data set. GRAPE handles multiple data types using data set for each type, and the genotype of GRAPE is the form of a linear string of integers. In this paper, we apply GRAPE to evolution of general sorting algorithm. The problem of evolution of general sorting algorithm is considered challenging problem, and it needs to prepare iteration or recursion mechanism.

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). Next, in section 4, we apply the proposed method to evolution of general sorting algorithm and show several experimental results. Finally, in section 5, we describe conclusions and future works.

II. RELATED WORKS

Automatic Programming is a method of generating computer programs automatically. Genetic Programming (GP) [2], [3] 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).

Various representations for GP have been proposed so far. GP with index memory [4], [5] 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) [6], [7] 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) [8], [9] 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.

There are various representations using a graph. Parallel Algorithm Discovery and Orchestration (PADO) [10]–[12] 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) [13]. 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) [14], [15] was developed from a representation that was used for the evolution of digital circuits and represents a program as a graph. In certain respects, it is similar to the graph-based technique PDGP. However, CGP adopts the genotype-phenotype mapping. 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 [16] is the extension of Linear GP and Linear-Tree GP [17]. 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) [18], [19] 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.

Several researchers have investigated evolution of general sorting algorithm. Kinnear evolved general iterative sorting algorithm using Genetic Programming. He prepared special functions for the sorting problem, such as Iteration Function which is an iterative operator.

Recently two interesting Automatic Programming techniques were proposed, PushGP [20]–[22] and Object Oriented Genetic Programming (OOGP) [23]–[25]. 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. PushGP evolves programs using a Push language proposed by Spector, et al. Push is a stack-based programming language. They showed that PushGP evolved a general sorting algorithm [21]. OOGP evolves Object Oriented Programs instead of the form of LISP parse tree. OOGP evolved a general recursive sorting algorithm using its recursive mechanism [25].

III. GRAPH STRUCTURED PROGRAM EVOLUTION (GRAPE)

A. Overview

Various extensions and representations for GP have been proposed so far. However, it seems that more improvements are necessary to obtain more complex programs automatically. Graph Structured Program Evolution (GRAPE) [1] constructs graph structured programs automatically. The graph structured programs is composed of arbitrary directed graph of nodes and data set.

B. Structure of GRAPE

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

Examples of node in GRAPE are shown in Fig. 2. “No.1 node” adds data[0] to data[1] and substitute for data[0] using integer data type, and moves to the node “No.2”. “No.2 node” decides the 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 Fig. 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 Fig. 1 “No.7 node” outputs integer data[0]. Although the GRAPE program has only one start node, it can have 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. GRAPE 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 mapping is used in GRAPE system.
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 \( N \times (n_c + n_a + 1) + 1 \), where \( N \) is the number of nodes, \( n_c \) is the maximum number of connections and \( n_a \) is the maximum number of arguments.

C. 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:

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

D. Features of GRAPE

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 difference 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.

IV. EXPERIMENTS AND RESULTS

In this section, we apply GRAPE to the problem of automatic construction of sorting algorithm.

A. Settings of Experiments

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 sorting 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. Elements are randomly chosen from the range of \([0, 255]\).

The fitness function used in this experiment is given in (1). The range of this fitness function is \([0.0, 1.0]\). The higher the numerical value indicates the better performance.

\[
\text{fitness} = 1.0 - \frac{\sum_{i=1}^{n} (1 - \frac{1}{2d_{ij}})}{n}
\]

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 data set. If the fitness in (1) is reached 1.0, the fitness function is calculated using (2).

\[
\text{fitness} = 1.0 + \frac{1}{S_{exe}}
\]

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.

We used Minimal Generation Gap (MGG) as the generation alternation model. The MGG model [26]–[28] 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 elite 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 \( P(t+1) \).
TABLE II
GRAPE NODE FUNCTIONS FOR THE EXPERIMENT.

<table>
<thead>
<tr>
<th>Name</th>
<th># Connections</th>
<th># Args.</th>
<th>Argument(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>1</td>
<td>3</td>
<td>x, y, z</td>
<td>Use integer data type. Add data[x] to data[y] and substitute for data[z].</td>
</tr>
<tr>
<td>−</td>
<td>1</td>
<td>3</td>
<td>x, y, z</td>
<td>Use integer data type. Subtract data[x] from data[y] and substitute for data[z].</td>
</tr>
<tr>
<td>×</td>
<td>1</td>
<td>3</td>
<td>x, y, z</td>
<td>Use integer data type. Multiply data[x] by data[y] and substitute for data[z].</td>
</tr>
<tr>
<td>/</td>
<td>1</td>
<td>3</td>
<td>x, y, z</td>
<td>Use integer data type. Divide data[x] by data[y] and substitute for data[z].</td>
</tr>
<tr>
<td>=</td>
<td>2</td>
<td>2</td>
<td>x, y</td>
<td>Use integer data type. If data[x] is equal to data[y] connection 1 is chosen else connection 2 is chosen.</td>
</tr>
<tr>
<td>&gt;</td>
<td>2</td>
<td>2</td>
<td>x, y</td>
<td>Use integer data type. If data[x] is greater than data[y] connection 1 is chosen else connection 2 is chosen.</td>
</tr>
<tr>
<td>&lt;</td>
<td>2</td>
<td>2</td>
<td>x, y</td>
<td>Use integer data type. If data[x] is less than data[y] connection 1 is chosen else connection 2 is chosen.</td>
</tr>
<tr>
<td>SwapList</td>
<td>1</td>
<td>2</td>
<td>x, y</td>
<td>Use integer type and a list data. Swap list[data[x]] for list[data[y]].</td>
</tr>
<tr>
<td>EqualList</td>
<td>2</td>
<td>2</td>
<td>x, y</td>
<td>Use integer type and a list data. If list[data[x]] is equal to list[data[y]] connection 1 is chosen else connection 2 is chosen.</td>
</tr>
<tr>
<td>GreaterList</td>
<td>2</td>
<td>2</td>
<td>x, y</td>
<td>Use integer type and a list data. If list[data[x]] is greater than list[data[y]] connection 1 is chosen else connection 2 is chosen.</td>
</tr>
<tr>
<td>LessList</td>
<td>2</td>
<td>2</td>
<td>x, y</td>
<td>Use integer type and a list data. If list[data[x]] is less than list[data[y]] connection 1 is chosen else connection 2 is chosen.</td>
</tr>
<tr>
<td>OutputList</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>Output a list data and then the program halts.</td>
</tr>
</tbody>
</table>

5) Stop if a certain specified condition is satisfied, otherwise set \( t = t + 1 \) and go to step 2.

In these experiments we used \( m = 50 \).

The parameters of GRAPE are given in Table I. The number of nodes was 10, 30, and 50. In order to avoid the problem caused by non-terminating structures we limited the execution step to 3,000. When a program reaches the execution limit, the individual is assigned the fitness 0.0. The node functions are shown in Table II. We prepare simple node functions, arithmetic functions, swap the elements of list and compare the elements of list. We do not prepare special functions such as iteration functions.

In the experiment a list of integers and integer data type are used, and the size of integer data in GRAPE is 15. Initially, we set the size of input list (the list length) on the \( data[0] \) to \( data[4] \), constant value 0 on the \( data[5] \) to \( data[9] \) and constant value 1 on the \( data[10] \) to \( data[14] \).

B. Results

Results are given for 100 different runs with the same parameter set. Fig. 3 shows transition of average fitness of 100 trials. Fig. 4 shows transition of success rate. The success rate is computed as:

\[
\text{Success rate} = \frac{\text{Number of successful runs}}{\text{Total number of runs}}. \tag{3}
\]

We apply the elite individual generated by GRAPE to the test data set for each run. The test set is 50 random lists. We use the length of the list between 10 and 50 as the test set. Elements are randomly chosen from the range of \([0, 255]\). The results are shown in Table III.

Fig. 5 is an example of obtained structure which is general sorting algorithm. This GRAPE program has two loop structures and sorts any sequences of numbers.

C. Discussion

GRAPE successfully generates the sorting algorithm automatically, and the obtained structure is unique and solves completely the problems. In Fig. 4 the success rate rises greatly after 2,000,000 fitness evaluations, and finally reaches 67% when the number of nodes is 50. At the beginning of the evolution, the success rate is 0% until about 2,000,000 fitness evaluations. It shows that the evolution of sorting algorithm is very difficult. Although the success is 0% at the beginning of the evolution, the fitness rises little by little in Fig. 3. Therefore, it shows that the evolutionary method is functionally effective.

Table III provides the success rate for the training set and the test set. The results shows high success rate (“node30” is 62%, “node50” is 67%) for the training set. It shows that GRAPE is a powerful Automatic Programming technique. For the test set, “node30” shows best performance (39%), and “node50” is 28%. We can consider that these programs which success for the test set are general sorting algorithm. When the number of nodes is 10, the result is not good (the success rate is 3% for training set and test set). Because it is hard to construct the sorting programs automatically for the training set using “node10”. However, all of the successful individuals (programs) for the training set (3%) also succeed with the test set.

In the experiment, we use (2) as the fitness function. This fitness function means the less execution step is the better solution. For this reason, GRAPE tends to construct the exclusive sorting algorithm for the training set. Thus, the fitness function and selection of the training set are very important for GRAPE.

Fig. 5 is an example of obtained structure which is general sorting algorithm. This GRAPE program has two loop structures and sorts any sequences of numbers.
Fig. 3. Average performance over 100 trials by GRAPE with the number of node 10, 30 and 50.

Fig. 4. Transition of success rate over 100 trials by GRAPE with the number of node 10, 30 and 50.

obtained structure briefly translates to the Fig. 6 like C language form. This algorithm is similar to the selection sort.

The representation of GRAPE is graph structure, therefore it can easily represent loops and branches. The programs of sorting algorithms can be represented using recursion. A lot of elegant and efficient sorting algorithms (e.g. merge sort, quick sort) are best expressed as recursive functions. Although we have not prepared recursion function in this paper, GRAPE has constructed the best programs using the branches, the loops and the multiple data types. If we introduce recursion functions or modularity mechanisms (like ADFs; Automatically Defined Function [3]) to GRAPE, the performance of GRAPE may improve.

V. CONCLUSIONS AND FUTURE WORKS

In this paper, we apply Graph Structured Program Evolution (GRAPE) to evolution of general sorting algorithm. 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 confirm that the general sorting algorithm is obtained by GRAPE. The obtained structures (programs) of GRAPE are unique and have loops and branches. GRAPE obtain high success rate for the training and the test data set. As a result we showed that the evolutionary method is functionally effective. We show that GRAPE is a powerful Automatic Programming technique.

As for further research topics, we will introduce recursion functions or modularity mechanisms to obtain more elegant and efficient sorting algorithm or 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, control of autonomous agent, multi agent system and signal processing.

<table>
<thead>
<tr>
<th></th>
<th>Training set</th>
<th>Test set</th>
</tr>
</thead>
<tbody>
<tr>
<td>MGG node10</td>
<td>3%</td>
<td>3%</td>
</tr>
<tr>
<td>MGG node30</td>
<td>62%</td>
<td>39%</td>
</tr>
<tr>
<td>MGG node50</td>
<td>67%</td>
<td>28%</td>
</tr>
</tbody>
</table>

TABLE III

THE SUCCESS RATE FOR THE TRAINING SET AND THE TEST SET.
data[4] = ListLength;
while(1) {
    data[11] = 0;
    if(data[4] == 0) {
        return List[];
    }
    else {
        do {
            if(List[data[11]] > List[data[4]])
                SwapList(data[4], data[11]);
        } while(data[4] == data[11]);
    }
}

Fig. 6. This C language like program is translated from the GRAPE program in Fig. 5.

REFERENCES


