Graph Structured Program Evolution
with Automatically Defined Nodes

Shinichi Shirakawa and Tomoharu Nagao
Yokohama National University, Japan
Table of contents

1. Introduction
   - Related works
   - Graph Structured Program Evolution (GRAPE)

2. GRAPE with Automatically Defined Nodes (ADN)
   - Proposed method

3. Experiments and Results

4. Summary
Graph Structured Program Evolution with Automatically Defined Nodes

Background

- Automatic programming
  - Generate computer programs automatically
  - Various automatic programming techniques
  - Typical example is genetic programming (GP)
Various GP techniques

- **Genetic Programming (GP)**
  - Tree structure

- **Linear Genetic Programming (LGP)**
  - Linear representation of computer programs

- **Grammatical Evolution (GE)**
  - Grammar based GP

- **Cartesian Genetic Programming (CGP)**
  - Graph representation

- **Genetic Network Programming (GNP)**
  - Graph representation

- **Graph Structured Program Evolution (GRAPE)**
  - Graph representation
Graph Structured Program Evolution with Automatically Defined Nodes

**GRAph structured Program Evolution (GRAPE)**

**Phenotype (Structure of GRAPE)**

![Diagram of a graph representing the structure of GRAPE with nodes and connections]

**Genotype (integer string)**

<table>
<thead>
<tr>
<th>No. 0</th>
<th>No. 1</th>
<th>No. 2</th>
<th>No. 3</th>
<th>No. 4</th>
<th>No. 5</th>
<th>No. 6</th>
<th>No. 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>....</td>
<td>....</td>
<td>....</td>
<td>....</td>
<td>....</td>
<td>....</td>
<td>....</td>
<td>....</td>
</tr>
</tbody>
</table>

Examples of nodes

Examples of node

- Processing part

- Branching part

- It can handle several data types (e.g. integer, Boolean, list, image)
- Special nodes: start node, output node
Genetic operator

- **Uniform crossover**

  ```
  Parent 1: 0 1 2 3 4 5 6 7 8 9 ···
  Parent 2: a b c d e f g h i j ···
  Offspring 1: 0 b c 3 4 f 6 h 8 9 ···
  Offspring 2: a 1 2 d e 5 g 7 i j ···
  ```

- **Mutation**

  ```
  0 1 2 3 4 5 6 7 8 9 ···
  a 1 c 3 4 5 6 7 i j 9 ···
  ```
Features of GRAPE

1. Arbitrary directed graph structures
   → It can represent branches and loops

2. Handle multiple data types using the “data set”
   → e.g.) integer types, Boolean types, list types, image types

3. Genotype of integer string
   → It can use a usual Genetic Algorithm (GA)
   → (It does not need prepare special genetic operators)
Evolution of functions or modules

- **Genetic Programming (GP)**
  - Automatically Defined Function (ADF)
  - Module Acquisition (MA)
  - Automatically Defined Macro (ADM)

- **Cartesian Genetic Programming (CGP)**
  - Embedded Cartesian Genetic Programming (ECGP)

- **Genetic Network Programming (GNP)**
  - Automatically Generated Macro Nodes (AGM)
The main aims of this work

Motivation

- We introduce the concept of ADFs in Graph Structured Program Evolution (GRAPE)

Present work

- We propose a method called automatically defined nodes (ADN)
- The proposed GRAPE program includes a main program and several subprograms
Whole program consists of a main program and several sub-programs.

The main program considers an ADN as one of the nodes.

The same data set is processed or used at the nodes in the main program and ADN (the data set is global).
Structure of GRAPE with ADN

- The presence of one ADN in another is restricted to reduce non-terminating programs.

- The connections in the main program are feedforward structure, 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 $n$th node can only connect to $(n+1)$ and higher nodes. (These connection restrictions are performed at the genotype level)
Graph Structured Program Evolution with Automatically Defined Nodes

Experiments

- **Experiment 1**
  - Factorial \( a! \)

- **Experiment 2**
  - Exponentiation \( a^b \)

- **Experiment 3**
  - List sorting e.g. \((2 4 3 1) \rightarrow (1 2 3 4)\)
Fitness function

- Factorial, Exponentiation

\[
\text{fitness} = 1.0 - \sum_{i=1}^{n} \frac{|Correct_i - Estimate_i|}{|Correct_i| + |Correct_i - Estimate_i|}
\]

- List sorting

\[
\text{fitness} = 1.0 - \frac{\sum_{j=0}^{n} (1 - \frac{1}{2^{d_j}})}{n}
\]

- If the fitness is 1.0

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

$S_{\text{exe}}$: execution step
Experimental settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of evaluations</td>
<td>50000000</td>
</tr>
<tr>
<td>Population size</td>
<td>500</td>
</tr>
<tr>
<td>Children size (for MGG)</td>
<td>50</td>
</tr>
<tr>
<td>Uniform crossover rate $P_c$</td>
<td>0.1</td>
</tr>
<tr>
<td>Crossover rate</td>
<td>0.7</td>
</tr>
<tr>
<td>Mutation rate $P_m$</td>
<td>0.02</td>
</tr>
<tr>
<td>Maximum number of nodes</td>
<td>30 (main)</td>
</tr>
<tr>
<td></td>
<td>30 (ADN)</td>
</tr>
<tr>
<td>Number of ADNs</td>
<td>5</td>
</tr>
<tr>
<td>Execution step limits</td>
<td>500 (factorial)</td>
</tr>
<tr>
<td></td>
<td>1000 (exponentiation)</td>
</tr>
<tr>
<td></td>
<td>3000 (list sorting)</td>
</tr>
</tbody>
</table>

- There are a maximum of 30 nodes in the main program and ADN.
- The number of ADN is five.
- The maximum number of nodes in conventional GRAPE are set to 30 and 180, respectively.
Graph Structured Program Evolution with Automatically Defined Nodes

Node functions

<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 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 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>

- We prepare 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.
When an individual whose fitness value is equal or greater than 1.0 is found, the run is counted as a successful run.

GRAPE with ADN shows a more stable number of successful runs compared with conventional GRAPE.
The transition of the fitness value doesn’t have a big difference.
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 number of successful runs

Test data

<table>
<thead>
<tr>
<th></th>
<th>GRAPE with ADN</th>
<th>GRAPE (30)</th>
<th>GRAPE (180)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factorial</td>
<td>60</td>
<td>69</td>
<td>52</td>
</tr>
<tr>
<td>Exponentiation</td>
<td><strong>39</strong></td>
<td>34</td>
<td>32</td>
</tr>
<tr>
<td>List sorting</td>
<td>20</td>
<td><strong>33</strong></td>
<td>22</td>
</tr>
</tbody>
</table>

- We apply the elitist individual generated by GRAPE to the test set for each run.
- 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.
Automatic generation of recursive programs

- Programs with recursive structures are evolved by GRAPE with ADN.

- To evolved 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 structure.
Obtained program for factorial

This program returns factorial for all inputs.
Graph Structured Program Evolution with Automatically Defined Nodes

Obtained program for exponentiation

- This program returns exponentiation for all inputs.

- These programs call AND in ADN (recursive structure), demonstrating that GRAPE with ADN can evolve recursive programs, although they are simple and trivial recursive programs.

Data used:

Main

- start

ADN

- start
- data[13] > data[8]?

- end

Output data[14]
Summary

- We introduced the concept of ADFs to GRAPE.
- We applied the proposed method to evolve programs, and confirmed that ideal programs were obtained.
- Moreover, we confirmed that the modified proposed method obtained recursive programs.
Thank You!!