Genetic Programming Using a Turing-Complete Representation: Recurrent Network Consisting of Trees

Taro YABUKI* Hitoshi IBA†

2003

Abstract

In this chapter, a new representation scheme for Genetic Programming (GP) is proposed. We need a Turing-complete representation for a general method of generating programs automatically, i.e. the representation must be able to express any algorithms. Our representation is a recurrent network consisting of trees (RTN), which is proved to be Turing-complete. In addition, it is applied to the tasks of generating language classifiers and a bit reverser. As a result, RTN is shown to be usable in evolutionary computing.

1 Introduction

Genetic Programming (GP) is a technique for generating programs or functions automatically[6]. GP is a type of evolutionary computing that aims to solve problems through the repetition of modification and selection of prospective solution candidates. Various representations for programs or functions have been used. The most popular GP (standard GP) uses a single parse tree (S-expression) as a program representation. S-expressions (e.g. (plus (times x 5) (minus y x))) are made by combining non-terminals (e.g. plus, minus, times, and divide) and terminals (e.g. x, y, and integers).

We have proposed a substitution for the single S-expression. It is a recurrent network consisting of trees (RTN). In the following paragraphs, we will explain why a new representation is required for GP.

When using GP, we must set various configurations. For example: the representation of individuals, the components of representation, the way to use the representation and evolutionary operators, etc. The strategy to set these configurations depends on the objective tasks.

The objective tasks of GP can be classified as follows:

1. Programs that can be easily written by humans.
2. Programs that are simpler or more efficient than those written by humans.
3. Programs that solve unsolved problems.

If the task belongs to either the first or second class, then the configurations can be decided with reference to the previously known solution. However, if the task belongs to the third class, then it is not easy to set the configurations. Choosing a smaller non-terminal set and restricting the expressiveness of individuals may make the search easier. However, should the search fail, it will be impossible to find out whether it is attributable to GP or the configurations. For example, suppose we try to generate a classifier for the language $\{ww|w \in \{0,1\}^*\}$. If we use a representation whose repertoire is the same as one of the pushdown automaton, then we will never succeed.

One conceivable approach is to start with simple settings and gradually introduce complex one. One method proposed composes the S-expression of basic arithmetic functions in the early stage, and then introduces a loop or recursion as the search progresses[6]. A strategy like this is adoptable for a task belonging to the first or second classes mentioned above, but not for the unsolved problems, because it is not clear how a loop or recursion affects the expressiveness of individuals. For unsolved problems, a strategy that we can confirm the increase of expressiveness is desirable. In the ideal case, the expressiveness of an individual finally becomes Turing-complete. In other words, an individual will be able to express any algorithms.

Additionally, there are other requirements for the new representation for GP.

*The University of Tokyo, Japan
†The University of Tokyo, Japan

Simplicity For example, representations which use too many terminals are not easy to use.

Extensibility If it is known that some functions, e.g. trigonometric functions, are essential for the problem, then it must be easy to add these functions.

Similarity to standard GP A natural extension of the standard GP is desirable, because of its widespread use.

The RTN proposed in this chapter meets those requirements.

This chapter is organized as follows. In section 2, the expressiveness of the standard GP is summarized. In section 3, RTN is proposed with an example. In section 4, the proof that RTN is Turing-complete is given. In section 5, evolutionary operators for RTN are reviewed. In section 6, RTN is applied to evolve a language classifier. In section 7, RTN is applied to evolve bit reverser. We give some discussion in section 8 and conclude in section 9.

2 Expressiveness of the Standard GP

2.1 Theory of Computability

According to the theory of computability[10], the ability of a computer can be classified as follows:

Table Reply to the input by querying the table. It cannot accept regular expressions, etc.

Finite states automaton (FSA) Machine with finite states. It cannot decide whether parentheses are matched or not.

Pushdown automaton (PDA) FSA with an infinite stack. It cannot accept the language like $\{ww|w \in \{0,1\}^*\}$.

Turing machine (TM) FSA with an infinite tape. The tape (or head) can move both forward and backward. TM is the model of today's digital computers. It cannot decide whether an arbitrary TM halts or not.

Although, TM has the largest repertoire, there are tasks that it cannot handle. However, it is thought that all machinery procedures can be simulated by a TM (Church-Turing Thesis). In addition, we should distinguish exact methods from approximate methods, using concepts from the theory of computation. In this chapter, we restrict our interest to exact methods.

2.2 Expressiveness of the Standard GP Individual

The expressiveness of a GP individual depends on both its components and the way the individual is used. If we compose an S-expression of basic arithmetic functions and use it as a program, then the repertoire of such a program is limited. For example, the S-expression composed of four arithmetic functions, variables and constants cannot even include the repertoire of a table.

It is obvious that if we can repeat the evaluation of S-expression with a finite memory and non-terminals to access it, then the expressiveness of the GP individual is equivalent to the one of FSA. Similarly, if there are non-terminals to access an infinite stack, then the expressiveness is equivalent to the one of PDA. If there are non-terminals to access an infinitely indexed memory and we can repeat the evaluation until the data stored in the memory meets a halting condition, then the expressiveness is equivalent to the one of a TM, i.e. Turing-complete[11].

In short, the expressiveness of a GP individual can be extended as shown in Table 1. However, it is not easy to shift naturally from the PDA phase (the non-terminal set includes PUSH and POP) to TM phase (the non-terminal set includes READ and WRITE). On the other hand, RTN described in the next section can shift from table to TM naturally.

3 Recurrent Network Consisting of Trees

An outline of RTN is given as follows. An example of RTN is shown in Figure 1 (P in Figure 1 is a function which returns a remainder of its argument divided by 2). Each node has both a value and a pure function expressed by an S-expression. (In this chapter, S-expression is written as a normal expression for the sake of readability.) The functions of the RTN are as follows:

Table 1: Expressiveness of the standard GP.

Class	Repetition of evaluation	Memory
Table	No	No
FSA	Yes	Finite
PDA	Yes	Infinite stack
TM	Yes	Infinite indexed memory

$$\begin{aligned} &\#1: \ (c - P(c))/2, \\ &\#2: \ P(a)d. \end{aligned} \tag{1}$$

The function has at most four parameters (i.e. a, b, c, and d). These parameters are bound to the value of nodes. In this case, the parameter a of the node #2 is bound to value of the node #1, because the a of #2 and the value of #1 are connected in Figure 1.

The value (v) at the next time step is bound to the output of the function. Thus, RTN of Figure 1 can be rewritten as follows (the asterisk means the next time step):

$$\begin{aligned} v_1^* &= (v_1 - P(v_1))/2, \\ v_2^* &= P(v_1)v_2. \end{aligned} \tag{2}$$

Note that we do not need the terminals like v_1 or v_2. Even if the network becomes very large, we need at most only four terminals as the variables of S-expressions.

The programs are executed according to the discrete time steps. Define the function and the value of the i-th node at time t as f_n and $v(n,t)$, respectively, the number of parameters as k, and the index of linked node as $l_{n,i}$. The value at $t+1$ will be

$$v(n, t+1) = f_n(v(l_{n,1}, t), \cdots, v(l_{n,k}, t)). \tag{3}$$

Suppose the value of the first node (#1) is bound to the input data and the value of second node is 1 at $t = 0$. For example, when the input data is a binary digit 1011, the transition of RTN is given in Table 2. When the value of #1 becomes 0, the value of #2 is 0 if and only if the inputted binary digit contains 0.

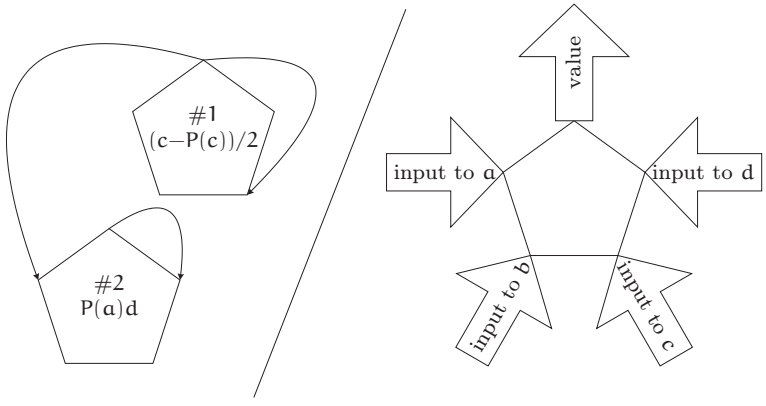

Figure 1: Example of RTN (left) and the relation between variables and link (right).

3.1 Description of an RTN

RTN is defined by the following six factors:

1. List of S-expression

Table 2: Transition of RTN of Figure 1.

time step	0	1	2	3	4
#1 value	1011	101	10	1	0
#2 value	1	1	1	0	0

2. Network topology vector
3. Initial state
4. The way to input data
5. Halting condition
6. The way to output a result

In the case of the RTN shown in Figure 1, the list of an S-expression is given as

$$\{(c - P(c))/2,\ P(a)d\}. \tag{4}$$

The network is described as follows. Link of #1 can be expressed by $\{*, *, 1, *\}$, because the third parameter i.e. c is bound to the value of #1. The behavior of this RTN does not change even if any integer enters the position with a character '$*$', because there are no other parameters to be bound except for c. Similarly, the link of #2 is $\{1, *, *, 2\}$. Network topology vector is the concatenation of these links i.e. $\{*, *, 1, *, 1, *, *, 2\}$. Practically, integers between 1 and the network size are put in the position of '$*$' and the network topology vector becomes $\{1, 2, 1, 2, 1, 1, 1, 2\}$.

Additionally, the initial value of all nodes is one. The value of #1 is bound to the input data. The halting condition is that the value of #1 is 0. The result is the value of #2 at the time when the RTN halts. Consequently, the RTN shown in Figure 1 is defined completely.

3.2 Differences between RTN and Standard GP

RTN is a natural extension of the standard GP. Standard GP uses a single S-expression as an individual representation. On the other hand, RTN uses plural S-expressions. Special non-terminals are not needed, but four arithmetic functions and a function P are needed to make RTN Turing-complete as discussed in the next section. The number of variables is at least four. Variables of the standard GP are bound to the input data. On the other hand, the values of nodes are bound to those.

In this chapter, we deal only with one-input programs. One-input programs can simulate multi-input programs in principal, because one-tape Turing machines can simulate arbitrary multi-tape Turing machine. In such a case, a method of coding multi-tape in one-tape is needed. Aside from this, if there are more than four nodes, then RTN can simulate a multi-tape Turing machine directly.

4 Turing-Completeness of RTN

RTN can simulate an arbitrary TM if there are four nodes and the function of each node is composed of arithmetic functions, a function P, four variables and constants. In short, RTN can represent any algorithms. Firstly, we prove that the extended RTN is Turing-complete. Secondly, we prove that the standard RTN is Turing-complete.

4.1 Extended RTN—List as a Node Value

The proof is straightforward. RTN to simulate TM is constructed. The characters for the tape of the TM used here is 0, 1, and blank. It is well known that only two characters e.g. 0 and 1 are essential. However we use three characters for the sake of understandability.

Suppose that the TM is in the following state, and the head is moving rightwards, i.e. $D(q_i, s_j) = 1$ (if the head is moving leftwards, then $D(q_i, s_j) = 0$):

$$\underbrace{\cdots\ b_3\ b_2\ b_1\ b_0}_{m}\ \underset{q_i}{\underline{s_j}}\ \underbrace{c_0\ c_1\ c_2\ c_3\ \cdots}_{n},$$

where s_j represents the character of the tape that is currently being read, and q_i represents the state of the head.

The value of a node mentioned above is a number. Here RTN is extended to deal with a list of numbers as a node value, and the tapes m and n are expressed by lists.

After the tape has been rewritten to $s_{ij} = R(q, s)$ and the head has moved rightwards, the TM is as follows:

$$\underbrace{\cdots\ b_3\ b_2\ b_1\ b_0\ s_{ij}}_{m^*}\ \underline{c_0}_{q_{ij}}\ \underbrace{c_1\ c_2\ c_3\ \cdots}_{n^*}.$$

This step can be described as a transition of q, s, m, and n. In this example, the new values q^*, s^*, m^*, and n^* are obviously:

$$\begin{aligned} q^* &= Q(q, s), \\ s^* &= \mathrm{car}(n), \\ m^* &= \mathrm{cons}(R(q, s), m), \\ n^* &= \mathrm{cdr}(n), \end{aligned} \tag{5}$$

where $Q(q, s)$ is the subsequent state, "car" is a function to return the first element of the list given as its argument, "cdr" is a function to return a list containing all but the first element, and "cons" is a function to take an expression and a list and return a new list whose first element is the expression and whose remaining elements are those of the old list. In addition, car returns blank if its argument is an empty list, and cdr returns blank if the number of elements of its argument is either 0 or 1. We use "nil" of Lisp instead of blank hereafter, because it satisfies above conditions. (We sometimes use "{}" instead of "nil" in this chapter.)

It is easy to check that the transition of TM can be expressed in ordinary cases as follows:

$$\begin{aligned} q^* &= Q(q, s), \\ s^* &= \mathrm{if}(D(q, s), 0, \mathrm{car}(m), \mathrm{car}(n)), \\ m^* &= \mathrm{if}(D(q, s), 0, \mathrm{cons}(R(q, s), m)), \mathrm{cdr}(m)), \\ n^* &= \mathrm{if}(D(q, s), 0, \mathrm{cdr}(n), \mathrm{cons}(R(q, s), n)), \end{aligned} \tag{6}$$

where the non-terminal "if" is defined as:

$$\mathrm{if}(a, b, c, d) := \text{if } a = b \text{ then } c \text{ else } d. \tag{7}$$

It is clear that the functions defined as tables i.e. $Q(q, s)$, $R(q, s)$, and $D(q, s)$ can be composed of if, basic arithmetic functions, and constant. After all, the essential non-terminal set is basic arithmetic functions, if, car, cdr, and cons. During the evolutionary computation, S-expression which breaks syntax of above non-terminal set may be generated. Thus, it is necessary to devise the definition of non-terminals to be applicable to both numbers and lists. When syntax is broken, all functions return its first argument, for example, $\mathrm{plus}(4, \{1, 0, 1\}) = 4$ and $\mathrm{car}(12) = 12$.

4.2 Simulating TM by RTN

TM can be simulated by the RTN shown in the Figure 2. The relationship between TM and RTN is shown in Table 3. Note that a, b, c, and d are local variables of the node. In short, the variable a of #1 and that of #2 have no relation. (In the case of this example, the value of #1 is inputted to both the variable a of #1 and that of #2 for the sake of readability.)

Table 3: The relationship between TM and RTN.

	Meaning of value	Expression
#1	State of TM	$Q(a, b)$
#2	Letter at head	$\mathrm{if}(D(a, b), 0, \mathrm{car}(c), \mathrm{car}(d))$
#3	Left tape	$\mathrm{if}(D(a, b), 0, \mathrm{cons}(R(a, b), c)), \mathrm{cdr}(c))$
#4	Right tape	$\mathrm{if}(D(a, b), 0, \mathrm{cdr}(d), \mathrm{cons}(R(a, b), c))$

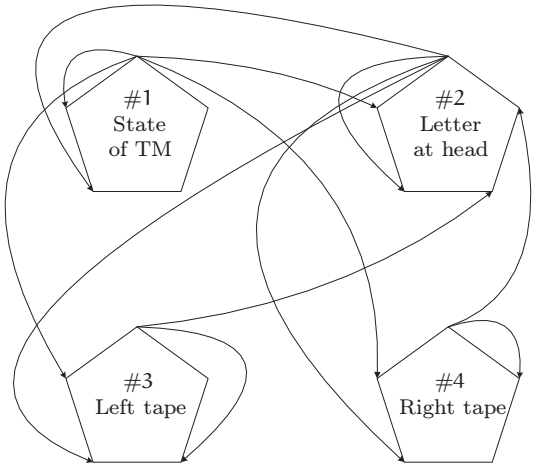

Figure 2: RTN to simulate TM.

4.3 Standard RTN

In the case of the extended RTN, the node must be able to have a list as its value. However, this condition can be removed. In other words, the standard RTN is Turing-complete. The proof is straightforward. Firstly, the TM is arithmetized. Secondly, RTN to simulate the arithmetized TM is constructed.

The tape of an arithmetized Turing machine is not a string of characters but an integer[7]. Arithmetization makes it possible to express the movements of a TM in the form of simple arithmetic, rather than the symbol operations.

Again, suppose that the TM is in the following state, and the head is moving rightwards, i.e. $D(q_i, s_j) = 1$:

$$\underbrace{\cdots\ b_3\ b_2\ b_1\ b_0}_{m}\ \underset{q_i}{\underline{s_j}}\ \underbrace{c_0\ c_1\ c_2\ c_3\ \cdots}_{n},$$

The symbols s, m, and n are redefined as follows:

$$s = s_j,\ m = \sum_{i=0}^{\infty} b_i 2^i,\ \text{and}\ n = \sum_{i=0}^{\infty} c_i 2^i. \tag{8}$$

As a result, the state of the tape can be expressed by three numbers s, m, and n. (The sum of the above is actually finite, because the number of '1' in the tape is finite.) Furthermore, by expressing the state of the TM q_i by integer q, the complete state of the TM can ultimately be expressed by four numbers q, s, m, and n.

After the tape has been rewritten to $s_{ij} = R(q, s)$ and the head has moved rightwards, the TM is as follows:

$$\underbrace{\cdots\ b_3\ b_2\ b_1\ b_0\ s_{ij}}_{m^*}\ \underset{q_{ij}}{\underline{c_0}}\ \underbrace{c_1\ c_2\ c_3\ \cdots}_{n^*}.$$

This step can be described as a transition of q, s, m, and n. In this example, the new numbers q^*, s^*, m^*, and n^* are obviously:

$$\begin{aligned} q^* &= Q(q, s), \\ s^* &= P(n), \\ m^* &= R(q, s) + 2m, \\ n^* &= H(n), \end{aligned} \tag{9}$$

where H and P are the quotient and the remainder divided by 2, respectively.

It is easy to check that the transition of TM can be expressed in ordinary cases as follows:

$$\begin{aligned} q^* &= Q(q, s), \\ s^* &= P(m)(1 - D(q, s)) + P(n)D(q, s), \\ m^* &= (R(q, s) + 2m)D(q, s) + H(m)(1 - D(q, s)), \\ n^* &= (R(q, s) + 2n)(1 - D(q, s)) + H(n)D(q, s). \end{aligned} \tag{10}$$

By using these equations instead of those shown in Table 3, the standard RTN can simulate any TM.

The final task is to check that the node of RTN can express $Q(q,s)$, $R(q,s)$, and $D(q,s)$ defined as a table. Assume that the number of the internal state is k. Function $f(q,s)$ that returns $Q(q,s)$ can be written in the form of one equation like:

$$f(q,s) = (1-s)\sum_{i=0}^{k}\left(A_{i,0}\prod_{j=0,j\neq i}^{k}(q-j)\right) + s\sum_{i=0}^{k}\left(A_{i,1}\prod_{j=0,j\neq i}^{k}(q-j)\right), \tag{11}$$

where A is a rational number like,

$$A_{a,b} = \frac{Q(a,b)}{\prod_{i=0,i\neq a}^{k}(a-i)}. \tag{12}$$

(Note that s will be 0 or 1, and q will be an integer between 0 and k.)

For example, in the case of $k=2$, the function will be:

$$\begin{aligned}(1-s)\ ((q-2)\ (q-1)\ Q(0,0)/2 - (q-2)\ q\,Q(1,0) + (q-1)\ q\,Q(2,0)/2)\\ +s\,((q-2)\ (q-1)\ Q(0,1)/2 - (q-2)\ q\,Q(1,1) + (q-1)\ q\,Q(2,1)/2)\end{aligned} \tag{13}$$

As a result, an RTN can simulate an arbitrary TM if there are four nodes and each node consists of arithmetic functions, a function P, four variables and constant (only 1 is necessary, other integers can be made by arithmetic operations on 1). We do not assert that this is the minimum non-terminal set.

In addition, if we do not worry about the size of the non-terminal set, then the above discussion can be omitted by introducing non-terminal "if". In which case, the transition of the TM will be represented as follows:

$$\begin{aligned}q^* &= Q(q,s),\\ s^* &= \mathrm{if}(D(q,s),0,P(m),P(n)),\\ m^* &= \mathrm{if}(D(q,s),0,R(q,s)+2m,H(m)),\\ n^* &= \mathrm{if}(D(q,s),0,H(n),R(q,s)+2n).\end{aligned} \tag{14}$$

It should be obvious that the expression of $Q(q,s)$, $R(q,s)$, and $D(q,s)$ becomes simpler.

Similarly, FSA and PDA can be simulated by RTN (Table 4).

Table 4: Expressiveness of RTN.

	Num. of nodes	Num. of variables	Non-terminals
Table	1	1	Arithmetic functions
FSA	2	2	Arithmetic functions and P
PDA	3	3	Arithmetic functions and P
TM	4	4	Arithmetic functions and P

4.4 Halting Condition

The halting condition of RTN can be set in neither the node description nor the network topology. It corresponds to the fact that there are no halting conditions in the transition table of a TM.

The halting condition of a TM can be defined as some particular state. In the case of the TM mentioned above, assign a state for halt and stop the transition of the TM when q becomes that particular state. RTN adopts the same method. Particular node and its value for halt are assigned in advance. Transition of RTN is stopped when the node has the particular value.

It is well known that there are no general methods of deciding whether TM will halt or not in advance[10]. Consequently, a fundamental difficulty exists in evolutionary computing using Turing-complete representation. Perfect evaluation of individual in finite time is impossible. However, this does not matter in practice, because the solution we want to generate must be executed in a reasonable time.

5 Evolutionary Operators

As mentioned above, the data structure of RTN of Figure 1 is like:

$$\{\text{list of S-expressions}, \text{network topology vector}\} = \{\{(c - P(c))/2, P(a)d\}, \{1, 2, 1, 2, 1, 1, 1, 2\}\}. \quad (15)$$

Various evolutionary operators for this data structure can be introduced:

1. Replacement of S-expression by randomly generated S-expression
2. Exchange of S-expressions between two RTNs
3. Mutation on network topology vector
4. Exchange of links of nodes between two RTNs
5. Crossover of two network topology vectors
6. Increment (or duplication) and decrement of node.

1 and 3 can be used together. Similarly, 2 and 4 can be used together. Crossover of two network topology vector is the same as the crossover used in the ordinary Genetic Algorithm. However, the node number of parents i.e. the size of network topology vector must be the same. The number of nodes is changed by the node increment or decrement operators. Thus, these operators must be used to keep the node number of all RTNs in the population the same.

The expressiveness of RTN (i.e. Turing-complete) does not change even if the node number is greater than four. However, the redundancy from the tree increment or duplication may yield good effects on the search.

An operator that exchanges the clusters in the network between two RTNs could be introduced. However, a randomly generated RTN i.e. each node has four randomly connected links, is hardly clustered. In other words, there are few isolated groups of nodes in RTN. Therefore, cluster exchange operator is not introduced. (Considering the free variable in the S-expressions and nodes that do not have a causal relationship, a network may be clustered. However, it's expensive to check these clusters.)

6 Example 1: Language Classifier

For example, we use RTN to generate a language classifier. Target languages are Tomita languages (Table 5). The number of training data used in [4] is adopted.

Table 5: Tomita languages[4].

Language	Definition			
	Number of strings (Length ≤ 10)		Number of training data	
	Positive	Negative	Positive	Negative
L1	$1*$			
	11	2,036	9	8
L2	$(10)*$			
	6	2,041	5	10
L3	No odd-length 0-string after an odd-length 1-string			
	716	1,331	11	11
L4	Any string not containing 000 as a substring			
	1,103	944	10	7

6.1 Objective of Search

The objective of search is to generate RTN that behaves as follows:

1. Set the value of all nodes to nil.

2. Input strings letter by letter to the particular node according to time step.

3. If no letter is left, then input nil to the node.

4. RTN decides whether the string belongs to the language or not within particular time since input ends. In other words, value of the output node becomes 1 if the string belongs to the language and 0 if it does not.

Among the six factors to define RTN, the list of S-expressions and network topology are searched. Instead of inputting letter by letter, the entire string is inputted all together. For that purpose, S-expressions of #1 and #2 are frozen to be $\mathrm{cdr}(a)$ and $\mathrm{car}(a)$, respectively. Additionally, links of #1 and #2 are both frozen to be $\{1, *, *, *\}$. In short, transition rule of #1 and #2 are $\mathrm{cdr}(v_1)$ and $\mathrm{car}(v_1)$, respectively. As a result, the value of #1 is the string at beginning and shortened letter by letter. The value of #2 is the first element of #1 at previous time step. It would be possible to let the node #1 and #2 evolve. However, we did not try this, because it was expected that the search became harder. In addition, the configuration used here is equivalent to inputting the data to the node #2 letter by letter, thus this configuration does not mean a hint for the search.

Among the remaining four factors, initial state, the way to input data and the way to output a result are involved in the specifications of RTN, i.e. these are not searched. The other factor, the halting conditions, i.e. when and which node gives an answer are determined during the training process. Another halting condition like "when the value of #x is y, halt the RTN" is adoptable, but not adopted in our work.

6.2 Halting Condition and Fitness Function

The halting condition is determined by finding a node which decides whether the string belongs to the language or not with the highest probability. The concrete procedure is as follows:

1. Input a string to #1.

2. Run RTN until the value of #1 becomes nil, and reset the time to zero.

3. Run RTN until the time becomes the network size. Consequently, table $v(k, n, t)$ is gotten, where k is the index of training data, n is the index of node and t is the time step.

The network size, i.e. the number of nodes, is the maximum time for one node to affect another node. Thus, time limit in above phase 3 means that RTN must output a result before the ending of the string affects the entire network.

Next, check whether a node has an answer or not. Define a function g as follows:

$$g(k, n, t) = \begin{cases} 1 & (\text{positive data} \wedge v(k, n, t) = 1) \vee (\text{negative data} \wedge v(k, n, t) = 0) \\ 0 & \text{otherwise.} \end{cases} \tag{16}$$

Find out when and which node has the answer with the highest probability. In other words, determine n and t such that $\Sigma_k\, g(k, n, t)$ is maximized.

Time series of RTN run must be like the one shown in Table 6. After running the RTN until the value of #1 becomes nil, the time is reset to zero (under the horizontal line). The time taken for this process depends on the string (in this case, 3). At $t = i$, the value of $\#n$ has an answer. The constants i and n must be independent from the string.

Fitness value of an individual is defined as the success probability of the process mentioned above. In short:

$$\frac{\text{the maximum value of } \Sigma_k\, g}{\text{the number of training data}}. \tag{17}$$

Training data are strings whose length is shorter than 11. These are generated randomly at every generation. (It is possible to use all $2,048$ strings whose length is shorter than 11, because we use other longer strings for validation. However, it is very time-consuming to use $2,048$ strings for each fitness evaluation. Thus, we use a small number of strings for fitness evaluation.)

Table 6: Transition of solution.

t	#1	#2	···	#n	···
0	{1, 0, 1, 1}	{}		{}	
1	{0, 1, 1}	1			
2	{1, 1}	0			
3	{1}	1			
0	{}	1			
1	{}	{}			
⋮					
i	{}	{}		1 or 0	

6.3 Setting of Evolutionary Computing

Setting of evolutionary computing is shown in Table 7. In addition, each evolutionary operator makes one-third of the offspring. The non-terminal "divide" returns its first argument directly if the second argument is 0. Similarly, "p" returns its argument directly if it is not an integer.

While an RTN is running, if the value of any node is either larger than 10^{10} or smaller than 10^{-10}, then it is forced to be 0. This corresponds to the fact that it is impossible to deal with the infinite tape of a TM in practice.

We do not assert this setting is optimal. We set it by trial and error. For example, we could not evolve the solution of L2 and L3 when the population size is 1,000. It was confirmed that the search performance is dependent on the number of nodes. However, the best number is not known.

Table 7: Setting of evolutionary computing.

Population size	1,000 for L1, L4 and bit reverser in section 7
	5,000 for L2, L3
Number of nodes	20
Evolutionary operators	Replacement of S-expression
	Mutation on the network topology vector
	Crossover of two network topology vectors
Selection method	Tournament (The tournament size is 5.)
Terminals	a, b, c, d, nil, integer from 0 to 10
Non-terminals	plus, minus, times, divide, p, if, car, cdr, cons

6.4 Results

Evolutionary computing was carried out using strings shorter than 11 as training data. Fitness scores of generated RTNs are all 1, i.e. generated RTN classifies the training data perfectly.

Only nodes with a causal relationship with the node that outputs an answer are shown. Note that terminals like v_i are not needed. These are used only for the sake of readability. Only four terminals are needed for RTN.

6.4.1 L1

RTN for L1 was generated at fifth generation. Its ninth node outputs an answer after four steps since the value of #1 becomes nil.

$$
\begin{aligned}
\nu_1^* =& \mathrm{cdr}(\nu_1)\\
\nu_2^* =& \mathrm{car}(\nu_1)\\
\nu_9^* =& \mathrm{times}(\mathrm{if}(\nu_{19}, \mathrm{divide}(\nu_{19}, \mathrm{minus}(\mathrm{divide}(\nu_{19}, \mathrm{cdr}(\mathrm{if}(\nu_{19}, \nu_{15}, \mathrm{times}(\nu_{20}, \nu_{15}), \mathrm{cdr}(\nu_{19})))), \nu_{19})),\\
& 1, \nu_{20}), \nu_{15})\\
\nu_{15}^* =& \nu_{17}\\
\nu_{17}^* =& \mathrm{times}(\nu_{19}, \nu_{17})\\
\nu_{19}^* =& \nu_2\\
\nu_{20}^* =& \nu_{19}
\end{aligned}
\tag{18}
$$

6.4.2 L2

RTN for L2 was generated at 14th generation. Its 18th node outputs an answer after two steps since the value of #1 becomes nil.

$$
\begin{aligned}
\nu_1^* =& \mathrm{cdr}(\nu_1)\\
\nu_2^* =& \mathrm{car}(\nu_1)\\
\nu_{10}^* =& \mathrm{cdr}(\mathrm{p}(\nu_{16}))\\
\nu_{16}^* =& \mathrm{minus}(\mathrm{divide}(6, \nu_1), \mathrm{times}(\mathrm{car}(\mathrm{p}(\mathrm{times}(\nu_2, 3))), \nu_{10}))\\
\nu_{18}^* =& \mathrm{times}(\nu_{20}, \nu_{10})\\
\nu_{20}^* =& \mathrm{cdr}(\mathrm{p}(\mathrm{if}(\mathrm{cons}(\mathrm{car}(\nu_2), \nu_2), \mathrm{cons}(\mathrm{p}(\mathrm{if}(\nu_{18}, \nu_2, \mathrm{cons}(\mathrm{cons}(4, \mathrm{car}(\nu_2)), \mathrm{divide}(\nu_2, \nu_2)),\\
& \mathrm{minus}(\mathrm{car}(\mathrm{minus}(\nu_2, \nu_2)), \mathrm{car}(\mathrm{plus}(\mathrm{divide}(0, 3), \nu_2))))), \nu_{18}), \mathrm{plus}(4, \nu_2), \mathrm{minus}(9, \nu_2))))
\end{aligned}
\tag{19}
$$

6.4.3 L3

RTN for L3 was generated at 249th generation. Its fourth node outputs an answer after four steps since the value of #1 becomes nil.

$$
\begin{aligned}
\nu_1^* =& \mathrm{cdr}(\nu_1)\\
\nu_2^* =& \mathrm{car}(\nu_1)\\
\nu_4^* =& \mathrm{times}(\nu_{17}, \nu_{10})\\
\nu_6^* =& \mathrm{if}(\nu_{17}, 1, \nu_{10}, \nu_4)\\
\nu_8^* =& \mathrm{times}(\mathrm{cdr}(\nu_2), \nu_{17})\\
\nu_{10}^* =& \nu_6\\
\nu_{17}^* =& \mathrm{p}(\mathrm{plus}(5, \mathrm{minus}(\nu_2, \nu_8)))
\end{aligned}
\tag{20}
$$

6.4.4 L4

RTN for L4 generated at 40th generation. Its 19th node outputs an answer after nine steps since the value of #1 becomes nil.

$$
\begin{aligned}
\nu_1^* =& \mathrm{cdr}(\nu_1)\\
\nu_2^* =& \mathrm{car}(\nu_1)\\
\nu_3^* =& \nu_{20}\\
\nu_5^* =& \nu_6\\
\nu_6^* =& \mathrm{cdr}(\mathrm{plus}(\nu_{19}, \nu_{19}))\\
\nu_9^* =& \mathrm{if}(1, \nu_{16}, \nu_{16}, \mathrm{if}(\nu_{10}, \mathrm{p}(\nu_{16}), 3, 8))\\
\nu_{10}^* =& \mathrm{if}(\mathrm{times}(1, \nu_{15}), \nu_5, \mathrm{cons}(\nu_3, \nu_3), \nu_{10})\\
\nu_{12}^* =& \nu_9\\
\nu_{13}^* =& \mathrm{cdr}(\nu_{19})\\
\nu_{15}^* =& \nu_{20}\\
\nu_{16}^* =& \nu_2\\
\nu_{19}^* =& \mathrm{p}(\nu_{12})\\
\nu_{20}^* =& \mathrm{if}(\nu_{12}, \nu_{20}, \nu_{20}, \nu_{13})
\end{aligned}
\tag{21}
$$

As mentioned above, the network of RTN is hardly clustered (Figure 3). However, by cutting down unnecessary variables, some links can be removed. As a result, network is clustered (Figure 4).

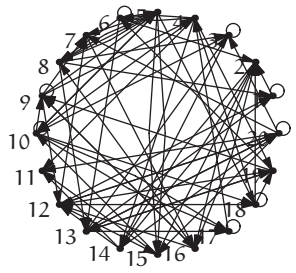

Figure 3: Generated RTN for L4.

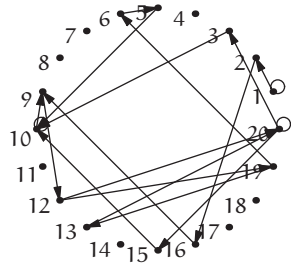

Figure 4: Essential part of RTN of Figure 3.

6.5 Validations

Evolutionary computing used strings whose length is smaller than 11. To validate the results, we tried to classify all strings whose length is smaller than 17 using generated RTNs. As a result, RTNs generated by the evolutionary computing could classify the strings perfectly.

7 Example 2: Bit Reverser

For a second example, we use RTN to generate bit reverser. Bit reverser is a program which receives a bit string and returns a reversed string. For example, if we input $\{1, 0, 0, 1, 1\}$, then bit reverser returns $\{1, 1, 0, 0, 1\}$. This problem is tackled by MIPs[1].

The halting condition and the fitness function are similar to those of the language classifier. When RTN halts, the particular node must output the reversed bit string. The fitness score is a ratio of the right responses to the number of training data. Training data are 10 strings whose lengths are less than 11. They are made randomly in each generation. The setting of evolutionary computing is presented in Table 7.

RTN for this problem was generated at fifth generation as follows:

$$\begin{aligned} \nu_1^* &= \mathrm{cdr}(\nu_1) \\ \nu_2^* &= \mathrm{cdr}(\nu_1) \\ \nu_{17}^* &= \mathrm{cons}(\nu_2, \mathrm{if}(\{\}, \mathrm{cdr}(\mathrm{divide}(\mathrm{minus}(4, \mathrm{times}(\nu_{17}, 1)), 5)), 1, \mathrm{minus}(\nu_{17}, \nu_{17}))). \end{aligned} \tag{22}$$

Its 17th node outputs the reversed string after a step since the value of #1 becomes nil. This RTN can reverse a string of any length, because we can confirm that the equation for #17 is simplified like:

$$\nu_{17}^* = \mathrm{cons}(\nu_2, \nu_{17}). \tag{23}$$

Note that both $\mathrm{times}(\nu_{17}, 1)$ and $\mathrm{minus}(\nu_{17}, \nu_{17})$ are ν_{17} when ν_{17} is a list. Accordingly, the transition of this RTN is as shown in Table 8.

Table 8: Transition of generated RTN.

t	#1	#2	#17
0	{1,0,0,1,1}	{}	{}
1	{0,0,1,1}	1	{}
2	{0,1,1}	0	{1}
3	{1,1}	0	{0,1}
4	{1}	1	{0,0,1}
0	{}	1	{1,0,0,1}
1	{}	{}	{1,1,0,0,1}

8 Discussion

8.1 Turing-Completeness and Graph Representation

Various representations for GP have been proposed so far. Representations related to our work are as follows.

In the case of the standard GP, an individual is represented as a single S-expression which consists of non-terminals (functions) and terminals (variables and constants). Variables are bound to the input data. Output of the program is the evaluated value of the S-expression. It is proved that the expressiveness of the individual of the standard GP becomes Turing-complete by adding functions to access an indexed memory and repeating the evaluation of S-expression[11].

There are other Turing-complete representations, for example, Cellular Automaton and recurrent artificial neural network, etc. One of the drawbacks of these representations is that it is not easy to introduce new functions. On the other hand, GP using S-expression can be introduced using new functions easily by adding new non-terminals.

There are various representations using a graph. For example, GP-automata[2], Parallel Distributed Genetic Programming (PDGP)[9], PADO[12] and Multiple Interacting Programs (MIPs)[1]. Linear-Graph GP also uses a graph, but its nodes are not S-expressions[5].

GP-automata is different from RTN in expressiveness. Each node of GP-automata corresponds to a state of finite states automaton (FSA). Thus, the expressiveness of GP-automata is the same as the one of FSA.

PDGP is different from RTN in the fact that each node of PDGP is not an S-expression but a non-terminal. As mentioned above, it is easy to show the Turing-completeness of RTN, because the nodes of RTN are S-expression. The expressiveness of PDGP is not clear.

PADO is different from RTN in the fact that PADO has an indexed memory, and there are non-terminals like READ and WRITE. Thus, the data flowing in the network are sometimes dealt not as a pure data, but as a memory index. Consequently, in introducing new non-terminals, we must consider the case in which they are used to calculate a memory index. On the other hand, RTN has nothing like an indexed memory. Additionally, non-terminal like "read the value of $\#n$" cannot be used. Thus, users of RTN need not consider this problem.

8.2 Multiple Interaction Programs

Multiple Interacting Programs (MIPs) is very similar to RTN. It is different from RTN in two points as described below. There are some drawbacks because of these differences.

Firstly, many terminals are needed for MIPs. For example, in case of MIPs, the program shown in Figure 1 is represented as follows:

$$\begin{aligned} v_1^* &= (v_1 - P(v_1))/2, \\ v_2^* &= P(v_1)v_2. \end{aligned} \tag{24}$$

On the other hand, it is represented as follows:

$$\begin{aligned} v_1^* &= (a - P(a))/2, \\ v_2^* &= P(a)b, \end{aligned} \tag{25}$$

where the value of #1 is connected to the variable a in #1 and #2. This difference is conspicuous when the network becomes large. In case of RTN, only four variables are needed.

Secondly, a policy to decide the network size, the link number of each node and non-terminal set, etc. is not given for MIPs. If proper non-terminal set is not given, then search may be difficult. For example, generation of bit reverser is reported in [1]. The non-terminal set used there includes functions like sin and cos. As a result, the generated program is complicated and limited to 5-bit. On the other hand, we confirmed experimentally that RTN can generate the bit reverser easily without any special non-terminals except what is essential for Turing-completeness.

8.3 Language Classifier

Generation of Tomita language classifier was tackled in [3]. However, their result was obtained by limiting the expressiveness. This task was tackled by $\Re$-STROGANOFF[4], which is an integrated system of the standard GP and recurrent neural networks. However, its highest score for Tomita language L4 was 91%. On the other hand, our score is 100%. This comparison will clearly show the effectiveness of our approach.

It is known that the expressiveness needed for Tomita language classifier is the same as the one of FSA. Thus, if we limit the expressiveness to that of FSA, the search may be easier. Nevertheless, the attempt to use Turing-complete representation is meaningful for two points:

1. This is in the early stage of the general method of generating programs automatically. As mentioned above, the representation used in program generations must be Turing-complete, because we can use representations with limited expressiveness only when we know the limitation to express the answer in advance.

2. The success of search using a Turing-complete representation means the success with fewer clues to the answer, because the limitation of the expressiveness of individuals results in giving helpful hints to search algorithms unconsciously. It is desirable the search succeeds in spite of human does not give many hints. Although it is not easy to show quantitatively, the larger expressiveness becomes, the fewer clues exist. If the non-terminal set is small, so much the better.

9 Conclusion and Future Works

We proposed a representation for GP. It was a recurrent network of trees (RTN). RTN was proved to be Turing-complete. As an example, RTN was applied to generate a classifier for Tomita languages. Generated classifier was better than that generated by using GP.

However, the tasks tackled here can be achieved with limited representations i.e. finite state automaton. In other words, Turing machine is not needed to solve the tasks. Thus, the task which is impossible for such a limited representation is one of the future works.

We expect the node of RTN will be a good unit to encapsulate, introduce and keep knowledge, i.e. a good building block for GP. The research about this topic including a yardstick for judging this matter is also one of our future works.

The network used in this chapter is a nearly random network, in other words, the probability that any two nodes are connected is constant if the differentiation between input and output is ignored. Other network topology, e.g. the scale-free network, could be considered.

There are still few cases in which programs have actually been evolved in a system using Turing-complete representations[8, 11]. Further study is required, including the question as to whether such programs can evolve within the framework of evolutionary computing.

References

[1] P.J. Angeline. Multiple interacting programs: A representation for evolving complex behaviors. *Cybernetics and Systems*, 29(8):779–806, 1998.

[2] D. Ashlock. GP-automata for dividing the dollar. In *Genetic Programming 1997: Proceedings of the Second Annual Conference*, pages 18–26, 1997.

[3] C.L. Giles et al. Learning and extracting finite state automata with second-order-recurrent neural networks. *Neural Computation*, 4:393–405, 1992.

[4] H. Iba et al. Temporal data processing using genetic programming. In *Proceedings of the Sixth International Conference ICGA-95*, pages 279–286, 1995.

[5] W. Kantschik and W. Banzhaf. Linear-graph GP — a new GP structure. In *Genetic Programming, Proceedings of the 5th European Conference, EuroGP 2002*, volume 2278 of *LNCS*, pages 83–92, 2002.

[6] J.R. Koza et al. *Genetic Programming III.* Morgan Kaufman, 1999.

[7] M.L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, Inc., 1967.

[8] P. Nordin and W. Banzhaf. Evolving turing-complete programs for a register machine with self-modifying code. In *Proceedings of the Sixth International Conference ICGA-95*, pages 318–325, 1995.

[9] R. Poli. Evolution of graph-like programs with parallel distributed genetic programming. In *Genetic Algorithms: Proceedings of the Seventh International Conference*, pages 346–353, 1997.

[10] M. Sipser. *Introduction to the Theory of Computation.* Thomson Learning, 1997.

[11] 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, 1994.

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