Algorithmically Transitive Network: A Self-organizing Data-Flow Network with Learning

Hideaki Suzuki[1], Hiroyuki Ohsaki[2], and Hidefumi Sawai[1]

[1] National Institute of Information and Communications Technology
588-2, Iwaoka, Iwaoka-cho, Nishi-ku, Kobe, 651-2492, Japan
{hsuzuki,sawai}@nict.go.jp

[2] Graduate School of Information Science and Technology, Osaka University
1-5 Yamadaoka, Suita, Osaka, 565-0871, Japan
oosaki@ist.osaka-u.ac.jp

Abstract. A novel non-von Neumann computational model named "Algorithmically Transitive Network" (ATN) is presented. The ATN is a data-flow network composed of operation nodes and data edges. The calculation is propelled with node firing and token creation on the edges. After it finishes, teaching values are given to the answer nodes and an energy function is evaluated, which causes backward propagation of differential coefficients with respect to token variables or node parameters. The network's topological alteration takes place based on these calculation/learning processes, and as a result, the algorithm of the network is refined. The basic scheme of the model is explained, and some experimental results on symbolic regression problems are presented.

Keywords: data-flow network, learning, neural network, back-propagation, artificial chemistry.

1 Introduction

For a long time, network and computation have been in a close relationship with each other in several disciplines of information sciences. Back in the 1970s to 1980s, the 'data-flow computer' (DFC) [18] was studied in many institutes in the world in the hope that parallel algorithms represented by the 'data-flow network' might remedy the 'bottleneck' problem which a von Neumann computer suffers from. Today, the DFC is neither a commercial-based computer nor a hot research topic, partly because the DFC's algorithm expressed as a network is difficult for a human to design or maintain. A network is a very natural way to represent a computational algorithm, but to utilize the data-flow network, we might have to incorporate an additional function to maintain programs.

Another famous research approach on network-based computation is 'artificial neural networks' [3]. Unlike the data-flow network, an artificial neural network consists of nodes with (quasi-)homogeneous functions modeled by McCulloch-Pitts [11] or Hodgkin-Huxley [4]. An algorithm obtained through learning is indirectly and distributedly coded in the weight vectors for synapses. For this reason, it is hard or impossible for a human to analyze an established algorithm in the artificial neural network.

Though most models for the artificial neural network are primarily focused on the cellular activities in the brain, a natural neuron is, of course, made up of a huge

J. Suzuki and T. Nakano (Eds.): BIONETICS 2010, LNICST 87, pp. 59–73, 2012.
© Institute for Computer Sciences, Social Informatics and Telecommunications Engineering 2012

number of biomolecules from a nanoscopic point of view. These molecules govern all of the activities in a cell, which makes a cell behave dynamically on signal transmission processes. To make a more realistic model for the neurons, it might be better to implement molecular agents (agents with active functions) in artificial neurons and make them move around the neural network, giving rise to a functional change in the neurons.

One such network-based computational model with molecular agents is 'Modified Network Artificial Chemistry' invented by Suzuki [21]. In association with this model, Suzuki [21] proposed a paradigm of 'program-flow computing', wherein programs (agents) move from node to node, bringing different functions to CPUs (nodes). This is also closely related to 'active network' [26] proposed in the 1990s in the area of computer network. The active network enables a router (node) to have various functions by delivering packets (agents) with encapsulated programs.

Based upon these studies, very recently, the authors presented a new concept for computation and learning, named "Algorithmically Transitive Network (ATN)" [22,23,24]. The distinctive features of the ATN are summarized as:

- A program is represented by a 'data-flow network' whose nodes execute arithmetic/logic operations and edges transmit data, like the DFC.
- After the calculation, triggered from the teaching signals, the network transmits data backward and revises network parameters, like the artificial neural network.
- The network topology (algorithm) can be modified/improved through execution of movable agents' programs.

The ATN's learning process is primarily done by the famous 'back-propagation' (BP) learning. Since its first proposal by Werbos and Rumelhart *et al.* [15,16,27], the BP has been successfully applied to a number of real world problems and has been one of the most widely-used learning algorithms among the artificial neural network researchers [17]. Like the BP in the artificial neural network, the BP in the ATN uses the steepest descent method to revise network parameters. A similar idea was also mentioned by Kumazawa [9] who considered the possibility of using the BP to train an 'operation network'. In both the operation network and the ATN, the node operations transmit signals from inputs to outputs so directly that the learning coefficient should be adjusted more minutely in the ATN than in the artificial neural network. The paper presents a formula for the learning coefficient, as well.

The ATN's basic concepts [22,23] and an initial experimental result [24] were briefly given in the previous reports. Following these works, this paper presents the full description of the model's framework and experimental results to reveal the performance of the ATN on a class of symbolic regression problems. In the following, Section 2 explains detailed design of the model, and Section 3 presents the experimental results. In Section 4, concluding remarks and discussion on the model's meanings and future possibility are given.

2 Method

2.1 Data-Flow Network

As in the DFC, the ATN's nodes read the input 'tokens' on their incoming edges, fire, and create the output tokens on their outgoing edges during calculation. This constructs

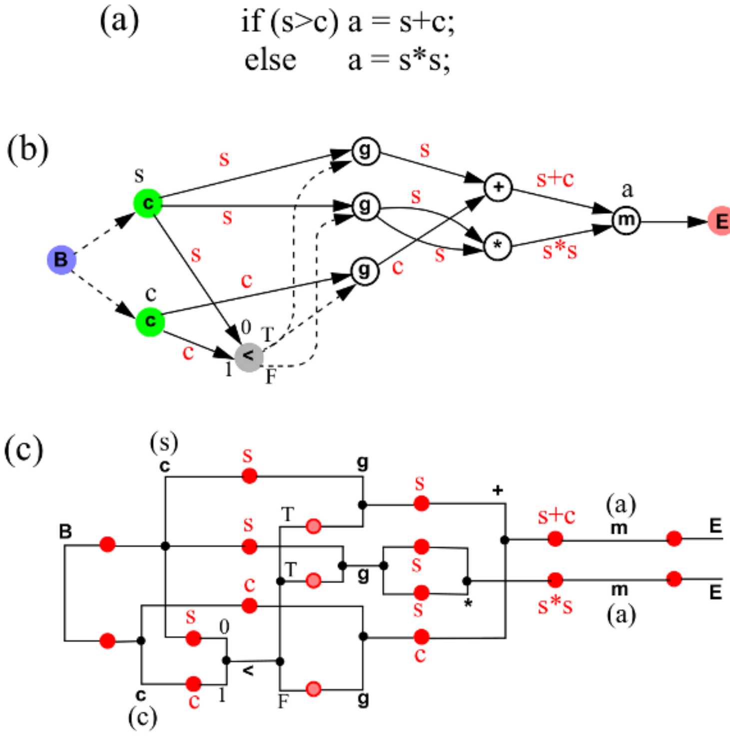

Fig. 1. (a) Higher language program of a simple conditional branch, (b) data-flow network (ATN), and (c) fire-token pedigree produced by the calculation. The variable name s represents the graph's input (sensor) signal, and the a represents the graph's output (answer) value. The pedigree's top ancestor is the initial fire at the 'B' node, and its last descendants are firings at the 'E' node or nodes with no outgoing edge. In (b), arithmetic edges are expressed as the solid arrows, and regulating ones are expressed as the broken arrows.

a 'fire-token pedigree' whose nodes (tokens) represent variables or mathematical expressions in the original program, and whose hyper-edges (firings) represent arithmetic/logical operations used to create the tokens. An example of these relationships is shown in Fig. 1. As explained below, the pedigree is used for the BP learning. The node operations used in this paper are listed in Table 1.

2.2 Simulation Procedure

An ATN simulation proceeds as follows:

Step (1) [Initialization]. Create an initial network randomly or through the transformation of a higher language program. (More arguments on this matter are given in Section 4.)

Step (2) [Calculation]. Substitute new sensor value(s) for the s node's v(s) and make the Begin node fire. Repeat a forward propagation (FP)'s time step operation that

Table 1. Node operations

Name	Oper. code	Input num.	Arith. /Regu.	X	R
Begin	`B`	0	R	–	1
End	`E`	1(a)	R	–	–
Negative	`n`	1(a)	A	$-x_0$	r_0
Inverse	`i`	1(a)	A	$1/x_0$	r_0
Add	$+$	2+	A	$\sum x_i$	$\min(r_i)$
Multiply	$*$	2+	A	$\prod x_i$	$\min(r_i)$
Subtract	$-$	2	A	$x_0 - x_1$	$\min(r_0, r_1)$
Divide	$/$	2	A	x_0/x_1	$\min(r_0, r_1)$
Less than	$<$	2	R	–	$R_0 = \sigma \cdot \min(r_0, r_1)$ $R_1 = (1-\sigma) \cdot \min(r_0, r_1)$
Greater than	$>$	2	R	–	$R_0 = (1-\sigma) \cdot \min(r_0, r_1)$ $R_1 = \sigma \cdot \min(r_0, r_1)$
Equal to	$==$	2	R	–	$R_0 = (1-\delta) \cdot \min(r_0, r_1)$ $R_1 = \delta \cdot \min(r_0, r_1)$
Not equal to	$!=$	2	R	–	$R_0 = \delta \cdot \min(r_0, r_1)$ $R_1 = (1-\delta) \cdot \min(r_0, r_1)$
Logical AND	`A`	2+	R	–	$\min(r_i)$
Logical OR	`O`	2+	R	–	$\max(r_i)$
Logical NOT	`N`	1(a)	R	–	$1 - r_0$
Gate	`g`	2	A	x_0	$\min(r_0, r_1)$
Merge	`m`	1(a)	A	x	r
Arith. constant	`c`	1(a)	A	v	r
Regul. constant	`C`	1(a)	R	–	$R_0 = u$, $R_1 = 1 - u$

The operations are classified as asynchronous (`E`, `n`, `i`, `N`, `m`, `c`, and `C`) or synchronous (the others). '(a)' in the third column represents 'asynchronous'. An asynchronous node fires every time a token is created on any incoming edge, whereas a synchronous node fires only when the tokens are created on all the incoming edges. '2+' in the third column means that the node can have two or more incoming edges. $\sigma \equiv \mathrm{sig}(\kappa\beta_\mathrm{r}(x_0 - x_1))$ where $\mathrm{sig}(z) \equiv \frac{1}{1+\exp(-z)}$, and $\delta \equiv \mathrm{delta}(\kappa\beta_\mathrm{r}(x_0 - x_1))$ where $\mathrm{delta}(z) \equiv 4\mathrm{sig}(z)\mathrm{sig}(-z)$. Note that all the X and R's functions have differentiable formulas. The variables v (for `c`), u (for `C`), and β_r (for judging operations $<$, $>$, $==$, and $!=$) are the node parameters adjusted by the learning. The inverse temperature coefficient κ is a predefined constant.

makes nodes fire, until there is no node able to fire. Calculate the final answer of the network.

Step (3) [Learning]. Calculate partial differential coefficients in tokens on the `a`'s outgoing edges (the last descendant tokens in the pedigree). Repeat the BP's time step operation that 'extinguishes' the firings, until the firing on the Begin node is extinguished. Calculate the learning coefficient η and revise node parameters.

Step (4) [Topological Reformation]. Conduct the agents' graph modifying operations. Move agents to the next nodes/edges if required.

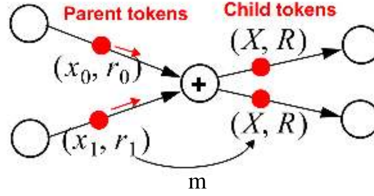

Fig. 2. The FP's unit process in the data-flow network: firing of a '+' node

Step (5) [Supplying Agents]. Randomly choose nodes and supply agents to them at some constant rates.

Step (6) [Termination or Recursion]. If a certain termination condition is satisfied, stop the simulation. Otherwise, go to Step (2).

2.3 Calculation by the Forward Propagation (FP)

In the current implementation, each token has a real value vector (x, r) delimited by $-\infty < x < \infty$ and $0 \leq r \leq 1$, where x is the arithmetic value for the calculation, and r is the regulating value that represents the probability of the token itself existing in the network. When a node fires, the output vector (X, R) is calculated from the operand vector(s) (x_i, r_i)s of the input token(s) using the functions in Table 1. For example, if a '+' node fires, it calculates $X = \sum_i x_i = x_0 + x_1$ and $R = \min_i(r_i) = \min\{r_0, r_1\}$, and creates tokens with (X, R) on all the outgoing edges (Fig. 2).

To make the network learnable, the ATN's judgment nodes (such as '$<$') give a 'fuzzy' result. See the formulas for R_0 and R_1 in Table 1. If x_0 is much smaller than x_1 in a '$<$' node for example, we get $R_0 = 0$ and $R_1 = 1$, hence, we create a token only on the outgoing edge with label '`T`' (we 'kill' the token on the '`F`' edge). If x_0 is similar to x_1, on the other hand, we have both positive R_0 and R_1, hence, we create tokens on both outgoing edges with labels '`T`' and '`F`'.

This fuzzy judgment reproduces tokens in judgment nodes, and in general, after the FP, an `a` node has two or more firings whose child tokens have vector (x_j, r_j)s, where j is the firing number at the `a` node. With these, we calculate the resultant answer value as

$$a = \frac{\sum_j x_j r_j}{\sum_j r_j}. \tag{1}$$

2.4 Learning by the Backward Propagation (BP)

The ATN's learning is conducted by propagating differential coefficients vector (D_x, D_r) from child tokens to parent tokens in the token-fire pedigree. This begins with the calculatin of the energy function

$$E = \frac{1}{2}(t-a)^2 + \mu\left(1 - \sum_j r_j\right)^2 + \nu \sum_j r_j^2 (1 - r_j)^2 \tag{2}$$

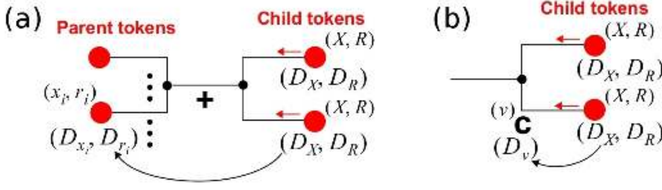

Fig. 3. The BP's unit processes in the fire-token pedigree: extinguishing of (b) a '+' fire and (c) a 'c' fire

at the a node. Here, t is a teaching signal given from the outside, a is a value calculated from Eq. (1), and μ and ν are predefined constants. Again, the summation for j is taken for all the firings at the a node. The μ- and ν-terms are the 'penalty' terms added in order to make the sum of $\{r_j\}$ be one and make each r_j close to zero or one, respectively. By partially differentiating Eq. (2), we can calculate differential coefficients vector $(D_{x_j}, D_{r_j}) = (\frac{\partial E}{\partial x_j}, \frac{\partial E}{\partial r_j})$ of the a node's output tokens. (Through this paper, $D_{\square}$ signifies the energy function E's partial differential coefficients with respect to $\square$.)

Once (D_X, D_R) is obtained for all the child tokens of a firing (here, (D_x, D_r) is expressed as (D_X, D_R) to indicate that it is for a child token), the firing is extinguished and the parent tokens' (D_{x_i}, D_{r_i}) is calculated with

$$D_{x_i} = \sum_{\text{children}} \left(D_X \frac{\partial X}{\partial x_i} + D_R \frac{\partial R}{\partial x_i} \right), \tag{3a}$$

$$D_{r_i} = \sum_{\text{children}} \left(D_X \frac{\partial X}{\partial r_i} + D_R \frac{\partial R}{\partial r_i} \right). \tag{3b}$$

$\frac{\partial X}{\partial x_i}$, $\frac{\partial R}{\partial x_i}$, $\frac{\partial X}{\partial r_i}$, and $\frac{\partial R}{\partial r_i}$ are calculated from Table 1. The summation '$\sum_{\text{children}}$' is taken for all the child tokens of the fire (Fig. 3(a)).

Specifically, when a firing is extinguished at a node with parameter $\{v\}$, $\{u\}$, or $\{\beta_{\text{r}}\}$ (which are all expressed as $\{z\}$ hereafter), we also calculate the partial differential coefficient of E with respect to z using

$$D_z = \sum_{\text{children}} \left(D_X \frac{\partial X}{\partial z} + D_R \frac{\partial R}{\partial z} \right) \tag{4}$$

(Figure 3(b)). $\frac{\partial X}{\partial z}$ and $\frac{\partial R}{\partial z}$ are also calculated from Table 1. The chain rule (differentiation rule for the composite function) ensures that D_z evaluated from Eq. (4) gives the partial derivatives of the original energy function E [3].

After D_z is obtained at all of the c, C, and judging firings, we finally revise the node parameters using the steepest descent method as:

$$z \to z - \eta \frac{\partial E}{\partial z} = z - \eta D_z. \tag{5}$$

Here, with a predefined constant $\eta_{\rm lp}$, we require that the revision by a one-pass propagation (a pair of the FP and BP) make E become $1-\eta_{\rm lp}$ times the former value as:

$$
\begin{aligned}
(1-\eta_{\rm lp})\cdot E(\{z\}) &= E\left(\left\{z-\eta\frac{\partial E}{\partial z}\right\}\right)\\
&\simeq E(\{z\})-\eta\sum_z\left(\frac{\partial E}{\partial z}\right)^2\\
\therefore\ \eta &= \frac{\eta_{\rm lp}\cdot E}{\sum_z\left(\frac{\partial E}{\partial z}\right)^2}.
\end{aligned}
\tag{6}
$$

The linear approximation of the Taylor expansion of $E(\{z\})$ was used. The convergence of the parameter learning by Eq. (5) is ensured by Eq. (6).

2.5 Topological Reformation

The topological reformation of an ATN is conducted by agent operations listed up in Fig. 4. A CON (constantification) agent changes the node operation into c or C (Fig. 4(a)), a DIV (division) divides a constant node into two (Fig. 4(b)), a BRG (bridge) constructs a bridge between an edge and a node (Fig. 4(c)), a MKV makes a new variable node (Fig. 4(d)), a MGT merges two adjacent addition/multiplication operations (tuples) (Fig. 4(e)), and a MGN merges two or more constant operand nodes within an addition/multiplication operation (Fig. 4(f)). Among these, CON, MGT, and MGN simplify (reduce the node/edge numbers of) the graph, whereas the others complexify the graph.

Though not shown in Fig. 4, the seventh agent WAR rewires the background '**wa**' edges which are compared to 'van der Waals' bonds known as the weakest interaction between bio-molecules [19,20,21]. Some agent operations except for CON and DIV are concerned with two or more nodes. One way to do such inter-node operations is that an agent moves through the network and collects information by themselves, but in the current implementation, most agents stay at nodes and gather information through '**wa**' edges created by the WAR agents which move around the network in lieu of them.

In what follows, we take CON, DIV, and BRG and explain their detailed operations.

A CON agent usually stays at a variable node. During the stay, it observes (x, r) of the firings created at the node, and if the value does not change for a long time, it changes the node operation into c or C depending on whether the node operation is arithmetic or regulating. At the same time, the incoming edges of the node are cut except for an edge with the minimum sum of the past r values which is conserved to make the node 'firable'.

A DIV agent usually stays at an arithmetic/regulating constant node with two or more outgoing edges. It observes (D_x, D_r) of firings at the node for a long time, and if they contradict each other, divides the node into two. After this division, v (or u) of the divided nodes are able to learn toward different directions.

A BRG agent usually stays at an edge. It randomly chooses a neighboring node 'a' and constructs a bridge between the current edge and node 'a' as shown in Fig. 4(c).

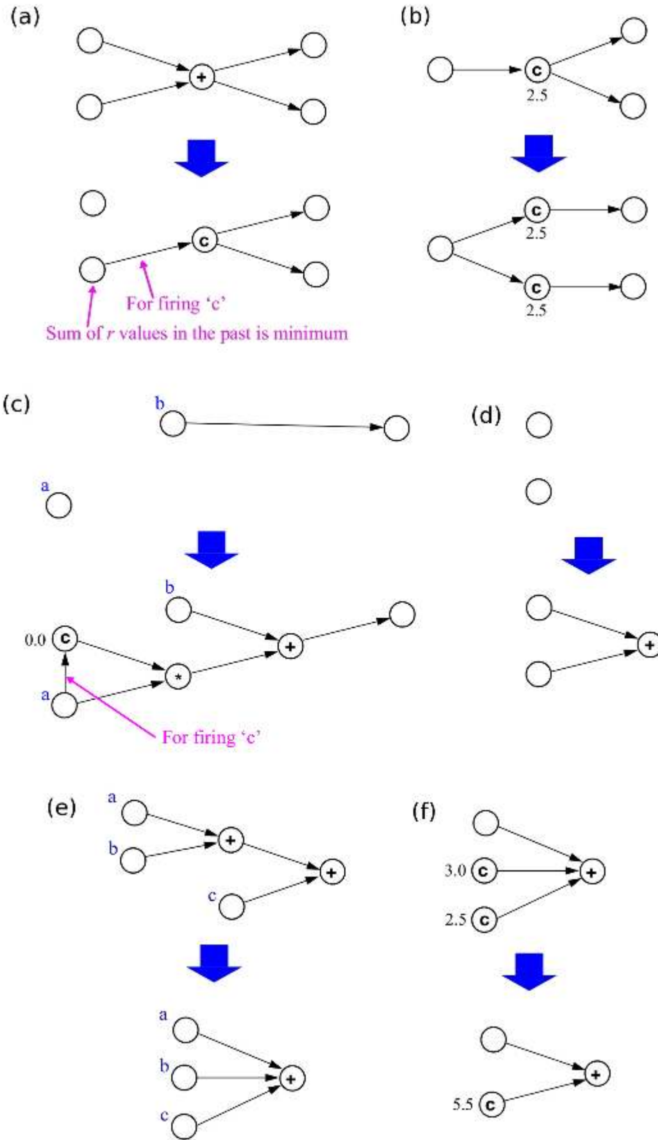

Fig. 4. Graph modifying operations by agents

Three nodes with the operations $\star$, +, and c (with $v = 0.0$) are newly created for the bridge. Since $a * 0.0 + b = b$, this modification does not change the calculation result; however, after this operation, differential coefficients propagated to the newly created c node gradually changes v, which makes node 'a' affect the calculation result.

Through all the graph reformation processes, the current implementation prohibits a 'loop' from being created in the ATN. The future possibility of loosening this constraint is discussed in Section 4.

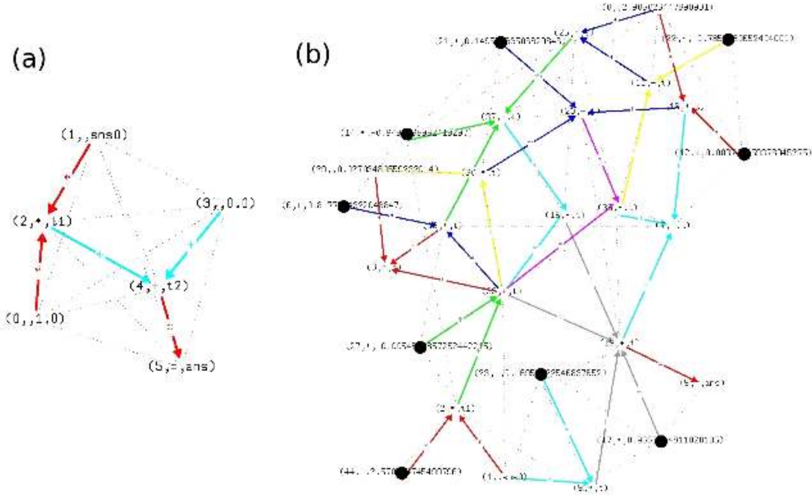

Fig. 5. (a) Initial ATN and (b) final ATN (after $80,033$ time steps) obtained for a regression problem for the one-variable quadratic function (Eq. (7a)). The graphs are drawn with a commercial software named aiSee [1]. The final ATN produces the a value with a cubic function $a = 0.965 + (0.169s) + (-0.00548 + 2.57s) + ((-0.00548 + 2.57s)(-0.948 + (0.817(-0.00548+2.57s)) + (2.90(-0.785+((-0.00548+2.57s)(0.149+(0.00374*2.90)+(0.0270(-0.00548+2.57s))))))))$. The result was obtained from a one-minute simiulation run.

3 Experiments

To demonstrate the ATN's basic capability to explore algorithms, here we apply the ATN to some symbolic regression problems. The simulation constants are taken to be: $\mu = 1.0$, $\nu = 1.0$, $\beta_{\text{r-init}} = 0.3$ (β_{r}'s initial value), $\kappa = 100.0$, $\eta_{\text{lp}} = 0.5$, $N_{\text{tgt}} = 300$ (target node number), $M_{\text{wa-tgt}} = 5.0$ (**wa** edge's target degree per node), $\lambda = 1.0$ (mean free path for rewiring **wa** edges; a **wa** edge is joined to a closer node with a smaller this value [20]), $T_{\text{xrUcCON}} = 20$ (number of passes until a node is judged to be constant), $r_{\text{diff-ngl}} = 1.0 \times 10^{-4}$ (threshold of r under which tokens are killed), and $T_{\text{watchStblty}} = 10$ (number of passes until DIV decides to split a node). The simulation program is implemented in Java and is run on a standard desktop computer with Intel Duo processor (1.86GHz).

3.1 Polynomial Functions

We prepare one-variable quadratic, cubic, and quartic functions

$$t = 1 - 6s + 10s^2 \quad (7a)$$

$$t = -0.5 + 11s - 35s^2 + 28s^3 \quad (7b)$$

$$t = 0.2 + 35s - 175s^2 + 300s^3 - 160s^4 \quad (7c)$$

within the domain $0 \le s \le 1$ as targets, and examine the ATN's learnability.

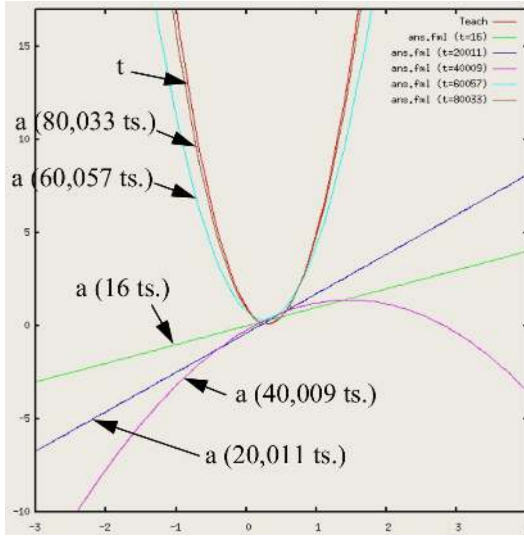

Fig. 6. The s-a (sensor-answer) and s-t (sensor-teach) plots for the run in Fig.5

Figures 5 and 6 show a representative result for the quadratic function. In this experiment, we start from a handmade 6-node ATN that represents a linear function $a = 0 - s$ (see Fig. 5(a)), then after 80,000 time steps (about 800 passes), we obtain a 39-node 158-agent network (Fig. 5(b)). Figure 6 shows the change of the s-a (sensor-answer) plot during this run. We can see from this figure that the final function of the ATN almost perfectly agrees with the target function within the domain. Results for ten different runs plotted in Fig. 7 show that nine out of ten runs succeed in finding desirable functions for this regression problem.

Using the same method, we also tested the ATN's learnability with the cubic and quartic functions (Eqs. (7b) and (7c)). The representative results are shown in Fig. 8. For both functions, we tested hundred runs and ten runs with different random number sequences and found that 60% and 80% runs succeeded in creating the desirable functions, respectively.

3.2 Fraction Function

Next we apply the ATN to symbolic regression of a 'fraction' function defined as

$$t = \frac{11 - 62s + 88s^2}{6.5 - 32s + 40s^2} \tag{8}$$

within the domain $0 \leq s \leq 1$.

We tested ten runs using different random number sequences, but all the runs failed to create solution networks that produce the desirable answer for this much more difficult function. Figure 9 shows a representative result.

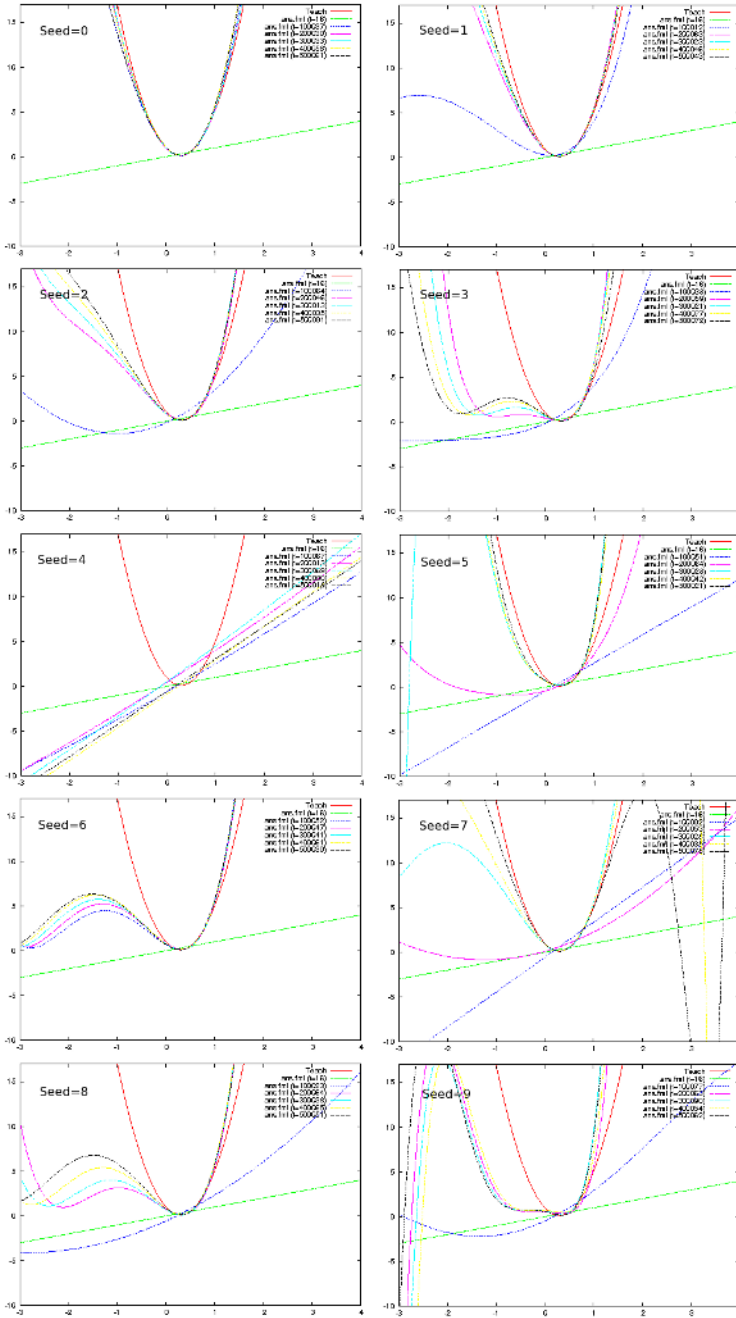

Fig. 7. s-a and s-t plots obtained for a regression problem for the one-variable quadratic function (Eq. (7a)). Out of ten different runs that use the same parameter setting, nine runs find desirable results (only the run with Seed $=$ 4 fails). Note that the s is given only within the domain $0 \leq s \leq 1$.

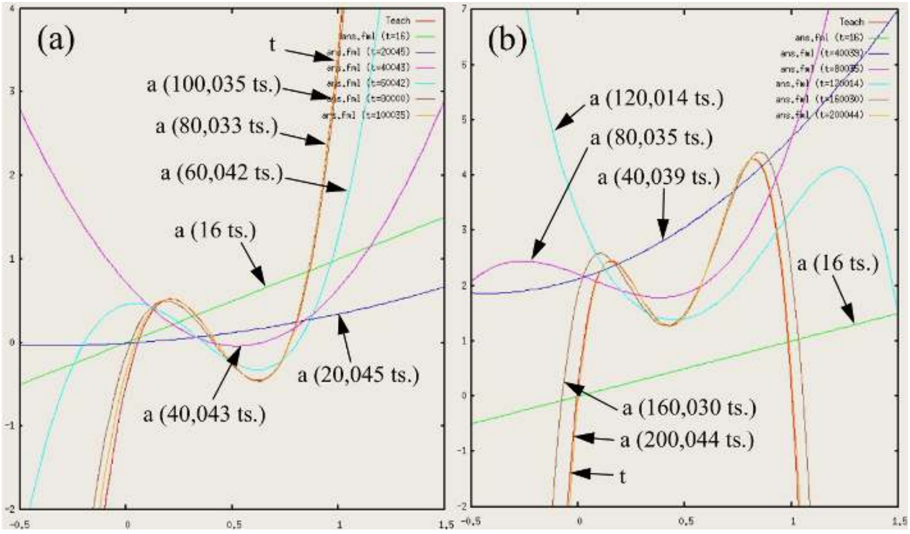

Fig. 8. The s-a and s-t plots for representative runs for the (a) cubic function (Eq. (7b)) and (b) quartic function (Eq. (7c))

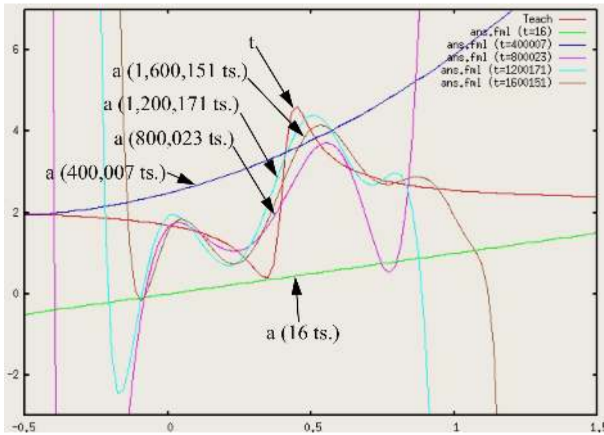

Fig. 9. The s-a and s-t plots for a representative run for the fraction function (Eq. (8))

3.3 Conditional Branch

Finally, we take a conditional branch and conduct a preliminary experiment. We set a teaching function

$$\begin{array}{ll} \text{if } s > 0.6, & t = 1 - s \\ \text{else} & t = 4s + 1 \end{array} \tag{9}$$

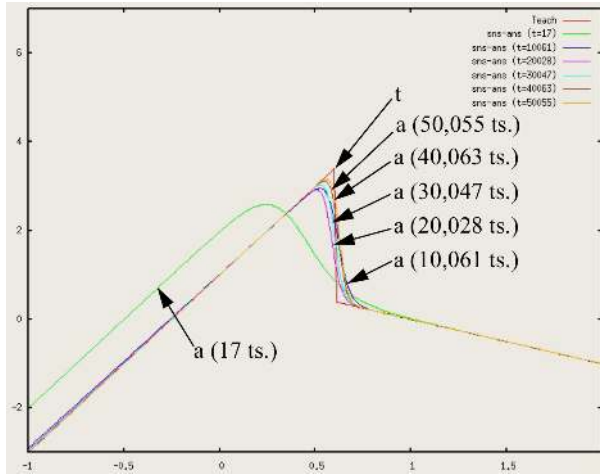

Fig. 10. s-a and s-t plots obtained for a conditional branch problem (Eq. (9))

within the domain $0 \leq s \leq 1$. We hand-design the initial ATN whose topology directly represents Eq. (9) but whose network parameters are different from the target values, and examine the ATN's capability of adjusting the parameters towards the desirable values. No agents for topological reformation are supplied.

Figure 10 shows a representative result. We can see from this figure that the ATN is able to optimize not only the threshold value but also the branch's fuzziness parameter β_{r}.

4 Discussion

A novel computing and learning model, algorithmically transitive network (ATN), was proposed. The ATN is represented by a data-flow network used to describe an algorithm of the data-flow computer (DFC). After the calculation by the forward propagation, the ATN propagates differential coefficients backward and adjusts node parameters with the steepest descent method. Moreover, based on the information obtained from this learning, the ATN modifies topological structure of the network through the agent operations, leading to the renovation of the network's algorithm. The model was successfully applied to a few simple symbolic regression problems, but the limitation of the learning capability was also found with more complex functions. Revising the model's framework and improving the ATN's learnability remains a future research subject.

4.1 Supervised Learning in Computation

As another auto-programming tool that represents algorithms by a network, we have Genetic Programming (GP) proposed by Koza [6,7,8]. Though the original GP used only a program graph with tree topology, but such research as PADO [25], Cartesian GP [12,13], and GNP [10] adopted networks with free topology and have extended the GP's domain. In addition, some researchers have also incorporated reinforcement

learning in the GP: node parameters were adjusted by Q-learning [5,2,10] or by the multiple linear regression analysis [14]. The ATN, on the other hand, combines the GP-like program graphs with the artificial neural network and enables more powerful supervised learning in computation.

4.2 Translation from Programming Language

Though the ATN experiments presented in this paper start from a small random/hand-made network, if we were able to develop a tool to translate a higher language program (written in C, Java, or whatever) into the ATN, the presented method would be used to improve/optimize programs designed by the humans. Of course, a usual higher language program includes a number of different control flows not tested in this paper: loop, subroutine, and so on. Translating a program with these elements into the ATN and revising it with the presented method are one of the problems to be tackled in the future.

References

1. aiSee: Commercial software for visualizing graphs with various algorithms such as rubber-band, http://www.aisee.com/
2. Downing, K.L.: Reinforced genetic programming. Genetic Programming and Evolvable Machines 2(3), 259–288 (2001)
3. Haykin, S.: Neural networks and learning machines. Prentice-Hall, Inc. (2009)
4. Hodgkin, A.L., Huxley, A.F.: A quantitative description of membrane current and its application to conduction and excitation in nerve. Journal of Physiology 117, 500–544 (1952)
5. Iba, H.: Multi-agent reinforcement learning with genetic programming. In: Koza, J.R., et al. (eds.) Genetic Programming 1998: Proceedings of the Third Annual Conference (GP 1998), pp. 167–172 (1998)
6. Koza, J.R.: Genetic Programming: on the Programming of Computers by Means of Natural Selection. MIT Press, Boston (1992)
7. Koza, J.R.: Genetic Programming II: Automatic Discovery of Reusable Programs. MIT Press, Boston (1994)
8. Koza, J.R., Bennett III, F.H., Andre, D., Keane, M.A.: Genetic programming III: darwinian invention and problem solving. MIT Press, Boston (1999)
9. Kumazawa, I.: Learning and neural network. Morikita Publishing Company, Tokyo (1998) (Japanese)
10. Mabu, S., Hirasawa, K., Hu, J.: A graph-based evolutionary algorithm: genetic network programming (GNP) and its extension using reinforcement learning. Evolutionary Computation 15(3), 369–398 (2007)
11. McCulloch, W.S., Pitts, W.: A logical calculation of the ideas immanent in nervous activity. Bullet. Math. Biophysics 5, 115–133 (1943)
12. Miller, J.F.: An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In: Banzhaf, W., et al. (eds.) Proceedings of the Genetic and Evolutionary Computation Conference, vol. 2, pp. 1135–1142. Morgan Kaufmann (1999)
13. Miller, J.F., Smith, S.L.: Redundancy and computational efficiency in cartesian genetic programming. IEEE Transactions on Evolutionary Computation 10(2), 167–174 (2006)

14. Nikolaev, N.Y., Iba, H.: Regularization Approach to Inductive Genetic Programming. IEEE Transactions on Evolutionary Computation 5(4), 359–375 (2001)
15. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. Nature 323, 533–536 (1986)
16. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning internal representations by error propagation. In: McClelland, J.L., Rumelhart (eds.) The PDP Research Group: Parallel Distributed Processing, vol. 1. MIT Press, Cambridge (1986)
17. Sejnowski, T.J., Rosenberg, C.R.: Parallel networks that learn to pronounce English text. Complex Systems 1, 145–168 (1987)
18. Sharp, J.A. (ed.): Data flow computing: Theory and practice. Ablex Publishing Corp., Norwood (1992)
19. Suzuki, H.: Mathematical folding of node chains in a molecular network. BioSystems 87, 125–135 (2007)
20. Suzuki, H.: An approach toward emulating molecular interaction with a graph. Australian Journal of Chemistry 59, 869–873 (2006)
21. Suzuki, H.: A network cell with molecular agents that divides from centrosome signals. BioSystems 94, 118–125 (2008)
22. Suzuki, H., Ohsaki, H., Sawai, H.: A Network-Based Computational Model with Learning. In: Calude, C.S., Hagiya, M., Morita, K., Rozenberg, G., Timmis, J. (eds.) UC 2010. LNCS, vol. 6079, pp. 193–193. Springer, Heidelberg (2010)
23. Suzuki, H., Ohsaki, H., Sawai, H.: Algorithmically Transitive Network: a new computing model that combines artificial chemistry and information-communication engineering. In: Proceedings of the 24th Annual Conference of Japanese Society for Artificial Intelligence (JSAI), pp. 2H1-OS4-5 (2010) (Japanese)
24. Suzuki, H., Ohsaki, H., Sawai, H.: An agent-based neural computational model with learning. Frontiers in Neuroscience. Conference Abstract: Neuroinformatics (2010), doi:10.3389/conf.fnins.2010.13.00021
25. Teller, A., Veloso, M.: PADO: Learning tree-structured algorithm for orchestration into an object recognition system. Carnegie Mellon University Technical Report, CMU-CS-95-101 (1995)
26. Tennenhouse, D.L., Wetherall, D.J.: Towards an active network architecture. ACM Computer Communication Review 26(2), 5–18 (1996)
27. Werbos, P.J.: The roots of backpropagation: From ordered derivatives to neural networks and political forecasting. In: Adaptive and Learning Systems for Signal Processing, Communications and Control Series. Wiley Interscience (1994)