Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2014, Article ID 362968, 11 pages
http://dx.doi.org/10.1155/2014/362968

Research Article

Autonomous Development of Algorithmic Concepts for Program Comprehension

Guojin Zhu, Kai Zhang, and Jiyun Li

School of Computer Science and Technology, Donghua University, Shanghai 200051, China

Correspondence should be addressed to Guojin Zhu; gjzhu.dhu@163.com

Received 18 March 2014; Accepted 9 July 2014; Published 5 August 2014

Academic Editor: Jingjing Zhou

A developmental model of algorithmic concepts is proposed here for program comprehension. Unlike traditional approaches that cannot do anything beyond their predesigned representation, this model can develop its internal representation autonomously from chaos into algorithmic concepts by mimicking concept formation in the brain under an uncontrollable environment that consists of program source codes from the Internet. The developed concepts can be employed to identify what algorithm a program performs. The accuracy of such identification reached 97.15% in a given experiment.

1. Introduction

Our idea of autonomous development of algorithmic concepts for program comprehension is inspired by the autonomous development paradigm [1] for constructing developmental robots.

Program comprehension [2, 3], also known as program understanding, is concerned with ways to analyze source codes for achieving some intentions, such as code reuse [4], code plagiarism detection [5, 6], algorithm recognition [7–11], and programming tutoring [12]. Over the past decades, scientists have proposed many approaches. Most of them depend on predesigned representations, which include [13] mental models that describe human mental representations of the program to be understood, cognitive models that describe the cognitive processes and temporary information structures in the programmer's head that are used to form mental models, and programming plans that are generic fragments of code that represent typical scenarios in programming. In these traditional approaches, a machine cannot do anything beyond the predesigned representation. For example, the traditional approaches of algorithm recognition are unable to recognize the algorithms whose programming plans or templates [7–10] are not defined in the library of algorithm templates.

In contrast, the autonomous development paradigm, called autonomous mental development (AMD), enables machines to develop their minds autonomously when they interact with their environments [1, 14]. With AMD, a robot can learn any tasks, including those whose representations are not defined before the robot is born. Our previous work shows that applying the AMD theory to program comprehension could avoid predefining templates for algorithm recognition [11].

Unlike predesigned representations that will not change when the machine runs, the representation proposed in this paper will change or develop gradually from randomness to algorithmic concepts when the machine interacts with its environment that consists of program source codes. This is similar to the internal representation in a human brain that develops from no idea of any apple at the birth of the human being to a concept for apples after the brain obtains enough information about apples.

Although the brain has no idea of any apple when the human being encounters an apple at the first time, some information about the apple (i.e., an image of the apple) will reside in the memory of the brain, which makes something change in the memory. When the human being encounters an apple again, the brain may not think this second apple so strange by recalling the image of the first apple. Moreover, the brain may be more familiar with apples after the human being encounters the second apple. This means that the image of the first apple is updated with some information obtained from the second apple, resulting in the fact that the updated image

stands for both apples rather than just for the first one. In this way, the apple image in the memory of the brain is updated whenever the human being encounters an apple, leading to the brain being more and more familiar with apples. Finally, the brain is so familiar with apples that almost nothing of the apple image needs updating. At this time, the apple image in the brain represents all the apples in the world, and we think that a concept for apples is formed in the brain (i.e., the apple image becomes the concept for apples).

Our developmental approach, which is on a basis of a computational neural network, mimics the above procedure of concept formation in the brain to develop algorithmic concepts autonomously with algorithmic information extracted from program source codes. This might be an easy task if each of the program source codes implements one of two simple algorithms. For instance, one might apply a concept-formation neural network [15] to divide these program source codes into two groups with each group representing a concept for one of the two algorithms. In this case, the task becomes classifying (or clustering) the program source codes.

On the other hand, it might be a difficult task to mine program source codes that come from online judge systems [16] in the Internet. The algorithmic knowledge underlying these program source codes, which are submitted by college students from all over the world, is very valuable for programming tutoring. However, discovering such algorithmic knowledge seems a challenge because the space consisting of these program source codes is changing uncontrollably every day. We are hardly able to know what and how many algorithmic concepts lie in these program source codes in advance. We are hardly able to predict what new algorithmic concepts will emerge in the changing space consisting of these program source codes. In simple words, this changing environmental space is muddy, so that the developmental approach is necessary for such muddy task [14] in this case.

In this paper, we propose a developmental model for concept formation under such an environmental space that is unknown but is changing uncontrollably as described above. The algorithmic concept generated by this developmental model is readable for both machine and human. This feature is desirable in program comprehension. In a concept assignment task [17], for instance, a machine with this developmental model is able to associate its perceived program with one of its developed concepts that stands for the algorithm performed by the program, while at the same time the same machine is able to display the associated concept in a human-readable expression, making it possible for a human to understand what algorithm the program performs.

2. Algorithmic Concept Development

2.1. Algorithmic Concepts. A person understands a program at an algorithmic level when being able to explain the sequence of operations that the program performs; that is, a program could be understood as a sequence of operations.

Figure 1(a) shows a program that can be understood as a sequence of concrete operations, or *concrete algorithm*, as described by the flowchart in Figure 1(c). The idea behind this specific sequence of concrete operations is to use an array as a function table. The function table describes a function f by tabulating all the arguments x and their corresponding function values $f(x)$. In Figure 1(a), the array named a is used as the function table. When obtaining an argument value n (e.g., $n = 0$) from its input, the program can return the corresponding function value (e.g., 8899) easily, just by outputting the value of $a[n]$ (e.g., $a[0]$).

The same idea of using an array as a function table is also behind the flowchart in Figure 1(d), which expresses the concrete algorithm of the program in Figure 1(b). The main distinction of the program in Figure 1(b) from the one in Figure 1(a) is that the outputs of the former are vectors; for example, the output $str[0]$ is the vector $(1, 2, 3, 4, 5, 6)$. This implies that the two concrete algorithms described in Figures 1(c) and 1(d) could be conceptualized as an abstract algorithm, as expressed by the flowchart in Figure 1(e).

An algorithmic concept is a common idea behind all concrete algorithms that have the same characteristics. The idea behind the two concrete algorithms above, for example, is such an algorithmic concept, which also refers to the abstract algorithm expressed in Figure 1(e). Each of the two programs above is regarded as an instance of this algorithmic concept.

There are many algorithmic concepts in brains of human programmers. Each algorithmic concept is a common notion shared by programmers to refer to similar concrete algorithms. Thus, an algorithmic concept represents a class of similar concrete algorithms. To understand similar concrete algorithms as an algorithmic concept makes programmers manipulate programs easier. Such manipulation includes creating, maintaining, explaining, reengineering, and reusing [17]. It is convenient for programmers to communicate with each other by using algorithmic concepts.

Different algorithmic concepts represent different classes of concrete algorithms. Each concrete algorithm in a class is an instance of the algorithm concept that represents the class.

2.2. The Developmental Model for Concepts. From our developmental standpoint, the internal state S of the brain B depends on the environment E that the agent of the brain B explores, formally written as $S = B(E)$. More specifically, the environmental area E_t that the brain B has sensed from the very beginning to the current time $t > 0$ determines the internal state S_{t+1} of the brain B at the next time $t+1$; that is, $S_{t+1} = B(E_t)$. Biologically, the brain runs by sending signals of discrete spikes. So we think that the brain works in discrete time (i.e., $t = 0, 1, 2, \ldots, k, \ldots$).

In this work, the environment E consists of program source codes that come from online judge systems [16] in the Internet. We assume that at each time instance t the brain senses one and only one program e_t. Thus, we have $E_t = \{e_0, e_1, \ldots, e_t\}$. The distinction of E_t from E is that the elements in E_t are ordered in time whereas those in E are not. In other words, E_t denotes a sequence which consists of elements from E. Note that E_t may have duplicate elements; for example, $e_1 = e_{10}$ if the brain encounters the program e_1 again at the time instance $t = 10$. Moreover, there are many possible

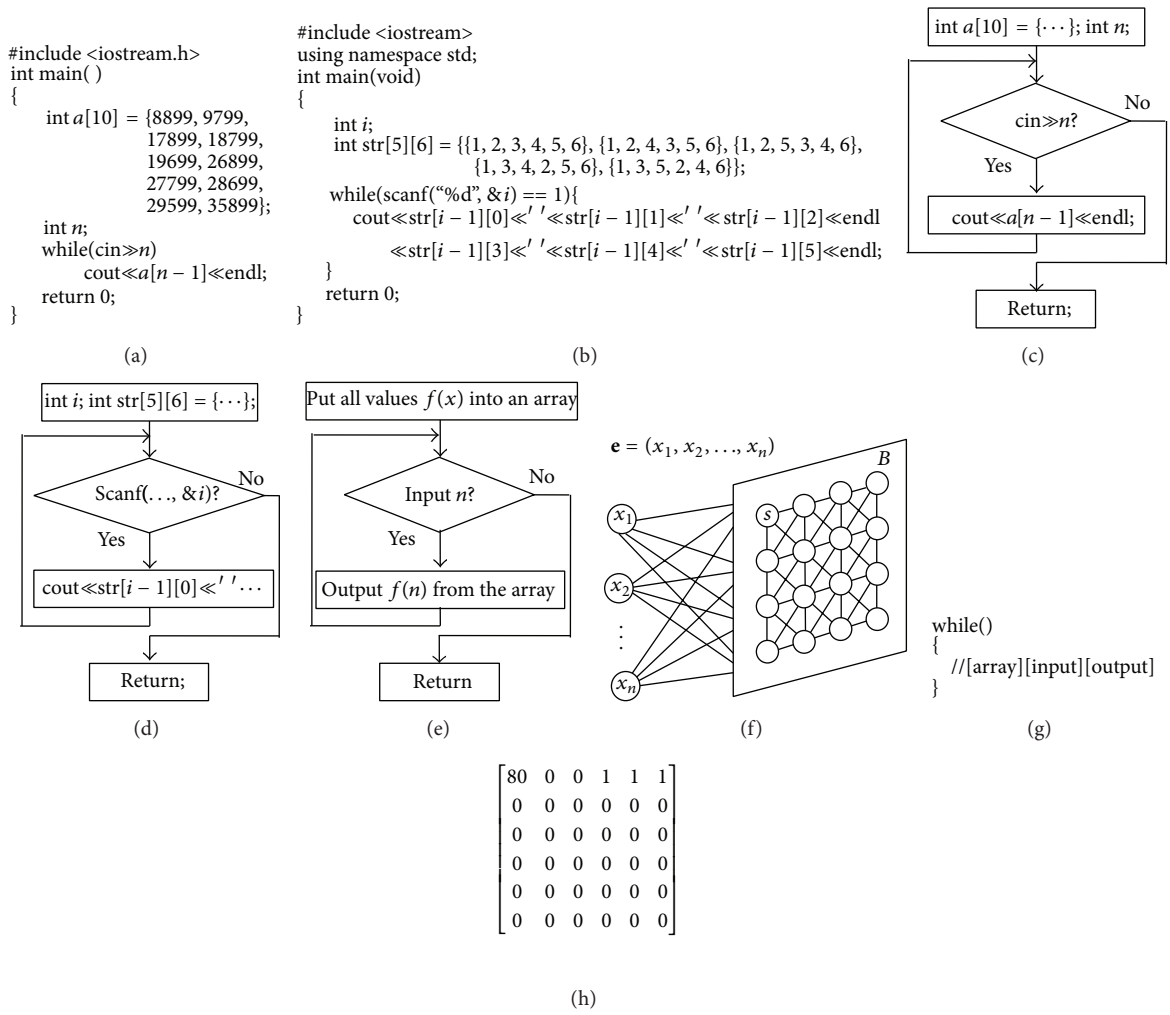


FIGURE 1: (a) For each input number, this program prints a corresponding number from a one-dimensional array. (b) For each input number, this program prints a corresponding vector from a two-dimensional array. (c) This flowchart depicts the concrete algorithm for the program in (a). (d) This flowchart depicts the concrete algorithm for the program in (b). (e) The abstract flowchart for both programs in (a) and (b): for each input n, output the corresponding function value $f(n)$ from a function table stored in an array. (f) The brain B is implemented as a one-layer neural network which has $k \times k$ neurons. Its inputs are n-dimensional vectors. Each input vector $\mathbf{e}$ will trigger the procedure $R(\mathbf{e})$ to recall a neuron s in the brain B. The synaptic vector $\mathbf{s}$ of the neuron s will be updated by the procedure $U(\mathbf{s}, \mathbf{e})$. (g) The algorithmic signature of the program in (a). (h) The matrix for the algorithmic signature in (g).

sequences that may form the E_t at the time instance t for a given E because the brain may explore the environment E randomly.

The internal state S is modeled as a set of elements, called images. Each image s in the set S will be recalled when the brain obtains the algorithmic information of a program e in E. This can be formally written as $s = R(e)$, where R denotes the procedure of recalling s triggered by e. In addition, the image s will be updated into s'; that is, $s' = U(s, e)$, where U denotes the procedure of updating s with the algorithmic information of the program e. The effect of this updating is equivalent to adding the updated image s' into the set S from which the image s has been removed. Thus, the change of the internal state S can be formulated as $S_{t+1} = (S_t \setminus \{s\}) \cup \{s'\}$, where $s = R(e_t)$, $s' = U(s, e_t)$, and e_t denotes the program that the brain encounters at the time instance t.

At the very beginning (i.e., $t = 0$), the brain has no idea of any program but may remember some algorithmic information of any program that it encounters. For this reason, we assume that the initial state S_0 of the set S consists of some things that will change to remember some algorithmic information of programs. In addition, for computational convenience, we use the same notation $s = R(e)$ for something s in S_0 that will change to remember some algorithmic information of the program e, so that, for each time instance t, there always exists one and only one element s in S_t that satisfies $s = R(e)$ for each given program e in E. Biologically, the initial state S_0 refers to innateness [18] which is the result of evolution.

Being innate, each element s in the initial state S_0 contains no algorithmic information at all. After updated by the procedure $U(s, e)$, however, the element s turns into s', which

contains some algorithmic information of the program e. More and more elements in the internal state S will be updated to contain algorithmic information when the brain senses more and more programs, while some of them are updated many times to contain rich information about algorithms. When some element s in S is updated slightly by the procedure $U(s, e_t)$ at a time instance t (i.e., its updated image s' is almost the same as the element s itself), the updated image s' is regarded as an algorithmic concept. Note that the algorithmic concept s' is developed from something having nothing to do with any algorithmic information.

This developmental model for concept formation is described formally as the following algorithm, where all developed concepts are collected in a set C (which is initialized empty, i.e., $C = \emptyset$).

Step 1. $C \leftarrow \emptyset$

Step 2. $S \leftarrow S_0$

Step 3. At each time instance $t = 0, 1, 2, \ldots$

Step 3a. $e \leftarrow e_t$

Step 3b. $s \leftarrow R(e)$

Step 3c. $s' \leftarrow U(s, e)$

Step 3d. $S \leftarrow (S \setminus \{s\}) \cup \{s'\}$

Step 3e. $C \leftarrow C \cup \{s'\}$ if s' is almost the same as s.

In summary, this developmental model consists of three elements: a brain B, its internal state S, and its environment E. The behavior of the brain B is characterized as its recalling procedure R and updating procedure U. Each time the brain B senses a program e in the environment E, the procedure R will recall an image s in the internal state S that satisfies $s = R(e)$, and then the procedure U will update the image s such that the updated image $s' = U(s, e)$. When the difference between the image s and its updated image s' becomes very small after much many updates, we regard the updated image s' as an algorithmic concept and put it into the set C of developed concepts.

2.3. The Internal Representation for Development. We apply the theory of lobe component analysis (LCA) to implementation of our developmental model for concept formation. The LCA, which meets the idea of AMD, was proposed for developing feature detectors (neurons) in neural networks [19].

The brain B in our developmental model is implemented as a one-layer network which consists of $k \times k$ neurons. The internal state S of the brain B is composed of all the synaptic vectors that represent synaptic weights of neurons in the brain B (see Figure 1(f)). Every synaptic vector $\mathbf{s}$ in S can contain the algorithmic information from programs in the environment E. All these synaptic vectors of neurons are initialized with random values to represent the initial state S_0 of S at the time $t = 0$. This means that each neuron s in the brain B is "born" with no relation to algorithmic information.

The inputs to the brain B are also vectors, which have the same dimension as the synaptic vectors. Each input vector $\mathbf{e}$ represents the algorithmic information of a program source code e in the environment E.

The LCA neural network contains two cell-centered mechanisms, called lateral inhibition and Hebbian learning, respectively. The former can be applied to implementation of our recalling procedure $R(\mathbf{e})$ and the latter to implementation of our updating procedure $U(\mathbf{s}, \mathbf{e})$.

The recalling procedure $R(\mathbf{e})$ is implemented as follows. We computer the response $P_{\mathbf{e}}(\mathbf{s})$ of every synaptic vector $\mathbf{s}$ in S by $P_{\mathbf{e}}(\mathbf{s}) = \mathbf{e} \cdot \mathbf{s}/\|\mathbf{s}\|$, where $\|\mathbf{s}\|$ denotes for the norm of the vector $\mathbf{s}$, and $\mathbf{e}\cdot\mathbf{s}$ for the inner product of the two vectors $\mathbf{e}$ and $\mathbf{s}$. The response $P_{\mathbf{e}}(\mathbf{s})$ is the projection of the input vector $\mathbf{e}$ on the synaptic vector $\mathbf{s}$. The greater the response $P_{\mathbf{e}}(\mathbf{s})$ is, the closer the synaptic vector $\mathbf{s}$ is to the input vector $\mathbf{e}$. A synaptic vector $\mathbf{s}_1$ is closer to the input vector $\mathbf{e}$ than another synaptic vector $\mathbf{s}_2$ if its response $P_{\mathbf{e}}(\mathbf{s}_1)$ is greater than the response $P_{\mathbf{e}}(\mathbf{s}_2)$ of the synaptic vector $\mathbf{s}_2$. If there is no synaptic vector closer to the input vector $\mathbf{e}$ than a synaptic vector $\mathbf{s}_m$, then the synaptic vector $\mathbf{s}_m$ has the maximal response $M(\mathbf{e})$, where

$$M(\mathbf{e}) = \max_{\mathbf{s} \in S} P_{\mathbf{e}}(\mathbf{s}). \tag{1}$$

Thus, it is reasonable to think that the synaptic vector $\mathbf{s}_m$ that satisfies $P_{\mathbf{e}}(\mathbf{s}_m) = M(\mathbf{e})$ represents the image that the brain B will recall when receiving the input vector $\mathbf{e}$; that is, $R(\mathbf{e}) = \mathbf{s}_m$. We also say that the procedure $R(\mathbf{e})$ recalls the neuron s that the synaptic vector $\mathbf{s}_m$ belongs to.

The updating procedure $U(\mathbf{s}, \mathbf{e})$ is formulated as

$$U(\mathbf{s}, \mathbf{e}) = \omega_1(n_s)\,\mathbf{s} + \omega_2(n_s)\,M(\mathbf{e})\,\mathbf{e}, \tag{2}$$

where $\omega_1(n_s)$ denotes the retention rate and $\omega_2(n_s)$ the learning rate and $\omega_1(n_s) + \omega_2(n_s) = 1$. Both the retention rate $\omega_1(n_s)$ and the learning rate $\omega_2(n_s)$ are functions of the age n_s of the neuron s that the synaptic vector $\mathbf{s}$ belongs to. The age n_s keeps the number of times that the synaptic vector of the neuron s has been updated.

The retention rate $\omega_1(n_s)$ is a monotonically increasing function. Initially, the age n_s is equal to 1, and the retention rate $\omega_1(n_s)$ is equal to 0, so that $U(\mathbf{s}, \mathbf{e}) = \omega_2(n_s)M(\mathbf{e})\mathbf{e}$, which means that no information in the synaptic vector $\mathbf{s}$ will remain in the updated synaptic vector $\mathbf{s}'$ of the neuron s at its first update. When the age n_s increases, the retention rate $\omega_1(n_s)$ is no longer equal to 0, so that the first term $\omega_1(n_s)\mathbf{s}$ in (2) is not zero, which means that some information in the synaptic vector $\mathbf{s}$ will remain in the updated synaptic vector $\mathbf{s}'$ of the neuron s. When the age n_s becomes a large number, the retention rate $\omega_1(n_s)$ will be approaching 1, leading to the fact that almost all information in the synaptic vector $\mathbf{s}$ will remain in the updated synaptic vector $\mathbf{s}'$ of the neuron s.

The learning rate $\omega_2(n_s)$ is a monotonically decreasing function. When $n_s = 1$, the learning rate $\omega_2(n_s)$ is equal to 1, which means the updated synaptic vector $\mathbf{s}'$ of the neuron s will accept much information of the input vector $\mathbf{e}$. When the age n_s becomes a large number, the learning rate $\omega_2(n_s)$ will be approaching 0, which means the updated synaptic vector $\mathbf{s}'$ of the neuron s will accept little new information from the current input vector $\mathbf{e}$ at the update.

Thus, the effect of the updating procedure $U(\mathbf{s}, \mathbf{e})$ depends on the age n_s of the neuron s that the synaptic vector $\mathbf{s}$ belongs to. When the age n_s becomes infinite, we have

$$\lim_{n_s \to \infty} \left[\omega_1 (n_s)\, \mathbf{s} + \omega_2 (n_s)\, M (\mathbf{e})\, \mathbf{e}\right] = \mathbf{s}. \tag{3}$$

This means that the resulting vector $\mathbf{s}'$ of the updating procedure $U(\mathbf{s}, \mathbf{e})$ will be almost the same as the synaptic vector $\mathbf{s}$ when the age n_s becomes a large number. For this reason, we say that the neuron s is *mature* when its age n_s becomes large enough.

2.4. Formation of Algorithmic Concepts. Each program e in the environment E is an instance of some algorithmic concept. For example, the program in Figure 1(a) is an instance of the algorithmic concept expressed in Figure 1(e). Initially, the brain B has no idea of any algorithmic concept, so that the set C of the developed concepts is empty. All the synaptic vectors in the internal state S are initialized with random values to represent the initial state S_0 at the time $t = 0$, which are evidently independent of the environment E.

Thus, the brain B has no idea of any instance of an algorithmic concept F (e.g., the one expressed in Figure 1(e)) when it encounters the first instance f_1 (e.g., in Figure 1(a)) of the algorithmic concept F at the time $t = t_1$. However, the brain B will remember some algorithmic information of the instance f_1 by updating the synaptic vector $\mathbf{s}_F^{(0)}$ of some neuron s_F with the input vector $\mathbf{e}^{(1)}$ that represents the algorithmic information of the instance f_1. The synaptic vector $\mathbf{s}_F^{(0)}$ is the closest one to the input vector $\mathbf{e}^{(1)}$ in comparison with other synaptic vectors in the internal state S, so that the procedure $R(\mathbf{e}^{(1)})$ will recall the neuron s_F; that is, $\mathbf{s}_F^{(0)} = R(\mathbf{e}^{(1)})$. Because this is the first time that the neuron s_F is recalled to update its synaptic vector (i.e., its age is 1, $\omega_1(1) = 0$, and $\omega_2(1) = 1$), we have the following result $\mathbf{s}_F^{(1)}$ of the updating procedure $U(\mathbf{s}_F^{(0)}, \mathbf{e}^{(1)})$ by (2):

$$\mathbf{s}_F^{(1)} = M\left(\mathbf{e}^{(1)}\right)\mathbf{e}^{(1)}, \tag{4}$$

where $M(\mathbf{e}^{(1)})$ is the response of the synaptic vector $\mathbf{s}_F^{(0)}$. This means that the updated synaptic vector $\mathbf{s}_F^{(1)}$ of the neuron s_F will contain some algorithmic information of the instance f_1 from the input vector $\mathbf{e}^{(1)}$. This process is equivalent to $S \leftarrow (S \setminus \{\mathbf{s}_F^{(0)}\}) \cup \{\mathbf{s}_F^{(1)}\}$. The age of the neuron s_F is increased by one before the next update of its synaptic vector.

When the second instance f_2 (e.g., in Figure 1(b)) of the algorithmic concept F arrives at the time $t = t_2$ ($t_2 > t_1$), the brain B will not think the instance f_2 so strange since the synaptic vector $\mathbf{s}_F^{(1)}$ of the neuron s_F contains some algorithmic information of the first instance f_1 which is similar to the second instance f_2. The brain B will recall the same neuron s_F again, because its current synaptic vector $\mathbf{s}_F^{(1)}$ is the most similar to the input vector $\mathbf{e}^{(2)}$ that represents the algorithmic information of the instance f_2; that is, $\mathbf{s}_F^{(1)} = R(\mathbf{e}^{(2)})$. Thus, the synaptic vector of the neuron s_F will be updated again, and its age becomes 2. By (2) and (4), we have the following result $\mathbf{s}_F^{(2)}$ of the updating procedure $U(\mathbf{s}_F^{(1)}, \mathbf{e}^{(2)})$:

$$\mathbf{s}_F^{(2)} = \omega_1 (2)\, M\left(\mathbf{e}^{(1)}\right)\mathbf{e}^{(1)} + \omega_2 (2)\, M\left(\mathbf{e}^{(2)}\right)\mathbf{e}^{(2)}. \tag{5}$$

Because $\omega_1(2) \neq 0$ and $\omega_2(2) \neq 0$ in (5), the newly updated synaptic vector $\mathbf{s}_F^{(2)}$ of the neuron s_F will contain algorithmic information of both instances f_1 and f_2 from vectors $\mathbf{e}^{(1)}$ and $\mathbf{e}^{(2)}$, respectively. At this time, the neuron s_F stands for both instances f_1 and f_2 rather than just for the first instance f_1. This means that the brain B becomes more familiar with the algorithmic concept F.

For the reasons above, the same neuron s_F in the brain B will be recalled to update its synaptic vector whenever an instance of the algorithmic concept F arrives, leading to its synaptic vector containing more and more information about the algorithmic concept F. When the ith instance f_i of the algorithmic concept F arrives at the time $t = t_i$ ($i = 3, 4, 5, \ldots$), the neuron s_F will be recalled to update its synaptic vector for the ith time. Obviously, the number i is actually the age of the neuron s_F at the time t_i. By (2), the result $\mathbf{s}_F^{(i)}$ of the ith update $U(\mathbf{s}_F^{(i-1)}, \mathbf{e}^{(i)})$ is obtained as follows:

$$\mathbf{s}_F^{(i)} = \omega_1 (i)\, \mathbf{s}_F^{(i-1)} + \omega_2 (i)\, M\left(\mathbf{e}^{(i)}\right)\mathbf{e}^{(i)}, \tag{6}$$

where $\mathbf{s}_F^{(i)}$ denotes the ith-updated synaptic vector of the neuron s_F and $\mathbf{e}^{(i)}$ denotes the input vector that represents the algorithmic information of the ith instance f_i of the algorithmic concept F. By inspecting (6), we can conclude that the synaptic vector of the neuron s_F will be updated greatly (i.e., great difference between $\mathbf{s}_F^{(i-1)}$ and $\mathbf{s}_F^{(i)}$) when the age i is small and slightly (i.e., slight difference between $\mathbf{s}_F^{(i-1)}$ and $\mathbf{s}_F^{(i)}$) when the age i becomes large, because $\omega_1(i)$ is an increasing function from 0 to 1 whereas $\omega_2(i)$ is a decreasing function from 1 to 0. As the age i becomes large enough, the synaptic vector of the neuron s_F will be almost unchanged.

On the other hand, a large age i means that the synaptic vector $\mathbf{s}_F^{(i)}$ contains algorithmic information from a large number of instances of the algorithmic concept F. The larger the age i is, the more instances of F the neuron s_F represents. Finally, the neuron s_F represents almost all instances of the algorithmic concept F when its age i is greater than a very large number. At this time, we think that the synaptic vector $\mathbf{s}_F^{(i)}$ is developed into a representation for the algorithmic concept F (i.e., an algorithmic concept is formed in the brain B). Thus, the neuron s_F is collected as a developed concept, that is, $C \leftarrow C \cup \{s_F\}$, when its age i is greater than a threshold which is a very large number.

From the concept formation above, it can be seen that the developed concept stands for an idea shared by a group of programs that have similar concrete algorithms. In other words, the developed concept represents a common idea that programmers have when they are writing similar programs. Thus, it is helpful to apply the developed concept to program understanding. Note that the developmental process of algorithmic concepts is unsupervised. This feature is desirable for understanding what algorithmic concepts are employed by

the programs that come from online judge systems in the Internet.

3. Representation for Algorithmic Information

3.1. Algorithmic Signatures. An algorithm is a finite list for calculating a function [20]. There are many kinds of notation to express an algorithm, such as natural languages, flowcharts, pseudocode, and problem analysis diagrams. In Figure 1(c), for example, a flowchart describes a concrete algorithm. From the flowchart, we can find some algorithmic information, described as an *algorithmic signature* in Figure 1(g). This algorithmic signature consists of a control flow statement *while* and three noncontrolling language points *array*, *input*, and *output*. It characterizes the concrete algorithm of the program in Figure 1(a).

Generally, an algorithmic signature consists of several units. Each unit contains two parts: a control flow statement and its relevant language points. The control flow statements are the most important components, because they decide the control flow of an algorithm. There are four control flow statements: *while*, *for*, *if*, and *switch*. The noncontrolling language points in a control flow statement are regarded as relevant to the control flow statement. The algorithmic signature in Figure 1(g) has only one unit, where there are three relevant language points in the control flow statement *while*.

There are two relations between the units: sequence relation and nesting relation. The program in Figure 2(a) has four control flow statements, so that its algorithmic signature has four units as shown in Figure 2(b). The first unit consists of a control flow statement *while* with its three relevant language points *equal to*, *input*, and *output*. This first unit has three nesting relations to the three other units, respectively. The pair "{" and "}" of the *while* statement shows the nesting relation. The three units nested in the first unit have sequence relations with each other, arranged from up to down. They all have one and the same relevant language point *greater than*.

The algorithmic signature of each program can be generated from the parse tree of its source code. It is not difficult to identify control flow statements, noncontrolling language points, and their relationships in the parse tree. Every program can be converted into its algorithmic signature by a depth-first traversal of its parse tree.

3.2. Matrixes for Algorithmic Signatures. An algorithmic signature can be represented by a $d \times d$ matrix. Each row of the matrix may consist of a controlling number followed by noncontrolling numbers to represent a unit in the algorithmic signature. The controlling number denotes the control flow statement in the unit, whereas the noncontrolling numbers indicate whether there are some noncontrolling language points in the unit or not.

The first row of the matrix in Figure 1(h) represents the unit of the algorithmic signature in Figure 1(g). The rest rows are all filled with zeros, meaning that there are no more units in this algorithmic signature. The first number 80 in the first row is a controlling number, which indicates that the control flow statement of the unit is a *while* statement. Following the controlling number are all noncontrolling numbers, where the three 1s indicate that there are three noncontrolling language points *array*, *input*, and *output*, respectively, in the control flow statement.

The matrix in Figure 2(c) represents the algorithmic signature in Figure 2(b). The top four rows represent the four units of the algorithmic signature, respectively. Because the second unit is nested in the first unit, all numbers except the last in the second row move right a position, whereas the last number moves onto the first position. Thus, the controlling number of the second row is in the second position from the left. The number 190 indicates that the control flow statement of the second unit is an *if* statement. The second row and the third row are the same because the second unit and the third unit are the same and they have a sequence relation to each other.

Each position for a noncontrolling number in a row is associated with a noncontrolling language point. Positions from the second to the sixth in the first row of the two matrixes above are associated with language points *greater than*, *equal to*, *array*, *input*, and *output*, respectively. There are only two values 0 and 1 for a noncontrolling number. The value 1 of a noncontrolling number indicates that its associated language point exists in the unit of the algorithmic signature, whereas the value 0 indicates that its associated language point does not exist in that unit.

The controlling number is designed to be greater than the noncontrolling number. For the controlling number, there are four values 80, 110, 190, and 220, which denote the four control flow statements *while*, *for*, *if*, and *switch*, respectively. Note that the two controlling numbers for *while* and *for* have a smaller difference because both of them are iteration statements.

3.3. Signatures of Developed Concepts. Figure 2(d) shows the procedure from program source codes to signatures of developed concepts. The program source code (e.g., in Figure 1(a)) is converted into its algorithmic signature (e.g., in Figure 1(g)) through its parse tree. The algorithmic signature will be converted into a matrix (e.g., in Figure 1(h)). The matrix will be converted into the input vector of the brain B which is based on LCA. From the output of the brain B are vectors which stand for developed concepts.

The relationship between a matrix and its corresponding vector is shown in Figure 2(e). The first column of the matrix maps into the first n components of the vector, where n is the number of rows in the matrix. The second column of the matrix maps into the second n components of the vector, and so on. In this way, we can convert the matrix of an algorithmic signature into its corresponding vector as an input to the brain B. Reversely, the first n components of the vector map into the first column of the matrix, the second n components of the vector map into the second column of the matrix, and so on. In this way, we can convert the developed vectors from the output of the brain B into their corresponding matrixes.

Figure 2(f) shows a matrix which is derived from a developed concept. It is a simplified version of a matrix

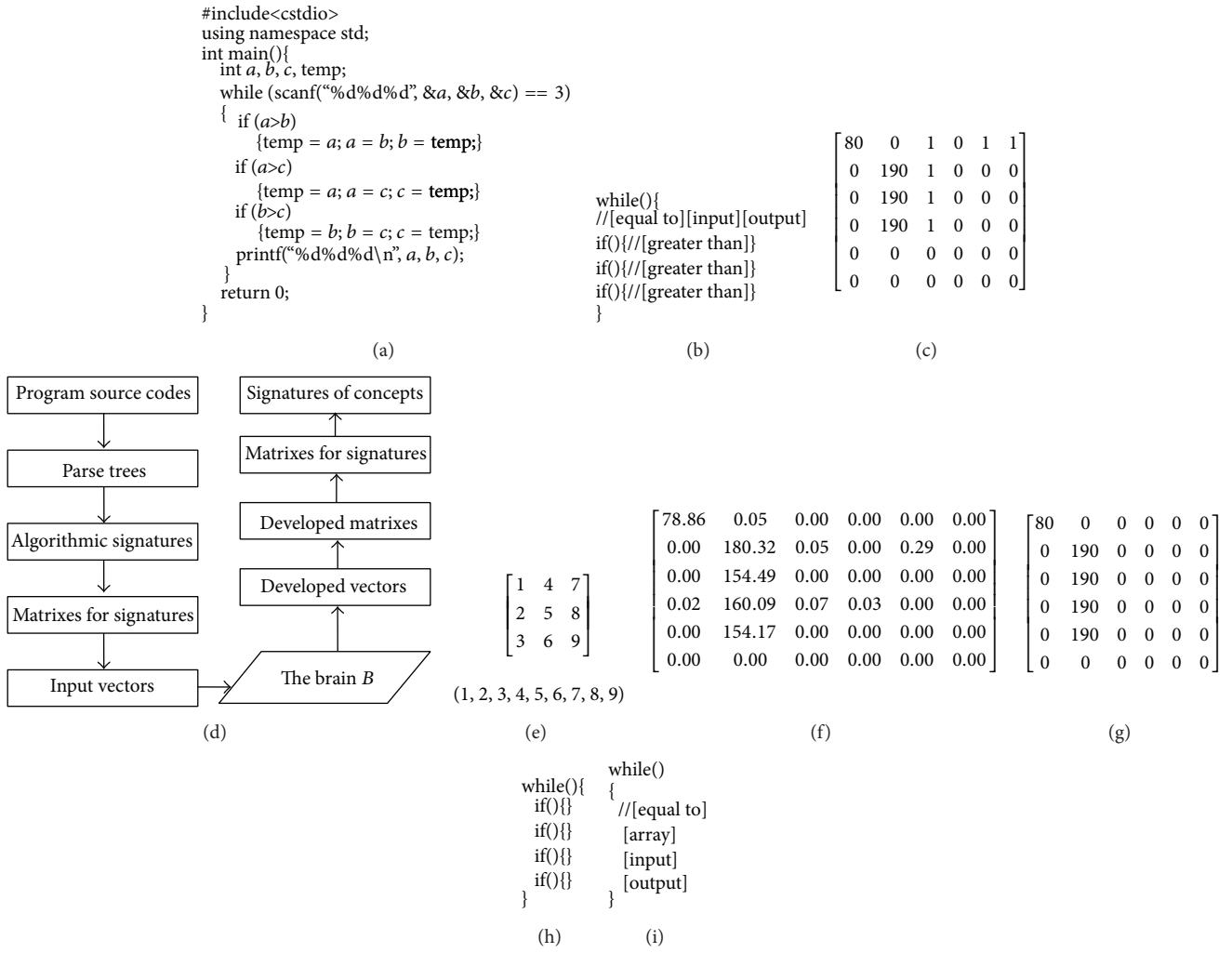


Figure 2: (a) A C++ program which has four control flow statements. (b) The algorithmic signature of the program in (a). (c) A matrix for the algorithmic signature in (b). (d) From program source codes to signatures of developed concepts. (e) The relationship between a matrix and its corresponding vector. (f) A simplified version of a developed matrix from our experimental results. (g) This matrix is generated from one in (f). (h) This signature is produced from the matrix in (g), which represents a developed concept. (i) This signature characterizes the algorithmic concept shared by both programs in Figures 1(a) and 1(b).

from our experimental results for convenience in discussion. We can see that the maximum number in the first row of the matrix is 78.86. We suppose that 2 is the threshold to determine whether a row represents a unit or not. Because the maximum number 78.86 is greater than the threshold number 2, we can treat this row as a unit. In the same way, we find four other units, whose maximum numbers are 180.32, 154.49, 160.09, and 154.17, respectively.

In order to get signatures of developed concepts, these maximum numbers should be replaced by their corresponding controlling numbers. We replace all maximum numbers with their closest controlling numbers. For example, we replace the maximum number 78.86 by the controlling number 80 because 80 is the controlling number closest to the maximum number 78.86. For the same reason, the maximum numbers 180.32, 154.49, 160.09, and 154.17 are replaced by the controlling number 190, respectively.

The next step is to determine noncontrolling numbers in a unit. Each nonmaximum number (e.g., 0.05 in the first row) in a unit should be replaced by a noncontrolling number. We supposed that 0.5 is the threshold: if a number is greater than 0.5, we replace it with 1, otherwise 0. Because all the five nonmaximum numbers in the first row are less than 0.5, they are replaced with 0, respectively. We do the same thing for the other four rows. Figure 2(g) shows the result of the conversion.

With these controlling numbers and noncontrolling numbers, we can convert the matrix into a signature (e.g., in Figure 2(h)). This signature is not an algorithmic signature of a program although both of them have the same format. The former characterizes a developed concept which is employed by a group of programs, whereas the latter only represents the concrete algorithm of one program. For example, the signature in Figure 2(i) characterizes the developed concept shared

by a group including the two programs in Figures 1(a) and 1(b), whereas the algorithmic signature in Figure 1(g) represents the concrete algorithm of the program in Figure 1(a) only.

4. Experiment and Results

In this experiment, the brain B is composed of a one-layer network of 20×20 neurons, each of which has a 900-dimensional synaptic vector. For the updating procedure $U(\mathbf{s}, \mathbf{e})$ in (2), the retention rate is defined as $\omega_1(n_s) = (n_s - 1 - \mu(n_s))/n_s$ and the learning rate $\omega_2(n_s) = (1 + \mu(n_s))/n_s$, where $\mu(n_s)$ is an amnesic function as follows [19]:

$$\mu(n_s) = \begin{cases} 0, & \text{if } n_s \le t_a \\ \dfrac{c(n_s - t_a)}{(t_b - t_a)}, & \text{if } t_a < n_s \le t_b \\ c + \dfrac{(n_s - t_b)}{m}, & \text{if } n_s > t_b. \end{cases} \quad (7)$$

The parameters in $\mu(n_s)$ were set as follows: $t_a = 20$, $t_b = 200$, $c = 2$, and $m = 5000$.

After a developing phase for concept formation, the same brain B with all its 400 updated synaptic vectors was applied to a concept assignment task in program comprehension for an evaluation of the developed concepts.

4.1. Experimental Data. The environment E is composed of 2341 C++ program source codes from an online judge system submitted by sixty college students for solving sixty simple problems. All of these program source codes were judged correct by the online judge system. The number of control flow statements is not greater than ten in each of these program source codes. The nesting level of control flow statements is not greater than six.

In the developing phase, each of these 2341 program source codes was chosen randomly 300 times to form the environmental sequence E_t. It means that every program source code would occur 300 times randomly in the environmental sequence E_t and the length of the sequence is 2341×300. During the test for the evaluation of the developed concepts, however, each of these 2341 program source codes was chosen randomly only once for the concept assignment task.

Each program source code was converted into a 30×30 matrix which represents the algorithmic signature of the program source code. In the algorithmic signature, each unit is composed of a control flow statement and at most 25 relevant language points. Because the control flow statement is more important than its relevant language points, its controlling number was allocated to occupy five adjacent positions in a row of the matrix, whereas each noncontrolling number was allocated to occupy only one position for a corresponding language point. The first five positions in the first row of a matrix, for example, are all filled with the same value of the controlling number that represents the control flow statement of the first unit in an algorithmic signature. The other 25 positions for noncontrolling language points in the first row are listed in the Table 1, where all positions in a row are numbered $1, 2, \ldots, 30$ from the left to the right.

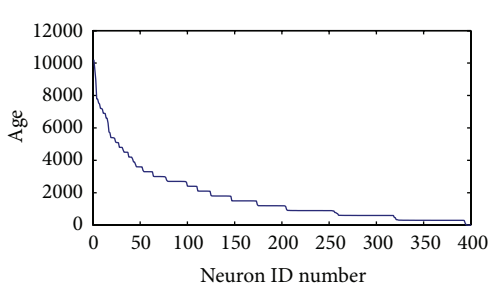
Table 1: Positions for noncontrolling language points in the first row of a matrix.

Position	Language point (node in the parse tree)
6	Assignment expression
7	Addition
8	Subtraction
9	Multiplication
10	Division
11	Modulus
12	Logical and
13	Logical or
14	Logical negation
15	Greater than
16	Less than
17	Greater than or equal to
18	Less than or equal to
19	Equal to
20	Not equal to
21	Array
22	Variable declarator
23	Variable
24	Decimal integer constant
25	Octal integer constant
26	Return statement
27	Continue statement
28	Break statement
29	Input
30	Output

Controlling numbers were designed to be greater than noncontrolling numbers for distinguishing the control flow statement from noncontrolling language points. The values of controlling numbers are 80, 110, 190, and 220 to denote the four control flow statements *while*, *for*, *if*, and *switch*, respectively. However, each noncontrolling number has only two values: 1 for existence of the corresponding language point in a unit and 0 for nonexistence. It should be pointed out that numbers in a nesting unit should move right five positions but numbers in a sequence unit should not (see Section 3.2).

Finally, each matrix was converted into a 900-dimensional vector as an input vector to the brain B.

4.2. Developmental Results. Figure 3(a) shows the ages of all the 400 neurons in the brain B at the end of the developing phase. The neurons are numbered $1, 2, \ldots, 400$ in the descending order of their ages. Those whose ages are greater than or equal to 3000 are regarded as mature enough to represent algorithmic concepts. We found 78 mature neurons and put them into the set C of developed concepts. They are numbered $1, 2, \ldots, 78$ in the same order as theirs in Figure 3(a).



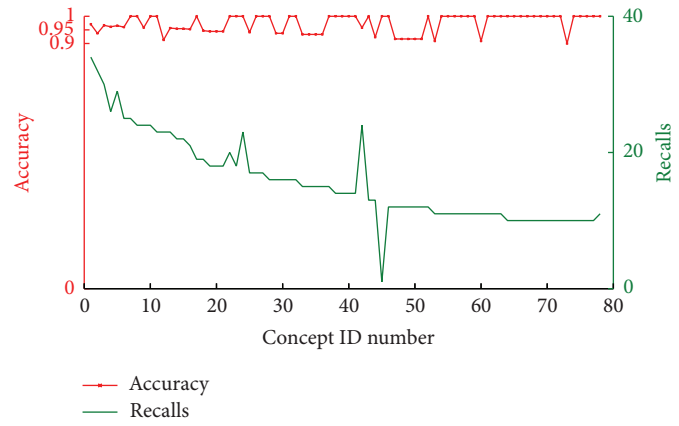
(a)

78.86	78.86	78.86	78.86	78.86	0.05	0.00	0.00	0.00	0.00
0.05	0.00	0.29	0.00	0.00	180.32	180.32	180.32	180.32	180.32
0.00	0.00	0.00	0.00	0.00	154.49	154.49	154.49	154.49	154.49
0.07	0.03	0.00	0.00	0.02	146.78	146.78	160.09	160.09	160.09
0.00	0.00	0.00	0.00	0.00	154.17	154.17	160.09	160.09	160.09
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

(b)

```
while( ) {
    //[variable]
    if( ){
    //[logical and] [equal to] [variable]
      [decimal integer constant] }
    if( ){
    //[assignment expression] [less than] [variable]
      [decimal integer constant] }
    if( ){
    //[assignment expression] [addition]
      [less than] [variable] [decimal integer constant] }
    if( ){
    //[assignment expression][addition]
      [less than] [variable] [decimal integer constant] }
}
```

(c)



(d)

Figure 3: (a) It shows the ages of all the 400 neurons at the end of the developing phase. The neurons are numbered $1, 2, \ldots, 400$ in the decreasing order of their ages. (b) This is a part of the matrix derived from the first developed synaptic vector in (a). The corresponding controlling numbers occupy five adjacent positions in every unit. (c) The signature of the first developed concept. (d) The concept assignment accuracy of each developed concept (indicated by a series of "×" scaled by the left vertical axis) and the recalls that each developed concept has (indicated by the continuous curve scaled by the right vertical axis).

The matrix of the first developed concept (i.e., numbered 1 in the set C) is shown partially in Figure 3(b), which presents the top left part of the matrix. This matrix implies a five-unit signature because each of the top five rows has numbers that are greater than the threshold number 2. In each of the top five rows, there are five numbers in five adjacent positions that are greater than 2. This means that the controlling number occupies five adjacent positions in each of these rows. The value of the controlling number is the one closest to the average of these five numbers. For example, the average of the five numbers in the first row is 78.86. Among the four values 80, 110, 190, and 220 for controlling numbers, the value 80 is the closest to the average 78.86. Thus, the controlling number of the first unit is 80, which means that a *while* statement is the control flow statement of the unit. In a similar way, we can know that the control flow statements of the other four units are all an *if* statement. All these four units are nested in the first unite because their first numbers (from the left) that are greater than the threshold number 2 are in the sixth column, which mean their corresponding controlling numbers are five positions right to the first row, indicating their nesting relations with the first unit. Moreover, these four units have sequence relations with each other because their corresponding controlling numbers are in the same columns.

Figure 3(c) shows the signature of the first developed concept. The four *if* statements are presented within the pair "{" and "}" of the *while* statement, indicating that their units are nested in the first unit. In addition, these four statements are presented from up to down, indicating that their units have sequence relations with each other. Following the symbol "//" are the noncontrolling language points relevant to their corresponding control flow statement. It can be seen, for example, that the second unit has four relevant language points *logical and*, *equal to*, *variable*, and *decimal integer constant* because they follow the symbol "//" within the pair "{" and "}" of its control flow statement *if*. This signature

Table 2: The fourth developed concept had twenty-six associated programs.

Problem ID	Number of programs
1	17
2	8
60	1

is readable in some sense, from which we can see that the first developed concept refers to four actions, each of which may or may not be executed according to its corresponding condition in every loop.

4.3. Concept Assignment Results. To test the developed concepts for evaluation, we stopped the development of the brain B by disabling its updating procedure $U(\mathbf{s}, \mathbf{e})$. When receiving an input vector $\mathbf{e}$ that represents a program e in E, the brain B would still activate its recalling procedure $R(\mathbf{e})$ and recall a neuron s such that its synaptic vector $\mathbf{s} = R(\mathbf{e})$. The recalled neuron s would be assigned to (or associated with) the program e if the recalled neuron s was in the set C of developed concepts (i.e., s was a developed concept), which means that the program e is supposed to perform the abstract algorithm that the developed concept s represents.

In this way, the 78 developed concepts in the set C were associated with 1229 out of the total 2341 program source codes in the environment E. This proportion is 52.50% in all the program source codes. The continuous curve scaled by the right vertical axis in Figure 3(d) shows the number of program source codes that each developed concept was associated with. 77 developed concepts (i.e., 98.72% of the 78 developed concepts) have 10 or more recalls (associated program source codes). Only one has less than 10 recalls; that is, the 45th developed concept has only one associated program source code. By comparing the matrix of the 45th developed concept with the matrix of the 42nd developed concept, we found that they were almost the same. This probably led to the fact that some program source codes which should be assigned to the 45th developed concept were wrongly assigned to the 42nd developed concept, resulting in the fact that the 42nd developed concept had a great number of recalls whereas the 45th developed concept had only one recall.

4.4. Evaluation of Developed Concepts. We found by inspection that program source codes which are assigned to the same developed concept always perform the same abstract algorithm if they solve the same problem. Table 2 shows that the fourth developed concept had twenty-six associated program source codes. Among them, seventeen solve problem 1, eight solve problem 2, and one solves problem 60. By inspecting these programs one by one, we found that all the programs that solve problem 1 perform the same abstract algorithm and that all the programs that solve problem 2 also perform the same abstract algorithm. Moreover, we found that the abstract algorithms of these two groups of programs are the same. However, we also found that the program that solves problem 60 performs a different abstract algorithm. We think that this program had a wrong concept assignment and that the other twenty-five programs had their correct concept assignments. Thus, the fourth developed concept has its assignment accuracy of $25/26 \approx 96.15\%$.

Table 3: The accuracy and recalls of the first ten developed concepts.

ID number	Recalls	Accuracy
1	34	97.06%
2	32	93.75%
3	30	96.67%
4	26	96.15%
5	29	96.55%
6	25	96.00%
7	25	100.00%
8	24	100.00%
9	24	95.83%
10	24	100.00%

Table 3 lists the concept assignment accuracies of the first ten developed concepts as well as their recalls. The series of "×" scaled by the left vertical axis in Figure 3(d) shows the assignment accuracy of each developed concept. Out of the 1229 program source codes that were associated with the 78 developed concepts, 1194 had their correct concept assignments. The overall accuracy of these concept assignments is $1194/1229 \approx 97.15\%$.

5. Conclusion

An algorithmic concept is an idea behind a group of programs that have similar concrete algorithms. Human programmers have many algorithmic concepts in their minds, which guide them to design programs. To understand what algorithmic concept is used to design a given program is a muddy task, especially in an uncontrollable environment that consists of program source codes from online judge systems in the Internet.

Our developmental model, inspired by an autonomous development paradigm, aims at challenging this muddy task. Unlike traditional approaches that cannot do anything beyond their predesigned representations, our model is able to develop its internal representation from randomness into algorithmic concepts that are not designed in advance. On a basis of an LCA neural network, the developmental process mimics the recalling procedure and the updating procedure in the human brain. Every program arrival will trigger the recalling procedure to recall a neuron in the neural network. The updating procedure will modify the synaptic weights of the recalled neuron with some algorithmic information of the program that triggers the recalling procedure. The algorithmic information of a program is characterized as an algorithmic signature which will be converted into a vector as an input to the neural network. After a neuron has been recalled to update its synaptic weights for enough times, its synaptic weights become a representation of an algorithmic

concept shared by the programs that trigger the recalls of the same neuron.

Such developed concepts are applicable to an algorithmic concept assignment task for program comprehension under the uncontrollable environment described above. In our experiment, 78 developed concepts were grown up under an environment consisting of 2341 simple programs that come from an online judge system. During an algorithmic concept assignment task, 1229 out of the 2341 simple programs were associated with the 78 developed concepts with an overall concept assignment accuracy of 97.15%. Moreover, each of developed concepts can be converted into its signature which is readable in some sense, so that a human could understand what algorithm its associated programs perform.

Since the initial state of the internal representation has no relation with any algorithmic concept, we believe that the principle of our model can also be applied to the development of concepts in other areas.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

This research is supported by the National Natural Science Foundation of China (NSFC) under Grant no. 60973121.

References

[1] J. Y. Weng, J. McClelland, A. Pentland et al., "Autonomous mental development by robots and animals," *Science*, vol. 291, no. 5504, pp. 599–600, 2001.

[2] A. von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995.

[3] S. Letovsky, "Cognitive processes in program comprehension," *The Journal of Systems and Software*, vol. 7, no. 4, pp. 325–339, 1987.

[4] S. Duszynski, J. Knodel, and M. Becker, "Analyzing the source code of multiple software variants for reuse potential," in *Proceedings of the 18th Working Conference on Reverse Engineering (WCRE '11)*, pp. 303–307, Limerick, Ireland, October 2011.

[5] J.-S. Lim, J.-H. Ji, H.-G. Cho, and G. Woo, "Plagiarism detection among source codes using adaptive local alignment of keywords," in *Proceedings of the 5th International Conference on Ubiquitous Information Management and Communication (ICUIMC '11)*, ACM SIGAPP; ACM SIGKDD; Seoul: Your Complete Convention Center; Ajou University, Seoul, Republic of Korea, 2011.

[6] A. Ohno and H. Murao, "A new similarity measure for in-class source code plagiarism detection," *International Journal of Innovative Computing, Information and Control*, vol. 5, no. 11, pp. 4237–4247, 2009.

[7] C. Alias and D. Barthou, "Algorithm recognition based on demand-driven data-flow analysis," in *Proceedings of the 10th Working Conference on Reverse Engineering*, pp. 296–305, Victoria, Canada, November 2003.

[8] C. Alias and D. Barthou, "On the recognition of algorithm templates," in *Proceedings of the Compiler Optimization Meets Compiler Verification (COCV '03)*, pp. 395–409, Warsaw, Poland, April 2003.

[9] S. Kim and J. H. Kim, "Algorithm recognition for programming tutoring based on flow graph parsing," *Applied Intelligence*, vol. 6, no. 2, pp. 153–164, 1996.

[10] A. Quilici, Q. Yang, and S. Woods, "Applying plan recognition algorithms to program understanding," *Automated Software Engineering*, vol. 5, no. 3, pp. 347–372, 1998.

[11] G. Zhu and X. Zhu, "Autonomous mental development for algorithm recognition," in *Proceeding of the International Conference on Information Science and Technology (ICIST '11)*, pp. 339–347, Nanjing, China, March 2011.

[12] T. Juan and X. Wen, "Intelligent tutoring system based on computing conceptual graphs," in *Proceedings of the International Conference on Artificial Intelligence and Education (ICAIE '10)*, pp. 60–62, Hangzhou, China, October 2010.

[13] M. Storey, "Theories, tools and research methods in program comprehension: past, present and future," *Software Quality Journal*, vol. 14, no. 3, pp. 187–208, 2006.

[14] J. Weng, "Task muddiness, intelligence metrics, and the necessity of autonomous mental development," *Minds and Machines*, vol. 19, no. 1, pp. 93–115, 2009.

[15] S. I. Amari, "Neural theory of association and concept formation," *Biological Cybernetics*, vol. 26, no. 3, pp. 175–185, 1977.

[16] A. Kurnia, A. Lim, and B. Cheang, "Online judge," *Computers & Education*, vol. 36, no. 4, pp. 299–315, 2001.

[17] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, "Program understanding and the concept assignment problem," *Communications of the ACM*, vol. 37, no. 5, pp. 72–82, 1994.

[18] J. Weng and W. Hwang, "From neural networks to the brain: autonomous mental development," *IEEE Computational Intelligence Magazine*, vol. 1, no. 3, pp. 15–31, 2006.

[19] J. Weng and M. Luciw, "Dually optimal neuronal layers: Lobe component analysis," *IEEE Transactions on Autonomous Mental Development*, vol. 1, no. 1, pp. 68–85, 2009.

[20] D. E. Knuth, *The Art of Computer Programming*, Addison-Wesley, Boston, Mass, USA, 3rd edition, 1998.