Automatic Organization of Programming Resources by Neural Computing

Guojin Zhu
Dept. of Computer Science, Donghua University, Shanghai 201620, China
Email: gjzhu.dhu@163.com

Le Liu
Dept. of Computer Science, Donghua University, Shanghai 201620, China
Email:liulefirst@163.com

***Abstract*—There are numerous of programming resources on the Internet, such as programming problems on online judge systems and program codes that solve these problems. Although these resources are valuable for students to practice programming, they are not effectively organized to facilitate students learning. Students and teachers may both hope that all these programming resources are organized as a tutoring sequence. For this purpose, an approach which is based on neural computing is proposed here to organize the programming resources automatically into a tutoring sequence. 2456 source codes were mined in our experiment, resulting in 97 mainstream solutions to 105 programming problems, respectively. These mainstream solutions were sorted by their complexities to form a tutoring sequence which organizes the problems together with their program codes from easy to difficult.**

***Index Terms*—programming resource, data mining, lobe component analysis, self-organizing maps, knowledge representation**

I. INTRODUCTION

There are lots of online judges [1] that provide thousands of problems for students to solve on a purpose of programming practice. This results in abundant program source codes (submitted by students as solutions to the problems) on the Internet. All these problems and their program source codes, however, are not organized effectively for teaching and learning. It is often difficult for students to choose suitable problems to practice. To organize these programming resources, a method [2] based on Formal Conception Analysis (FCA [3]) was proposed to discover the knowledge behind the source codes of the problem solutions. Another method [4] was proposed on a basis of predefined knowledge structures to identify the programming knowledge points in the solution reports. However, they are both based on predefined knowledge bases. Methods that depend on a predefined knowledge base have several disadvantages. For example, it is difficult to determinate how many knowledge points are needed to put into the knowledge base in advance [5].

To address such issue, we propose an approach to organize these programming resources automatically without any predefined knowledge base. The main idea of this approach is to mine the program source codes by neural computing to discover mainstream solutions to programming problems. Moreover, the discovered mainstream solutions are sorted to form a sequence from simple to complex. This sorted sequence, in which each of the programming problems can find its mainstream solution, functions as a tutoring sequence which arranges the programming problems together with their program source codes from easy to difficult.

However, the program source code, a sequence of strings, or a sequence of alphabets, is always trusted as a non-vectorial item. This makes it difficult to mind the program source codes by neural computing, which is often based on vector computation. For this reason, we use a vector, called *code vector*, to present the program source code. The code vector indicates whether or not the abstract syntax tree (AST [6-8]) of the program source code contains some special sub-trees, called *central sub-trees*.

Fig.1 shows the procedure to organize programming problems and their program source codes into a tutoring sequence by the proposed neural computing. The first step is to convert the program source codes into their abstract syntax trees by using the Java Complier Complier (JavaCC [7,8]).

Usually, an AST is composed of several sub-trees. The second step is to cut the ASTs into their sub-trees. We find that most sub-trees are similar to others, so that we try to use a central sub-tree to represent a group of similar sub-trees. In the third step, we introduce a clustering method which is called Self-Organizing Maps (SOM [9-13]) to mine the central sub-trees for groups of similar sub-trees. SOM is a clustering method considered as an unsupervised variation of the artificial neural network.

In the forth step, we construct the code vector of each program code on a basis of the central sub-trees. The number of components of the code vector is equal to the number of the central sub-trees. Each component of the code vector is corresponding to a central sub-tree, and

Corresponding author: Guojin Zhu; Email gjzhu.dhu@163.com.

doi:10.4304/jsw.8.6.1471-1478

vice versa. The value of a component of the code vector is the similarity value between a sub-tree of the program code and the central sub-tree corresponding to the component.

After obtaining the code vectors of program source codes, we could apply Lobe Component Analysis (LCA [14,15]) to mining them to discover the mainstream solutions to the programming problems in the fifth step. LCA is another type of the artificial neural network.

Finally, we generate a tutoring sequence by sorting the discovered the mainstream solutions (represented by vectorial templates) from simple to complex and associating each of the mainstream solutions with its corresponding problems as well as its corresponding group of program source codes.

The rest of the paper is organized as follows. In Section II, we introduce the abstract syntax tree (AST) briefly and present two methods to measure the similarity between ASTs. In Section III, we introduce the concept of the central sub-tree, and explain how to obtain a central sub-tree from similar sub-trees by SOM. In Section IV, we convert source codes into their code vectors on a basis of central sub-trees and mine the solutions to problems from the code vectors by LCA. In Section V, we propose a method to generate a tutoring sequence by associating the problems with the mined solutions. Section VI shows the experiment results which demonstrate that our approach is feasible. Finally, we draw the conclusion for our paper.

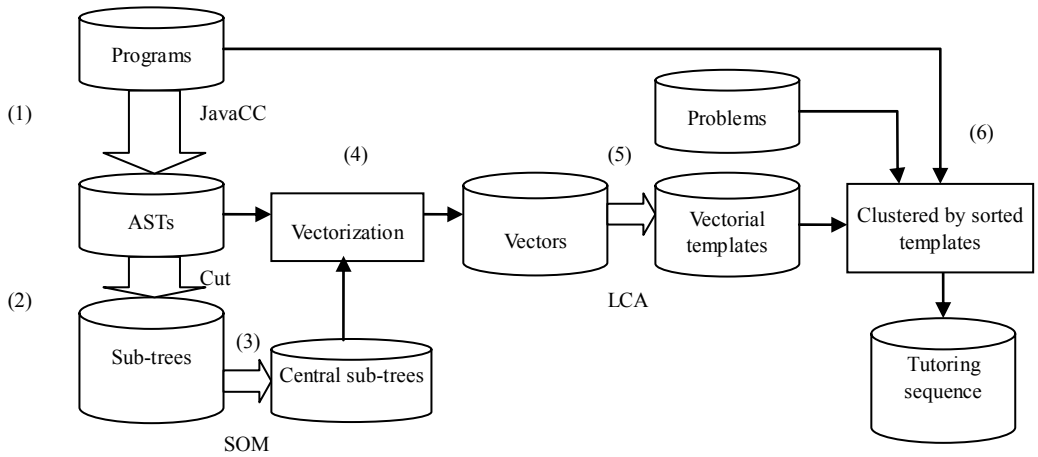

Figure 1. The procedures of organize programming resources by neural computing

II. Abstract Syntax Trees and their Similarities

A. Abstract Syntax Trees

In the field of computer science, an abstract syntax tree (AST), or just a syntax tree, is a tree representation of the abstract syntactic structure of a program source code written in a programming language. An AST is often the output of a parser and it forms the input to semantic analysis and code generation. The Java Complier Complier (JavaCC [7,8]) is a tool of parser-generator written in Java that allows the parser to produce ASTs. Each source code can be converted into its AST. Fig.2 shows an AST of a simple source code (displayed in the up right corner) generated by JavaCC. Each node in the tree represents a constant, variable, operator, or statement.

B. Tree Similarity

We define that the size of a tree is the number of nodes that the tree contains. We define the *tree size similarity* to measure the similarity between two trees.

Given two trees X and Y, the tree size similarity, denoted by $TreeSizeSimilarity(X, Y)$, measures the degree that the tree Y is similar to the tree X in size, which is defined as follows:

$$TreeSizeSimilarity(X,Y)=\frac{Node(X)-|Node(X)-Node(Y)|}{Node(X)} \quad (1)$$

where $Node(T)$ denotes the number of nodes in the tree T.

It is not enough to measure the similarity between two trees only by the tree size similarity. Two trees that have the same size may have different structures. For this reason, we define the similarity between two structures, called *tree structure similarity*. The tree structure similarity is based on the tree edit distance [16] (TED).

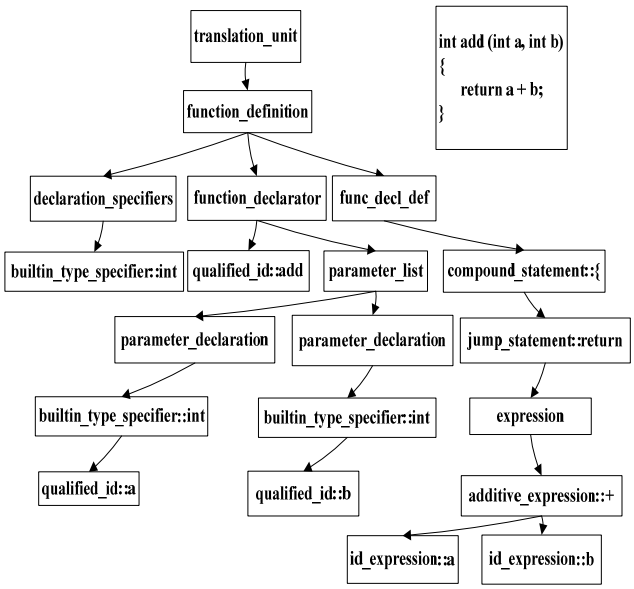

Figure 2. The abstract syntax tree of a simple source code

The tree edit distance is a measurement used to measure the similarity between tree structured data. The tree edit distance is defined as the minimum-cost sequence of node edit operations that transform one tree into another. If T is an ordered tree, these edit operations are defined as follows:

- Rename the label of a node x in T;
- Delete a non-root node x in T with parent x', making the children of x become the children of x';
- Insert a node x as a child of x' in T, making x the parent of a consecutive subsequence of the children of x'.

Given two trees X and Y, the tree structure similarity, denoted by *TreeStructureSimilarity*(X, Y), measures the degree that the tree Y is similar to the tree X in structure. It is defined as follows:

$$TreeStructureSimilarity(X,Y) = \frac{Node(X) - TED(X,Y))}{Node(X)} \qquad (2)$$

where $TED(X, Y)$ is the tree edit distance between X and Y. Obviously, the smaller the tree edit distance $TED(X, Y)$ is, the bigger the tree structure similarity value will be.

III. Mining Central Sub-trees

A. Sub-tree and Central Sub-tree

In computer programming, a block is a section of code which is grouped together. Usually, a block is composed of one or more declarations and statements. Each block implements a special functionality. In an AST, each block is corresponding to a sub-tree. The blocks in C/C++ programming are delimited by curly braces. We cut the AST into its sub-trees such that each of its sub-trees represents a block of the source code.

We find that most sub-trees of ASTs are similar to others, so that we try to use a central sub-tree to represent a group of similar sub-trees. Fig.3 shows three groups of similar sub-trees. They are represented by circles, stars and squares. Each circle represents a sub-tree. The sub-trees that the circles represent are similar to each other. We can see that the circles are clustered together. So are the stars and squares. The circle nearest to the center of the cluster of circles is regarded as the central one. The central circle represents the *central sub-tree* of the similar sub-trees that the circles represent.

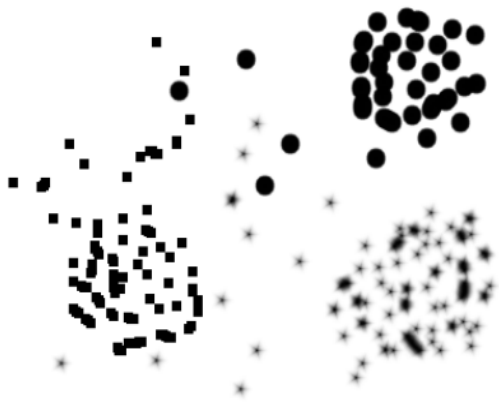

Figure 3. The clusters of sub-trees

B. Self-Organizing Maps

The Self-Organizing Map (SOM) is a clustering method considered as an unsupervised variation of the artificial neural network. Compared with other artificial neural networks, SOM uses neighborhood function [17] to preserve the topological properties of the input space. In maps consisting of thousands of nodes, it is possible to perform cluster operations [18-19] on the map itself.

Let us briefly introduce the idea of Self-Organizing Maps in terms of our trees. An SOM map consists of a group of neurons. Each neuron includes two parts: its corresponding tree on the SOM map, and a list of trees under the neuron. The trees in the list under the neuron are similar to its corresponding tree on the SOM map, so that we think that its corresponding tree on the map represents all the trees under the neuron. The SOM algorithm forms a semantic map where similar trees are mapped closely together and dissimilar ones apart.

These are the variables needed in the SOM algorithm.

1) t is the index of the input tree;
2) $T(t)$ is an input tree;
3) n is a neuron in the map;
4) T_n is the tree corresponding to the neuron n;
5) L_n is the list of trees under the neuron n including the neuron tree T_n;
6) s is the current iteration, and e is the iteration limit;
7) *similarity*(x, y) is the formula to calculate the similarity between the tree x and the tree y;
8) *neighbor*(n) is the neighborhood of the neuron n;
9) *trees*(n) is a set of all trees in the neighborhood of the neuron n.

The SOM algorithm is described as follows.

1) Initialize the map such that each neuron n has a randomly-chosen tree T_n and an empty list L_n of trees under the neuron (i.e., $L_n = \varnothing$).
2) For each input tree $T(t)$, do the following:
   a) For each neuron n in the map, calculate the similarity between the input tree $T(t)$ and the neuron tree T_n by the formula *similarity*(T_n, $T(t)$);
   b) Choose the neuron m whose corresponding tree T_m has the largest similarity value of *similarity*(T_m, $T(t)$). Add the input tree $T(t)$ into the list L_m of trees under the neuron m.
3) For each neuron n, do the following:
   a) Choose the tree M_n from the set *trees*(n) of all trees in the neighborhood *neighbor*(n) such that the tree M_n has the biggest sum S of similarities

$$S = \sum_{T \in trees(n)} similarity(M_n, T) \qquad (3)$$

   b) Replace the neuron tree T_n by the chosen tree M_n.
4) Empty the list L_n of every neuron n, and increase s.

5) Repeat from Step 2) to Step 4) while $s < e$.

Each neuron tree T_n in the map is a central tree which represents the trees in the list L_n at the end of the above algorithm.

C. Obtaining Central Sub-trees by SOM

For a given set of sub-trees, we apply the SOM algorithm in two steps to obtain the central sub-trees, as is shown in Fig.4.

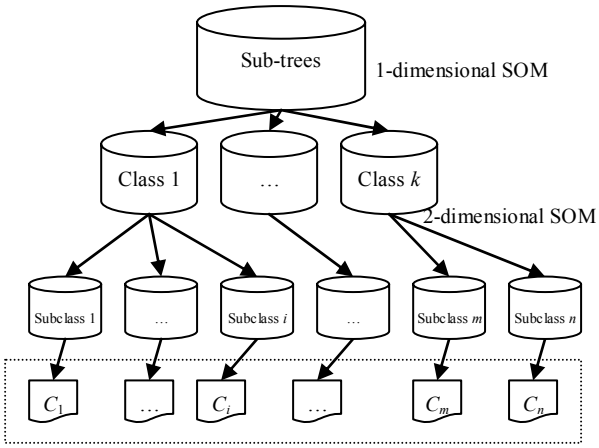

Figure 4. Using SOMs to obtain central sub-trees

We take a 1-dimensional SOM method to divide all sub-trees into several classes. We use the tree size similarity to measure the similarity between two sub-trees, e.g., the similarity between the input sub-tree $T(t)$ and the neuron tree T_n is measured by *TreeSizeSimilarity*(T_n, $T(t)$). For each neuron n, we treat neurons at its left side and right side including itself as its neighborhood *neighbor*(n). When the 1-dimensional SOM operation is finished, the list L_n of sub-trees under each neuron n is a class of sub-trees (e.g., *Class* 1, *Class k* in Fig.4). The sub-trees in the same class are similar in size.

Moreover, each class obtained by the 1-dimensional SOM is further divided into several subclasses by a 2-dimensional SOM method. We use the tree structure similarity to measure the similarity between two sub-trees in the same class, e.g., the similarity between the input sub-tree $T(t)$ and the neuron tree T_n is measured by *TreeStructureSimilarity*(T_n, $T(t)$). For each neuron n, we use Moore neighborhood algorithm [20] to determinate the neurons including itself as its neighborhood *neighbor*(n). When the 2-dimensional SOM operation is finished, the list L_n of sub-trees under each neuron n is a subclass of sub-trees (e.g., *Subclass* 1 and *Subclass m* in Fig.4). The sub-trees in the same subclass are similar not only in size but also in structure. Each neuron tree T_n in the 2-dimensional SOM is a central sub-tree (e.g., C_1 and C_n in Fig.4) at the end of the 2-dimensional SOM operation.

IV. Mining Solutions

A. Code Vectors

Fig.5 shows the procedure to construct the code vector $V = (v_1, v_2 \ldots v_n)$ for a given source code, which is described as follows.

1) Convert the source code to its abstract syntax tree X, and cut the abstract syntax tree X into its sub-trees $x_1, x_2, \ldots, x_k$ such that each of them represents a block of the source code, denoted by $X = \{x_1, x_2 \ldots x_k\}$.

2) For each sub-tree x in X, do the following:

   a) Construct a set $S(x)$ of all central sub-tress that are similar to the sub-tree x in size such that every central sub-tree c in $S(x)$ satisfies *TreeSizeSimilarity*(c, x) > K, where K is between 0.7 and 0.9 (e.g., 0.8).

   b) If the set $S(x)$ is not empty, choose the central sub-tree c_m that is the most similar to the sub-tree x in structure, and set v_m = *TreeStructureSimilarity*(c_m, x), where v_m is the component of the code vector V corresponding to the central sub-tree c_m.

3) Set each of the rest components of the code vector V equal to 0.

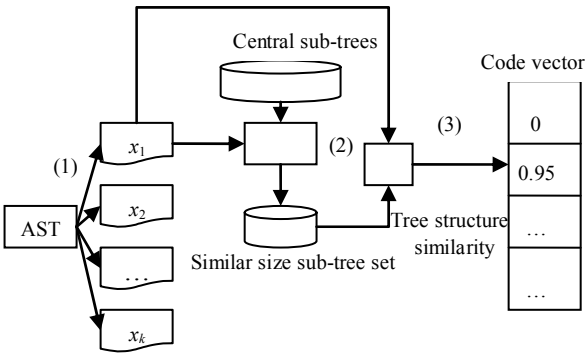

Figure 5. The procedure to construct the code vector

For example, suppose an abstract syntax tree $X = \{x_1, x_2, x_3\}$ and $C = \{c_1, c_2, c_3, c_4, c_5, c_6\}$ be the set of central sub-trees. For the sub-tree x_1, there are two central sub-trees c_2 and c_4 that are similar to it in size (larger than 0.8), i.e., the set $S(x_1) = \{c_2, c_4\}$. The set $S(x_1)$ is not empty. Thus, we choose the central sub-tree c_2 that is the most similar to the sub-tree x_1 in structure, and set the component v_2 of the code vector V corresponding to the central sub-tree c_2 equal to *TreeStructureSimilarity*(c_2, x_1) (e.g.,. $v_2 = 0.98$). For the sub-trees x_2 and x_3, use the same method to generate other two components of the code vector V. In the end, the code vector is obtained, e.g., V = (0, 0.98, 0, 0.92, 0, 0.87). Generally, the number of non-zero components in the code vector is equal to the number of sub-trees in the abstract syntax tree X.

B. Lobe Component Analysis for Mining Solutoins

The Lobe Component Analysis (LCA) is a type of artificial neural network that supports incremental learning. Its idea is to use as little as possible of the vectors to represent the entire space of the data.

Now, let's introduce the principle of LCA briefly. Given a limited resource, LCA divides the sample space R into c mutually non-overlapping regions called *lobe regions*:

$$R = r_1 \cup r_2 \ldots \cup r_c \qquad (4)$$

where $r_i \cap r_j = \varnothing$, if $i \neq j$. Each region r_i is represented by a single unit feature vector called *lobe vector*.

As we have mentioned earlier, each code vector corresponds to a program code that solves a programming problem. The solution to the problem is implemented by the program code. Some program codes may implement the same solution. LCA would divide all code vectors into several mutually non-overlapping lobe regions. Each of the lobe regions is a group of code vectors that implement the same solution. Thus, we think that each lobe vector represents a solution whereas each code vector in the lobe region represents an instance of the solution (i.e., each program code is a solution instance). The lobe vector is called *vectorial template* in Fig. 1.

Each component of the lobe vector is a real number. To make it more clearly to analysis, we convert the lobe vector into a zero-one vector, called *solution vector*. For a given lobe vector $L = (l_1, l_2 \ldots l_k)$, its corresponding solution vector $O = (o_1, o_2 \ldots o_k)$ is such that $o_i = 1$ if l_i is greater than 0, or $o_i = 0$ otherwise. For example, $L = (0.95, 0, 0.9, 0.65, 0)$, its solution vector $O = (1, 0, 1, 1, 0)$. It is easy to see that the more ones in the solution vector, the more complexity the solution is.

C. Mainstream Solutions

Some solutions have many instances (i.e., code vectors) in their corresponding lobe regions. Others have few instances. The more instances in its corresponding region, the more frequently used the solution is.

TABLE I
SOLUTIONS TO PROBLEMS

Solution	Problems
s_1	**110-5(15)**
s_2	121-2(4), **110-5(2)**
s_3	131-3(16)

The second column indicates the information of the problems in the solution. For example, 110-5(15) indicates that the problem whose ID number is 110-5 has 15 source codes in the solution s_1.

A problem may several solutions, among which the one that is used most frequently to solve the problem is called the *mainstream solution* of the problem. For example, suppose there are three solutions s_1, s_2 and s_3 obtained by LCA as shown in Table I. The solution s_1 can be used to solve a problem whose ID number is 110-5. The solution s_2 can be used to solve two problems whose ID numbers are 121-2 and 110-5, respectively. The solution s_3 can be to solve a problem whose ID number is 131-3. For the problem "110-5", there are two solutions s_1 and s_2 that can be used to solve it. However, the problem "110-5" has 15 code vectors in the solution s_1 and 2 code vectors in the solution s_2. So we treat the solution s_1 as the mainstream solution of the problem "110-5".

Below are the variables needed in the algorithm for finding mainstream solutions from the solutions obtained by LCA.

1) $P = \{p_1, p_2, \ldots, p_n\}$ is a set of problems, where n is the number of problems;
2) $S = \{s_1, s_2, \ldots, s_k\}$ is a set of solutions, where k is the number of solutions;
3) N is a temporary set that used to store the data.

The algorithm to discover the mainstream solutions is described as follows.

For each problem p_i in the set P, do the following:

1) Empty the set N, i.e., $N \leftarrow \varnothing$;
2) Select from the set S all the solutions that can solve the problem p_i, and put them into the set N;
3) For each solution in the set N, calculate the number of code vectors of the problem p_i;
4) Choose a solution s_x from the set N that has the largest number of code vectors of the problem p_i as the mainstream solution of the problem p_i.

Each problem will find its mainstream solution by the algorithm above. Moreover, a mainstream may solve more than one problem.

V. Organizing Program Source Codes

In this section, we present a method to generate the tutoring sequence according to the mainstream solutions and their corresponding problems.

These are the variables and operations needed in the method.

1) *mask* is a variable of a zero-one vector, initialized all ones, i.e., *mask* = (1, 1, … ,1).
2) & denotes the bitwise AND operation of two zero-one vectors.
3) * denotes the dot product operation of two vectors.
4) ~ denotes the reverse operation for a given zero-one vector.
5) $S = \{s_1, s_2, \ldots, s_m\}$ is the set of mainstream solutions
6) $O = \{o_1, o_2 \ldots o_m\}$ is the set of the solution vectors. Each solution vector o_i in the set O corresponds to a solution s_i in the set S.
7) *SP* is the set that are used to store the temporary solutions
8) *Sequence* is the list to store the sorted solutions.

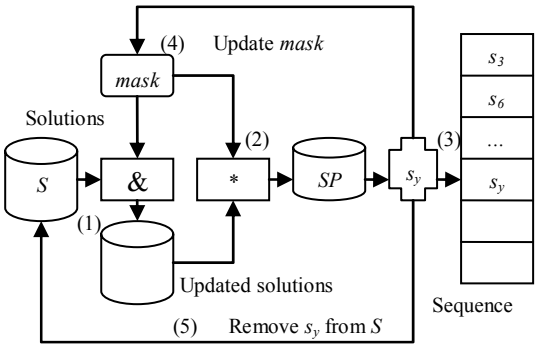

Figure 6. The steps to generate the tutoring sequence

Fig.6 depicts the algorithm to generate a tutoring sequence, which is described as follows.

1) For each solution s_i in the set S, update its solution vector o_i by $o_i \leftarrow o_i$ & *mask*.
2) For each solution s_i in the set S, calculate the dot product of the vector variable *mask* and its solution vector o_i by $x_i = mask * o_i$ (x_i is an integer). Choose the

solutions which have the smallest value of the dot product, and add these solutions into the set *SP*.

3) Choose a solution s_y from the set *SP* that has more instances than each of others in the set *SP* has, and append the solution s_y at the end of the list *Sequence*.

4) Update the vector variable *mask* by $mask \leftarrow \sim o_y$.

5) Remove the solution s_y from the set *S*, and empty the set *SP*.

6) Repeat from Step 1) to Step 5) until $S = \Phi$.

By applying the algorithm above, we can get a sequence of mainstream solutions in the list *Sequence* sorted from simple to complex. A tutoring sequence is formed by associating each problem with its mainstream solution in the list *Sequence*.

TABLE II
MAINSTREAM SOLUTIONS

Solution	Number	Solution vector	Problems
s_1	7	(1, 0, 1, 0, 0)	203-20(5)
s_2	7	(1, 0, 1, 0, 1)	123-1028(7)
s_3	12	(1, 0, 1, 0, 0)	206-42(10)
s_4	24	(1, 0, 1, 1, 1)	110-5(17), 203-24(8)

Number is the number of source codes in the solution.
Solution vector is the vector of the solution.
Problems indicate the problems in the solution. For example, 110-5(17) indicates that the problem whose ID number is 110-5 has 17 source codes in the solution s_4.

For example, suppose we have four mainstream solutions shown in Table II. The solution s_1 can be used to solve a problem whose ID number is 203-20. The solutions s_2 and s_3 can solve the problems "123-1028" and "206-42", respectively. The solution s_4 can be used to solve two problems whose ID numbers are 110-5 and 203-24, respectively. During the first iteration, the set *SP* = $\{s_1, s_3\}$. However, the solution s_3 has 12 instances, which are more than 7 instances that the solution s_1 has. So we append the solution s_3 at the end of the list *Sequence*.

After removing the solution s_3 from the set *S*, the rest of solutions are shown in Table III, where the solution vectors has been updated by the bitwise AND operation $o_i \leftarrow o_i$ & *mask*.

TABLE III
AFTER THE FIRST IERATION

ID	Number	Solution vector	Problems
s_1	7	(0, 0, 0, 0, 0)	203-20(5)
s_2	7	(0, 0, 0, 0, 1)	123-1028(7)
s_4	24	(0, 0, 0, 1, 1)	110-5(17),203-24(8)

At the end of the algorithm, we get a sequence of mainstream solutions stored in the list *Sequence* = (s_3, s_1, s_2, s_4). A tutoring sequence is obtained by associating each problem with its mainstream solutions. We can denote the tutoring sequence by problem ID numbers, e.g., (206-42, 203-20, 123-1028, 110-5, 203-24), which corresponds to the list *Sequence* = (s_3, s_1, s_2, s_4).

VI. EXPERIMENTS

Our experiment is based on 105 problems which consist of 45 simple problems fro freshmen and 60 medium problems for sophomores. We gathered 2456 corresponding C language source codes submitted by 118 students. All of these source codes were downloaded from the online judge at http://acm.dhu.edu.cn/dhuoj.

A. Central Sub-trees

We converted the 2456 source codes to their ASTs. 11837 sub-trees were obtained after cutting the 2456 ASTs according to their curly braces.

TABLE IV
INFORMATION OF CLASSES

ID	Number	Scope	Minimum	Average
1	1298	[8, 11]	0.8	0.902
2	**687**	**[12, 14]**	**0.923**	**0.95**
3	627	[15 ,18]	0.882	0.941
4	547	[19 ,22]	0.905	0.954
5	469	[23 ,25]	0.958	0.973
6	563	[26 ,29]	0.929	0.962
7	530	[30 ,34]	0.938	0.962
8	519	[35 ,39]	0.946	0.967
9	447	[40 ,44]	0.930	0.969
10	482	[45 ,49]	0.957	0.976

Number is the number of sub-trees in the class;
Scope is the range of sizes of sub-trees in the class;
Minimum is the smallest value of tree size similarities;
Average is the average value of tree size similarities.

By the clustering method of a 1-dimensional SOM, we obtained twenty-five classes, ten of which are listed in Table IV. We can see that sub-trees which are similar in size are gathered in one class. Take the second class for example. There are 687 sub-trees in this class. The size of each sub-tree in the class is between 12 and 14 inclusive. The minimum tree size similarity between these sub-trees is 0.923.

118 subclasses were generated by applying a 2-dimensional SOM to the twenty-five classes. Thus, we obtained 118 central sub-trees. Table V lists 4 central sub-trees of them. Take the forth row for example, the ID number of the central sub-tree is 593. There are 348 sub-trees which are similar to this central sub-tree. The minimum value of the tree structure similarities between the central sub-tree and these sub-trees is 0.929.

TABLE V
CENTRAL SUB-TREES

No	Center	Number	Minimum
1	584	34	0.867
2	160	185	0.929
3	600	73	0.867
4	**593**	**348**	**0.929**

Center is the ID number of the central sub-tree;
Number is the number of sub-trees in the subclass;
Minimum is the smallest value of tree structure similarities between the central sub-tree and the sub-trees in the subclass.

B. Solutions

2456 code vectors were constructed on a basis of the 118 central sub-trees according to the method presented in Section IV. In the LCA progress, we randomly select 400 solution instances (code vectors) to initialize their corresponding solution vector.

400 solutions were obtained after 3000 times training of the LCA, and 97 mainstream solutions were obtained from these solutions according to the method presented in Section IV.

C. Tutoring Sequence

A tutoring sequence was generated in our experiment which organized the problems together with their program codes from easy to difficult. It is shown in Table VII in Appendix. We can see that the simple problems appear in the front of the tutoring sequence, whereas the medium problems appear in the back of the tutoring sequence. However, a medium problem (i.e., 202-18 Accumulate Formula) appears near the front of the sequence, and a simple problems (i.e., 110-46 Deal) appears near the back of sequence. Such phenomenon is normal, because a simple problem may need more code blocks to solve it (i.e., if…else if…else statements). For the medium problem, there is a clever approach to solve it.

VII. Conclusion

There are lots of archives of problems for programming practice on the Internet. The problems in these archives, however, are not organized effectively for programming tutoring. Teachers may hope that the problems are organized into a tutoring sequence from easy to difficult. In this paper, we proposed an approach to organize programming problems and their program source codes into a tutoring sequence by neural computing. This approach includes converting the source codes into their corresponding abstract syntax trees, applying Self-Organizing Maps to extracting central sub-trees from the abstract syntax trees, mining the program codes to obtain the mainstream solutions by Lobe Component Analysis, and producing a tutoring sequence of the mainstream solutions from simple to complex. It is hoped that such a tutoring sequence is useful for teachers and students.

Acknowledgment

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

References

[1] http://en.wikipedia.org/wiki/Online_judge

[2] Z. Guojin and Z. Zhishou, "Knowledge unit discovery for programming tutoring based on Formal Concept Analysis," in *Educational and Information Technology (ICEIT), 2010 International Conference on*, 2010, pp. V3-476-V3-479.

[3] R. Wille, "Knowledge acquisition by methods of formal concept analysis," presented at the Proceedings of the conference on Data analysis, learning symbolic and numeric knowledge, Antibes, 1989.

[4] G. Zhu and L. Fu, "Automatic Organization of Programming Resources on the Web," in *Advances in Computer Science and Information Engineering*. vol. 168, D. Jin and S. Lin, Eds., ed: Springer Berlin Heidelberg, 2012, pp. 675-681.

[5] Z. Guojin and Z. Xingyin, "Autonomous mental development for algorithm recognition," in *Information Science and Technology (ICIST), 2011 International Conference on*, 2011, pp. 339-347.

[6] J. Jones, "Abstract Syntax Tree Implementation Idioms," in *Proceedings of the 10th Conference on Pattern Languages of Programs (PLoP2003)*, 2003.

[7] G. Zhu and C. Deng, "Mining Source Codes of Programming Learners by Self-Organizing Maps," in *Advances in Computer Science and Information Engineering*. vol. 168, D. Jin and S. Lin, Eds., ed: Springer Berlin Heidelberg, 2012, pp. 683-688.

[8] V. Kodaganallur, "Incorporating language processing into Java applications: a JavaCC tutorial," *Software, IEEE,* vol. 21, pp. 70-77, 2004.

[9] EA. Ferrán, B. Pflugfelder, and P. Ferrán, "Self-organized neural maps of human protein sequences," *Protein science : a publication of the Protein Society,* vol. 3, pp. 507-521, 03/ 1994.

[10] Z. Guojin and Z. Xingyin, "The Growing Self-organizing Map for Clustering Algorithms in Programming Codes," in *Artificial Intelligence and Computational Intelligence (AICI), 2010 International Conference on*, 2010, pp. 178-182.

[11] T. Kohonen, "The self-organizing map," *Proceedings of the IEEE,* vol. 78, pp. 1464-1480, 1990.

[12] Q. Xiao, X. Qian, and Liao, "Clustering Algorithm Analysis of Web Users with Dissimilarity and SOM Neural Networks". *Journal of Software*, North America, 7, nov. 2012.

[13] J. Vesanto and E. Alhoniemi, "Clustering of the self-organizing map," *Neural Networks, IEEE Transactions on,* vol. 11, pp. 586-600, 2000.

[14] W. Juyang and M. Luciw, "Dually Optimal Neuronal Layers: Lobe Component Analysis," *Autonomous Mental Development, IEEE Transactions on,* vol. 1, pp. 68-85, 2009.

[15] M. D. Luciw and W. Juyang, "Laterally connected lobe component analysis: Precision and topography," in *Development and Learning, 2009. ICDL 2009. IEEE 8th International Conference on*, 2009, pp. 1-8.

[16] K. Zhang and D. Shasha, "Simple Fast Algorithms for the Editing Distance between Trees and Related Problems," *SIAM Journal on Computing,* vol. 18, pp. 1245-1262, 1989.

[17] Y.-P. Qin, L.-Z. Lü, F.-W. Zhang, B.-B. Zhang, and J. Zhang, "The Neighborhood Function and Its Application to Identifying Large-Scale Structure in the Comoving Universe Frame," *The Astrophysical Journal,* vol. 669, p. 692, 2007.

[18] X. Li, "A New Text Clustering Algorithm Based on Improved K_means," *Journal of Software*, North America, 7, jan. 2012.

[19] C. Huang, J. Yin, and Hou, "Text Clustering Using a Suffix Tree Similarity Measure," *Journal of Computers*, North America, 6, oct. 2011.

[20] http://en.wikipedia.org/wiki/Moore_neighborhood

APPENDIX A

TABLE VII
TUTORING SEQUENCE

No.	Problem title	No.	Problem title	No.	Problem title
1	112-59 Date	36	200-3 Pure composite number	71	201-11 Translate letters
2	107-13 The day number of a month	37	207-46 Different date	72	201-9 Spiral square
3	123-1028 Count decimals-2	38	109-40 Complex formula	73	203-19 Three ships
4	112-43 Palindrome digital	39	106-24 Bred problem	74	203-22 Chess board
5	206-42 Josephus-2	40	106-25 Sum	75	204-25 Magic ratio
6	111-35 Real value calculate	41	201-7 Special Four-digit	76	113-39 Reverse order
7	110-55 Count letter	42	205-35 The three prime numbers	77	207-45 Seek prime
8	112-32 Encryption	43	202-14 Series	78	200-4 Count Series number
9	203-24 Lost pages	44	109-2 Magic number	79	209-60 Gold Bach's Conjecture
10	202-18 Accumulate Formula	45	203-23 Construct sequence	80	106-42 Sort
11	113-8 Count decimals-1	46	208-51 Tree	81	209-58 The multiple of 7
12	202-16 Formula result	47	205-34 Ring	82	208-54 Adder
13	201-12 A multiple of T	48	105-53 Prime	83	209-59 Prime
14	203-20 Multiplication of tribal people	49	105-19 Sum of approximate	84	110-5 Absolute value
15	106-21 Seek big, small, average	50	112-18 Data reversal	85	105-38 Strange number
16	203-21 Bus	51	107-36 How about you score	86	204-28 Decimal
17	111-28 Shipping rate	52	205-32 Table tennis	87	206-38 Palindrome-1
18	111-37 Busy dog robber	53	209-57 Maximum	88	105-9 Check prime number
19	108-14 Print letter	54	200-1 Triangle number	89	206-39 Moto
20	113-7 Count reverse letter	55	208-53 Palindrome-2	90	201-10 Array
21	111-27 Function value	56	209-56 Series-2	91	204-27 Security system
22	105-10 Count zero	57	108-47 Print*	92	123-1026 Strange Shape
23	105-17 Sum of number	58	108-26 Back and front of letter	93	200-2 pure prime
24	107-11 Class problem	59	123-1025 Data division	94	110-46 Deal
25	106-56 Open/close light	60	209-55 Series-1	95	207-47 Magic square
26	110-45 SysConvert-1	61	204-26 Output Diamond-1	96	105-6 Greatest common divisor
27	110-48 SysConvert-2	62	202-15 Abc	97	200-6 Print sequence in order
28	202-17 Maximum benefits	63	106-44 Binary	98	207-44 Count String
29	200-5 Highest frequency	64	206-41 Josephus-1	99	207-43 The start of String
30	107-20 Which day	65	207-48 Longest letter	100	204-30 A multiple of N
31	111-52 A simple problem	66	106-22 Fibonacci	101	205-36 Matrix transfer
32	107-58 Triangle	67	201-8 Seek the biggest	102	206-37 Letter sort
33	204-29 Sparse matrix	68	109-12 Ticket price	103	205-31 poker
34	208-52 Triangle	69	208-50 Sequence	104	202-13 Divisible
35	205-33 Max and min	70	206-40 Numeric string processing	105	208-49 Scholarship

The problem title which starts with "1" indicates an easy problem, e.g., "112-59 Date" is an easy problem. The problem title which starts with "2" indicates a medium problem, e.g., "208-50 Sequence" is a medium problem. When two programs have the same mainstream solution, put the one that has more program codes before the other.

Guojin Zhu is an associate professor at the Department of Computer Science, Donghua University (DHU), Shanghai, China. He received his M.S. and Ph.D. degrees from DHU in 1991 and 2007, respectively. He was a visiting scholar at the Department of Computer Science and Engineering, Michigan State University, East Lansing, Michigan, USA from November 2007 to November 2008. His current research interests include semantic web, knowledge discovery, and neural computing.

Le Liu is a graduate student of Computer Application Technology at Donghua University. He was born in Hunan province, P. R. China in 1987. His current main research interest is computer network and AI.