Structural Diagnosis Method for Computer Programs Developed by Trainees

Daniel Gaydachuk[1], Olena Havrylenko[1], Juan Pablo Martínez Bastida[1], Andrey Chukhray[1]

[1] National Aerospace University, KhAI, Kharkiv, Ukraine
dvrch@mail.ru, {lm77191220, jpbastida, achukhray}@gmail.com

Abstract. The individual professional skills training via intelligent tutoring systems is one of the high-priority scientific and applied problems. This paper considered the issue that arises while training professional skills of algorithmization and programming, specifically – how the sample program stored in ITS will be compared with the program developed by a trainee. Structural diagnosis method for computer programs developed by trainees is proposed. Its advantages are the speed increment in comparison with known methods and a better adjustment of tutoring purposes. The results have been verified by means of the module testing, the tutoring system prototype implementation and introduction to a studying process.

Keywords: Intelligent Tutoring System, M-ary tree, edit distance, trace, trainee's program, code diagnose.

1 Introduction

Today one of the most actual scientific and engineering problems is ensuring of the effective professional skills individual learning. Thus, the limited human psychophysiological abilities is due to that one teacher cannot adaptively teach every student in a group with twenty or thirty people. A promising solution of this situation may be the development and implementation of intelligent tutoring systems (ITS). Such programs may have potentially unlimited resources and high performance.

One of the issues that could improve quality of computer tutoring in professional skills of algorithmization and programming is considered in this paper. The great challenge is how sample program stored in ITS will be compared with the program developed by a trainee [3, 4]. Commonly, such tutoring systems check solution written by a user on some programming language by means of input-output tests. If some test has failed and the trainee is not able to fix it by himself, then the tutoring system need to determine where the mistake occurred and help to resolve it.

Problem statement. It is required to create a method for finding the minimum edit distance and trace between two m-ary trees. It could be achieved by modernization of the method published in [6] that finds only tree distance. This method should provide trainee's code diagnose on the structural level.

Assume, that each node in the tree has signed by a label which is a serial number beginning from the bottom to the top and from the left to the right. Then following edit operations can be defined: insert operation of the node, delete operation of the node and exchange of two node labels. The operations are measured by means of the weight factor, which is a metric described in [5]. In this case the weight of label exchanging with the same label is equal to zero. The weight of label insertion is equal to the weight of label deletion. The weight of the two different labels exchanging can't be greater than the weights sum of the equivalent deletions and insertions, which transform the source tree the same way. In this paper, it is assumed that weight of an operation is equal to zero (label exchanging with the same label) or one (in other cases).

The example of parameters determining, such as l(T) – an array of the leftmost tree T leaves and LR_keyroots(T) – a set of such k, for which $l(T)[k] \neq l(T)[p(T)[k]]$, is represented on Fig. 1. Parameter *p(T)* is an array of the ancestors for all nodes in the tree T in ascending order, index in [] specify the number of root node. After finding trace between two trees one tree can be converted into another.

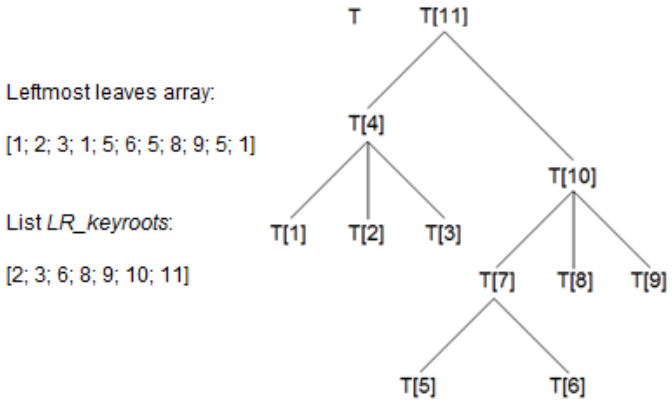

Fig. 1. The example of determining *l* and *LR_keyroots* for tree *T*

The method implemented in this article implies that each tree has input data represented as two arrays: the first – an array of ancestors of every node in the tree: *p1* (if it is the initial tree) or *p2* (if it is an ending tree), the second – an array of labels for each of the nodes (*T1* and *T2* respectively). Further indexation of array fields will start from zero (enumeration of tree nodes remains the same).

In order to solve the problem of finding the tree distance and trace the method covers, the calculation of *l* and *LR_keyroots* for the initial tree (*l1* and *LR_keyroots1* respectively) and for the tree to which you want to convert the initial one (*l2* and *LR_keyroots2* respectively), as follows. For each tree, all the nodes have been analyzed starting with the number one in ascending order.

For instance, a tree T is considered and nodes that at least once has been read or modified, become marked. Originally, leftmost leaf array *l* filled with zero values and *LR_keyroots* (as a list) is empty.

Obviously,

$$l(T)[1] = l(T)[p(T)[1]] = l(T)[p(T)[p(T)[1]]] = \ldots = l(T)[|T|] = 1 \quad (1)$$

where $|T|$ is a number (the cardinality) of tree nodes (hereafter we will use the notation «*nT1*» (*nT1* = $|T1|$) and «*nT2*» (*nT2* = $|T2|$)).

First, the leftmost leaf array l is filled by these values. Further, the next unmarked node is determined in turn. If it is found, it will be a tree leaf. For it the relation *l(T)[i]* = *i* is valid. At the same time, the field *l(T)[p(T)[i]]* is set to *i*, if the ancestor was not marked, otherwise value *i* is added to an array *LR_keyroots*, and then the shift to the next unmarked node is done. At the end, tree *T* root number is added to the array *LR_keyroots*. In order to solve the problem-parallelized version of the method presented in [6], two arrays are used:

1. *treedist* size *nT1 * nT2* (*treedist[i1, j1]* contains the edit distance between two trees, which roots are node *T1[i1]* of tree *T1* and node *T2[j1]* of tree *T2*, respectively).
2. *dist* sizing $nT1^2 * nT2^2$ (*dist[i, j][i1, j1]* contains the edit distance between two forests, which includes nodes numbered *l1[i]* to *i1* and the tree *T1* from *l2[j]* to *j1* in the tree *T2*, respectively).

By using an array of that size, allowed to work with trees in which the number of nodes is not more than two hundred. In this regard, the optimization and extension were performed for the base parallelized method proposed in [6], excluding the parallelization itself.

Below, there are the changes undergone by this method. In [6] it is proved that

$$\begin{cases} dist[i,\ j][l1[i]-1, l2[j]-1] = 0 \\ dist[i,\ j][i1, l2[j]-1] = dist[i,\ j][i1-1, l2[j]-1] + 1, \\ dist[i,\ j][l1[i]-1,\ j1] = dist[i,\ j][l1[i]-1,\ j1-1] + 1 \end{cases} \quad (2)$$

$$\{i1 \in [l1[i]; nT1],\ j1 \in [l2[j]; nT2]\}$$

In order to reduce the number of fields in *dist* array, as one of the options, it is necessary to replace some of them to analytical expression. As following it is shown:

$$\begin{cases} dist[i,\ j][l1[i]-1, l2[j]-1] = 0 \\ dist[i,\ j][i1, l2[j]-1] = i1 - l1[i] + 1, \\ dist[i,\ j][l1[i]-1,\ j1] = j1 - l2[j] + 1 \end{cases} \quad (3)$$

$$\{i1 \in [l1[i]; nT1],\ j1 \in [l2[j]; nT2]\}$$

In addition, it should be noted that some of the fields *dist[i, j][i1, j1]* of array *dist* containing the distance between two non-empty forests are not used due to the performance of the following relationships:

$$
\begin{aligned}
& i \in LR\_keyroots1, \\
& j \in LR\_keyroots2, \\
& i1 \in [l1[i]; i],\ j1 \in [l2[j]; j]
\end{aligned} \tag{4}
$$

Let the source array *dist* which will be referred to as «*dist0*», be the so-called "original", *dist* array (different from the original) "image" that includes only the fields which contain the distance between two non-empty forests. It is noticed that between the index of the field list *LR_keyroots* and the field value one-to-one ratio exists. It can also be noticed that field of indices that do not satisfy the conditions imposed on *i1* and *j1* will never be refered.

Thus, the *dist* array is ragged, namely: it can be represented as a two-dimensional matrix of size |*LR_keyroots1*|*|*LR_keyroots2*|, the elements of which are two-dimensional matrices; each has the size determined by the expression:

$$
\begin{aligned}
& (LR\_keyroots1[i0] - \\
& - l1[LR\_keyroots1[i0]] + 1) \times \\
& \times (LR\_keyroots2[j0] - \\
& - l2[LR\_keyroots2[j0]] + 1)
\end{aligned} \tag{5}
$$

where *i0, j0* are the row and column numbers of the first matrix, respectively. The match of array *dist0* fields to array *dist* may be written as:

$$
\begin{aligned}
& dist0[i, j][i1, j1] \equiv \\
& \equiv dist[i0, j0][i1\_i, j1\_j], \\
& \{i = LR\_keyroots1[i0], \\
& j = LR\_keyroots2[j0], \\
& i1\_i = i1 - l1[i], \\
& j1\_j = j1 - l2[j]\}
\end{aligned} \tag{6}
$$

Furthermore, the *treedist* array is converted and it can replace it with an array *dist0* in accordance with the following identity:

$$
\begin{aligned}
& treedist[i1, j1] = dist0[u, v][i1, j1], \\
& \{u \in LR\_keyroots1, l1[u] = l1[i1], \\
& v \in LR\_keyroots2, l2[v] = l2[j1]\}
\end{aligned} \tag{7}
$$

It should be noted that one-to-one relationship is observed between the specified value $u \in LR\_keyroots(T)$ and value $l(T)[u]$.

Let specify array *LR_keyroots_inverse(T)*, for which the equality is as follows:

$$(LR\_keyroots\_inverse(T)[l(T)[i]-1]=i0)\equiv$$
$$\equiv (l(T)[i]=l(T)[i0]\cap i0\in LR\_keyroots(T)) \tag{8}$$

then,

$$\begin{aligned} &treedist[i1,\ j1]=dist[\\ &LR\_keyroots1\_inverse[l1[i1]-1],\\ &LR\_keyroots2\_inverse[l2[\ j1]-1]][\\ &i1-l1[i1],\ j1-l2[\ j1]], \end{aligned} \tag{9}$$

$$\begin{aligned} &\{LR\_keyroots1\_inverse=\\ &=LR\_keyroots\_inverse(T1),\\ &LR\_keyroots2\_inverse=\\ &=LR\_keyroots\_inverse(T2)\} \end{aligned}$$

Thus, the distance between trees *T1* and *T2* is the value equal to:

$$\begin{aligned} &dist[|\ LR\_keyroots1\ |-1,\\ &|\ LR\_keyroots2\ |-1][\\ &nT1-1,nT2-1] \end{aligned} \tag{10}$$

The search for trace between two trees is carried out based on the third and fifth lemmas of the article [6] by using the result of the calculation from the array *dist*.

The algorithm is implemented as code in C# programming language. Table 2 presents the information about the approximated time the algorithm took for selected sizes of trees, and the sizes of both trees are the same but their depth is equal to two.

Table 1. Data-time comparison of the algorithm

The number of nodes of each tree	Approximate run time		The exception
	the distance search	the trace search	
not more than 100	less than 0.1 s	less than 0.1 s	–
250	1.15 s	0.15 s	
500	9 s	0.25 s	
1000	1 m 11 s	0.4 s	
2000	9 m 21 s	1.4 s	
4000	1 h 15 m 21 s	–	Stack Over Flow
more than 6500	–		Out Of Memory

2 Conclusion

The proposed algorithm is a modification of the algorithm Shasha-Zhang. Nevertheless, the latter does not allow determining the trace between two trees. In the modified algorithm the used memory grows slower than in the Shasha-Zhang algorithm, when the number of trees nodes increases. If count of tree nodes was 6500 or more, the algorithm could be interrupted with a message about insufficient memory.

The prototype of the tutoring system was developed to verify method's reliability. It analyzes trainee's C++ code by means of a set of tests. In case a user could not solve a task, the system starts lexical code analysis (comparison of token arrays), and then parse it (comparison of abstract syntax trees), in order to identify errors and prompts hints to assist the trainee in accomplishing the proper writing of the required algorithm.

References

1. Chukhray, A.: On a method of verification of professional skills algorithmization (in Russian). In: Journal Radio-electronic and computer systems. vol.4(38), pp.84–86. (2009)
2. Chukhray, A.: Models and methods for adaptive computer systems support of the acquisition of knowledge and skills in solving algorithmic learning tasks (in Russian). In: Journal Radio-electronic and computer systems. vol. 5(64), pp. 390–402. (2013)
3. Horwitz, S.: Identifying the semantic and textual differences between two versions of a program. In: Proceedings of the SIGPLAN 90 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices. vol. 25(6), pp. 234–245. White Plain, New York. (1990)
4. Yang, W.: Identifying syntactic differences between two programs. In: Software-Practice and Experience. vol. 21(7), pp. 739–755. (1991)
5. Tai, K.-C.: The tree-to-tree correction problem. In: Journal of the ACM. vol. 26(3), pp. 422–423. (1979)
6. Zhang, K., Shasha, D.: Simple fast algorithms for the editing distance between trees and related problems. In: SIAM Journal of Computing. vol. 18(6), pp. 1245–1262. (1989)