Case-Based Adviser for Near-Miss Programs

Hiroyoshi WATANABE, Kumiko TAKAI, Masayuki ARAI and Shigeo TAKEI
School of Science and Engineering, Teikyo University,
1-1 Toyosatodai, Utsunomiya-shi 320-8551, Japan
{hiro,kumiko,arai,takei}@ics.teikyo-u.ac.jp

Abstract. This paper describes a method of giving advice for near-miss programs using a case-based reasoning approach in programming education. A near-miss program is a program that does not run properly but is rather close to correct programs. Two case-based methods of generating advice sentences are proposed; one is based on similarity to past advice cases and the other is based on the difference from correct programs. In addition to these, a model of giving step-by-step advice is proposed. Based on these ideas, a case-based advice system for a simple assembly language is implemented. The system had been utilized in actual classes and the results suggested that the case-based advice system was useful.

1. Introduction

In programming education, programming exercise courses play an important role, because writing programs is indispensable to learning programming. In programming exercise classes, teachers give students various problems in order to get the students to understand important concepts in programming. Students write programs that satisfy the requirements of the problem and submit the program and/or report to the teacher. The teacher reads the programs and reports to see if the students understand the concepts. However, the teachers' workloads in such programming classes tend to be very heavy. It is very important to reduce the teachers' workloads and let teachers focus their energy on guidance that students really need.

We have proposed a case-based evaluation assistant model of novice programs and developed a case-based evaluation assistant system for a simple assembly language CASL. The target evaluation task is to judge whether students' programs satisfy the requirements of the given problem. First, the system automatically checks the actions of the students' programs using prepared sample data, and then the system evaluates the implementation of the programs based on evaluation cases. The system has been used in actual classes at our university and the results showed that the system can reduce teachers' evaluation work drastically [1]-[3].

In the next step, we aim at giving advice on students' programs that do not run properly; but are rather close to correct programs. Such programs with a few bugs are called "near-miss programs" in this paper. We propose a model of giving step by step advice, not giving detailed advice at one time. We also propose a method of generating advice sentences for near-miss programs by a case-based reasoning approach [4] [5]. Using the model and the generation method, we implemented an advice system for near-miss programs of a simple assembly language CASL-II. CASL-II is an improved version of CASL and it is adopted in examinations for information-technology engineers certified by the Japanese ministry of international trade and industry, instead of CASL.

There has been much research on automated diagnosis of learners' programs [6]-[10]. Most of the researches took a knowledge-based approach. The advantages of the knowledge-based approach include that perfect diagnosis would be done in the range of acquired knowledge in diagnosis systems and detailed diagnosis reports could be generated based on deep knowledge. Some systems can even explain the reasons of errors in the

learners' programs. However, the knowledge acquisition is a big problem and few knowledge-based diagnosis systems are used practically. We aim at implementing a practical system by taking a case-based reasoning approach.

2. Basic Ideas

2.1 Giving Step-By-Step Advice

When teachers give advice to students, they give small hints at first and then give more hints or detailed advice later. Students will gain more confidence when they solve problems with small hints than in such cases where they got detailed advice. We call such a style of giving advice "step-by-step". Although many teachers will take this pedagogical style of giving advice to students, most of the published automated diagnosis systems give detailed advice, or show full reports of the diagnosis, all at once. We propose a model of giving step-by-step advice to students in Section 3 and adopt the model to implement a case-based advice system.

When we reflected on how we give advice to students, we noticed that we gave advice in three or four steps. At first, we tell students generally where bugs may exist in the students' programs. Then, we give small hints about the bugs one or two times. Finally, we explain about the bugs and how to fix them. Based on such reflection, we adopted three or four steps of giving advice including indicating the location of bugs as the first piece of advice.

2.2 Two Methods of Advice Sentence Generation

There are two methods of generating advice sentences using two different types of cases.

The first method is to generate advice sentences based on the similarity to advice cases, which include a near-miss program as a problem description and advice sentences on the program as a solution description. The advice system retrieves the most similar advice case to a target near-miss program, compares the selected case with the target program to generate correspondence information, and modifies the sentences from the advice case using the correspondence information to adapt to the target program.

The second method is advice generation based on the difference between the target near-miss program and a correct program. There are many correct programs in evaluation case-bases of the evaluation assistant system we already implemented [1]-[3]. The advice system retrieves the most similar evaluation case to a target near-miss program and adopts the correct program in the retrieved case if the difference between these programs is small enough. Advice sentences can be generated by analyzing the difference.

We adopt both of these methods because there are many more variations of buggy programs than correct programs; the more resources for giving advice the better, in order to deal with many variations.

3. Model of Giving Step-By-Step Advice

This section describes a model of giving step-by-step advice. When a student requests advice for a certain program for the first time, advice about locations of bugs is presented. After that, basically, each time when the same student requests advice for the same program, advice sentences in the next level are presented. However, the next level of advice is not presented if the minimum interval time has not passed from the time when the current level of advice sentences is presented.

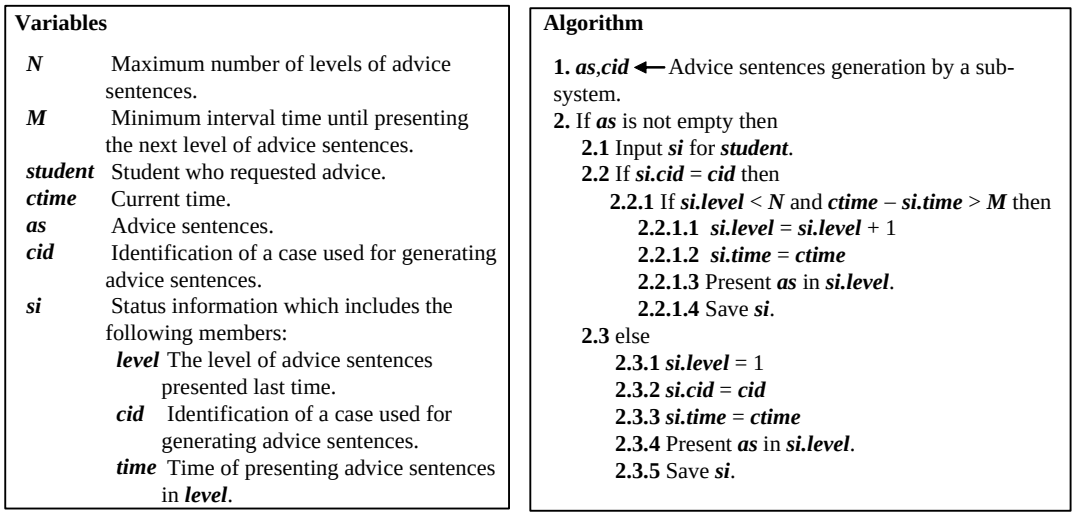


Figure 1. Data and an algorithm for presenting advice sentences.

Data and an algorithm for a model of giving step-by-step advice are shown in figure 1. This algorithm is launched when a student requests advice about his/her program. First, a sub-system of advice sentences generation (ASG sub-system) tries to generate advice sentences (*as*). The ASG sub-system needs to be capable of generating *N* levels of advice sentences. When the ASG sub-system cannot generate advice sentences, that is, *as* is empty, the system shows such a message as "Sorry, there is no advice resource available for your program right now", to the student.

When the ASG sub-system generated advice sentences, the system judges whether the system should present the next level of advice sentences or not, based on status information (*si*). Status information of each student includes an identification of a case used for generating advice sentences (*si.cid*), a level number of advice sentences presented so far (*si.level*), and the time of the first presentation of advice sentences in the current level (*si.time*).

A target program for the current advice request is regarded as almost the same as a target program for the advice request last time, when the case identification in the status information (*si.cid*) is the same as an identification of a case used for generating advice sentences (*cid*) this time. In this case, advice sentences of the next levels are presented to the student and status information of the student is updated, if an interval between the current time and the time in status information (*si.time*) is longer than the pre-defined minimum interval, *M*. Otherwise, the system shows a message such as "Please, try to find bugs again with the advice presented so far. You have to wait for a while to get the next set of advice", to the student.

If *si.cid* is not the same as *cid*, the student is considered to have changed his/her program. In this case, the first level of advice sentences is presented to the student.

4. Generation of Advice Sentences Based on Similarity

4.1 Advice Cases

Advice sentences for a student's program can be generated using advice cases. Main parts of an advice case are a near-miss program as a problem description and *N* levels of advice sentences as a solution description. (*N* =3 in our implemented system.) Examples of a target program and advice sentences in an advice case for a problem p-01 described in table 1 are shown in the right side of figure 2.

Table 1. An example of problem presented to students.

No.	Description
p-01	Sum the given N integers decreasing a value of an index register by one. The number of integers is saved in the address labeled N. Integers are saved in the memory area whose start address is labeled DATA. The sum should be saved in the address labeled ANS.

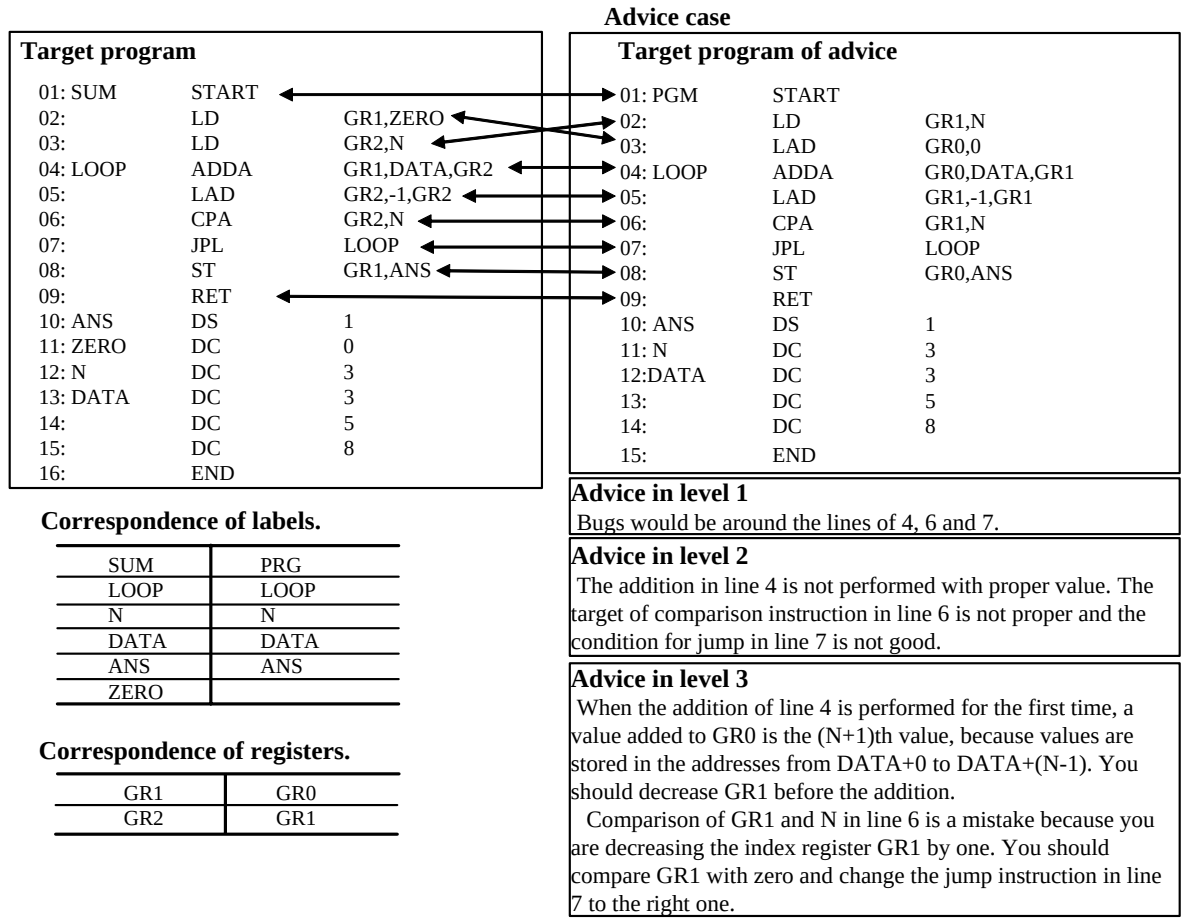


Figure 2. Examples of an advice case and matching a student's program and the case.

An advice case also includes retrieval information and maintenance information. The retrieval information includes a problem identification that the near-miss program is written for and numbers of every operation code in the near-miss program. The maintenance information includes the name of a person (a teacher or a co-learner) and the date of adding and updating the advice case.

Three level indexes of advice cases are constructed using information of generalized program lists, that is a program list which covers several variations of program lists. The case index for advice cases is almost the same as the index for evaluation cases, which is described in [2] and [3]. The difference is that variations of redundant instructions are not used for indexes of advice cases. An instruction is regarded as redundant when the execution results are correct regardless of whether the instruction is removed or not. The reason why redundant instructions are not dealt with in indexes of advice cases is because it is difficult to detect redundant instructions in near-miss programs which are not be executed correctly.

4.2 Processes of Generating Advice Sentences with Advice Cases

First, the most similar case to a given student's program is retrieved from the advice case-base by following the case index. The retrieved case is compared with the student's

program in detail in order to investigate whether the advice sentences in the case can be applied to the student's program. The comparison is called a program matching process. The processes of retrieval and the program matching are the same as ones used in case-based program evaluation [2].

If the selected case matches the student's program perfectly, advice sentences in the case are used for the student's program after simple modification. That is, label names, register numbers and line numbers in advice sentences from the case are replaced with corresponding ones in the given program, using correspondence information generated in the program matching process. When no case matches student's program perfectly, advice sentences can not be generated.

The advice case matches the student's program perfectly in figure 2. While advice sentences in level one and two are used as they are, the advice sentences in level three are modified, that is, GR0 is replaced with GR1 and GR1 is replaced with GR2 based on correspondence information (two tables in figure 2).

5. Generation of Advice Sentences Based on Difference

5.1 Outline of Difference Based Generation

The second method is generation based on the difference between a target near-miss program and a correct program. A correct program used for advice generation is called a base program. In the domain of programming in assembly languages, there are five types of difference and advice sentences depending on the type of difference.

(a) Mistakes of instructions. Operation codes or operands are different in corresponding instructions between the target near-miss program and the base program.
(b) Lack of instructions. The target near-miss program lacks some instructions, which should correspond with instructions in the base program.
(c) Obstacle instructions. The target near-miss program has some instructions, which do not correspond with any instructions in the base program.
(d) Differences in order of instructions. The order of some corresponding instructions is different between the target near-miss program and the base program.
(e) Differences of label locations. The locations of corresponding labels are different between the target near-miss program and the base program.

Correct programs in evaluation case-bases are available for the base programs. An evaluation case consists of a problem description, a solution description, retrieval information and maintenance information. The problem description is a target program list for evaluation. The actions of all programs in evaluation cases are correct, because the actions of submitted programs are checked using prepared sample data and programs that do not run correctly are rejected before case-based evaluation. The solution description is evaluation results, i.e., the judgment of acceptability and written advice, although they are not used for generating advice sentences for near-miss programs. For the details of evaluation cases, please refer to [2].

5.2 Retrieving Base Program and Program Matching

The most similar program to a student's near-miss program is retrieved from an evaluation case-base. The retrieved program is compared with the student's program in detail (program matching) and correspondence information and difference information are generated. Difference information of each type described in Section 5.1 is the following:

(a) Mistakes of instructions. A line number of an instruction in the target program, a line

number of the corresponding instruction in the base program and a location of the mistake that is one of “the operation code”, “the first operand”, “the second operand”, or “the third operand”.

(b) Lack of instructions. A line number of an instruction in the base program, which does not have a corresponding instruction in the target program.

(c) Obstacle instructions. A line number of an instruction in the target program, which does not have a corresponding instruction in the base program.

(d) Differences in order of instructions. A list of serial line numbers in a part of the target program where order of corresponding instructions is different.

(e) Differences of label locations. A line number of an instruction in the base program and a type of the difference, that is one of “(type1) two labels do not correspond”, “(type2) an instruction in the target program has a label but one in the base program does not” or “(type3) an instruction in the base program has a label but one in the target program does not”.

Program matching processes for generating correspondence information and difference information are as follows:

(1) Generating candidates of correspondence information. For each instruction in the target program, instructions in the base program which have the same operation code as the instruction in the target program are added to a candidate list of instruction correspondences. Pairs of registers and pairs of labels which are derived from the candidates of the instruction correspondences are also added to a candidate list of register and label correspondences.

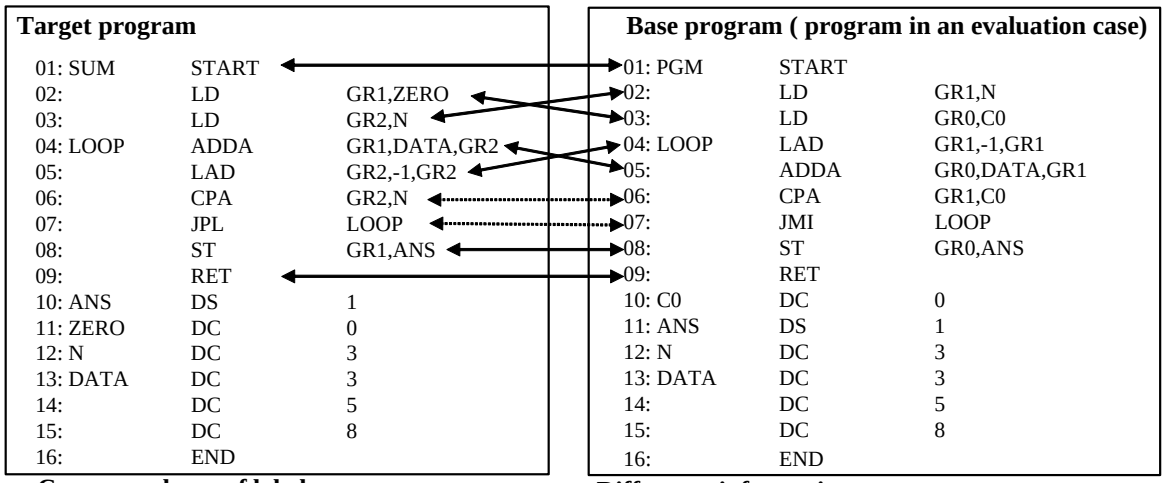
(2) Pruning the candidates and regenerating correspondence information. Candidates are pruned in order to make consistent correspondences of instructions, labels and registers. Because the base program should not match the target program perfectly, the strategy that maximizes numbers of consistent correspondence is adopted. As a result of pruning the candidates, final correspondence information is generated.

(3) Generating difference information. First, instructions which do not have corresponding ones in the other program are picked up and difference information on (b) lack of instructions and (c) obstacle instructions is generated. Second, the generated information on (b) and (c) are compared and pairs of instructions that correspond except for a small difference are detected as (a) mistakes of instructions. Information on instructions which are detected as (a) is removed from (b) and (c). Finally, (d) differences in order of instructions and (e) differences of label locations are derived from correspondence information.

Figure 3 shows examples of a target near-miss program and base program for a problem P-01 in table 1. After the process of (2), solid arrows between the target and base programs and correspondence tables in figure 3 are generated as correspondence information. In the process of (3), lines 4 and 5 in the base program are picked up as (b) lack of instructions and lines 4 and 5 in the target program are picked up as (c) obstacle instructions. Comparison of (b) and (c), two elements of (a) mistake instructions are detected and (b) and (c) become empty. Finally, one element of (d) differences in order of instruction is detected. The difference of order in lines 2 and 3 is not picked up because the difference is regarded as trivial. For conditions of trivial differences of order, please refer to [2].

5.3 Generating Advice Sentences

Advice sentences are generated using correspondence and difference information generated by program matching. For the first level, advice sentences on the location of bugs are generated regardless of the types of difference, while advice sentences in levels 2 and 3 depend on the types of difference.



Correspondence of labels.

SUM	PRG
LOOP	LOOP
N	N
DATA	DATA
ANS	ANS
ZERO	C0

Correspondence of registers.

GR1	GR0
GR2	GR1

Difference information.

(a) Mistakes of instructions
(a-1) Line 6 in the target program.
Line 6 in the base program.
The second operands are different.
(a-2) Line 7 in the target program.
Line 7 in the base program.
The operation codes are different.
(d) Difference in order of instructions
(d-1) Line 4 and line 5

Figure 3. Examples of matching results between a target program and a base program.

Advice sentences generated from the situation in figure 3 will be the following:

Level1: Bugs are around line 4 to line 7.

Level2: Check the order of instructions in lines 4 and 5. Check the instructions in lines 6 and 7.

Level3: Should the order of instructions in lines 4 and 5 be lines 5, 4? Check the operands in line 6. Check the operation code in line 7.

Level4: Should the second operand in line 6 be “N”? Should the operation code in line 7 be “JMI”?

6. Experience of Using Case-based Advice System in Actual Classes

Based on the ideas described so far, we implemented a case-based sub-system of generating advice sentences and added the sub-system to the evaluation assistant system, which we had already implemented. We used the system in two classes of the CPU and assembly language course at Teikyo University in 2002. The students in the classes numbered 91 and 90. We just included an explanation of the advice system on the web page which explains how to submit programs, but we did not emphasize the advice system, because we wanted to investigate the needs of such advice systems. We presented 28 problems in the two classes. There were evaluation case-bases for 11 of the 28 problems because we had presented the 11 problems before. There were advice case-bases for 8 of the 11 problems.

There were 486 advice requests for 28 problems. The numbers of advice requests depend on problems. The maximum number of requests for one problem was 61 and they were from 18 of 90 students. This result shows that needs for advice systems are high because the advice request from 20% of students is considered to be a high rate for traditional style classes in which students can ask questions of teachers or students. The rate will be higher in self-learning style classes in an e-learning environment.

Table 2. Results of counting advice request logs.

	CB	None	Total
a Advice is presented by similarity based method	72	0	72
b Advice is presented by difference based method	68	61	129
c total of advice requests	226	260	486
(a+b)/c x 100 (%)	61.9	23.5	41.4

CB: 11 problems which have past cases.
None: 17 problems whose case-bases was empty at first.

Table 3. Results of a questionnaire.

How the advice is useful?	Number
Advice was useful many times.	11
Advice was useful some times.	27
Advice was not very useful.	11
I did not request advice.	59

Table 2 shows results of counting advice request logs. The ratio of cases in which advice sentences were presented for advice requests is rather high (62%) when there were case-bases, although the ratio is low when the case-bases are empty. Table 3 shows results of a questionnaire after classes. Not all the students answered the questionnaire. Table 3 shows that the advice system was useful for 22% of students who used the system.

The usefulness of the case-based adviser is suggested through the experience with the implemented case-based advice system, although the performance of the system has not been sufficiently high so far. The performance of the system can be improved by adding new cases to its case-bases.

7. Conclusions

We have proposed a framework of a case-based adviser for near-miss programs. Based on the idea, we implemented a case-based advice system for a simple assembly language CASL II. Experience using the system in actual classes suggests that a case-based advice system is useful for students. Systems implemented based on the idea will be useful tools especially for e-learning environments. In the future, we plan to improve the performance of our system and investigate effectiveness of the proposed case-based adviser in detail.

This research was supported in part by JSPS Grants-in-Aid for Scientific Research No.14580428.

References

[1] H. Watanabe, M. Arai, and S. Takei, Automated Evaluation of Novice Programs Written in Assembly Language, *Proc. of ICCE99*, Chiba, **2** (1999) pp. 165-168.

[2] H.Watanabe, M.Arai and S.Takei, Case-based Evaluation of Novice Programs, *Proc of AI-ED2001*, San Antonio, pp.55 – 64(2001).

[3] H.Watanabe, M.Arai and S.Takei, A Method of Constructing Case-base for Evaluation Assistant of Novice Programs, *Proc. of ICCE2001*, Soul, **1** (2001) pp.313 – 320.

[4] J.Kolodner, Case-Based Reasoning, Morgan Kaufmann Publishers,Inc. (1993).

[5] D. Leake ed., Case-Based Reasoning: Experiences, Lessons and Future Directions, AAAI Press / MIT Press, (1996).

[6] A. Adam and J. P. Laurent, LAURA, A System to Debug Student Programs, *Artificial Intelligence*, **15** (1980) pp.75-122.

[7] W. L. Johnson, Understanding and Debugging Novice Programs, *Artificial Intelligence*, **41** (1990) pp.51-97.

[8] W. R. Murray, Automatic Program Debugging for Intelligent Tutoring Systems, *Computational Intelligence*, **3** (1987) pp.1-16.

[9] Schorsch,T. , CAP: An Automated Self-Assessment Tool to Check Pascal Programs for Syntax, Logic and Style Errors, *Proc. of SIGCSE95*, (1995) pp.168 -172.

[10] H. Ueno, Concepts and Methodologies for Knowledge-Based Program Understanding - The ALPUS's Approach, *IEICE Trans. Inf. & Syst.*, **E78-D**, no.2 (1995) pp.1108-1117.

[11] S. Kim and J. H. Kim, Algorithm Recognition for Programming Tutoring Based on Flow Graph Parsing, *Applied Intelligence*, **6**, iss.2 (1996) pp.153-164.