A Case-Based Reasoning Approach to Providing Feedback to Novice Programmers

Ryan Dimaunahan and Raymund Sison
College of Computer Studies
De La Salle University
ryan.dimaunahan@dlsu.edu.ph, raymund.sison@delasalle.ph

Abstract: Adaptive feedback contains information that individual users of a system will find helpful rather than cryptic. A case-based reasoning (CBR) approach to automatic feedback generation can provide feedback that is timely and adaptive; however, such an approach generally needs a sufficiently populated case base. In this paper, we describe a pedagogical programming tool called CBR-C that uses a CBR based approach to give meaningful and adaptive feedback to students learning the C programming language for the first time. CBR-C generates multiple levels of feedback depending on the number of cases in its case base and the required remediation of the student, and is able to give feedback despite having insufficient cases in its case base. Experiments for evaluating the feedback generation capability of CBR-C were conducted with students learning to program in C for the first time. These students were assigned to control and experimental groups, and each student was instructed to submit solutions to a programming problem incrementally until the student finally gets a correct answer, i.e., a C program that meets all the given programming requirements. The improvement in code quality of each submission was then determined to see whether the feedback generated by CBR-C had any effect on the code of the students. The improvement in code quality of the students who used CBR-C was greater, with mild statistical significance, than those who did not, indicating that receiving feedback from CBR-C regarding one's program is better than not receiving any feedback at all, at least as far as students learning C for the first time are concerned.

Keywords: Feedback, Case Based Reasoning, Pedagogical Programming Tool

1. INTRODUCTION

1.1 Research Description

Computer programming is a very challenging subject to learn for the first time, as learners are often forced to face multiple challenges simultaneously (Jenkins, 2002), thus increasing their cognitive loads (Winslow, 1996). Not only would the learner have to know how to devise program logic which he or she would then translate into code; he or she is also required to understand the semantics and syntax of the language itself; the former being the more difficult task of the two (de Barros, dos Santos Mota, Delgado, & Matsumoto, 2005).

In addition to this, learners are also required to understand how to use the Program Development Environment (PDE) they were instructed to use, the common examples of which are designed for use by experienced programmers, with features such as basic syntax highlighting, automatic keyword completion, in-depth debugging tools, and generation of entire code segments (Vogts, Calitz, & Greyling, 2008). The error messages returned by these PDEs, while informative, are not always helpful for learners as, according to Nienaltowski, Pedroni, and Meyer (2008), the usefulness of error messages does not always lay in the amount of information they contain; what matters is the manner by which these error messages were presented.

In the analysis of programming behavior by Kummerfeld and Kay (2003), students learning how to program for the first time are not as good in understanding error messages, even if the error messages were informative. Experienced programmers, who have a deeper understanding of a programming language, benefit more from informative error messages, and tend to react and formulate solutions faster with regards to them (Kummerfeld & Kay, 2003). These studies reveal a need for adaptive feedback, to instruct novice programmers on how to interpret error messages. This finding parallels the recommendations for formative feedback found in a review by Shute (2008); feedback must be "valid, objective, focused and clear" and "if feedback is not specific or clear, it can impede learning and frustrate learners" (Shute, 2008).

Several approaches were considered in this research for diagnosing code errors and providing adaptive feedback to learners, among them are the Value-based Diagnosis Model proposed by Mateis, Stumptner, and Wotawa (2000), the Modified Reiter's Algorithm used by de Barros, Delgado, and Machion (2004) for PROPAT, Intention-based Detection employed in PROUST (Johnson & Soloway, 1986) and partly by JITS (Suarez & Sison, 2008), and Case-Based Reasoning (Leake, 1996) implemented in an Intelligent Tutoring System by Reyes (2002).

Case-based reasoning (CBR) by Leake (1996) is a knowledge retrieval and acquisition technique that uses past experiences and solutions to solve future problems. CBR does this by representing previously encountered problems and their respective potential solutions as a case, and storing all experienced cases in a case base. When a novel problem is encountered, the case-base is searched for the closest possible match to the novel problem. The solution associated with this match is then adapted to the new problem; this adapted solution will then be evaluated based on how well it addressed the problem. This new problem solution pair will then be stored in the case-base as a new case, to serve as an additional reference when another new problem is encountered.

CBR has four major phases; Leake (1996) defines them as Case Retrieval, Case Adaptation, Case Evaluation and Case Storage, while Aamodt and Plaza (1994) defines them as Retrieve, Reuse, Revise and Retain. Fig. 1, adapted from Aamodt and Plaza (1994) illustrates these four major phases. Every problem that needs a solution is treated as a case in CBR (Aamodt & Plaza, 1994) describes a case as a "problem situation", which is an "experienced situation" that has been learned by the CBR system. It is a combination of the problem introduced to the CBR system and the suggested solution to the problem. When a new problem is introduced to the CBR, a case is retrieved from the case base; this is Case Retrieval. Before presenting this retrieved case, it is first adapted to more accurately solve the newly submitted problem; this is Case Adaptation. After this adapted case has been presented as a suggested solution, it is then evaluated to check if the solution addresses the new problem; this is Case Evaluation. Finally, this adapted solution along with the new problem will be stored as a case in the case base.

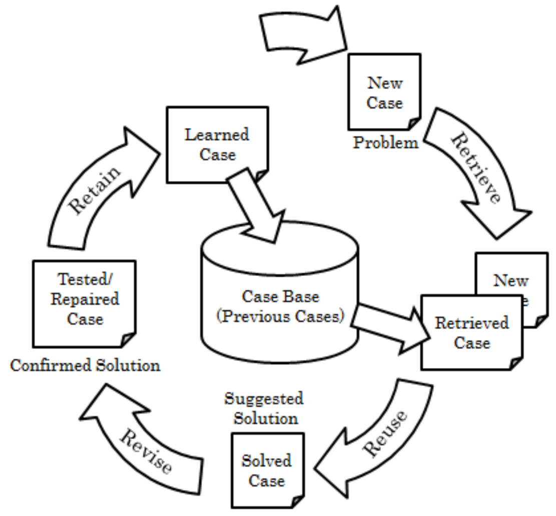

Fig. 1 The major CBR processes

A CBR based approach was selected for this research because of its advantages over traditional rule-based approaches, as outlined by Leake (1996). In terms of knowledge acquisition, traditional rule-based approaches might have difficulty generalizing rules from data, and if applied to real time diagnosis it might take time for rules to emerge as the system would require some amount of training. CBR on the other hand works with readily available cases, which, by the nature of CBR, is built in real time (Leake, 1996), which is required for this research as it involves real time feedback generation. Related to this, traditional rule-based approaches require adequate initial representation of possible scenarios, and would have to undergo retraining if novel scenarios are introduced. This makes it less suitable for diagnosis and remediation, as the misconceptions of students are not finite. Since CBR learns incrementally (i.e., it doesn't attempt to learn everything at once), it can adapt constantly. New cases may be added to the case base without having to reset the system (Leake, 1996). In addition, CBR based approaches can also store cases where feedback was not properly given, or the feedback did not completely address the misconception of the student, thus enabling CBR approaches to warn against problematic solutions (Leake, 1996). This is useful in feedback generation, as not all feedback will automatically result in successful remediation.

1.2 Research Objectives and Scope

The general objective of this research is to test the effectiveness of a pedagogical programming environment, implementing a CBR-based approach to diagnosis and remediation that provides meaningful feedback in the form of instructional error messages for logical errors on novice programmers learning the C language for the first time. In line with these objectives, we developed a pedagogical programming tool called CBR-C that makes use of a CBR (Leake, 1996) based approach to provide adaptive feedback to students learning how to program for the first time, which is able to give feedback despite having insufficient cases in its case base.

This research focuses only on and is constrained by the subset of the C programming language covered by the Introductory to Computer Programming (COMPRO1 or COMMAT1) subject of De La Salle University Manila (DLSU). These topics are:

- Basic Programming Concepts and the Basic Program Skeleton
- Tokens and Expressions
- Basic Input and Output Statements (printf and scanf)
- Conditionals
- Iterative Statements (Event-controlled and Count-controlled loops)

Because of this, CBR-C was developed specifically for programmers learning the C language for the first time; CBR-C only works efficiently on programs of the said scale. In addition, only logical errors would be covered by this research; syntax errors were not addressed by CBR-C. It is expected that, before using the CBR-C, the novice programmer must have at least completed writing a whole program. The PDE will not be able to successfully provide appropriate feedback if CBR-C was used it in the middle of coding.

CBR-C generates four levels of feedback depending on the number of cases in its case base and the required remediation of the student. This approach was used to address the issue of insufficient cases in the case base. Table 1 shows these four feedback levels. CBR-C will always attempt to generate the highest feedback level possible.

Table 1 Levels of feedback provided by CBR-C

Level	Description
0	The correct code is revealed to the student.
1	The results of test cases will be revealed.
2	The differences between the closest non-faulty code to the submitted code will be revealed to the student.
3	Explanation on the underlying misconception will be given to the student. The teacher has the option to give this feedback if the students were not prepared to give these explanations.

Level 3 feedback can only be given if there exists a case in the case base that sufficiently matches the new case. This feedback level provides a detailed description of the error found in the student's code, and is given either by other students who have solved the same problem in the past, or by the teacher if the student cannot adequately explain the error. In the absence of Level 3 feedback, CBR-C will expose the difference between the closest matching non-faulty code and the code of the student. The teacher is required to introduce at least one correct solution to CBR-C; this could be used as a basis for Level 2 feedback if the code of the student matches this correct solution enough. Level 1 feedback is given when both Level 2 and 3 feedbacks cannot be given. CBR-C will simply reveal the results of testing the code of the student against a set of test cases introduced to CBR-C. Finally, should the student give up, Level 0 feedback will be given.

2. METHODOLOGY

2.1 Evaluating Feedback Generation of CBR-C

To test the effects of CBR-C generated feedback on students learning to program in C for the first time, a preliminary experiment was conducted on twenty six Manila Science High School students. These students are incoming third year high school students who would be learning C programming on their third year. The hypothesis was that programming with the help of feedback from a tool like CBR-C would be better than programming without. In this regard, other features of CBR-C were not factored into the experiment and were disabled.

These students were divided into two groups, a Control group which did not receive any feedback apart from whether their submitted code was buggy or not, and an Experimental group, which received feedback from CBR-C. The division of these two groups was done at random. Both groups were given access to Dev C++ only until they were able to remove any Syntax Errors manifested in their code. Afterwards, both groups were required to use only a text editor (Notepad) to debug their code. The students were to submit their code to the experimenter in charge of their group once they have successfully removed any syntax errors. Their code was sent via a network tool to the machine of the experimenter.

For the Control group, once the experimenter was notified of a submission, the experimenter then compiled the submitted code. The compiled program was checked against test cases to see if the code is buggy or not. If it was, then the student was informed that his or her code was no longer buggy; otherwise, the executable (.exe) file was sent back to the student for them to test and fix. For the Experimental group, the same procedure was used but instead of compiling the student's submitted code, the experimenter introduced the code to CBR-C. The feedback generated by CBR-C was then presented to the student and the executable (.exe) file was sent back to them to test and fix. This was repeated until the submission of the student had met all the given programming requirements.

For the experiment, Level 2 Feedback was not given, and Level 3 Feedback was given as much as possible. In the absence of relevant Level 3 Feedback, Level 1 Feedback was given instead. Level 2 Feedback was not given because student profiling could not be feasibly performed. Extra cases were also added into the Case Base of CBR-C, since the primary goal of the experiment is to see if giving feedback actually helps students determine the underlying misconceptions in their code and if doing so could help improve code quality. These cases were built from problems given to the students prior to the experiment.

2.2 The Odd Sum Problem Experiment

The programming problem the students were tasked to solve was the Odd Sum Problem shown in Fig. 2.

Problem Description: Until the user inputs 0, keep asking for integer inputs. Get the sum of all the odd inputs and display the result. Do not put input or output prompts in your solution.

Fig. 2 The description of the Odd Sum Problem

Four test cases as shown in Table 2 were used to test the correctness of the student submissions. In addition, eleven extra cases were introduced into CBR-C. While not all possible misconceptions were represented in these cases, since these came from past submissions of the students involved in the experiment, it is hoped that these cases were adequate enough to represent common misconceptions for the group.

Table 2 The test cases for the Odd Sum Problem

Test Case Number	Input	Expected Output
0	0	0
1	1 3 5 7 0	16
2	2 4 6 8 0	0
3	1 2 3 4 5 0	9

3. RESULTS AND DISCUSSION

3.1 Effects of CBR-C Feedback on Students

Of the students in the Experimental group, five were able to resubmit again after their first submission. These were the students who were given feedback. Two of these were given Level 3 Feedback while three were given Level 1 Feedback. From the Control group, nine students were able to give a first and second submission. They were told that their code was incorrect, no other feedback or guidance was given to them. Table 3 contains the average code quality of the first and second submissions of the Experimental and Control group and the average difference in code quality between two, computed using the pqGram Tree Edit Distance Approximation metric (Bille, 2005) against the correct code of the teacher. It was observed that the codes submitted by the students did not deviate from the correct code of the teacher, and that the buggy codes submitted had the same intention as the correct code of the teacher, therefore code quality can be checked against the code of the teacher.

Table 3 Results of the Odd Sum Problem Experiment

	Average Code Quality of First Submission	Average Difference between the Second and First Submissions	Average Code Quality of Second Submission
Experimental	72.04%	9.92%	81.97%
Control	72.43%	2.18%	74.61%

Following Sison (2009), the Wilcoxon Rank Test for statistical significance was used to determine if there was a significant difference between the improvement of the code qualities of the submissions of the Experimental and Control groups. This was because it was not immediately clear that the data gathered falls under the normal distribution. The test yielded a p-value of 0.0548, which means that the difference between the improvements of the quality of the codes was mildly statistically significant given the chosen significance level (0.05).

4. CONCLUSIONS

In this research, CBR-C, a Pedagogical Programming Tool that implements a Case-Based reasoning approach, has been developed. This tool provides meaningful feedback for diagnosing and remediating misconceptions that give rise to logical errors found in the code of novice programmers learning the C language for the first time. This meaningful feedback is in the form of instructional error messages that provides more information through the form of hints, clues or explanations of the underlying misconceptions of the students, as suggested by Sison, Numao, and Shimura (2000).

Experiments showed that receiving feedback from CBR-C results in higher code quality than receiving no feedback, therefore receiving feedback is better than not receiving feedback.

CBR-C does not incorporate student profiling into its approach. Student profiling, even as simple as a survey form to be filled up prior to using the tool, could be helpful in identifying what kind of feedback a student might appreciate. A student might not want in depth feedback immediately; he or she might prefer to be given hints first before receiving a full explanation of the misconceptions present in his or her code. This could serve as an additional factor in deciding which level of feedback to give a student.

5. REFERENCES

Aamodt, A., & Plaza, E. (1994). Case-Based Reasoning: Foundational Issues, Methodological Variations, and System Approaches. *AICom - Artificial Intelligence Communications*, 39-59.

Bille, P. (2005). A Survey on Tree Edit Distance and Related Problems. *Theor. Comput. Sci.*, 217-239.

de Barros, L., Delgado, K., & Machion, A. (2004). An ITS for programming to explore practical reasoning. *Proceedings of the Brazilian Conference on Computer Education.*

de Barros, L., dos Santos Mota, A., Delgado, K., & Matsumoto, P. (2005). A tool for programming learning with pedagogical patterns. *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange* (mp. 125-129). San Diego, California: ACM.

Jenkins, T. (2002). On the Difficulty of Learning to Program. *3rd annual Conference of LTSN-ICS.* Loughbourgh.

Johnson, W., & Soloway, E. (1986). PROUST: knowledge-based program understanding. Sa *Readings in artificial intelligence and software engineering* (mp. 443-451). San Francisco: Morgan Kaufmann Publishers Inc.

Kummerfeld, S., & Kay, J. (2003). The neglected battle fields of syntax errors. *Proceedings of the fifth Australasian conference on Computing education - Volume 20* (mp. 105-111). Darlinghurst: Australian Computer Society, Inc.

Leake, D. B. (1996). *Case-based reasoning: Experiences, lessons and future directions.* Cambridge, MA: MIT Press.

Mateis, C., Stumptner, M., & Wotawa, F. (2000). A value-based diagnosis module for java programs. *11th International Workshop on Principles of Diagnosis (DX).*

Nienaltowski, M., Pedroni, M., & Meyer, B. (2008). Compiler error messages: what can help novices? *Proceedings of the 39th SIGCSE technical symposium on Computer science education* (mp. 168-172). New York: ACM.

Reyes, R. (2002). Using Case-Based Reasoning for Adaptive Tutoring. *Doctoral Degree in Computer Science Dissertation.* Manila, Philippines: De La Salle Univerisy-Professional Schools Inc.

Shute, V. (2008). Focus on formative feedback. *Review of educational research*, 153-189.

Sison, R. (2009). Investigating the Effect of Pair Programming and Software Size on Software. *Proceedings of Software Engineering Conference, 2009. APSEC '09. Asia-Pacific*, 187 - 193.

Sison, R., Numao, M., & Shimura, M. (2000). Multistrategy Discovery and Detection of Novice Programmer Errors. *Machine Learning*, 157-180.

Suarez, M., & Sison, R. (2008). Automatic Construction of a Bug Library for Object-Oriented Novice Java Programmer Errors. *Proceedings of the 9th international conference on Intelligent Tutoring Systems* (mp. 184-193). Montreal: Springer-Verlag.

Vogts, D., Calitz, A., & Greyling, J. (2008). Comparison of the effects of professional and pedagogical program development environments on novice programmers. *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology* (mp. 286-095). New York: ACM.

Winslow, L. (1996). Programming pedagogy: A psychological overview. *SIGCSE Bull.*, 17-22.