LEARNING TO PROGRAM USING PART-COMPLETE SOLUTIONS

Stuart Garner

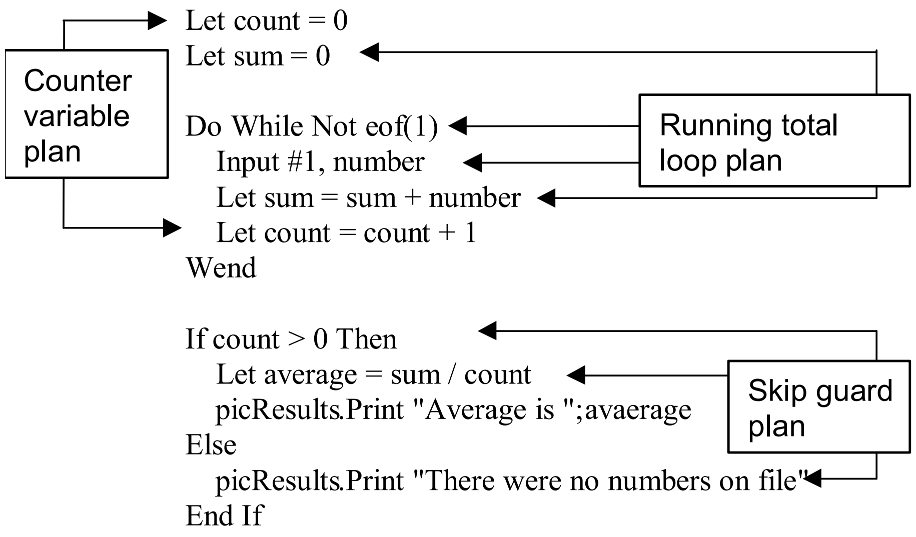
ABSTRACT
Learning to write computer programs is not an easy process for many students with students experiencing high levels of cognitive load. This paper discusses the "completion" method of learning to program that requires students to complete solutions for incomplete programs that have been given to them. A software tool called CORT (Code Restructuring Tool) which supports the "completion" method of learning to program and that has been developed by the author is then described. Initial evaluations of the tool indicate that students who use CORT require less time to solve problems and require less resources than non-CORT students. There was no difference in the performance of both groups of students in a final examination.

KEYWORDS
Learning programming, completion method, computer assisted learning

INTRODUCTION

Learning to write computer programs is not an easy for many students and low levels of achievement are experienced by many in first programming courses. Guzdial et al (1998) suggest that of all the potential project activities that a student might engage in, programming is probably one of the most difficult. Students have difficulty programming, both with doing it and with learning it. Most students typically learn little about programming in their first programming classes (Kurland et al., 1986), and, even in later classes, their programs tend to be buggy and they make inappropriate assumptions (Soloway et al., 1982). Winslow (1996) concludes that even when students know how to solve a problem by hand, they do not know how to translate it into a valid program.

Expert programmers have the necessary cognitive schemata to easily perform familiar programming tasks and also to interpret unfamiliar situations in terms of their generalised knowledge (Van Merrienboer & Paas, 1990b). In the domain of programming these specific schemata are known as programming plans and they are learned programming language templates, or stereotyped sequences of computer instructions, that form a hierarchy of generalised knowledge. Examples of programming plans are shown in figure 1.



```
Let count = 0
Let sum = 0

Do While Not eof(1)
   Input #1, number
   Let sum = sum + number
   Let count = count + 1
Wend

If count > 0 Then
   Let average = sum / count
   picResults.Print "Average is ";avaerage
Else
   picResults.Print "There were no numbers on file"
End If
```

Figure 1. Examples of Programming Plans

The question then arises as to how educators can encourage the development and construction of such schemata in novice programmers. One approach is to use the “reading” method of learning programming in which students study algorithms. However, there is a need to require students to mindfully “abstract” (think about) the algorithms and plans being studied and one approach is to present students with part-complete solutions that the students have to attempt to complete.

A software tool called CORT (code restructuring tool) has been developed by the author to support this method of learning programming. This paper describes CORT together with its preliminary evaluation with students.

THE READING METHOD OF LEARNING TO PROGRAM

The “reading” approach to learning programming emphasises the reading, comprehension, modification and amplification of non-trivial, well-designed working programs and an introductory programming course using this approach has four phases:

1. Students run and evaluate the strengths and weaknesses of working programs.
2. Students read and hand trace well structured working programs. Specific language features are learned by the study of these concrete programs.
3. Students modify and amplify existing programs. They are therefore introduced to design and coding.
4. Students generate programs on their own, developing design and structured coding skills.

This approach would appear to be an ideal one to follow in introductory programming courses. We do not, for example, expect students to construct English essays without having first read other essays and books, and so why should we expect students to learn programming without first studying existing programs? Most programming texts include worked examples for students to study. However this approach is not often used and I would put forward two possible reasons. Firstly it is difficult to motivate students to sit down and hand trace existing code. Unless there is some form of assessment associated with this process, students tend to skip and gloss over it. However a method where students were encouraged to study existing code would help them abstract appropriate schemata for use in subsequent problem solving. Secondly, the creation of appropriate worked examples for students to modify and amplify is a time consuming process. A great deal of thought is also required by the instructional designer in the selection of appropriate examples.

USE OF PART-COMPLETE SOLUTIONS

A lot of the work in this area has been carried out in the use of part-complete solutions and the learning of programming by Van Merrienboer and his colleagues (Van Merrienboer, 1990a; Van Merrienboer, 1990b; Van Merrienboer & De Croock, 1992; Van Merrienboer, Krammer, & Maaswinkel, 1994; Van Merrienboer & Paas, 1990). They argue that the traditional approach to the teaching and learning of programming is ineffective and that although the "Reading" approach is a better one to follow, the presenting of worked examples to students is not sufficient as the students may not "abstract" the programming plans from them. "Mindful" abstraction of plans is required by the voluntary investment of effort and the question then arises as to how we can get students to study the worked examples properly. In practice, students tend to rush through the examples, even if they have been asked to trace them in a debugger, as they often believe that they are only making progress in their learning when they are attempting to solve problems.

To encourage such mindful abstraction, use can be made of incomplete, well-structured and understandable program examples that require students to generate the missing code or "complete" the examples. This approach forces students to study the incomplete examples as it would not be possible for their completion without a thorough understanding of the examples' workings. An important aspect is that the incomplete examples are carefully designed as they have to contain enough "clues" in the code to guide the students in their completion. It is suggested that this method facilitates both automation, students having blueprints available for mapping to new problem situations, and schemata acquisition as they are forced to mindfully abstract these from the incomplete programs (Van Merrienboer & Paas, 1990b).

In one study, two groups of 28 and 29 high-school students from grades 10 to 12 participated in a ten lesson programming course using a subset of COMAL-80 (Van Merrienboer, 1990a). One group, the "generation" group, followed a conventional approach to the learning of programming that emphasised the design and coding of new programs. The other group, the "completion" group, followed an approach that emphasised the modification and extension of existing programs. It was found that the completion group was better than the generation group in constructing new programs. It was found that the percentage of correctly coded lines was greater and that looping structures were more often combined with correct variable initialisation before a loop together with the correct use of counters and accumulators within the loop. It would appear that the completion strategy had indeed resulted in superior schemata formation for those students within that group.

A side effect of the research was also noted. The drop-out rate from the completion group was found to be lower than for the generation group, particularly for female students with low prior knowledge. It was suggested that perhaps the generation of complete programs is perceived as a difficult and menacing task and that the completion strategy overcomes this difficulty.

CORT PROGRAM

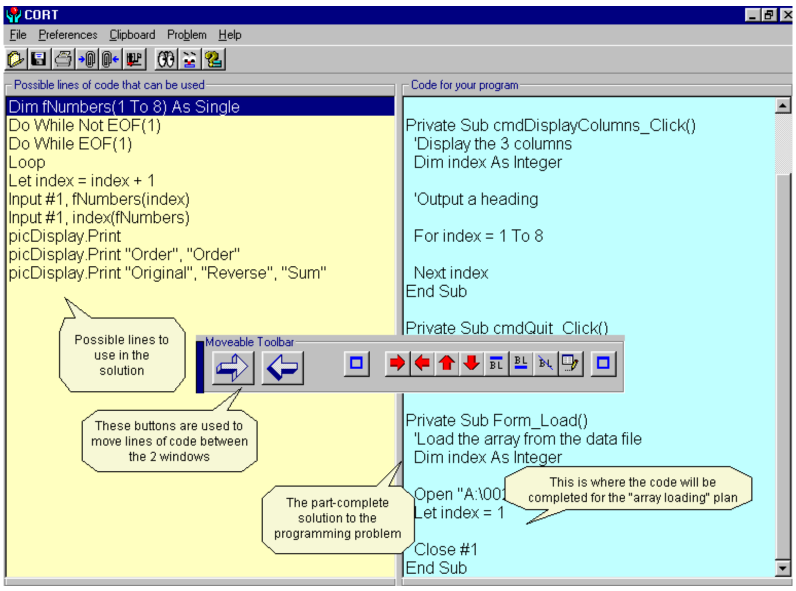
CORT (code restructuring tool) has been designed to support the "completion" method of learning to program and it has following features:

- Support for part-complete solutions to programming problems. Such solutions help in schemata creation and also reduce cognitive load.
- A mechanism so that missing statements can easily be inserted into a part-complete solution and also moved within that solution. This provides scaffolding for students.
- A facility so that students can add and amend lines of code. This would allow scaffolding to be reduced and for students to add more of their own code.

- For visual programming, a facility for students to easily view the target interface. The interface should be annotated with the various object names thereby reducing any split-attention effect and helping reduce cognitive load (Chandler & Sweller, 1991).
- A facility to access tutor created questions concerning the programming problems being attempted and for students to enter answers to those questions. This will promote reflection and higher order thinking.
- A facility to easily transfer a completed solution from CORT to the target programming environment.
- A facility to easily transfer programming code from the target programming environment back into CORT for further amendment.

The CORT interface is shown in figure 2.

USE OF CORT BY STUDENTS


CORT
File Preferences Clipboard Problem Help
Possible lines of code that can be used
Code for your program
Moveable Toolbar
Possible lines to use in the solution
These buttons are used to move lines of code between the 2 windows
The part-complete solution to the programming problem
This is where the code will be completed for the "array loading" plan

```
Dim fNumbers(1 To 8) As Single
Do While Not EOF(1)
Do While EOF(1)
Loop
Let index = index + 1
Input #1, fNumbers(index)
Input #1, index(fNumbers)
picDisplay.Print
picDisplay.Print "Order", "Order"
picDisplay.Print "Original", "Reverse", "Sum"
```

```
Private Sub cmdDisplayColumns_Click()
 'Display the 3 columns
 Dim index As Integer

 'Output a heading

 For index = 1 To 8

 Next index
End Sub

Private Sub cmdQuit_Click()

Private Sub Form_Load()
 'Load the array from the data file
 Dim index As Integer

 Open "A:\00
 Let index = 1

 Close #1
End Sub
```

Figure 2. CORT Interface

A student would typically use CORT as follows:

1. A student loads in a CORT file and the two windows display a part-complete solution to a problem together with possible lines to be used. There is a facility available for the contents of the two windows to be printed out.
2. The student can view the problem statement and the Visual BASIC solution interface by clicking on the appropriate buttons on the fixed toolbar. The problem statement may have already been provided to the student in the form of a handout, however there is also a facility to print it from within CORT.
3. The student moves certain lines from the left hand window to the right hand window in an attempt to complete the solution. Lines can be moved up or down, and indented or outdented in the right hand window. Some problems have too many lines in the left hand window, some of those lines being incorrect.
4. If necessary, the student can invoke a simple editor to amend, add or delete lines of code.
5. The student clicks on the appropriate button to copy the contents of the right hand window to the Windows clipboard.

6. The student invokes Visual BASIC and loads the file that contains the interface for the solution. This is in effect the Visual BASIC solution to the problem without the lines of code and was created by the tutor.
7. The student pastes the contents of the Windows Clipboard into the Visual BASIC editor and tests the program to determine if it works correctly. Use is made of the trace and debugging facilities of Visual BASIC. These facilities provide an insight to the workings of the notional machine.
8. If the student finds a problem with the working of the program, they can return to CORT and make the changes to the code there.
9. The student repeats steps 3 to 8 until they have a working program.
10. The student answers the tutor's questions concerning the programming problem that they have just attempted.

INITIAL EVALUATION OF CORT

Evaluation Design

An investigation took place over a period of one semester at Edith Cowan University in Western Australia, a semester being 14 teaching weeks. The unit that the students were taking was MIS2200, Software Development 2, which is an introductory programming unit for students within the School of Management Information Systems. Students are expected to gain fundamental programming knowledge in this unit including the three basic control structures, built-in functions, user-defined functions, event and general procedures, text file processing, and array processing.

The traditional way of delivering this unit is to have a two hour lecture, in which basic knowledge is introduced to the students together with methods of solving standard problems, and to have a one hour computer laboratory. In a laboratory, students are given programming problems to attempt to solve using Visual BASIC. If a student requires help then they usually ask fellow students or their tutor.

In the semester that the investigation took place, 58 students were enrolled in the unit and 53 went on to completion. Students had to enrol in one of four computer laboratories and they selected the particular laboratory at the time of enrollment. In the investigation, two of the laboratories were chosen to use the CORT program and the other two laboratories to have "conventional" programming exercises without CORT. In order to reduce possible bias, at the time of enrollment the students did not know whether they would be in the CORT or non-CORT group. Of the students who completed the unit, the number in the CORT group was 26 and the number in the non-CORT group was 27. The unit tutor had taught the unit before when the researcher was coordinating the unit. This was also the case during this research, all materials being provided for the tutor by the researcher.

The students in both the CORT and non-CORT groups had to fill-in journals after they had completed each problem. These journals were analysed and at the end of the semester as was the final closed-book examination.

Evaluation Results

The results of the closed book examination revealed that there was no difference in the performances between the CORT and non-CORT groups. However, the student journals revealed some interesting differences.

Time taken to complete programming problems

The sets of problems that the students had to attempt were different between the CORT and non-CORT groups, however they were similar in nature and their degree of difficulty. It was found that the CORT students took significantly less time than the non-CORT students. Figure 3 is a graph showing these differences.

Help Required to Solve the Programming Problems
Students could obtain help from the tutor in the computer laboratory, the textbook, or a fellow student. The research revealed that the CORT students required significantly less help than the non-CORT students. The results are shown in figure4.

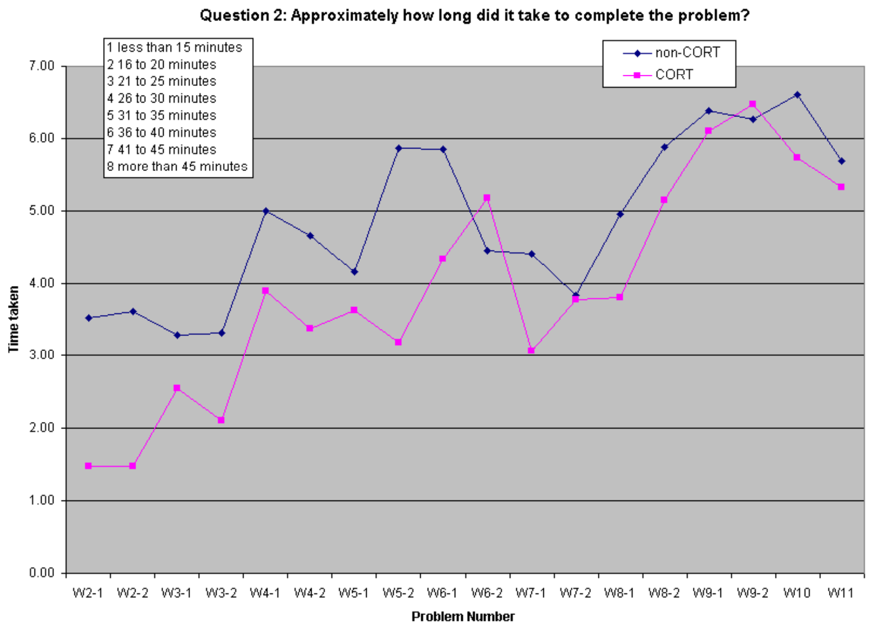


Figure 3. Time Taken to Complete Problems

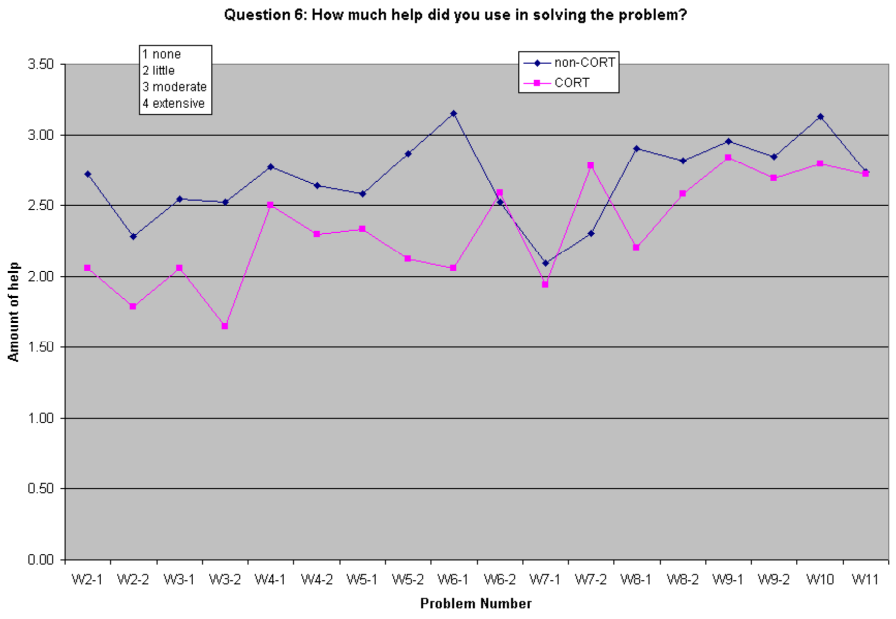


Figure 4. Help Required to Complete Problems

CONCLUSIONS

Results from the initial research into CORT and the "completion" method of learning to program suggest that there is no difference in the outcomes between CORT and non-CORT students. However, it can be seen that the CORT students took less time to complete their work and that they required less help. An argument can therefore be put forward that if the CORT students spent more time solving problems, such that their total time matched that of the non-CORT students, then they would most likely perform better in a final exam thereby suggesting that they had become better programmers.

The results are also of interest to e-learning and distance learning instructional designers. Learning to program “at a distance” is normally very difficult as students “hit brick walls” and require a lot of help from their tutors. The amount of help required by students should be reduced if CORT were utilised in their study.

REFERENCES

Chandler, P., & Sweller, J. (1991). Cognitive load theory and the format of instruction. Cognition and Instruction, 8, 293-332.

Guzdial, M., L., Hohmann, M., Konneman, C., & Walton, S., E. (1998). Supporting programming and learning-to-program with an integrated CAD and scaffolding workbench. Journal of Interactive Learning Environments, 6(1-2), 143-179.

Kurland, D., Clement, C., Mawby, R., & Pea, R.(1986). Mapping the Cognitive Demands of learning to Program. In Pea, R. & Sheingold, K. (Eds.), Mirrors of Minds (103-127). Norwood, NJ: Ablex.
Perkins, D. N., Schwartz, S., & Simmons, R. (1988). Instructional Strategies for the Problems of Novice Programmers. In R. E. Mayer (Ed.), Teaching and Learning Computer Programming: Multiple Research Perspective (pp. 153-178): Hillsdale, NJ: Erlbaum.

Soloway, E., Ehrlich, K., Bonar, J. & Greenspan, J. (1982). What do Novices Know about programming? In Badre, A. & Schneiderman, B. (Eds.), Directions in Human-Computer Interaction (87-122). Norwood, NJ: Ablex.

Van Merrienboer, J. J. G. (1990a). Strategies for Programming Instruction in High School: Program Completion vs. Program Generation. Journal of educational computing research., 6(3).

Van Merrienboer, J. J. G. (1990b). Instructional Strategies for Teaching Computer Programming: Interactions with the Cognitive Style Reflection-Impulsivity. Journal of research on computing in education, 23 Fall 1990(1).

Van Merrienboer, J. J. G., & Paas, F. (1990). Automation and Schema Acquisition in learning elementary computer programming. Computers in Human Behavior(6), 273-289.

Van Merrienboer, J. J. G., & De Croock, M. B. M. (1992). Strategies for computer-based programming instruction: program completion vs. program generation. Journal of Educational Computing Research, 8(3), 365-394.

Van Merrienboer, J. J. G., Krammer, H. P. M., & Maaswinkel, R. M. (1994). Automating the planning and construction of programming assignments for teaching introductory computer programming. In R. D. Tennyson (Ed.), Automating Instructional Design, Development, and Delivery (NATO ASI Series F, Vol. 119) (pp. 61-77): Springer Verlag, Berlin.

Winslow, L. (1996). Programming Pedagogy -- A Psychological Overview. SIGCSE Bulletin, 28(3).

Stuart Garner
Edith Cowan University
Pearson St.
Churchlands, 6023
Western Australia
Email: s.garner@ecu.edu.au