RoboProf and an Introductory Computer Programming Course

Charlie Daly
Department of Computer Applications
Dublin City University
Glasnevin, Dublin 9
353 (0) 1 704 5572

cdaly@dcu.ie

ABSTRACT

RoboProf is an online teaching system. It is based on WWW technology and can easily incorporate WWW-compatible media such as graphics, audio and video. It is structured as a self-paced course book: RoboProf presents the student with information on a closely-defined topic and then marks a set of exercises covering that material. When the students results are satisfactory, a new topic is introduced.

The idea behind RoboProf is to increase motivation by borrowing ideas from certain games. These ideas include providing a challenge, giving quick feedback, making progress visible and encouraging experimentation.

RoboProf was used to teach an introductory computer programming course. An introductory computer programming course must cover two main areas, the computer model (syntax and semantics of a programming language) and program design. In this paper I show how RoboProf can be effectively used to help teach the syntax and semantics of a programming language.

1. INTRODUCTION

Teaching and learning programming is not easy. Students must simultaneously learn a programming language and program design. It is ineffective to teach one of these in isolation from the others. Teaching design without the language is inefficient, as the student gets no practice in the techniques. Teaching the language without the design tends to encourage bad programming practice as well as providing little motivation for the students. Teaching both together is a juggling act that students find quite difficult.

RoboProf [2] teaches the language first using artificial motivation to encourage the students. Students log onto RoboProf and read the notes about the topic they are learning. When they are ready they are asked questions or given exercises on the topic. Their submissions are marked immediately and if they don't get full marks they are shown the correct answer. There are no penalties for an incorrect submission, a student must simply complete a similar problem. A student will work through the instructional material and gain practical coding experience by solving the exercises/problems.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ITiCSE '99 6/99 Cracow, Poland
© 1999 ACM 1-58113-087-2/99/0005...$5.00

At DCU we have a 12-week introductory programming course. There are two lectures and two laboratory sessions per week. This year RoboProf was used for the first 6 weeks to help teach a subset of the syntax, semantics and some idioms of C++. The module was delivered to 48 mathematics students.

The RoboProf exercises were part of the continuous assessment (25% of the module marks). All the students who completed the module passed the RoboProf section of the course after six weeks, six weeks before the deadline.

Almost all the exercises were small programming problems. For example, one of the exercises showed a *for* loop that printed the numbers 1 to 10. The student was then asked to adapt the program so that it printed the numbers 5 to 20. The student would submit the modified program and be marked based on output. The marking engine is modular, and easily modified/customised. A separate marking module can be designed for a particular problem. This could mark the program based on the source code or speed of execution etc. The current problems are however marked almost purely on output. For the above problem, RoboProf did ensure that a for loop was used to control the printing. Most exercises followed this format, i.e. the student had to modify a short program (about 10 lines) and adapt it by altering several lines. The problems covered if statements, loops, nested loops, conditional expressions, arrays and strings.

If the student's program didn't get full marks, they were shown what the program did produce along with what it should have produced. The student could then correct the program and resubmit. Although there was no penalty for resubmission the students were told that if the programs were being continuously modified in a hope that one version might accidentally work then they would be penalised. All program submissions were filed and could be later checked by the lecturer.

Students could log into RoboProf using a standard WWW browser and do the exercises at any time and anywhere that they had Internet access. However RoboProf was most heavily used during the scheduled (and supervised) lab sessions.

2. RoboProf in Action

In an introductory programming course students must write programs in order to develop a programming skill. It is however a lot of work to set programming tasks and to provide feedback

on all student submissions. Since many of the lecturers are themselves programmers it is not surprising that many automated systems have been developed, e.g. [1] [4] and [5].

Immediate feedback encourages students to try the problem and to feel satisfaction when they solve it. Each problem tends to build on the ones that went before, reinforcing the students' confidence in programming. It would very hard work to have so many exercises in a manual system and even if you did, the students would not get the benefits of immediate feedback.

Frequently in our labs students are quite shy about asking tutors for help. However because they had short clear goals, the students were focused on solving the problem and forgot their shyness. This increased their demand for tutors which we could only satisfy when we had a tutor for every 10 students. RoboProf helped with the administration of the exercises but it didn't eliminate the human help required to explain how one might go about solving a particular problem. This did place a large burden on the tutors and it didn't stop when the labs had finished as the students had learned the power of email. Undergraduate tutors are a useful resource that we used to fill the need.

The feedback provided by RoboProf is minimal but it is fast. More detailed feedback can be provided later as tutors get a report (via the Web) of students' progress. A tutor can send a message to a student if they notice anything funny about the code. This could include suggestions for alternative ways of solving the problem etc.

3. Implementation

RoboProf comprises a student database, a course database and a Java servlet (a servlet is the Java equivalent of a CGI program) which presents the module notes and problems. The servlet runs on a Unix server. An applet on the student's computer handles the program submissions. When a program is submitted it is compiled, and if successful it is run against a set of test data. The student is shown the output of the program and the expected output for each set of test data. The marking system can be specified separately for each problem. It is possible to plug in modules that analyse the submitted program and give a mark based on this. There is a number of default marking schemes, which mark a program depending on its output. This may compare the numbers in the output with the numbers in the expected output, or expect an exact match or use some other criterion. When a new programming problem is added to the system the instructor can specify use of a default marking scheme or may wish to supply a particular marking module. The test data can be fixed for each exercise or generated randomly for each submission, with the appropriate output generated by the engine to check the student's output. This could be useful if a student adapted a program to check what the input data was and simply print the appropriate output. In fact students sometimes submitted a program that they knew wasn't working just so they could see the sample input and output.

One big advantage of Java is that it allows dynamic loading of program modules. This means that the engine can load specific problem generation and testing modules without having to adapt the engine itself. Each problem can be generated at run-time and adapted to the ability of the student. The marking software can be made arbitrarily sophisticated for each problem. Of course this would require writing programs to add this behaviour. The default behaviour is to present a constant specification, to mark based on output and to read a file containing the required test data. In the current implementation almost all the problems were designed using the default behaviour.

4. The Problems

There were 39 problems which broke down as follows

Variables/Arithmetic	5
Control Flow (if/for/while)	19
Arrays	10
Strings	5

An example of an arithmetic problem specification was "ask the user for a number and to print that number squared". The control flow problems included finding factors of numbers and using nested loops, e.g. printing out a rectangle of asterisks. The most difficult problems were in the array section. One was to find the index of all numbers greater than 10; another was to locate the smallest number in the array. Building on that was the problem "locate the smallest number in the array and set it to zero". The string problems involved reversing a string and capitalising all words in a string.

The following criteria were used to design problems. Firstly they

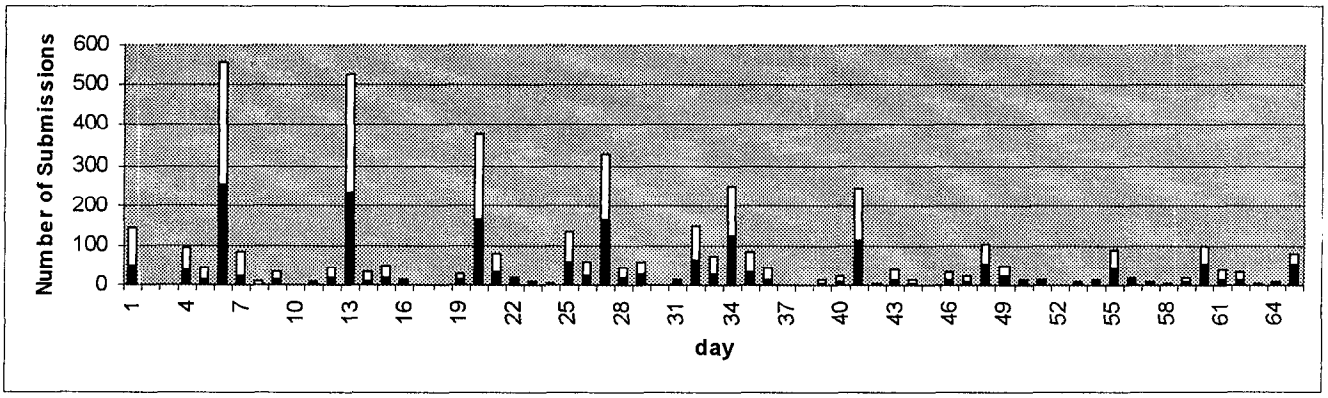

Figure 1. Number of program submissions per day. The shaded section indicates correct submissions.

had to be reasonably easy so that students would not despair or be tempted to plagiarise. Secondly they had to build on previous problems and finally they should help the student to build up a repertoire of useful programming idioms.

To help ensure that the output would match the input, the specification frequently included code that read the input and printed the output. This code could be copied directly from the browser to an editor.

5. Results

Forty-eight students used RoboProf over a nine-week period. All the students passed (got over 40%), and three-quarters got 90% or more. Figure 1 shows the problem submissions per day.

Most submissions were on a Wednesday (the lab session). On the first main lab session 230 correct programs were submitted. This averages at over four programs per student. And this is without the pressure of deadlines. Although students were encouraged to finish within six weeks (to keep up with the lectures) the deadline wasn't till the end of the semester (six weeks later). The number of submissions to RoboProf gradually decreases over time. Some of the more enthusiastic students completed all the problems early and stopped using the system (see figure 2).

Figure 3 shows the usage by student, i.e. the number of students who submitted a program on a given day. Over 41 students submitted a program on day 13. As students became more familiar with RoboProf they became less reliant on the labs and the supervisors and used the system on non-lab days.

Figure 4 shows the number of submissions per problem. There were 39 problems and the average number of submissions was 2.6. (I.e. on average a student attempted each problem two or three times.) One student managed to attempt problem 14 twenty times. If the number of attempts is high then it could indicate that the problem is difficult or that the specification is unclear.

6. Conclusions

RoboProf provided an environment where students could improve their programming skills by completing gradually more difficult tasks. Students completed the tasks well ahead of the deadlines. The challenge and the immediate feedback kept the students interested. They also appeared to be more focused in the lab.

The fact that a student had to compile and run the programs ensured that they were well versed in the low-level details of compiling a problem. RoboProf did not attempt to teach problem solving techniques. Rather the aim was that a student would have a good grasp of the syntax and semantics of a programming language and also become acquainted with a large number of programming idioms. This would place the student in a good position to learn program design and problem solving – the

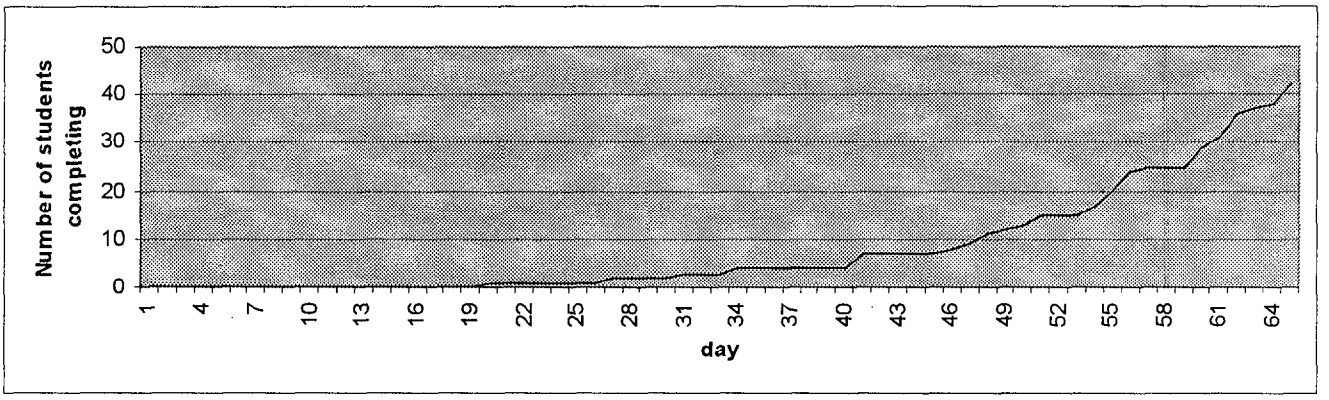

Figure 2. Number of students who completed all RoboProf problems per day

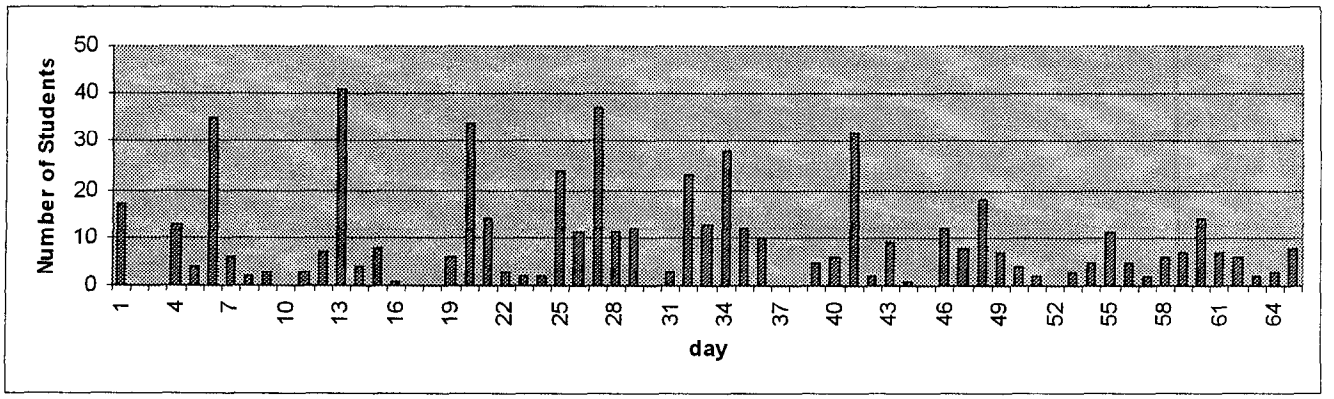

Figure 3. Student usage by day (number of students who submitted at least one program on a given day)

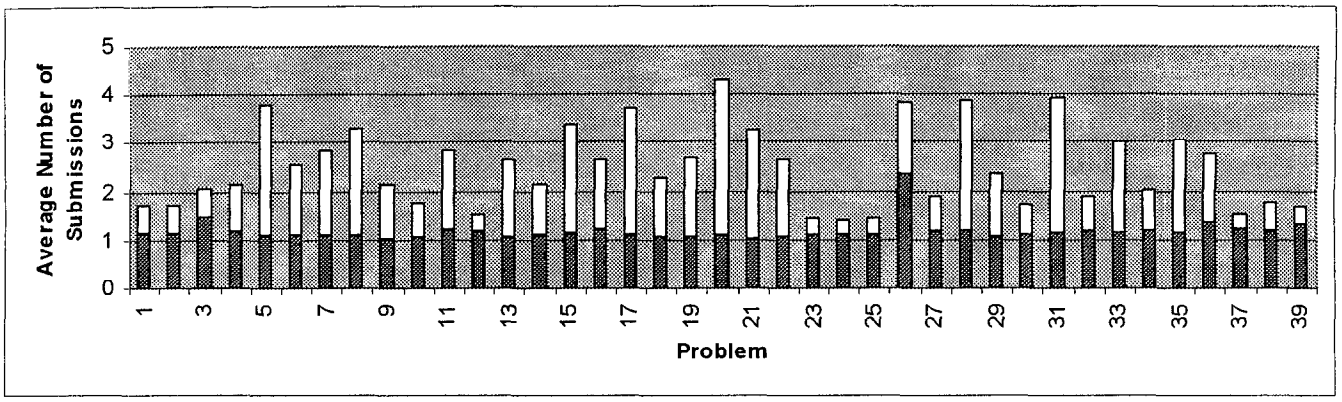

Figure 4. Average number of submissions per problem. Correct submissions are shaded

second part of the course.

The design and use of RoboProf was based on trying to capture the motivating power of some computer games. For example in the popular game Quake, a player completes challenges, gets immediate feedback and thus progresses through the game. The RoboProf engine supplies the framework and the problems must be designed to fit in with this concept. It must be easy to use, encourage experimentation (by not penalising incorrect submissions) and reward success.

Students did use the system consistently. In fact one problem was that some students found RoboProf quite addictive and spent too much time at it. It might be necessary to limit the time students spend on the system.

A final point: I suspect that the activity of surfing (whether surfing the Web or channel surfing with television) allows, in fact almost encourages, students to develop bad habits of concentration. It is too easy to click away from any difficult concepts. RoboProf endeavours to focus students in a way rarely achieved with Web based media.

6.1 Future work

Work has already commenced on a number of enhancements to RoboProf. A plagiarism detector has been added. A lot of effort is being put into analysing student answers. If a particular incorrect answer is recognised by the system then the student can be presented with a path through the notes to cater for that particular misconception. The many student submissions are being analysed to detect common errors. RoboProf will then be updated to recognise these errors and ask further questions to try to get the student to correct their misconception. See [3] P158 for a discussion of this.

The data based on number of attempts, time spent on the problems etc. is also being analysed to see are there indicators which might help locate students having difficulty.

7. ACKNOWLEDGMENTS

I would like to thank Dr Colin Coleman for his insightful and useful comments on the paper and RoboProf.

8. REFERENCES

[1] Benford S. D. et Al *A system to teach programming in a quality controlled environment* The software quality journal, V2, 1993.

[2] Daly C., McGee P. *RoboProf: an Automated learning environment* Proceedings CTC Dublin City University, 1995.

[3] Laurillard D. *Rethinking University Education*, Routledge 1993.

[4] Rosbottom, J. *Computer managed, open question, open book assessment* proceedings ITiCSE Uppsala Universitet 1997.

[5] Thorburn, G., Rowe G. *PASS - an automated program assessment system* Proceedings CTC Dublin City University, 1996.