Synthesis and Analysis of Automatic Assessment Methods in CS1

Generating intelligent MCQs

Des Traynor
Dept. Computer Science
National University of Ireland, Maynooth
Co.Kildare, Ireland
dtraynor@cs.may.ie

J. Paul Gibson
Dept. Computer Science
National University of Ireland, Maynooth
Co.Kildare, Ireland
pgibson@cs.may.ie

ABSTRACT

This paper describes the use of random code generation and mutation as a method for synthesising multiple choice questions which can be used in automated assessment. Whilst using multiple choice questions has proved to be a feasible method of testing if students have suitable knowledge or comprehension of a programming concept, creating suitable multiple choice questions that accurately test the students' knowledge is time intensive.

This paper proposes two methods of generating code which can then be used to closely examine the comprehension ability of students. The first method takes as input a suite of template programs, and performs slight mutations on each program and ask students to comprehend the new program. The second method performs traversals on a syntax tree of possible programs, yielding slightly erratic but compilable code, again with behaviour that students can be questioned about. As well as generating code these methods also yield alternative distracting answers to challenge the students. Finally, this paper discusses the gradual introduction of these automatically generated questions as an assessment method and discusses the relative merits of each technique.

Categories and Subject Descriptors

K.3.2 [**Computer and Information Science Education**]: Computer Science Education, Self-assessment

General Terms

Design, Human Factors

Keywords

Assessment, Program Comprehension, First Year Programming

1. INTRODUCTION

It is difficult to assess the programming ability of students in introductory computer science. This was highlighted in 2001[10] when an international study showed that the majority of students lack the basic programming skills after they have completed a year of computer science studies. If suitable programming assessment was in place, this lacuna in student knowledge would have been far more obvious. Whilst it is convenient to place the blame on students, and it may seem reasonable to place blame on lecturers, it is true to say that if effective programming assessment was enforced then the impact of such a large study would have been greatly reduced. For too long, improper assessment had incorrectly awarded students artificially high grades, thus leaving the discipline in a state of shock when the results from the study arrived. Assessment is the key to learning[16]. Poor assessment presents two problems, it does not highlight the weak students accurately, nor does it challenge the promising students appropriately[1].

Automated assessment methods involving machine analysis of output from programs[3] have proved to be a successful method of analysing students' ability to write code; however this is not the only learning outcome we expect from introductory computer science. It is also important that students can understand and comprehend code. It has also been argued that comprehension should be examined, before asking students to apply their knowledge, in Blooms taxonomy of educational objectives[2] comprehension is one stage before application.

One clear and objective means to estimate students' ability to comprehend programs is to use multiple choice questions (MCQs). It is a common misconception that MCQ tests are easy and the lazy way out for lecturers, it has been noted by Lister and others that *Quality MCQs are not the work of the lazy!*[9]. The MCQs used by Lister typically involves a piece of code, and questions regarding the behaviour of the code when executed. It is the goal of this research to automate the process of creating high quality MCQs through using automatically generated fragments of code and providing a restricted number of 'intelligently chosen' possible answers.

[1]An alternative outcome from poor assessment occurs when the assessment is too challenging. In this case students are demotivated by poor scores and the lecturer again gains little insight from analysis of the results.

Which of the following is the correct declaration for the main function in Java?

a) Begin Program ()

b) public start main()

c) public static void main(String args[])

d) int program;

e) System.out.println("Hello World");

Figure 1: Weak MCQs encourage surface learning

1.1 Overview of the paper

The next section introduces some of the difficulties in automated assessment, and discusses the requirements for reasonable multiple choice questions. The third and fourth section of this paper details the two approaches we have taken to solving the problem, and presents sample output from each attempt. The fifth section compares the output from both methods and discusses their strengths and weaknesses. The final section presents preliminary results from using this technology in Maynooth University and also discusses future work for the project.

2. AUTOMATED ASSESSMENT

The clear advantages of automated assessment are objectivity and convenience. Assessment involves creating questions, carrying out the examination and marking the scripts. It is both difficult and time consuming to mark 100 student scripts in an accurate, unbiased manner. If this procedure can be automated in some way, whilst maintaining the quality of assessment, this will allow the teacher/lecturer more time to concentrate on the preparation and delivery of course material. It is important to state that many projects such as ExamGen[13] offer what they refer to as automatic test generation. Whilst this may seem a similar goal, what distinguishes the work in this paper is that the questions themselves are automatically generated. The majority of the other projects simply provide students with a random subset of a large question bank.

Educationalists have argued that MCQs encourage *surface learning* where the student can pay lip service to a topic by skimming through their notes and achieve a sufficient score in the exam[1]. It has also been said that the only real advantage is that of speed and convenience. As a result of these misconceptions, departments are justifiably cynical about the adoption of MCQs as an assessment method. Often they will cite extremely weak MCQs they have seen as examples of why it is an unsuitable assessment method. For example, the multiple choice question in Figure 1 could be correctly answered by any student with little or no knowledge of programming in Java, simply through skimming throw the first section of notes on most programming courses. This method of learning deserves little reward in any assessment method.

Lister[8] shows a more extreme example of this where all incorrect answers have artificially low feasibility thus rendering the question of no value. Another common criticism from educationalists is that any form of selected response testing is only capable of testing the lower levels of Bloom's taxonomy[2], however in the field of computer science examining students' ability to comprehend a program is of great value. The next section discusses the specifications of the requirements as a set of criteria for evaluating the quality of the MCQs.

2.1 Requirements for effective MCQs

To ensure that the MCQs that will be generated by this system will be of a high quality, there are certain requirements that each question must meet[2]. The current requirements are detailed in the following list.

1 **Good quality code.** Any code presented to the students must be of a high quality, and follow all proper guidelines given to students. Obfuscated code and spurious examples within code should not be allowed.

2 **No Tricks.** The question should focus on teaching the normal behaviour of programs, not the badly coded exceptions. There should be no *tricks* in the majority of the questions. (For example in C++ the statement `if(i=size)`will compile and always returns true, but is bad programming, similarly the expression `x+=x++;` is valid code but not good practice).

3 **Quality Distractors.** The alternative incorrect answers (distractors) must have a suitably high feasibility so as to ensure students are challenged by each question.

These requirements are useful as they serve to both clearly define what can be generated, and also ensure the quality of the output. The approaches to generating both code and questions were influenced by these requirements.

We studied a variety of different approaches for generating code and questions, ranging from grammatical evolution[14] to various different techniques extracted from genetic programming[11]. Two approaches that we deemed achievable were that of template based mutations, and random walks through a predefined syntax tree. These approaches were selected as they lend themselves to formal definition and could be implemented within the allowed time. We decided to pursue both of these approaches with the intent of evaluating their success as a means of assessment.

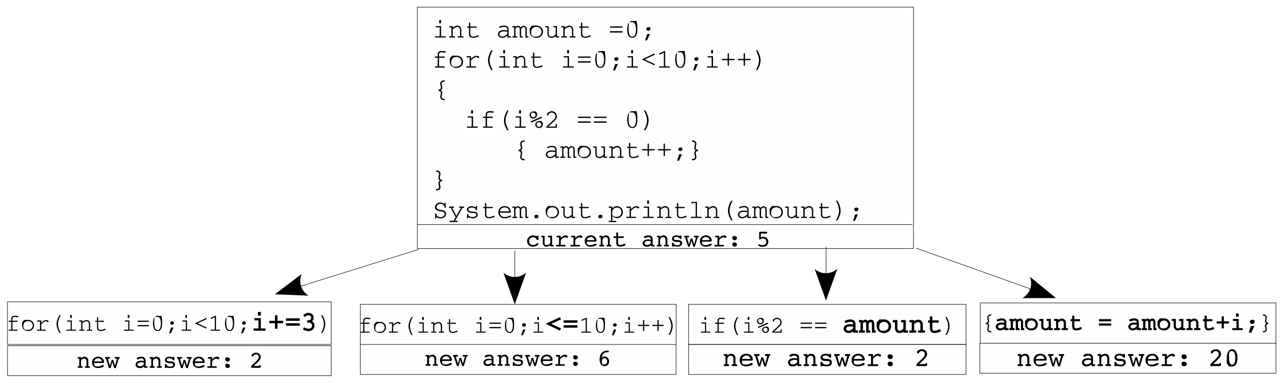
3. TEMPLATE BASED MUTATION

Template based mutation uses the mutation phase from genetic programming and applies it to a pre-approved population of problems.

In practice this means the lecturer will supply what they consider to be a suitable suite of problems, each one consisting of of a piece of code, and a question regarding its behaviour. This style of question was first proposed by Lister[7] in 2000 and has proved an effective method of probing student knowledge.

The mutations of the code take the form of minor changes that guarantee different behaviour from the program. The result of this is that one input template can result in a multitude of different questions generated. The mutations are

[2]It is current research beyond the scope of this paper to refine these requirements and construct models against which we can verify questions



```
int amount =0;
for(int i=0;i<10;i++)
{
  if(i%2 == 0)
     { amount++;}
}
System.out.println(amount);
current answer: 5
```

```
for(int i=0;i<10;i+=3)
new answer: 2
```

```
for(int i=0;i<=10;i++)
new answer: 6
```

```
if(i%2 == amount)
new answer: 2
```

```
{amount = amount+i;}
new answer: 20
```

Figure 2: One short code snippet can examine many concepts when altered slightly.

performed by randomly selecting a set of substitutions that should have an effect on the outcome of the code. These substitutions are usually related to the topic under examination. For example if a lecturer wished to examine students' knowledge of boolean operators, the substitutions could replace $<$ with $\leq$, $=$ with $\neq$ etc. The new program is then compiled and the correct answer is then retrieved through execution. Figure 2 gives a short example of how a piece of code can be automatically mutated to produce several different pieces of code, each with alternative outputs.

4. RANDOM TREE WALKS

The second approach investigated involves using randomization techniques for creating short pieces of code. Each piece of code is generated using a random *walk* through a very rich tree of potential programs. In Java there are certain lines of code that each program must display to compile and execute independently. The first part of the tree involves placing these declarations into the code and then making decisions based on the input parameters about how the program will be developed.

4.1 Input Parameters

The input parameters are used to define the constraints which govern the program being generated. The parameters are

- **The level of difficulty required.** This parameter (currently an integer 1..10), defines how difficult it should be to comprehend the program. It affects variables such as the length of statements, the levels of nesting (of both `if` statements and loops), the simplicity of loops(e.g. a simple loop would run from zero to some arbitrary number in increments of one, whereas a complex loop might have a variety of termination conditions, and non-trivial increments). In practice, code generated at a high level of difficulty can resemble obfuscated code which is not suitable as it does not satisfy the requirements.

- **The length of the program.** Depending on the length of the exam, certain lecturers may wish to have a small number of long programs, or a large number of short programs to test their students' knowledge. Whilst the authors find the latter yields more significant data, it was deemed necessary to include such a parameter to make the system adaptable. The length of a program is represented using an integer ranging from 1..10, where 1 would yield a program of approximately 5 lines, and 10 returns a program with between 40 and 60 lines of code.

- **The assumed knowledge of the students.** This parameter corresponds to a set of concepts that the students already have covered. For example, when writing a question based on arrays, it would aid in question construction to know that students have already covered `for` loops, `if` statements etc. This is implemented as an integer which corresponds to which level of learning the students are at. The lecturer supplies the level of learning and which concepts they represent.

- **The concept(s) being examined** This parameter represents the cynosure of the question. When creating the code the overall behaviour of the system should depend on this concept, thus making it crucial to understand when attempting to answer the question. In early tests of the system this necessity was overlooked and this resulted in poor questions. For example a question which alleged to examine `for` loops, would output code with `for` loops that would never be reached due to preceding `if` statements.

All of the above input parameters serve to influence the decisions made during the random code generation, for example if the difficulty of the question was set to 10 it is very likely that loops will nest inside each other, and that there will be more complicated boolean expressions inside `if` statements etc.

However generating code is not all that is required to generate questions. The third requirement was to generate a set of feasible answers that will distract the students sufficiently. This method for this is discussed in the next subsection.

4.2 Generating Feasible Answers

Automatically generating feasible answers requires a formal and reasonably rigorous definition of the concept of feasibility. This is a very difficult notion to define and represent in a program. Initial research showed that the most effective distractors were those that were based on student misconceptions. When writing feasible answers we believe the authors were intentionally mimicking the incorrect thoughts of

Misconception	Substitution
Division and Modulo	% with /
Confusing and with or	|| with &&
bounds checking	> with >=
if-else chains	`else if` with `if`

Table 1: Misconceptions and their Substitutions

their students, e.g students may confuse the AND operator (&&) with the OR operator (||) or confuse an array item's index with its value, so in a question examining knowledge of the operators one or more of the answers should check for this misconception. This is an effective method of generating feasible answers, and one which is also easily automated.

Novice misconceptions have been monitored in computer programming for a long time. Whilst the most significant work in this area covers older languages such as Basic[12] or Pascal [15] the work tends to be abstracted sufficiently to represent potential mistakes and misconceptions in most languages. In particular DuBoulay[4] identified the origins of these misconceptions and provided a list of mistakes common to all languages.

Modern, language specific studies have highlighted the most common programming mistakes in Java[6]. Complimentary to this a study of students' understanding of Java focused on many misconceptions that students use as rules when programming in the language[5]. Combining this recent research with the guidelines offered in the earlier works, it was decided to use misconceptions as a framework for creating alternative incorrect answers. This process is described as follows.

Each generated code segment is compiled initially and the correct answer is extracted by running the program. Then a set of substitutions based on the student misconceptions are performed and the code is recompiled and the alternative answer extracted. This is done once for each answer, until enough answers have been retrieved. If a substitution produces output that is already in the list of answers it is discarded. Table 1 shows an abbreviated list of the substitutions used for generating questions based on `for` loops and arrays; these substitutions can be performed in either direction and will usually alter the output of the program. During development it was found useful to also produce a highly infeasible answer, as this can identify students who are extremely weak in the subject and have resorted to clueless guessing. The final answer list is then shuffled to ensure a random ordering of the answers.

4.3 Example of output

Figure 3 shows a sample output from the random code generation where the topic examined was conditional statements (`if-else`) and while loops. The assumed knowledge in this case was operators, and this sample question had inputs of length=3 and difficulty=5. Applying the substitution technique yielded the following list of answers. The misconception associated with each answer is in square brackets.

a) test x=8,y=1 [correct answer]

b) x=11, y=25 [confusing % with /]

c) test x=11,y=31 [doesn't understand `if-else`]

```
int x = 89/9;
int y = x %5;
        if (x > y || y ==1)
        {
         System.out.println("test" );
        }
        else if (y<=x)
        {
                while(x <= 10)
                {
                 x++;
                 y+=x;
                }
        }
System.out.println("x=" + x + ", y="+ y );
```

Figure 3: The output from a typical run of random tree walking.

d) test x=9,y=4 [confusing && with ||]

e) test x=10,y=10 [Highly infeasible, student could only guess.]

5. RESULTS AND EVALUATION

This assessment method was in testing for the academic year 2003-2004, and was presented to students as an optional self-assessment method and proved extremely popular. Students were requested to provide feedback and comments about the system to aid in its development. Whilst the overall feedback was positive, when asked which style of questions (template based, or random) they preferred students provided some enlightening comments on which we can base future work.

The typical responses were that template based questions were good questions that examined given concepts very well. Students felt that if they scored highly in the template based tests, they were happy with their level of knowledge. The question were generated based on ten templates provided by the course lecturer.

The results were less encouraging for the fully random questions. Whilst the input from the lecturer was minimal (no work was required), the feedback was more critical. Students remarked that the questions did achieve a uniform difficulty and that sometimes the topic being examined was not central to the question. These issues were resolved later in the year, when the input parameters were modified to account for this.

In summary, whilst the template questions achieve a sufficiently high quality they require additional work from the lecturer; the randomly generated programs require little or no effort but produce less satisfactory results.

There will be a structured experiment in the following academic year comparing the re-developed system against traditional assessment methods. This will enable a full rigorous evaluation of automated assessment and The system will be deployed as a significant aspect of the students continuous assessment and feedback will again be requested to assist in further development.

6. CONCLUSIONS

The two alternative approaches have both shown that it is possible to achieve reasonable automated assessment. The system has been modified with the students' feedback taken into account and will be used in the following academic year for a series of assessment tests. In particular the output from template based mutations is highly satisfactory, whereas the full random code generation is still in its infancy as a method for question generation.

We acknowledge that automated assessment is by no means yet a suitable replacement for other, more traditional assessment methods. However we feel that this project provides a resource light additional assessment type which can work in unison with the more traditional methods. This complementary integration of traditional and innovative assessment is a more natural and pragmatic progression toward automating assessment techniques.

6.1 Future Work

We have decided to apply the techniques used in template based mutation to a variety of question styles. At present they are only suited to question asking students about how a program will behave (e.g. "What is the output? "), however there are a plethora of MCQ styles to which the techniques can be applied. At present all questions present code that will compile, however this technique could also be used to test if students can understand and apply basic rules of syntax and static semantics for compiling programs (e.g "Does this program compile, if not what is the error? ").

7. ACKNOWLEDGMENTS

The work presented in this paper is part of ongoing research in the National University of Ireland, Maynooth funded by the Irish Research Council for Science Engineering and Technology (IRCSET).

8. REFERENCES

[1] R. F. Biehler and J. Snowman. *Psychology Applied to Teaching*. Houghton Mifflin Company, 8th edition, 1997.

[2] B. S. Bloom and D. R. Krathowl. *Taxonomy of educational objectives*. McKay & Co, 1956.

[3] C. Daly and J. Waldron. Assessing the assessment of programming ability. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 210–213. ACM Press, 2004.

[4] B. duBoulay. Some difficulties of learning to program. In E. Soloway and J. C. Spohrer, editors, *Studying the Novice Programmer*, chapter 15, pages 283–301. Lawrence Erlbaum Associates, 1989.

[5] A. E. Fleury. Programming in Java: student-constructed rules. In *Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 197–201. ACM Press, 2000.

[6] M. Hristova, A. Misra, M. Rutter, and R. Mercuri. Identifying and correcting Java programming errors for introductory computer science students. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 153–156. ACM Press, 2003.

[7] R. Lister. On blooming first year programming, and its blooming assessment. In *Proceedings of the Australasian conference on Computing education*, pages 158–162. ACM Press, 2000.

[8] R. Lister. Objectives and objective assessment in CS1. In *Proceedings of the thirty-second SIGCSE technical symposium on Computer Science Education*, pages 292–296. ACM Press, 2001.

[9] R. Lister and J. Leaney. Introductory programming, criterion-referencing, and Bloom. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 143–147. ACM Press, 2003.

[10] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *SIGCSE Bull.*, 33(4):125–180, 2001.

[11] N. Pillay. Using genetic programming for the induction of novice procedural programming solution algorithms. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 578–583. ACM Press, 2002.

[12] R. Putnam, D. Sleeman, J. Baxter and, and L. Kuspa. A summary of misconceptions of high school BASIC programmers. *Journal of Educational Computing Research*, 2:459–472, 1986.

[13] A. Rhodes, K. Bower, and P. Bancroft. Managing large class assessment. In *Proceedings of the sixth conference on Australian computing education*, pages 285–289. Australian Computer Society, Inc., 2004.

[14] C. Ryan, J. J. Collins, and M. O'Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Proceedings of the First European Workshop on Genetic Programming*, pages 83–96. Springer-Verlag, 1998.

[15] J. C. Spohrer and E. Soloway. Novice mistakes: are the folk wisdoms correct? *Commun. ACM*, 29(7):624–632, 1986.

[16] R. C. Sprinthall, N. A. Sprinthall, and S. N. Oja. *Educational Psycholgy*. McGraw-Hill Education, 7th edition, 1998.