Computers & Education 54 (2010) 545–561

Contents lists available at ScienceDirect

Computers & Education

journal homepage: www.elsevier.com/locate/compedu

Marking student programs using graph similarity

Kevin A. Naudé *, Jean H. Greyling, Dieter Vogts

Department of Computer Science and Information Systems, Nelson Mandela Metropolitan University, Port Elizabeth, South Africa

ARTICLE INFO

Article history:
Received 10 February 2009
Received in revised form 31 August 2009
Accepted 3 September 2009

Keywords:
Programming and programming languages
Computer-assisted assessment
Graph similarity

ABSTRACT

We present a novel approach to the automated marking of student programming assignments. Our technique quantifies the structural similarity between unmarked student submissions and marked solutions, and is the basis by which we assign marks. This is accomplished through an efficient novel graph similarity measure (*AssignSim*). Our experiments show good correlation of assigned marks with that of a human marker.

1. Introduction

Automated marking has a long history of use in university programming classes. In very large classes, it brings an important pedagogical advantage: fast turnaround time for evaluation and feedback. This paper is about expanding the body of programs that may be marked automatically.

Traditionally, automated marking of programs takes the form of black-box testing. Student programs are subjected to a battery of test cases for which correct input–output is known a priori. Usually, the student's mark is the weighted sum of test cases passed.

This idea is very established. Hollingsworth's grader applied test data to punched card programs (Hollingsworth, 1960). There have been many systems since. Some examples are TRY (Reek, 1989), the Online Judge (Cheang, Kurnia, Lim, & Oon, 2003), and the recent Automatic Marker that integrates with Sakai (Suleman, 2008). Modern systems possess much greater sophistication, but carry the same main idea as Hollingsworth's original grader – that programs may be assessed indirectly by instead evaluating their output.

This approach does have problems. The most significant of these are that it:

- provides limited quality of feedback,
- places heavy constraints on what can be assessed,
- and can be very sensitive to small errors.

The first noted problem may be contested, because the feedback of failing test cases is sometimes quite useful. The problem is really that such feedback is not often pedagogically motivated. It also has the unfortunate side-effect of discouraging students from writing their own test cases – why do so when the marker program will tell you which test cases actually count? In this regard, ASSYST (Jackson & Usher, 1997) and the WEB-Cat Grader (Edwards, 2003) are commendable markers because they require students to submit their own test cases.

This paper focuses on the second and third problems. These issues deal with the validity and viability of marking based on black-box testing. It is clear that there are many varieties of programs that students may be expected to write which cannot reasonably be assessed by black-box testing. Examples of such programs are:

- Non-textual programs, such as
  - a turtle-graphics program to draw trees,
  - an animation of a bouncing ball;

* Corresponding author. Tel.: +27 41 504 2078; fax: +27 41 504 2831.
E-mail addresses: Kevin.Naude@nmmu.ac.za (K.A. Naudé), Jean.Greyling@nmmu.ac.za (J.H. Greyling), Dieter.Vogts@nmmu.ac.za (D. Vogts).

0360-1315/$ - see front matter © 2009 Elsevier Ltd. All rights reserved.
doi:10.1016/j.compedu.2009.09.005

```
bool isVowel(char ch)
{
    char TheChar = 'a';
    switch (TheChar)
    {
        case 'a': … case 'u':
            return true;
        default:
            return false;
    }
}
```

Fig. 1. Vowel function – almost correct.

- Interactive GUI programs, such as
  - a calculator program,
  - an address book database program;
- Tasks with specific algorithms, such as requiring
  - a merge sort (rather than, say, bubble sort), or
  - a recursive algorithm converted to use an explicit stack.

The third problem is that black-box testing does not produce stable scores. This is illustrated in Fig. 1 which offers a function for identifying vowels. The function is almost correct, having only one incorrect token. This error could easily be introduced by early testing. Unfortunately, the program also compiles and continues to fail 21 of 26 test cases. The indication is that small program changes can result in big changes to a student's mark. Clearly there is no inherent relationship between the specific number of test cases passed, and the amount of code which is correct.

Our approach is different in that we seek to assess program source code directly, by comparing the structure of submissions. Our main idea is to recognise, for example, that the code in Fig. 1 has very similar structure to the expected solution. Given a way to calculate the similarity as a percentage, we can assign a corresponding program score. Of course, practical details are more complex.

The contributions of this paper to the domain of automated assessment are listed below. It is noted that items 3–4 are relevant to a wider audience as graph similarity measures are useful in a number of distinct domains. The specific contributions are as follows:

1. We present a novel approach to automated marking that has wider application than black-box testing, but still measures task specific objectives.[1]
2. Since we use graph similarity in comparing programs, it is significant that none of the existing graph similarity heuristics produces usable scores in our domain. In Section 3.2 we explain why that is the case and in the first part of Section 4 we identify specific criteria that a suitable similarity heuristic should satisfy.
3. We develop a novel measure of graph similarity, *AssignSim*, in Section 4. While this measure satisfies our domain, we also offer a theoretical justification that it is an improvement over prior work in graph similarity (Section 4.1).
4. We show clearly how domain specific details may be incorporated into our graphs similarity measure (Section 5). This is important because graph labels frequently carry critical domain information, but surprisingly has only received limited treatment in the literature. We use labels to carry identifiers, constants, and operators.
5. We show that it is viable to mark free-response student programs by comparing their structure to that of programs with known mark values, such as the submissions made by students of the preceding year of study – or simply a modest subset marked by hand. Our findings indicate a strong correlation with a human marker which is very encouraging (Section 7). We expect to improve these results and some promising ideas are developed in Section 8.

The approach presented is distinct from prior assessment methods that use structural analysis. In addition, although this work does involve relating graphs that describe program semantics, it is also distinct from diagram assessment methods. Section 9 briefly observes relationships with these research efforts.

2. Marking from structure

There is a trade-off made in how tightly student programming assignments are specified. If the specification is very loose, it may be difficult to determine if a student's program satisfies the *spirit* of the assignment requirements. On the other hand, if the specification is very tight, students are left with very little room for creativity and exploration. The case for each has been made in the literature (Beaty, 2001; Harris, Adams, & Harris, 2004; Jackson, 2000).

Unyieldingly tight specifications favour automated marking because the solution space is narrow, and rigid external measures of success may be imposed. One way to allow students more freedom is to widen the solution space, and employ human markers instead.

[1] Unlike, for example, software metrics which are neutral to specific task success criteria.