Automatic Grading of Introductory Programming Labs

JD Kilgallin

University of Akron

jdk72@zips.uakron.edu

Abstract

Programming assignments are widely used as a course component and grading metric in introductory computer science courses. Grading of such assignments is generally done by the course professor or a teaching assistant, and may consist of static analysis of the code, examination of program output, and manual inspection of the source code. This manual examination is known to be a time-consuming process, and to frequently suffer from inconsistent results due to grader errors. In this paper, we examine the development of static analysis tools to perform some grading tasks formerly done by manual inspection, as applied to the Computer Science I course at the University of Akron. Such a tool, even when grading less than half of the point value for an assignment, results in grades 2%-5% lower due to a decreased false negative rate in detecting student errors.

# 1. Introduction

Like many other colleges and universities, the University of Akron relies on student submission of working programs as graded assignments in the introductory computer science courses. In the Computer Science I course, in particular, there are both weekly programming labs and a few longer-term, larger coding projects, all in the C++ programming language. Together, these account for 1/3 of the course grade, as established by the course rubric. The grading of these projects has historically been performed by the teaching assistants, who perform manual inspection of the code along with examination of the program output for a set of 1-5 test input cases. This examination is both tedious and error-prone, as found by other universities. For these reasons, we have developed and implemented an automated grading script in Python, which performs static analysis on the code, program output, and version control history in order to evaluate many components of the grading rubric. We evaluate the autograder’s performance against manual grading performed in the Fall 2013 010 sections and find that it catches many more errors than manual grading, with very few points erroneously deducted. In the rest of this paper, we describe the previous approach, without static analysis, in sections 2-4, then the autograder in sections 5 and 6, and finally the results from applying the program for support in actually grading the labs in the final sections.

# 2. Background

Many universities offer undergraduate and graduate degrees in computer science. Introductory courses are increasingly found even at the high school level. One of the first learning objectives in many such programs is proficiency in a programming language, together with an understanding of the accompanying logic, data structure construction and use, and algorithm development. At the University of Akron, this is achieved through a two-course sequence in C++. This first consists of 3460:209 Computer Science I, which introduces basic syntax, variable use, conditional branching, looping, function definitions and use, arrays, and pointers. The second course, 3460:210 Computer Science II, focuses on classes and object-oriented programming, along with advanced language features such as exceptions. Both courses contain both a lecture component and a lab component involving in-class programming. Like many introductory courses, student evaluation is performed through a combination of exams, written homeworks, in-class labs, and longer, larger, take-home programming assignments. The grading of these assignments is a complicated process requiring three course staff members – a professor, a lab instructor, and another teaching assistant – as well as a sizable infrastructure of software components, including an online Course Management System (CMS), an SVN repository, and a set of scripts and test input data for running the submitted labs. Many of the points allocated on the lab grading rubrics, however, are in areas that cannot immediately be assessed simply by running the program. For example, students are required to commit to the repository at least once for each part and are required to include their name and email. Such items are tedious to check and their grading may frequently overlook errors. They are, however, easily checked by the use of an additional script, which examines the lab report generated by the existing script for content in the program output, source code, and version control history.

# 3. Programming Assignments

The programs collected from students for this course fall into one of two categories. First, there was a series of 13 lab assignments conducted weekly. Many students attended a scheduled lab section in order to complete the labs, while others performed the labs at home or on their own time. All students submitted their code through an SVN repository, from which the laboratory instructor retrieved the student code, generated a report including the results of compiling and running the program, and evaluated the results in order to assign a score between 0 and 20. The second category of program is a set of 3 longer-term course projects, for which students were given several weeks. These were also submitted via SVN and graded by another teaching assistant. The programming assignments vary significantly between sections and by semester, and are therefore not very amenable to automatic grading through static analysis. The labs, however, are well-developed and relatively stable between semesters, with the same lab conducted in each section in any given semester. The number of labs collected, then, made the grading process into a prime candidate for automation.

# 4. Manual Grading Process

In previous and current iterations of the Computer Science I course at the University of Akron, grading of lab assignments is performed by one teaching assistant, who also serves as the lab instructor. Grading of programming projects is performed by another teaching assistant who serves as an aide to the professor during regular lectures. As described in section 3, grading of these programming projects is not amenable to automation. In both cases, though, and in the classic process, the teaching assistant retrieves the student code from the course SVN repository, runs it on a set of test cases through a script included in the repository infrastructure, and examines the output, source code, and version history in order to assign a grade based on a rubric with many components. In the case of the lab, there are usually 5-12 points in categories present across all labs. These include adequate commenting, commit frequency and commit messages, inclusion of the student’s name in a required header comment, and proper code style including indentation and line length limiting. The remainder of the points are dedicated to the crux of the lab – the proper use of loops, or functions, or arrays, or pointers; whatever comprised the lecture material of the week. It is this group of common rubric categories that proves particularly tedious, and it is easy to miss the fact that a line is too long, for example, when concentrating on the more important parts of the program. When this process is complete, the lab report is posted to the repository in both text and pdf format, and grades and comments are posted to Springboard, the CMS employed in classes at the University of Akron. Grading of a lab generally requires between 2 and 5 hours of work with approximately 30 submissions and 8 blank submissions per lab. The number of perfect scores when grading manually varies between a single student and nearly half the class, depending on the difficulty of the lab.

# 5. Static Analysis as a Grading Solution

In order to reduce the time consumption and tendency for false negatives, we examine use of a static analysis tool. Static analysis refers to the examination of one program’s source code or compiled executable, and through routine examination, properties of the program under examination are inferred without executing the program. Such analysis is extremely useful in both software testing and software security. In the former case, static analysis can be built into a test suite that runs on either a fixed schedule, e.g. nightly, or that runs on each change to the code. In the latter case, such tools may be used when a program is of unknown origin and effect, or is believed to be malicious. Such examination can range from simply scanning and printing the ASCII strings contained in the program data, to displaying a call graph, or searching for the program and presenting its characteristics based on a database of known programs. As applies to grading the common rubric components of the Computer Science I labs, a static analysis tool would need to be able to confirm the presence or absence of certain strings in the code, output, or version history, to count the occurrences of certain strings, and to detect if the student did not submit a solution or if it did not compile. Additionally, such a program should be able to allow small variations to many of these strings, such as differences in punctuation in user prompts presented when the lab is run. It is not feasible, in principle, to automatically grade all components of these labs, in part due to undecidability results and in part simply due to the number of acceptable variations and the necessary complexity of a program capable of correctly categorizing all of them. Rubric items not amenable to static analysis include correctness of loops involving one or more variable, logical conditions inside an if statement, or functional equivalence. For such components as can be statically analyzed, however, techniques for doing so have previously been employed at many universities, including a framework developed at Carnegie Mellon University in part by the author. A program with this functionality was therefore written to facilitate grading for Computer Science I, and the rest of this paper describes this program and the results from its use.

# 6. Implementation

The autograder developed for Computer Science I consists of a Python script which relies on the generated lab reports. The code for this program is attached as Appendix A. By examining a text file roster already present in the course repository infrastructure, the script loads each student’s lab report, parses it into three sections containing program output, source code, and version control history, respectively. First, the source control history is examined to determine whether the student even attempted the lab. If no commits by the student are present, a score of 0 is assigned and the program moves on to the next student. If the student has performed some work, further static analysis is performed for a variety of categories. The number of student commits are counted and compared to the number of parts of the program; if the student has committed fewer than this number then their commit history is considered incomplete and a point is deducted. Given that the commit history is generally worth 2 points out of 20 for each assignment, a further point is deducted if only one commit is made, for a score of 0 in this category. Similarly, the number of comments in the source code is evaluated and compared to the number of comments in the starter code. If the number has increased, then the student is known to have written a comment; otherwise a point is deducted for failing to comment the code. Only a single comment is required as the programs are largely very simple and few comments are needed to understand the bulk of the code. Furthermore, in evaluation of this method, the vast majority of students who had written at least one comment, had also written many other comments to thoroughly document their code. The exceptions to this were primarily students who had commented out a section of code, but not written any documenting comments. These students were not penalized for failing to document. The script also checks the required header comment for the mandatory inclusion of the student’s name and email address. The script was initially tested on labs 1-4 for the Fall 2013 section 010 of Computer Science I, which had already been graded at the time of development. After a few refinements and improvements, such as the ability to grade all students with a single command, and the design of a separate configuration file defining distinct constants for each lab, such as the number of commits and comments required. With proper configuration data for labs 5-9, then, this script was used in order to detect student errors. While this proved extremely useful, an additional component was then added to evaluate the text of the program output, source code, and commit messages for the required strings. This component utilized Levenshtein automata in order to perform fuzzy string matching and accept small variations from the required text.

# 7. Results

In the initial evaluation of the first implementation, which checked only a line length of 80 characters, and presence of the student’s name and email, as well as a hard-coded commit count and comment count. This script was tested on a random selection of 30 labs of approximately 130 submissions of the first 4 labs over 40 students. The results were then compared to the manual grading of these rubric elements, which had already been performed and submitted as the student’s official grade for the labs. The results revealed 13 errors worth 1 point apiece, for an average score 0.4 points lower out of 20, or 2% lower. Lab 5 was partially graded, so this version of the script was used to sweep for errors on this lab also. After refining and expanding the program capabilities to include per-lab configuration data and the ability to run all students’ labs, it was straightforward to apply the autograder to the remaining labs. For labs 6-9, which used this version of the script, Table 1 lists the average number of errors per lab detected in each category, based on a random sample of 10 submissions from each lab.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Lab | Lack of name/email | Lines over 80 chars | Too few comments | Too few commits |
| 6 | 0.3 | 0.1 | 0.2 | 0.2 |
| 7 | 0.1 | 0 | 0.6 | 0.1 |
| 8 | 0.2 | 0 | 0.3 | 0.2 |
| 9 | 0.1 | 0.4 | 0.8 | 0.2 |

Table 1: Average errors automatically detected per lab for labs 6-9

# 8. Conclusion

Based on the above results, automatic grading of Computer Science I labs can accommodate grading of all of the requirements common to every lab. Furthermore, it can do so quickly and reliably. These components range from 25% to 60% of the lab’s points, which indicates that it can perform a significant amount of the work previously done by the teaching assistants. Additional work can likely raise both the rubric coverage and the accuracy and interface of the script. Adaptations are most likely straightforward in order to accommodate other courses, such as Computer Science II. However, even as it stands, the script appears to have merit as a tool to assist teaching assistants with Computer Science I grading.

# 9. Future Work

The bulk of additional work to be done on the program is to add features to grade additional rubric components. Due to the simplicity of the programs submitted, checking program aspects such as correctness of loop termination conditions or basic algorithms may be feasible. A potential approach is to use machine learning techniques on existing submissions along with their known grades by component, in order to derive a rule for categorizing new submissions as correct or incorrect in each component. Some components may have simpler approaches that have simply not yet been implemented, such as confirmation of program indentation. Other changes may increase the program’s accuracy; for example, if a student comments out a section of code, this should not count as a descriptive comment and the script should thus check if a comment contains code or not. Finally, the program can be made more flexible by porting additional components to a configuration file. A Domain-Specific Language is likely appropriate for specifying a program rubric and grading criteria. This will allow the program to be adapted to changing lab requirements by semester, or to additional courses, and would thus further increase the value of the program.

# 10. References