Challenges Automating the Grading of Duke's Introductory Programming Assignments

Jack Zhang

Advisor: Robert C. Duvall

Thursday, April 18, 2013

Abstract

This project addresses several issues regarding grading assignments for Duke's primary introductory programming course: timeliness of evaluation, relevance of feedback, and quality and helpfulness of comments. The results of two surveys, one of comparable courses at other universities and one of students taking the course, were analyzed to offer insight on how assignment feedback can be improved. A review of automated grading systems was conducted to determine their suitability for the types of assignments given in the course. Then a framework was created that automates several common grading tasks, generating feedback for the student that can be supplemented by additional grader's comments. It is believed that this framework can be used to improve the usefulness of feedback given to students in all introductory programming courses.

1. Introduction

1.1 Context

Computer science is quickly overtaking other disciplines as a desirable and marketable major at many elite institutions [1]. The enrollment in Duke's primary introductory programming course, CompSci 201, has more than doubled in the last three years to almost 200 students (Appendix A). Such large class sizes increase the demand on professors and undergraduate teaching assistants (UTAs) to provide timely and relevant feedback to students.

Such feedback is crucial to learning to program because a student may find himself lost if he is uncertain about his grasp of the current materials. This is especially frustrating when he is constructing an understanding of computer science from the ground up, which could lead him to feel the discipline is harder than it actually is [2]. Ideally, feedback should be given immediately or within a week of the assignment's submission. This ensures that this assignment is still on the student's mind and that the feedback can be used productively on the current assignment..

Automation helps address these problems in a number of ways: providing some form of feedback immediately upon submission of an assignment; cutting down on the amount of time UTAs and instructors have to spend organizing student submissions; and scaling to accommodate the growing number of students enrolling in computer science courses. It may also help improve the type of feedback given by allowing UTAs to better utilize their time to generate specific and detailed comments about more efficient algorithms, implementation issues, or problem solving strategies.

1.2 Motivations and Experience

As a student who began learning computer science in college, I valued the feedback that I received on all my Compsci 201 assignments. Feedback helped me feel connected to the course outside of class and affirmed that I understood the current topics being taught in class. Feedback from project assignments was the only source to gauge my progress in class, however I noticed that I received feedback much faster for my essays in my writing class than for my assignments in Compsci 201. This was problematic because many of the Compsci 201 programming projects built upon concepts learned previously. Given the nature of the assignments in Compsci 201 and

the importance of feedback, I felt motivated to find a way to improve assignment feedback at Duke.

My experiences as an undergraduate teaching assistant has also shown me why it is difficult to generate feedback in a timely manner and why quality feedback is hard to maintain. The UTAs' responsibilities consisted of grading student submissions for assignments and midterms, assisting students during UTA office hours, and answering questions on the course's online bulletin board. Because UTAs are students as well, many have demanding schedules that often conflict with the responsibilities of being UTAs. As a busy student, I often found myself having to delay my evaluations because I could not set aside the time needed to correct the student submissions I was assigned.

2. Related Work

2.1 Studies and Publications

Senior lecturers who assessed the efficacy of automated feedback with students found that it has been more useful than traditional feedback in the learning process. They defined effective feedback as that which "indicates to learners where they have done well, where their misunderstandings are, and what follow-up work might be required. Such remarks should also be returned as quickly as possible if students are to take heed" [3]. Tutors provided automated feedback through Microsoft Outlook using drop-down menus, which could then be organized by the program. Students were divided into two groups those receiving electronic feedback and those receiving professor evaluated feedback. After students were allowed to compare their feedback with the feedback of others, the electronic feedback scored 0.9 units greater than traditional feedback on a Likert scale [4]. Tutors also noted that electronic feedback allowed for

a greater consistency compared to hand corrected version evaluation. Additionally, students who received electronic feedback did not consider the information less valuable than professor evaluated feedback; automation provided the added benefit of getting feedback back to students sooner while preserving the quality of feedback.

Automation to any degree has been shown to benefit both evaluators and students. A Microsoft Excel and Word marking assistant titled Electronic Feedback was found to have enabled many academics to "return more feedback, of higher quality, and in a shorter space of time" [5]. It was able to generate and email feedback reports after tutors had entered the appropriate data. The reports include standard comments selected from "a bank of statements that are anticipated to be required regularly during marking". Furthermore, the program was able to accentuate aspects of an assessment that caused more difficulty. The program was noted to have the potential to "inform future teaching, learning, and assessment strategies". The publication noted that "students attach great value to high quality feedback, considering it to be of greater importance than clear explanations and the stimulation of interest within the classroom" [6]. Additionally, Electronic Feedback noted that timeliness is indeed an important factor to quality feedback; "the educational benefit of even the most well crafted feedback is lost if it is returned too late" [7].

2.2 Existing Automation Systems

Types of Automation

Two main types of automation are currently used to evaluate code. JUnit or input output testing is used to test whether a specific method is correctly implemented; the method is given a variety of inputs and expected a specific outputs back [8]. Runtime testing is a type of testing

where the time of a method from start to finish is recorded. The runtime of different inputs are then compared to verify that a method exhibits expected behavior such as O(n) run time.

Web-CAT

Web-CAT is an automated system developed by Virginia Tech. It is “an advanced automated grading system that can grade students on how well they test their own code; Web-CAT is highly customizable and extensible, and supports virtually any model of program grading, assessment, and feedback generation” [9]. A few of Web-CAT’s unique features are that it is hosted online and thus relatively portable, it allows students to test unfinished code, assesses how students test their own code, and can support instructor-provided plug-ins for assessing student work. Web-CAT encourages test-driven development where "For each small feature or addition to be made, the programmer first writes a corresponding set of unit tests that express the expectations and intent for the feature's behavior, and then writes new code. The tests are used to check that the code meets the expressed expectations." Web-CAT is ideal when testing a relatively simple coding assignment where correctness can be evaluated by inputs and outputs, and details such as style can be measured by whether a Javadoc was included for every method.

Curator

Like Web-CAT, Curator is a web application for managing the submission of student assignment [10]. Curator follows from a line of automated grading systems dating back over 30 years, with the current version having been in use since 1997 [11]. The primary use for Curator has been to collect and archive student submissions as well as automatically scoring submitted assignments. Submissions require user validation and students are notified by e-mail when an

assignment has been processed by Curator. When Curator is configured to automatically grade assignments, email notifications include testing and scoring data.

Praktomat

Praktomat is "an automated system for managing the submission, test, and mutual reviewing of student's programs"; it is online platform that was built to enhance the quality of student code. Praktomat was developed in Python at Uni Passau in Germany, however the project has been discontinued. Praktomat streamlines program management by online submission, uses automated testing to assess program functionality, and provides voluntary mutual reviewing to improve readability and maintainability. The creators of Praktomat believe that reviewing other's code and having one's own code examined is helpful in producing and learning code [12].

The developers of Praktomat believed that there was inherent value in students viewing code from the perspective of the reviewer and gaining insight into the difficulty of understanding code from the perspective of a viewer. Students were allowed to resubmit their code any number of times before the deadline and incorporate insights from peer reviewing the code of other students and the feedback received from peer reviews. To counter the potential for plagiarism, students received personalized assignments and were only allowed to review another student's code after having submitted [13]. Other constructive measures were also implemented to ensure plagiarism did not occur. Assignments were evaluated based on a public and private test suite. The private test suite was only available to professors and evaluation also included manual examination by professors.

Praktomat garnered marked success upon implementation. 70.5% of students wrote at least one review. On average, a student submitted 3.27 reviews during the duration of the course. When students were asked whether Praktomat improved the quality of their code, 57.7% confirmed the automatic testing was helpful and over 60% confirmed the efficacy of the peer review process [13]. On average students who did not write a review at all had a lower average grade than that of students who wrote at least one review. It was found that receiving reviews improved readability even more than writing reviews.

ASSYST

ASSYST is the shortened name for assessment system, a software automation tool for grading student programs. ASSYST uses metrics for five areas of assessment: correctness, efficiency, style, complexity, and test data adequacy [14]. ASSYST provides a "graphical interface that can be used to direct all aspects of the grading process".

In each area the grader can specify four points that define the way scores are given (pictured right). A score lying between B and C received full marks but a score lying between A and B or C and D received partial credit while any score below A or above D will received no credit [14]. ASSYST used a few notable methods in assessing certain criteria.

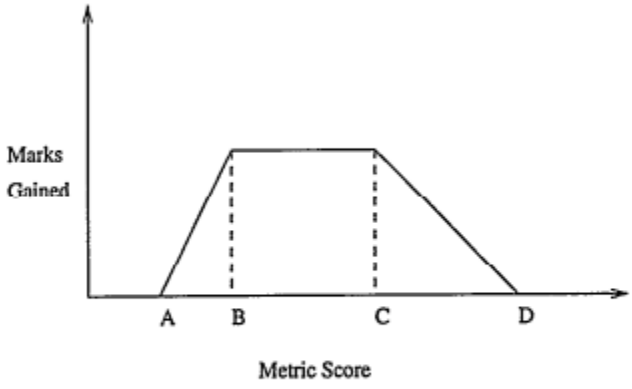


Figure 4: Marking graph

In assessing efficiency, ASSYST was able to count the number of statements executed in the student's code and compare this to the tutor's model solution. The student code would then be awarded points in efficiency based on boundaries set by the tutor. Programming style is assessed

via different characteristics such as module length, number of comment lines, and use of indentation, which can be adjusted by the tutor.

ASSYST demonstrates the "housekeeping" potential of an automated system to comprehensively organize code and apply a specific standard to every student submission. ASSYST's implementation offers some interesting insights into what information is available to the instructor when using an automated system and how such information can be effectively utilized to glean a more insightful picture of how a student's program is working compared to the model solution. ASSYST easily kept track of many useful metrics that a human grader cannot.

3. Independent Research

3.1 Automating Duke's Assignments

One of the challenges of automating Compsci 201 assignments is the "open-endedness" that characterizes many of these assignments. Such assignments often begin with a brief description of an interesting concept and are subsequently proposed as a computer science problem where the student is asked to implement specific functionality to complete the assignment. Students have the freedom to start on different portions of the code and are allowed to implement helper methods to assist in completing a specific method. Some assignments also provide bonus credit to students who implement extra functionality to the current assignment.

One example is DNA[15]. Students are given sample code that simulates a DNA strand and asked to create a new DNA strand object that uses a linked list (LinkStrand). Linkstrand must support the ability to append new DNA strands, reverse the DNA strand, and return the size of the current DNA strand. In order to provide consistent automation, limitations and constraints must be placed on naming conventions and the location of code. Thus, automating open-ended

assignments in the Duke computer science curriculum present problematic tradeoffs between fostering "open-endedness" and exploration in assignments and providing assignments that are very structured.

The current automation systems would be difficult to adopt for Duke's current assignments given nature. For example, Web-CAT is limited in the format of what can be tested and evaluated and the feedback that can be given from that evaluation. Much of the grading rubric for UTAs at Duke consists of verifying that a student understands concepts learned in class and has correctly implemented these methods within the assignment. Such criteria are difficult to verify via JUnit tests or other tests that simply match output. Adopting Curator at Duke suffers from the same type of problems as that of Web-CAT. Some Duke assignments include a JUnit tester class specifically to aid students in testing functionality of their program. Thus current automated systems would only be able to test a portion of student submissions, and some of this testing is already currently available to the students as a resource.

Though there is no automated system for Duke's computer science assignments, Duke does supplement these assignments with APTs (Algorithmic Problem-solving Tests) every week [16]. APTs are problems that require students to focus on one method that can be tested with input output matching; APTs are fully automated and provide immediate feedback to students detailing the correct and incorrect test cases. APTs only display the input, actual output, and the correct output.

3.2 Automation at Other Universities

To better understand existing methods of generating feedback and the prospect of automation, other universities were surveyed in September 2012 regarding their class sizes,

student to staff ratio, and efforts in assignment automation (Appendix B). The table of Stanford's curriculum is listed below:

A Table of Stanford's computer science Curriculum	
Average class size	1000 to 1500
Teaching assistant to student ratio	12 to 1
Typical feedback	15 minute "interactive grading" session
Average time to general feedback	30 minutes
Feedback type	Manually done by hand

Stanford has over a hundred undergraduate teaching assistants per year, and Stanford's model for generating feedback is similar to that of Duke's and requires many teaching assistants to succeed. Automation can be incorporated into the current evaluation process and feedback can be generated much more quickly via an automated system while preserving the ability to have interactive grading sessions. This is a valuable resource that Stanford provides and such a resource would be very valuable to Duke students that simply want to talk privately with a teaching assistant about topics that they do not feel comfortable mentioning in class.

In contrast to Stanford's feedback system which is largely generated by teaching assistants, Virginia Tech's curriculum is much more automated as seen in the table below:

A Table of Virginia Tech's computer science Curriculum	
Average class size	240 to 300
Teaching assistant to student ratio	2 to 60 (1 UTA and 1 Graduate TA)
Typical feedback	Web-CAT output
Average time to general feedback	30 seconds to 1 minute
Feedback type	Electronically online

Virginia Tech relies on Web-CAT to evaluate students and requires students to submit their own tests to gauge how well they understand the assignment. After the deadline has passed, TAs will manually grade submissions for design and other criteria. Virginia Tech's model has worked well for their large class sizes and the organization and instant feedback generated by Web-CAT has been shown to help to improve student code. Georgia Institute of Technology and Texas Tech University were also contacted, but no official response was received.

3.3 Survey Findings

On April 3rd, 2013, a voluntary survey was conducted of a Compsci 201 course at Duke with roughly 200 students, and 38 students responded. The following analysis will simply be to offer insight into how students currently feel towards the curriculum at Duke. The survey found that nearly 63% of students checked for new grades online at least once a week. In regards to the quality of feedback, students were allowed to select all areas that they felt feedback could improve. The survey found that 55% of students checked that feedback could improve; 23% selected that timeliness of feedback could improve, 17% stated quality and helpfulness of feedback could improve, and 15% marked that amount of feedback could improve. Of the 38

students who answered the survey, only 5% believe that feedback had a substantial influence on their code while 38% felt that feedback had no influence on their code whatsoever.

The survey found that 76.3% of students surveyed spend more than 5 hours on assignments. Though this is not unreasonable, some students have expressed there is a lack of clarity of the actual tasks that need to be completed for each assignment. For the survey "What resource would you like to have at your disposal as a computer science student that is not currently available?", multiple students affirmed that assignment descriptions could be clarified more. As class size increases, assignment specifications must be made more clear in order to ensure that students are not confused [17]. Furthermore, multiple students requested the ability to ask UTAs and instructors questions in a more personal setting. Though these concerns are not directly related to automation, it does indicate a need to strategically change the current Duke computer science curriculum to better aid students.

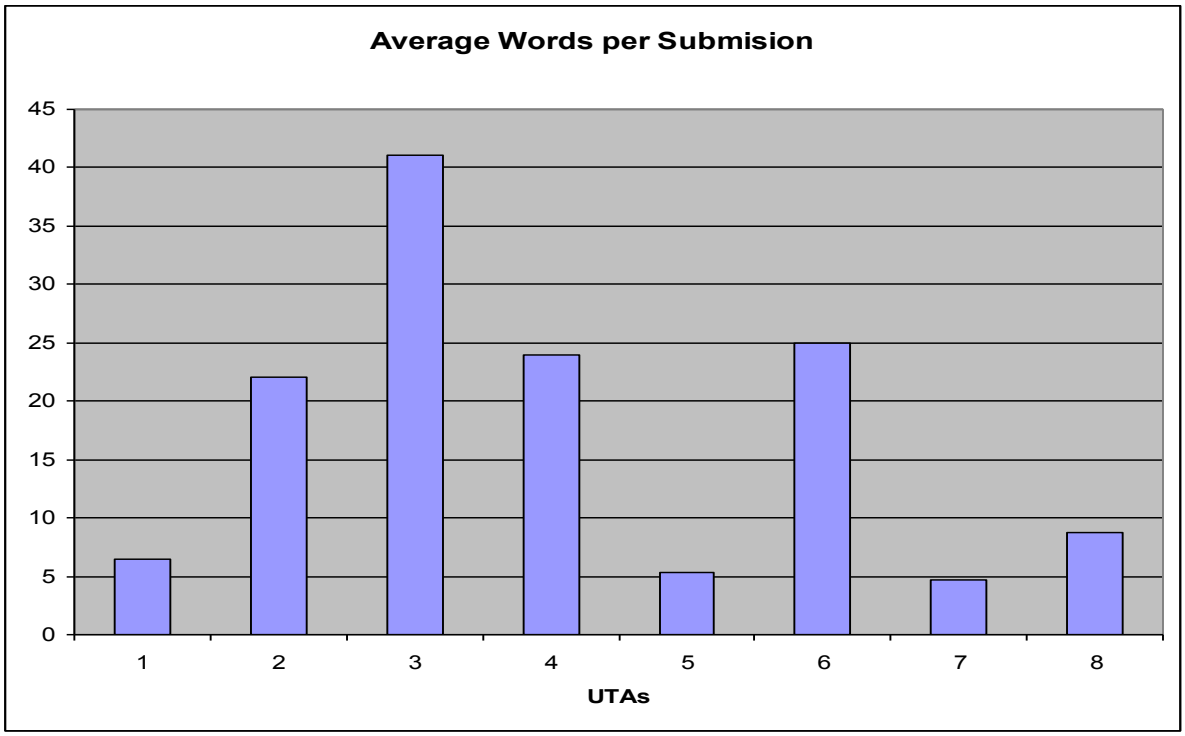
Students believed that more specific feedback would be helpful. Multiple students requested feedback regarding the efficiency of their code. Some students inquired about different approaches to the same problem. These responses indicate that students can sometimes grasp the problem and think of a solution, but feel that a more elegant solution is possible if they had a more thorough understanding. Another common request was feedback that was explained more thoroughly. Though this does not directly demonstrate the need for an automated system, automation can help UTAs save time when explaining why deductions were made so that they have more opportunities to be specific as to where the deduction occurred and what the proper solution would be.

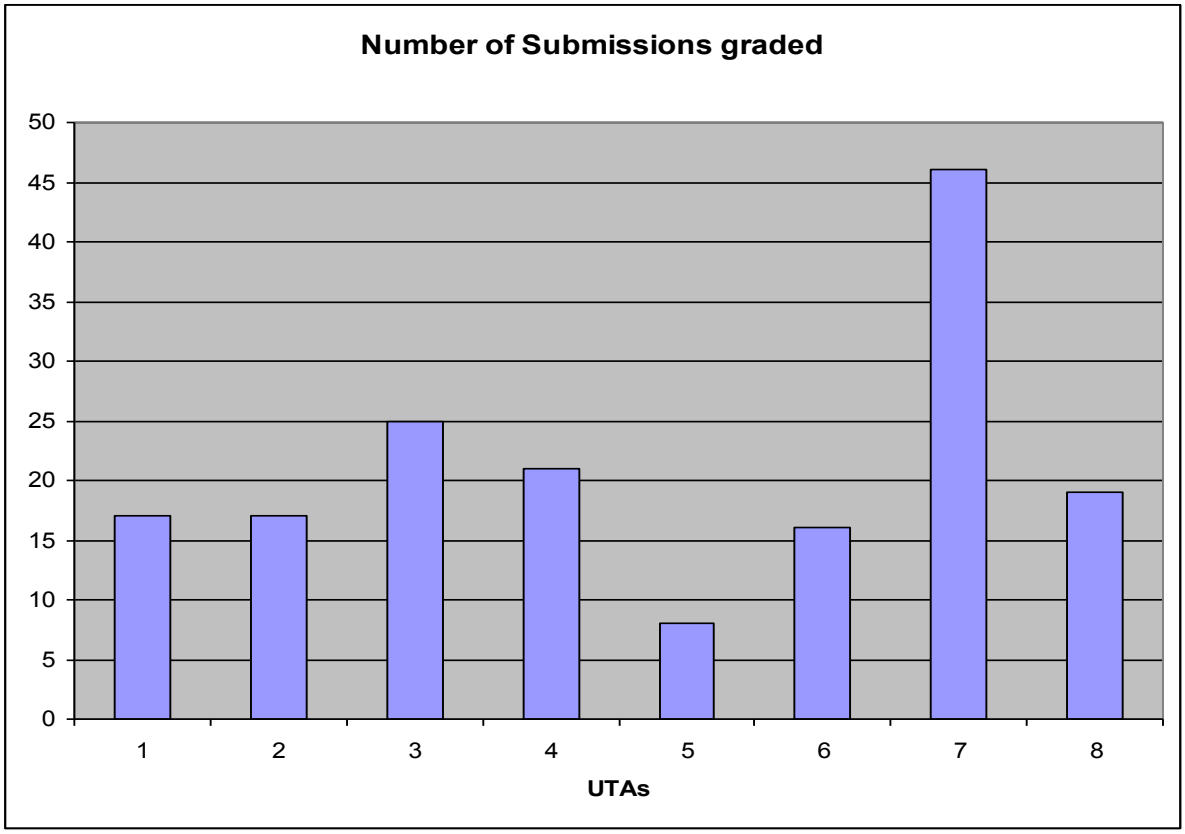
3.4 Feedback analysis

The feedback of 8 UTAs over 169 student submissions was analyzed to determine the quality of feedback for students. The assignment being graded was the DNA assignment mentioned in section 3.1 for the fall semester of 2011. Comments were split by the space character as a delimiter and the number of strings counted. In total, 2,837 words were used which suggests that on average each student submission received approximately 17 words as feedback. A breakdown of the number of words per submission is show below

	10 words or less	More than 10 words	More than 20 words	More than 30 words
Number of comments	98	69	49	20

Breaking the feedback down by evaluator, the average number of words per student submission ranged from a minimum of 4.74 words per submission to a maximum of 41.12 words per submission. The results can be seen in the Tables on the next page:


Average Words per Submision
45
40
35
30
25
20
15
10
5
0
1
2
3
4
5
6
7
8
UTAs


Number of Submissions graded
50
45
40
35
30
25
20
15
10
5
0
1
2
3
4
5
6
7
8
UTAs

These results reveal that, depending on the UTA each student is assigned, feedback can range from very short to fairly extensive. Interestingly, the UTA that had the highest number of words of feedback per submission also graded the second highest number of submissions. However, the UTA with the lowest average words per submission also graded the most submissions. From the two graphs, it is also clear that a significant portion of the grading averaged less than 10 words per submission. UTAs 1, 5, 7, and 8 graded more than half of all the submissions, yet all 4 averaged less than 10 words per submissions. Ideally with an automation system in place, UTAs with lower average words per submission can supplement their feedback via output from automation. Furthermore, if most of this feedback has already been generated, it would be possible to ask UTAs to spend more time on the analysis portion of the feedback and address areas where students performed poorly.

4. Automation Framework

4.1 Code Outline & Summary

A framework was developed to explore the logistics of developing an automated system that preserves the spirit of Duke's assignments. The framework utilizes different `Tester` classes to evaluate specific portions of a rubric; each `Tester` subclass use entirely different code for evaluation.

For example, two `Tester` subclasses are included in the generic framework that will allow instructors and teaching assistants to assess the existence of methods that students were instructed to implement and use and perform JUnit input output testing. Each specific section of the assignment rubric warrants a new `Tester` class.

AssignmentAuto is the main testing object that encapsulates all Tester classes while also allowing for easy iteration of assignment testing across a large directory of student submissions. Ideally, AssignmentAuto should initialize a list of all the desired Testers needed to properly assess a student and then iterate through these testers for every student submission available.

All tester classes contain a generateReport method that is passed to the Student class. The generateReport method returns a string of the feedback output for the Tester. This method is used to create an aggregate output file that can be given to students as feedback. The Student class is an object included to encapsulate useful information relevant to Duke students and generating the final feedback output file e.g. Net ID, first name, last name, email, etc.

To deploy this framework on an existing Compsci assignment, an instructor would create Tester classes that award points for specific objectives students were expected to implement in the assignment. Then, an AssignmentAuto class is created and passed an array of the Tester classes. Lastly, the processRoster method is called to scan in a file of student NetIDs and call the array of Tester classes on the student's submission.

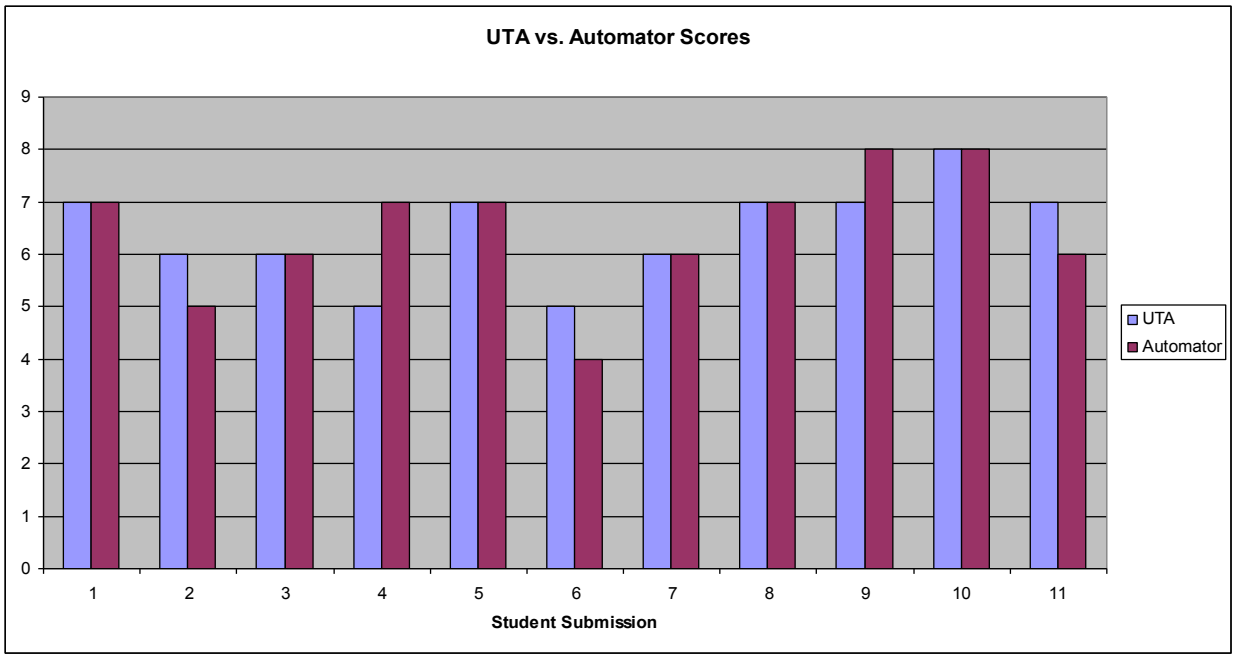
4.2 Automation Performance

The automated framework was used to evaluate the functionality portion of the DNA assignment splicing for student submissions in the fall semester of 2011. UTAs were asked to evaluate the LinkStrand Java class students created based on the following criteria (Appendix C):

1. If the LinkStrand passes all JUnit Tests from TestStrand
2. If it contains code that appends for the general IDnaStrand object as well as LinkStrand specifically
3. If it calls the size method with O(1) time
4. If it correctly reverses the LinkStrand

Students were given extra credit if they implemented the reverse method with a map that reused identical strings. The automator was tested on 11 student submissions and the resulting score compared to the actual UTA score.

TestStrand was modified to evaluate criteria 1 while DNA_Code_Tester was created to evaluate criteria 2, 3, and 4. The second criteria was evaluated using a modified version of the Code Tester class, which specifically looked within the append method by scanning the file line by line. The automator specifically looked for code that indicated that the student accounted for an instance of LinkStrand or an instance of SimpleStrand and an else statement for LinkStrand. It also verified that the student accounted for a general case where IDnaStrand was neither a SimpleStrand nor a LinkStrand, by scanning the code line by line specifically looking for the size method, accounting for variations in indentation, spacing, and comments and verified that the method only contained a return statement and no for loops. It scanned the method to ensure that it made use of nodes and also noted if the student made use of a map object within the method. The reverse method passed all of the JUnit tests within the TestStrand of criteria 1. The table below compares the UTA scores with the scores from the automator:



Of the 11 scores that were compared, 6 student submissions received the same score from the automator. In case 11, after looking at the student submission more closely, it was discovered that the append method required deductions that were overlooked by the UTA. The automator was unable to provide insight into whether a student specifically duplicated strings in their append method as well as implementation issues with the student's reverse method.

Though the automator for DNA evaluation was tailored to the evaluation rubric; discrepancies arose when UTAs scrutinized code closely and found specific implementation issues that warranted deductions. Several students named their LinkStrand file, "LinkedStrand.java", which prevented the automator from properly detecting whether the student accounted for specific instances of IDnaStrand in their append method. There was only one instance where the automator differed from the UTA score by more than 1. Much of the implementation correctness had to be verified through conditional statements that checked whether specific fragments of code were specifically utilized in a method. Even with a very

tailored approach to the assignment evaluation, UTAs did offer more insight when students did not implement a method correctly. Specific aspects of implementation were difficult to verify, such as whether in the reverse method students used toString to reverse the entire string instead of building the Linked List as asked. The output for the DNA automator was able to break down all deductions made and explain why deductions were made specifically (Appendix D).

4.3 Runtime Analysis

The current framework also has the capability of performing runtime analysis given an adequate number of iterations and increasing input. In the DNA assignment, students are asked to implement a LinkStrand that splices a string into an existing text file and to show that the SimpleStrand class splices DNA strands in O(n) time. The framework utilized a Java library to calculate linear regressions confirming that SimpleStrand ran in O(n) time. Below is a table of the R-Squared values, which measure whether run time was linear, for the linear regression of SimpleStrand's O(n) method:

	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5
R-Squared	0.969	0.963	0.969	0.9625	0.9678

The framework was also able to provide empirical evidence that the newly implemented LinkStrand class was not O(n) time. Below is a table of the R-Squared values for the linear regression LinkStrand's method, which was not O(n):

	Trial 1	Trial 2	Trial	Trial 4	Trial 5
R-Squared	0.0345	0.0038	0.0063	0.0194	0.02067

The R-Squared's of the LinkStrand output would lead us to believe that SimpleStrand's cutAndSplice method is O(n) while LinkStrand's is not. Runtime analysis testing would allow UTAs to quickly verify that a student's implementation is correct by looking at the linear fit. If a specific method should be completed in constant time, then the R-Squared values would be close to 0 while a method that should be O(n) time should have R-Squared values close to 1.

4.4 Guidelines for Automation of Future Assignments

Ideally, in order for an assignment to be easily automated, a few specific attributes must be present. First, the assignment must have specified sections where code must be evaluated. Both manual evaluation and automation become increasingly difficult without any information on where the evaluated code should be. The second trait that allows assignments to be easily automated is clear and concrete details that can be verified and tested. For example, a grading standard that specifies “Look at the code in this Java class and make sure that it looks correct” cannot be implemented in terms of automation. The assignment must be verifiable and testable in this regard. Lastly, the assignment must be conceptually organized in such a way that there are ordinal steps that can be followed and checked. This last quality is to ensure the coherence of the feedback generated and that the feedback is organized in a way that would be helpful for students. With these qualities, any assignment or project can be incorporated into the automation framework and meaningful feedback can be generated.

5. Findings and Challenges

5.1 File Management

One difficult problem encountered was the issue of testing all student submissions at the same time. The initial problem with so many student directories, all with different solutions to the same methods, was that evaluating each set of student files individually would require very meticulous organization. Furthermore, the program must constantly switch the tested classes with new ones, which would be impossible at runtime. To remedy this error, a shell script could be written that replaced the files being tested before the testers could be run. This meant an extra step in the process was required before the automation could smoothly iterate through all student submissions.

5.2 Naming Conventions

Uniformity of naming conventions and missing classes also proved to be a challenge. In order for the automation to run smoothly, strict naming conventions for both class names and method names had to be adhered to. If a student were to accidentally name LinkStrand to LinkedStrand, automation would assume the file does not exist, which would cause the output to be inaccurate or completely devoid of meaningful feedback. Unfortunately, this can only be addressed by stressing that all files be appropriately named and investigating all cases of output where an error arose. This solution could potentially compromise the open-endedness of many Duke computer science assignments because strict naming conventions would force students to follow a stringent guideline in terms of development.

6. Conclusions

As was seen, automation has the potential to remedy many of the problems facing computer science instructors at Duke today. An automated system tailored to Duke's curriculum can preserve the "open-endedness" of Duke assignments while still providing immediate feedback to students. Stanford and Virginia's Tech's feedback systems have demonstrated that, though feedback systems from both ends of the spectrum can work, it would be helpful to employ more automation so that the feedback system is scalable to larger class sizes. An automated system can cut down on the amount of time UTAs spend setting up assignment projects in Eclipse and other automatable tasks such as searching a method for specific pieces of code.

The current existing automated systems that make use of input output testing are incompatible with the Duke curriculum. Nevertheless the existing automation systems do shed light on the extent of automation and some of the interesting metrics that an automated system is capable of using. Duke's curriculum can benefit from being more test development driven; this has been shown to produce better quality code and encourages programming that is backed up by a true understanding of the code. As seen in ASSYST, automated systems have the potential to aggregate and organize many useful metrics that would otherwise be lost from manual grading. Furthermore any automation system allows for some degree of uniformity in terms of grading and imposes organization to all student submissions.

Just as Web-CAT and ASSYST have shown the need for an automation system, Praktomat has also shown that curriculum changes can be made to enhance the way students learn and apply code. Although having personalized assignments for an entire class is somewhat

impractical, having a few assignments that have a peer review element can be very helpful for students. Students are given an opportunity to view code from the perspective of an evaluator and are given the ability to use other classmates to receive useful evaluations.

The automation framework that was developed has shown that, even with the way Duke computer science assignments are structured, automation can be utilized to provide meaningful feedback. Automation can be used to verify implementation by checking for existing code, however there are specific limitations that must be placed on naming conventions and UTAs are able to offer more insight when errors occur. An automation system that can generate substantive feedback in a timely manner not only benefits the way students are learning concepts within courses, but also has a direct impact on the instructors and UTAs making the feedback. As the number of students enrolling in Duke's computer science courses rises, automation holds the key to generating feedback in a timely manner and improving the quality of feedback students receive.

Works Cited

1. Meyer, Robinson. "Stanford's Top Major Is Now Computer Science." The Atlantic. N.p., 29 June 2012. Retrieved. 01 May 2013. <http://www.theatlantic.com/technology/archive/2012/06/stanfords-top-major-is-now-computer-science/259199/>.
2. Ben-Ari, Mordechai, "Constructivism in computer science Education". SIGCSE '98 Proceedings of the twenty-ninth SIGCSE technical symposium on Computer science education, Pages 257-261, ACM 1998.
3. Denton, P., Madden, J., Roberts, M. and Rowe, P. (2008), Students' response to traditional and computer-assisted formative feedback: A comparative case study. British Journal of Educational Technology, 39: 486–500. doi: 10.1111/j.1467-8535.2007.00745.x 3 Feb. 2013.
4. McLeod, Saul. "Likert Scale." Likert Scale. N.p., 2008. Retrieved. 06 May 2013. <http://www.simplypsychology.org/likert-scale.html>.
5. Denton, Philip. "Generating Coursework Feedback for Large Groups of Students Using MS Excel and MS Word." University of Chemistry Education (2001): 1-8. Royal Society of Chemistry.
6. Ramsden, P. (1993) *Learning to Teach in Higher Education* London: Routledge.
7. Gibbs, G. and Habeshaw, T. (1993) *Preparing to Teach: An Introduction to Effective Teaching in Higher Education* Bristol: Technical and Educational Services Bristol.
8. "Junit: A Programmer-oriented Testing Framework for Java." JUnit. N.p., n.d. Retrieved. 05 May 2013. <http://junit.org/>.
9. "Home | The Web-CAT Community." *The Web-CAT Community*. National Science Foundation, n.d. Retrieved. 14 Apr. 2013.
10. "Curator: An Electronic Submission Management Environment." Curator Project Homepage. Virginia Polytechnic Institute and State University, 15 July 2004. Retrieved 14 Apr. 2013.

11. "Using Automated Grading Systems for Programming Courses - Do It Yourself computer science." DIY computer science. N.p., n.d. Retrieved 14 Apr. 2013.
12. Zeller, Andreas. "Making Students Read and Review Code." ITiCSE '00 Proceedings of the 5th Annual SIGCSE/SIGCUE ITiCSE conference on Innovation and Technology in Computer Science Education 32.3 (2000): 89-92. ACM.
13. Humayun, Amna, Wafa Basit, Ghulam Farrukh, Fakhar Lodhi, and Rabea Aden. "An Empirical Analysis of Team Review Approaches for Teaching Quality Software Development." Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (2010): 567-75, ACM
14. Jackson, David, and Michelle Usher. "Grading Student Programs Using ASSYST." SIGCSE '97 Proceedings of the twenty-eighth symposium on computer science education (1997): 335-39. ACM.
15. "CompSci 201: Data Structures & Algorithms." computer science 201: Data Structures & Algorithms. Duke University, n.d. Retrieved. 18 Apr. 2013. <http://www.cs.duke.edu/courses/compsci201/current/Assignments/DNA/dna.html>
16. "CompSci 201: Data Structures & Algorithms." computer science 201: Data Structures & Algorithms. Duke University, n.d. Retrieved. 29 Apr. 2013. <http://www.cs.duke.edu/courses/compsci201/current/Apts/apts.html>.
17. Kay, David G. "Large Introductory Computer Science Classes: Strategies for Effective Course Management." SIGSCE '98 Proceedings of the Twenty-ninth SIGCSE Technical Symposium on Computer Science Education (1998): 131-34. ACM.

Appendix A: Compsci 201 Class Sizes 2010-2013

CS201 Class Sizes	
2013 Spring Semester	193
2013 Fall Semester	180
2012 Spring Semester	143
2012 Fall Semester	124
2011 Spring Semester	102 (34 and 68)
2011 Fall Semester	165
2010 Spring Semester	79 (55 and 24)
2010 Fall Semester	98 (42 and 56)
2009 Spring Semester	55 (29 and 26)
2009 Fall Semester	74 (41 and 43)
2008 Spring Semester	61 (33 and 28)
2008 Fall Semester	80 (39 and 41)

Appendix B: Survey of Other Universities

Hi, my name is Jack Zhang and I am a senior working with Robert Duvall in the Computer Science Undergraduate Research Fellowship (CSURF) program at Duke University. I am researching how to improve the type and timeliness offeedback given to students on their programming projects in our CS1 and CS2 courses.

To get me started, it would be very helpful if you could answer the few short questions below and then direct me to someone who might be able to answer a few more detailed follow up questions:

- On average, how many students take your CS1 and CS2 (or equivalent) courses per year?
- What is the ratio of students to course staff (graduate or undergraduate TAs)?
- What is the primary form of feedback students receive on their programming assignments, homeworks, and tests?

Appendix C: DNA Grading Rubric

Notes for groups

This a group assignment so some of your assignments will be groups. If you're working with a group, please snarf the submission of every member of the group. If the students have followed the instructions, only 1 submission should have code and a writeup. The other in the team should just have readmes saying "please look at so-and-so's submission".
Some students may have submitted the same stuff with each member...annoying, but OK.
A couple may have submitted different writeups for each group member. If that's the case, just use the best writeup for the grade. Don't drive yourself crazy hunting for the best writeup for each particular question though - select the best writeup overall and use that.
Please enter the grades of each group member individually. But they all get the same grade.

Algorithm Correctness

1. Run all the unit tests in TATestStrand - they should all pass. If they do, give the student 3 points. If there is just 1 failure, give them 2 points. If there is at least one successful test, give them 1 point.
2. Take a look at their code for append(IDnaStrand)/append(String). The code should have a special case for LinkStrand objects and a general case for IDnaStrand objects. The special LinkStrand case should use nodes to avoid duplicating the strings, giving the desired O(B) time. If this all works perfectly, give them 2 points. If there's an attempt but you can see some bugs, give them 1 point. If the code is a mess or we're copying strings, give them 0 points.
3. Take a look at the code for size(). It should just return an instance variable (O(1) time)- the size should be correctly updated in append. If that's correct give them a point.
4. Take a look at the code for reverse. If it uses nodes and correctly reverses the list, give them 1 point. They do not get any credit if it just converts the whole list to a string with toString and reverses that. If it does it and reuses identical strings (this will require a Map that maps strings to reversed strings) also give them 1 point of *extra credit*.

Total possible algorithm points: 7. One point of extra credit is also possible.

Appendix D: DNA Automator Feedback Output

```
DNA Assignment Automated Feedback - John Smith

Functionality:
You passed 6 case(s) of reverse test functionality
You passed 7 case(s) of Splice functionality.
You passed 6 case(s) of Initialize functionality
You passed 6 case(s) of Size functionality
You passed 6 case(s) of toString functionality.
You passed the test case for double append.

Your functionality was implemented correctly. You earned 3 out of 3

Implementation:
You included the general case but forgot to include the special case for the
appending dna strands.

You earned 1 for the implementation of append.
You implemented the size function with constant time return.

You earned 1 for the implementation of size.
You implemented the reverse function correctly and did so with the use of Nodes.

You earned 1 for the correct implementation of reverse.
Your implementation was coded correctly. You earned 3 out of 4
```