


Fall 10-9-2010

From Walls to Steps: Using online automatic homework checking tools to improve learning in introductory programming courses

Dwayne Towell
Abilene Christian University

Brent Reeves
Abilene Christian University

Follow this and additional works at: https://digitalcommons.acu.edu/info_tech_computing

 Part of the [Educational Assessment, Evaluation, and Research Commons](#), [Online and Distance Education Commons](#), and the [Scholarship of Teaching and Learning Commons](#)

Recommended Citation

Towell, D. & Reeves, B. (2010) From Walls to Steps: Using online automatic homework checking tools to improve learning in introductory programming courses," ACET Journal of Computer Education and Research

This Article is brought to you for free and open access by the College of Business Administration at Digital Commons @ ACU. It has been accepted for inclusion in School of Information Technology and Computing by an authorized administrator of Digital Commons @ ACU.

From Walls to Steps: Using online automatic homework checking tools to improve learning in introductory programming courses

Dwayne Towell and Brent Reeves
Abilene Christian University
dwayne.towell@acu.edu
brent.reeves@acu.edu

Abstract

We describe the motivation, design, and implementation of a web-based automatic homework checker for Programming I and Programming II courses. Motivated by a problem-based-learning approach, we redesigned our first course to have over 70 short programming assignments. The goal was to change conceptual "walls" into "steps", so that students would not feel overwhelmed at any point in time. At each step along the way, it must be clear where the student is and the next step must feel attainable. Over the last 3 years, we have learned much about proper "step-size" and sequencing of problems. We describe how current computer science technologies both hurt and help our students. We conclude by a critique of the system, recommendations for undergraduate programming courses, and our goals for the next release.

Introduction

The motivation for this work arose from our observation that too many of our Programming I (CS1) students were completing the homework assignments but doing poorly on exams. It was not unusual for a student to have an A average on the homework and a D or F average on exams. Exams in these courses were both *written* and *practical*, meaning the students had to solve a programming problem in-class. As one possible explanation for this discrepancy in grades, we chose to investigate whether homework assignments were too infrequent. Perhaps the students did not do enough practice iterations, involving analysis, design, implementation and debugging of word problems, to be proficient on their own.

The traditional CS1 class includes several graded components: programming assignments to be completed out of class, proctored exams and possibly proctored programming exams. As noted above, given sufficient time and resources such as peers, study groups, course text, and web search tools, most students can manage to finish programming assignments. However, when under pressure to write working code on their own, such as during a practical exam, these same students were not able to accomplish simple programming tasks.

Several forces converge to bring this situation about. First, a typical course might include a dozen or so assignments, thus forcing each assignment to boldly push ahead and may include multiple new concepts. Given the limited number of assignments available, these "small steps" can appear to be "walls" to some students. Under pressure to complete assignments, enterprising students can always find some solution.

Unfortunately, their solutions tend to be aimed at getting the grade rather than practicing programming. Additionally, since each assignment is graded manually, usually the class has moved on to other topics by the time the feedback for an assignment is available to the student.

We decided students simply did not have enough practice or timely feedback. As we pursued possibilities, we looked around at other domains that possibly had a tradition of frequent practice and feedback. The obvious domain was Mathematics, because we were aware of the saying “The finger learns the math”, or “The pencil learns the math”, meaning the key in learning is repetition or much writing. Most mathematics courses, and especially introductory ones, have a large number of small assignments with answers readily available.

So our philosophy was to model homework assignments after Mathematics – there would be very many very small assignments. However, there is a practical limit to the number of assignments per semester using manual grading. There is not enough time to grade many assignments by hand, even with teaching assistants. Additionally, students deserve immediate feedback on their work, similar to having the answers in the back of the book in a mathematics course. So, our task became to somehow automate feedback for programming assignments (Reek 1989, Jones 2000, Jackson 1997). Out of this need was born Athene, our automatic program checker.

Athene

Athene is a web based system through which students submit their programming homework assignments. The “product” that students submit is a file of source code. Athene compiles, links, and runs the program through a test harness that inputs certain values and checks that the output is correct. A student navigates to the proper assignment, selects a local file to upload and within seconds receives a score on the program.

The system is written in PHP and uses PostgreSQL as a database server to store problems and submissions. We use gcc to compile the programs and apache as a web server. CodeIgniter served as PHP framework. A teacher makes assignments by choosing from existing problems and providing a due-date. Each problem has:

- Problem Description – an html page that describes what the challenge
- Scoring – a series of tests with expected outcomes and point values
- Test Suite – a set of test submissions to verify the scoring

Over time we expect to accumulate a set of many problems, from which teachers can select to serve their individual course needs. Problem descriptions vary from simple “Write a program that prints ‘Hello, World’” to as elaborate as necessary. They are html files and so can contain the usual items seen on web pages.

Scoring consists of a series of tests constructed with what we call “verbs”:

- `compile($args=' ')` : compile the program with any given arguments

- `source_contains_regex("/regex/s","description of regex for humans")` : does the source code contain the regular expression?
- `source_contains_C_function($return_type,$name)` : does the source code contain a function declaration with the given name and return type?
- `run($input)` : run the program and input the given values via stdin
- `output_equals($output)` : does stdout exactly match the given string?
- `output_contains($needle)` : does stdout contain the given string?
- `output_contains_lines($needle)` : do the given lines occur in the exact order in stdout?
- `output_contains_regex($needle,$description)` : does the regular expression is match stdout? If not, print the description. This separate description is necessary because we must tell the students what went wrong and regular expressions are difficult to understand.
- `output_does_not_contain($needle)` : is the output missing the given string?
- `score($number)` : set the current score to the given number; ranges from 0 to 1.0
- `message($msg)` : print the given message
- `hint($msg)` : provide a hint if an error occurs

These "verbs" allow us to check the output as well as the source code itself. The system is most useful for those problems that lend themselves to reading from stdin and answering to stdout, but has been used in other circumstances. Scoring consists of a series of tests and non-decreasing score assignments. As each test is passed, the score is expected to increase. This is different from the usual "task X is worth 10 points, task Y is worth 20 points, etc". Instead, the problem designer sequences the testing, knowing that the student is shown only one error at a time.

We make the distinction between "problem" and "assignment". We have accumulated many problems and categorized them initially in terms of courses (CS0, CS1, CS2). A teacher can choose any problem and create an assignment for their class. A few teachers willing to write problems make it possible to provide many choices and variety in assignments. We expect the quality and quantity of problems to increase over time. Since Athene logs all submissions, reviewing previous attempts provides feedback for improving existing problems and may suggest areas for new problems in the future.

What Happened

Athene is currently in the fourth semester of deployment for Programming I (CS1). A total of 114 students have made more than 18,000 submissions, each attempt was scored and feedback immediately provided to the student.

In an attempt to determine the value, if any, contributed by Athene in Programming I (CS1), overall grades from several sections of CS1 were compared. Six sections using Athene were compared with three sections that did not use automated scoring. Differences between sections could not be avoided, of course, however the same instructor using the same language and textbook were used in both cases. Of 46 students

in non-Athene sections, 46% received a final grade of A, B or C. Of 79 students in Athene sections, 71% received a final grade of A, B or C.

We present observations during our experience over the past three years: first to students, then to assignments, and finally to the system itself.

Impact on Students

Using Athene, students are allowed to submit work at any time up to the deadline set for that particular assignment. Additionally, they are allowed multiple attempts for each assignment without penalty. Unlike the traditional system which requires a grader, if a deficiency exists the student is immediately provided feedback about one specific issue with the solution and encouraged to correct it. This change in the usual grade mechanism has had both expected and unexpected consequences.

Before addressing changes observed, it is worth noting a change that was specifically not observed. Allowing students many attempts means that most of the time they are faced with negative feedback. It was anticipated that this overwhelming and continuous negative feedback could be discouraging, similar to a paper that the teacher has "bled all over". Happily, we have not seen this to be the case. In hindsight it seems likely this is due to the impersonal nature of the feedback.

As expected, program verification became the responsibility of the assignment author and most students became conditioned to perform very little, if any, verification before submitting an attempt at the solution. In a few extreme cases, it even led to clear violations of the intended solution, as students crafted a program with answers "hard-coded" for the test data. While this tendency was initially disappointing, it came with an unexpected and welcome side-effect unavailable with the traditional mechanism: the teacher is allowed to observe partial solutions.

Student attention to feedback from Athene not only allows the assignment author to help direct the student, but naturally draws the student into the problem, resulting in a higher than expected level of investment by the student. Once started, students tend to complete the assignment. Last semester, of 1524 assignments attempted, only 102 received less than a perfect score. One possible explanation is that the problems take on a challenging aspect which both encourages completion and imbues the task with elements similar to solving an intriguing puzzle.

It should be noted that not all students abandoned desk-checking and verification before submitting their work. For the more advanced students, solving each assignment in the fewest number of attempts is a badge of honor and provides a venue for good-natured peer-rivalry.

Programming I students submit about 75 assignments during the course of a semester long class. Many of the assignments share aspects with other assignments, such as utility functions or output formatting. These similarities force students to either gain practice

with a common solution or optimize for reuse code. Students tend to naturally choose between these options based on their individual need. Advanced students want to "save time" by reusing code, while those struggling simply need another opportunity to practice a construct. The bimodal nature of the student population insures there is a need for both options in most every class.

Crafting Curriculum

The most obvious curriculum change is the number of individual scored assignments each student completes during the semester. Without an automated scoring system, logistics limited the number of assignments per student per semester to about fifteen and the process delayed feedback about a week from the time the assignment was actually completed. Automated scoring allows the assignment author to create "throw away" assignments; ones so simple that previously they would not have been worth assigning. This is particularly true during the beginning of the course; currently half of the individual assignments are due during the first third of the semester. In addition to small assignments, a series of variations on a theme can be assigned to demonstrate contrasting alternative approaches. This also encourages students to reuse code as described above.

A challenge for any automatic scoring system is that it is not sophisticated enough to distinguish between trivial and serious errors. It might not be fair to receive poor marks on a program for forgetting to print a period; for example "Hello, World" instead of "Hello, World.". A human might decide both solutions score full marks. To add the kind of knowledge that even a young teaching assistant brings to bear would be prohibitively expensive. However, balanced against this disadvantage is instant feedback, 24x7 availability and multiple attempts. Given this tradeoff, we insist students get the output perfect, character for character since the system shows what their program printed and what the teacher expected. "Overly-picky" scoring is not a problem if the feedback is clear and students may submit multiple times.

When students become accustomed on the automated scoring system to provide verification new forces are introduced. Many students rely heavily on Athene as a test harness during debugging. Unfortunately, this can lead to myopic focus on the latest reported problem and current score. This encourages the unmotivated student to immediately halt work should a "solution" be found, even if that solution does not solve the general case. In extreme cases, frustrated or deceitful students have gone so far as to produce a program tailored to a particular data set. These student responses force assignment authors to take responsibility for complete verification of submissions, lest a student "get away" with a poor solution. At first, this was perceived as a negative force, but with additional experience we believe it encourages the author to define the problem more clearly.

Additionally, student reliance on verification allows well-crafted assignments to guide the student in a particular direction. Unlike compilers, which announce every possible problem in one vast disgorging of errors, Athene is designed to help the student focus on the next problem to be addressed. Using a combination of how the problem is stated and

the order in which requirements are verified, a student can be encouraged to address particular aspects of the assignment in an appropriate order. For example, if implementing the math is deemed more fundamental than producing a specific output format, then verifying the correct numeric answer can be checked before addressing the formatting issues.

Assignment authors have capitalized on both the peer-rivalry and challenging nature by revealing optional extensions to the assignment once it has been completed. Since the extension is optional and only revealed after the student has successfully completed the primary assignment, the extension is received as a challenge. Not all students accept the challenge, but for those that do, the time to complete the extension may be up to twice the time needed to solve the unextended assignment. This "up sell" of selected assignments is one way to serve both ends of bimodal student populations.

The collected set of assignments for a course represents a standard repository capturing the collective "wisdom" of many teachers. It also provides a standard rubric and allows comparisons across semesters and even teachers. Each semester submissions from the past semester are reviewed for anomalous submissions. In addition to exposing and patching "loopholes", this process allows troublesome assignments to be refined, re-sequenced or even replaced if needed.

One fortuitous consequence of this architecture is that a our step-size did not just decrease by virtue of going from a dozen assignments to 70 assignments, but in an important sense, the step size decreased by a factor of 5 - 10 because every test in the scoring file represents a step. Since the user sees only the next error, and since that error is completely designed by the problem designer, that designer has extremely fine control over the sequence in which they want to present all the requirements.

This fine grain control comes at no cost to the teacher, because the problem designer can make as few tests as they wish. In the extreme case, for example, "Hello, World.", there can be just one test, namely a string match. This fine grain control also comes at no "cost" to the student. The student never sees the tests that their program passes; that is, a correct program passes all tests and receives 100% without intermediate steps. The student is never forced to take tiny steps; they in essence, "jump over" the intermediate steps.

We believe the most important part of problems in our first three programming courses (CS0, CS1, CS2) are easily handled via a stdin/stdout testing architecture. While other testing options can be created, in many cases it is easier to change the specification of the problem instead. For example, in a course using Visual Basic Express, all of the textbook examples are "windowed" applications, not "console" applications. The teacher translated the problems into stdin/stdout. For example, one assignment has the user paint a window with two text boxes, a button, and a label. After entering two numbers, the user is expected to click the button and the problem then calculates something and sets a text label. The translation has the two textboxes as "writeline()" statements, input using "readline()", and finally printing the calculated value via another "writeline()".

System Upgrades

Since the original version of Athene, several changes have been introduced. Originally all submissions were verified as "black box" programs; only standard input could be manipulated and only standard output was examined. For some programs this worked well, however other assignments did not lend themselves to this format. So, several "injection" methods were developed to allow verification code to be inserted into student programs. While the particulars of the current methods are specific to C/C++, similar methods would work for other languages. In particular, languages such as Java which support reflection would allow even more control of the verification process. Depending on the intent of the assignment author, a submission may be compiled against several libraries and/or include verification code specific to the assignment. This level of inspection allows the author to be confident the student is solving the problem as specified by the assignment.

The review of submissions each semester led to a number of changes. While teachers can review any submission from any assignment or any student, reviewing hundreds of submissions is prohibitive. However the desire to allow teachers to more quickly identify anomalous submissions prompted a simple experiment using code metrics. Several code metrics were computed for each submission and included in submission directory listings, allowing anomalous values to be quickly spotted. McCabe Complexity was found to reliably suggest submissions worthy of human review (McCabe 1976). While we are pleased with this metric, we believe a metric that measures the size of program state, such as the number of variables, would also be useful.

As new interesting submissions are discovered each semester the need to regression test assignments grew and became cumbersome. The current system includes a test harness for assignments allowing the author to create or collect a large number of submissions to help evaluate each scoring routine.

Summary

We needed a way for our students to do many more programming assignments. We looked to the methods used in teaching Mathematics and developed the equivalent of the many small bite-sized exercises given in typical introductory to intermediate Mathematics courses. We developed Athene, a web based system that compiles and runs student's source code and subjects their program to various inputs, checking that proper output is produced.

Experience with the system shows students prefer the system to the usual dozen assignments that are graded slowly. The small step sizes do not impede the advanced students; because they understand the pattern or topic common to two homework assignments and are able to reuse the appropriate parts from previous homework. We were encouraged by the strong positive correlation between system use and the number of students earning A, B and C in CS1. Small steps work well.

Future Work

As anecdotal success stories spread, other teachers began to express an interest for use in other courses. Several other courses are planned for the near future including Web Development I, Introduction to Databases, and Object Oriented Programming. Our short term plans include adding support for Java and SQL. Adding support for text-based languages like these is not particularly difficult and we do not anticipate that architectural changes will be necessary.

Long term plans include graphics-based assignments. It would allow grading a program that supports graphics interaction, like a simple game in an introductory computer graphics course.

In addition to supporting a greater variety of assignments, we also look forward to better analysis of submissions. We will use static analysis tools on our existing and new submissions in ways that will help our students learn even better the fundamentals of computer programming.

References

Jackson, D. and User, M., "Grading Student Programs Using ASSYST," Proceedings SIGCSE '97, 1997, pp. 335-339.

Jones, Edward, "Grading student programs – a software testing approach," Proceedings of the 14th Annual CCSC Southeastern Conference, 2000, pp. 185-192.

McCabe, T.A., A complexity measure, IEEE Transactions on Software Engineering, vol. SE 2, no. 4, 1976, pp. 308-320.

Reek, K.A., "The TRY System or How to Avoid Testing Student Programs," Proceedings SIGCSE Bulletin vol. 21, no. 1, February 1989, pp. 112-116.