2021 ASEE ANNUAL CONFERENCE
Virtual Meeting | July 26–29, 2021 | Pacific Daylight Time
Paper ID #32796

Progression Highlighting for Programming Courses

Nabeel Alzahrani, University of California, Riverside

Nabeel Alzahrani is a Computer Science Ph.D. student in the Department of Computer Science and Engineering at the University of California, Riverside. Nabeel's research interests include causes of student struggle, and debugging methodologies, in introductory computer programming courses.

Prof. Frank Vahid, University of California, Riverside

Frank Vahid is a Professor of Computer Science and Engineering at the Univ. of California, Riverside. His research interests include embedded systems design, and engineering education. He is a co-founder of zyBooks.com.

Progression Highlighting for Programming Courses

Nabeel Alzahrani, Frank Vahid*

Computer Science and Engineering

University of California, Riverside

{nalza001, vahid}@ucr.edu

*Also with zyBooks

Abstract

Many CS instructors desire to see more than a student's final submission on a programming assignment, wanting visibility into how the student developed their code. Recently, program auto-graders have grown in usage. A popular cloud-based auto-grader, used in over 2,000 university courses per year, provides a log with an entry whenever a student does a develop run or submission run of their code. Beyond time/date and score, each entry links to a source code snapshot. Using that log file as input, we introduce "code progression highlighting" as a mechanism for instructors to gain visibility into students' code development. The highlighter starts with a statistical summary for each student in roster form, sortable by any statistic such as time spent, number of attempts, code size, a struggle metric, and more. For any student, an instructor can expand to view all code entries from that student, highlighted to show changes from the previous entry (the "progression"), with statistics per entry like time spent, characters changed, current score, etc. The progression highlighter helps instructors to assist students in office hours, to detect some cheating not detectable by similarity checkers, or to see where students are struggling. We intend to make the progression highlighter free on the web for instructors to use, simply by uploading the log file from the popular auto-grader, or from any system whose log file is converted to that format, and thus can serve the community of CS instructors to gain insights on their students' code development processes.

Introduction

The growth in usage of program auto-graders enables new possibilities for CS instructors. Previously, most schools graded manually, while some schools used custom-built auto-graders, or made use of the freely-available Web-CAT tool [1]. However, in the past few years, several cloud-based commercial auto-graders have appeared, such as zyBooks [2], Gradescope [3], Mimir [4], Vocareum [5], CodeLab [6], and MyProgramming-Lab [7], many emphasizing ease of use and immediate score feedback to students. Based on public information from and direct discussions with those companies, we can conservatively state that at least 500 universities and at least 1,000 courses have switched from manual grading to auto-grading in recent years, impacting well over 250,000 students per year. According to a recent whitepaper from zyBooks, there is a steep rise in the use of auto-graders in recent years. For example, courses that used zyLabs (their program grader) grew from 284 in 2016 (the year zyLabs was first released) to 2,175 in 2020, the number of students from 24,216 to 132,121, the number of instructors from 364 to 2,866. They indicate 20 million submissions were graded in 2020. This rise in auto-grader usage enables instructors to spend more time engaging with students and improving the education process. For example, instructors stated that switching to the auto-grader on average saved instructors 9 hours per week [8].

These new cloud-based auto-graders not only provide immediate score feedback to students, versus auto-graders run in batch mode by instructors after all programs are submitted, but some also provide built-in development environments where students develop their code, versus using an external integrated development environment or IDE. We conducted an online survey in Spring 2021 in one of our online CS1 C++ courses where 117 students participated. Most students were positive about auto-graders. For example, 98% liked the immediate score feedback, 80% disagreed that immediate score feedback caused stress and/or confusion, and 92% preferred an auto-grader over a human grader with feedback a week later.

The cloud-based auto-grader IDE provides a new opportunity of giving instructors a view into how their students develop their code -- when students started, how long they spent, how many times a student ran their code on their own ("develop" runs), how many times they submitted for grading and what score was received ("submission" runs), and so on. Auto-graders typically grade using input/output test cases, most of which are visible to the student, so students know how their grade is being determined, and can correct their program is response.

Student ID	Timestamp	Code link	Run	Score	Max score
102	1/1/20 9:20:03	URL1	Dev		
102	1/1/20 9:21:15	URL2	Sub	4	10
102	1/1/20 9:30:10	URL3	Sub	10	10
101	1/1/20 9:32:49	URL4	Dev		
101	1/1/20 9:33:40	URL5	Dev		

Figure 1: Example log file from an auto-grader (abbreviated).

We use the zyBooks program auto-grader. They recently began providing instructors with a detailed log of students' develop and submission runs for any programming assignment, depicted in Fig. 1 in simplified form. Each entry in the log includes the student's identifying info, the date and time, and a link to the run's source code. For submission runs, the log also shows the score received, and the max possible score.

As instructors of a CS1 course that serves about 1,500 students per year, we switched from our custom auto-grader to the zyBooks auto-grader several years ago -- but we note that our approach in this paper can be applied to any auto-grader's log files if the info is similar. We began developing scripts to process the provided log files to gain further insights into our students' behavior, such as script that estimates how long students spent on each program (and how long relative to the rest of class), counts the develop and submission runs, estimates whether detected a student is struggling (excessive submit runs or time spent), etc., as illustrated in Figures 2 and 3. Importantly, the instructor can sort by any column, such as lines of code, time spent, etc. This ability allows an instructor much flexibility in how to prioritize viewing of students. For example, an instructor might sort by time, and then examine one end of the roster to see why some students completed the program in almost no time, which is sometimes due to cheating (copy-paste of a solution obtained from a friend or online "tutor"), a student not following the requirement of submitting frequently and/or developing within the auto-grading environment, a highly-experienced student who simply got the program right quickly, or a solution that passes the test cases with much less code than the instructor expected (which could mean a problem with the test cases, or that the student found a clever simple solution).

Number of students	Average score	Average time spent	Average number of submissions	Average lines of code
31	7.5	17.8	13.4	30

Figure 2: Class statistics (abbreviated).

ID	Score	Time spent (min)	Number of submissions	Code size	Struggler metric
101	6	37	4	15	1.0
102	10	23	4	49	0.3
…	…	…	…	…	…

Figure 3: Student roster view with statistics (abbreviated).

While these statistical summaries were useful and form a part of our contribution in this paper, over time, we also desired a way to see the progression of a student's coding. The auto-grader’s web interface does provide links to all submitted code, but that was insufficient -- manually examining each code run didn't provide a quick or easy way to gain insight on what the student changed in the code.

This paper describes a tool we developed to automatically highlight differences between each run, and to provide statistical data for those runs, forming the main part of our contribution. The current tool implementation is language independent. We plan to make the tool freely available on the web (both the roster/statistics and the progression highlighting), to serve CS educators who may wish to gain insights on their students and/or to conduct research on programming assignments. The tool takes a log file with a simple format as described above, which is the default for zyBooks (and hence immediately usable for 2,000+ courses), but any auto-grader, commercial or custom, can have their log files auto-converted to that format for importing to our tool as well.

Assignment: Find max of three values

Spec: Given three input integer values, output the max value. If input is 5, 9, 3, output is 9.

Student1's submission (S1):

```
#include <iostream>
using namespace std;

int main() {
  int x, y, z;
  cin >> x >> y >> z;
  if ((x > y) && (x > z))
    cout << x;
  else if ((y > x) && (y > z))
    cout << y;
  else
    cout << z;
  cout << endl;}
```

Figure 4: Example of a simple programming assignment.

Code Progression Highlighting Tool

A programming assignment specifies a programming task for students. Fig. 4 is a sample programming assignment, followed by an example of a submission from a student whose name is shown as S1; real assignments are commonly much larger.

Fig. 5 provides an example of code progression highlighting. The input is the log file, which includes all develop and submit runs.

Code	Run type	Score	Min. since prev	Total min	Timestamp	Ins(+) Change(^) Del(-)

if ((x > y) && (y > z))
  cout << x;
else if ((y > x) && (y > z))
  cout << y;
else
  cout << z;
	Dev		0	0	1/1/20 9:20:03	

if ((x > y) && (x > z))
  cout << x;
else if ((y > x) && (y > z))
  cout << y;
else
  cout << z;
	Sub	4	0.5	0.5	1/1/20 9:21:15	^1

if ((x >= y) && (x > z))
  cout << x;
else if ((y > x) && (y > z))
  cout << y;
else
  cout << z;
	Sub	5	6.9	7.4	1/1/20 9:30:10	+1

Figure 5: Overview of the code progression highlighting tool, to help instructors quickly see a student's changes. The change from y to x is highlighted in the second run, and from > to >= in the third run.

Given the earlier mentioned roster view with statistics, if an instructor has decided to look further into a student's code, the instructor can click on a link to be taken to a page that provides highlighted code for every develop or submit run by that student, shown in Fig. 5.

The progression highlighting tool highlights any code that was added, changed or deleted from the previous run, so that instructors can quickly see the changes made.

For each code run, the tool also provides statistics, as shown in Fig. 5. The statistics include type of run (sub or dev), the number of characters added, changed, and deleted; the time spent between this run and the previous run, elapsed time (time spent so far since the first run), timestamp for each run, score for each run. If the time exceeds ten minutes between runs, we assume the student stepped away, and ignore that time.

We note that highlighting, and some statistics, are available from program version control systems like Git [9], Subversion [10], CVS [11], and Mercurial [12]. However, our approach differs in several ways. First, we don't require the use of such version control tools; students simply develop and submit using their class auto-grader. This simplicity can be especially important in early CS classes like CS1 or CS2. (Version control may be taught later in a class or course sequence) Second, our statistics are specifically intended for instructors, so include items not found in most version control systems. Third, we estimate time spent, which is lacking in version control systems.

The code progression highlighting tool is a web tool written in Python 3 using the Common Gateway Interface (CGI) specifications [13]. The tool is about 5K lines of code. The tool consists mainly of two parts: the first part is the student roster builder and the second part is the code progression highlighter. Moreover, each part consists of multiple subroutines and dictionary data structures. For example, for the student roster builder, there is a subroutine to read the log file and build a dictionary that contains the students (indexed by the students' IDs) with their codes and their codes' meta-data (such as timestamp, score, type of run, etc.). Another dictionary builds a class statistic, and another one for top quartile students. Likewise, the code progression part contains multiple subroutines and dictionaries. For example, one subroutine finds the difference between consequent codes and highlights them in yellow.

Problem statement

Numerous engineering and scientific applications require finding solutions to a set of equations. Ex: 8x + 7y = 38 and 3x - 5y = -1 have a solution x = 3, y = 2.

Given integer coefficients of two linear equations with variables x and y, use brute force to find an integer solution for x and y in the range -10 to 10.

Solution

```
int main() {
  int a1, b1, c1;
  int a2, b2, c2;
  int x, y;
  bool eqn1Solved, eqn2Solved;
  bool solutionFound = false;
  int xSolution, ySolution;
  cin >> a1 >> b1 >> c1;
  cin >> a2 >> b2 >> cin >> c2;
  for (x = -10; x <= 10; ++x)
    for (y = -10; y <= 10; ++y)
      eqn1Solved = ( (a1*x + b1*y) == c1 );
      eqn2Solved = ( (a2*x + b2*y) == c2 );
     if (eqn1Solved &&  eqn2Solved) {
       solutionFound = true;
       xSolution = x;
       ySolution = y;
     }
  if (solutionFound)
    cout << xSolution << " " <<
    ySolution << endl;
  else
    cout << "No solution" <<
    endl;
}
```

Figure 6: One example of the 5 selected programming assignments.

Applications

The progression highlighter can be applied by instructors in many ways, some of which we have begun doing in our own classes. Note: The progression highlighter and in fact any new analysis tool works best if students are required to do all development in the auto-grader's environment. In fact, for our CS1, we long ago started using the auto-grader's built-in development environment for most programs, thus avoiding issues with new students installing/learning external IDEs, and we are aware of hundreds of schools that do the same using that same auto-grader. However, for courses that want students using an external IDE, instructors can simply require frequent submissions, such as every 20 minutes or every 20 lines of code (for example). Again, we have found many courses already have such requirements, to encourage students to start early, to develop incrementally, and to decrease the likelihood students will just copy-paste someone else's solution.

One application is to bring up a student's highlighted progression for a student who has come to office hours in need of help. The progression gives the instructor a powerful view of the student's effort history. Did the student start early or late? Are they developing incrementally or writing large pieces of code all at once? Are they testing their created functions first or just using them untested in their larger code? When faced with an erroneous program, are they properly debugging or just making random changes to see what happens, or are they focused on the wrong region of code? We have found the progression highlighter exceptionally useful for such purposes, providing insights that we could not see before.

Another application is cheating detection, or even better cheating reduction/prevention. An instructor can sort students by number of runs or time spent, and investigate those students who spent the least time or runs, to see if they simply copy-pasted a solution. With the develop/submission requirement above, and by showing students the progression highlighter tool in class (perhaps in a fun manner during class like seeing how students did on a low-stakes programming task, and/or showing the tool with student names hidden), students may be less tempted to simply copy-paste a solution – students regularly observe that instructors can and do look at their progression, and thus simply pasting a solution will be a smoking gun. For in-class use, we include an "anonymize" button that hides student names/email from the display. Furthermore, if an instructor by other means suspects a student of plagiarizing code (perhaps via similarity detection from a tool like MOSS [14], or by dramatically higher programming assignment scores than exam scores), the instructor can examine that student's highlighted progression for further confirmation that the student did not develop the code. Note that statistics alone, like number of runs or time spent, can be more easily faked (via bogus runs before eventually copy-pasting a solution). But faking a code *progression*, though still possible, begins to approach the skill of just writing the code itself.

Logical errors	Details	Suggestions

Data type conversion	Assigning data (int) to the wrong data type variable (string) is an undetectable error (no error or warning message from the compiler).
	int integerValue;
	string stringAsBinary;
	stringAsBinary += integerValue % 2;
		Use if-else to convert between data types and check the result.

Return value	Missing return value.	Check the value of a function return before using it.
String indexing with for-loop	Not knowing that string contents are accessed from a[len-1] to a[0] in reverse order. for (i = userString.size(); i > 0; --i)	String indices are from 0 to n-1 for ascending order and from n-1 to 0 for descending order.

Loop counter wrong data type	Using unsigned int as a for-loop counter causes the for-loop to be infinite after subtracting one from zero.
	unsigned int i;
	string newString;
	newString="";
	for (i=userString.size()-1; i >= 0; --i){
	    newString=newString+userString.at(i);
	   }
		Use int instead of unsigned int data type when declaring a loop counter.

Figure 7: Errors by strugglers in the programming assignment “Convert to binary - functions”.

Many other applications exist, such as quickly gaining insight on what errors are causing a class to struggle, as we report on next.

Experiences using the tool to find causes of struggle

The progression highlighter helped us for a particular application. We sought to determine what coding mistakes caused students to struggle in our CS1 class. Previous research described common mistakes among learners [15, 16], but just because a mistake is common does not make the mistake bad; much learning comes from making and then fixing mistakes. In contrast, some mistakes cause students to struggle, which means the student cannot find their error, resulting in numerous runs, excessive time spent, and frustration. Frustrated students are more likely to resort to cheating, or to give up on the assignment, which may eventually lead to failing grades or dropping the class as well.

For each of 5 selected programming assignments that we assigned in Spring 2017, we uploaded the auto-grader's log file into our tool. One example is shown in Fig 6. Next, we sorted by time, which is one measure of struggle. (Number of runs is another). For the students who spent more than 30 minutes, we clicked on the link to be taken to the progression highlighter, and then scrolled through to quickly and easily see where the student was spending their effort, and could see the error they were making as shown in Fig 7. We discovered that the following errors were common logical errors among students that cause struggle (the main list is the programming assignment while the sub list are the errors per a program assignment):

- Brute force equation solver
    - Typos in formula
    - Convert to binary - functions
    - data type conversion
    - return value
    - string indexing
    - loop counter
- Palindrome
    - lack of plan
    - string indexing
- Count characters
    - lack of plan
    - using cin() instead of getline() to read a phrase
    - String indexing
- Leap year - functions
    - if vs, if-else

Program Assignment	strugglers / students	Total runs by strugglers	Avg. score out of 10	No. Errors	Teacher manual inspect time (min)
Brute force equation solver	3 / 40	199	8	1	12
Convert to binary - functions	11 / 56	414	6	4	50
Palindrome	7 / 49	476	7	2	38
Count characters	17 / 74	790	8	5	47
Leap year - functions	3 / 71	96	9	1	04

Figure 8: Strugglers (may or may not get full credit) in the five programming assignments. A struggler is a student who spends 30 min. or more in a programming assignment.

The summary of our findings is shown in Fig. 8. The time spent column on the right shows that we only needed tens of minutes per program to find common errors. In contrast, in the previous summer we did a similar analysis, but that required several weeks, since we had to manually examine code without the benefit of the progression highlighter, which not only took much longer but was also quite tedious and tiring.

Conclusion

This paper introduces an enabling technology for instructors of programming classes that emphasizes viewing a student's *progression* through the program development process. The technology is a tool that takes as input a log file from a program auto-grader used by thousands of classes (but could be used with other tools too if their logs were converted to the format). The tool provides instructors with a unique ability to sort their class roster according to various statistics like time spent, code size, or number of runs. Then, importantly, the tool allows instructors to view any student's highlighted progression through the program development process, seeing each run's code, with highlights showing additions/changes from the previous run, plus statistics for that code like time spent, number of additions/deletions, and score. The technology enables various new capabilities for instructors. For example, an instructor can open a student's highlighted progression for a student in office hours, to help better understand the student's approach to a program and to provide advice. Or, an instructor can sort by number of runs or time spent to detect students who might be struggling, choosing students with low scores at the top of that list, and examining their progressions to see what errors prevented them from scoring well, which we demonstrated in this paper can be done in just tens of minutes, yielding powerful insights that can improve an instructor's teaching. Or, an instructor can detect potential cheating by detecting students who got full credit with little effort (few runs or little time) -- or even better, to reduce/prevent cheating by showing students the tool (ideally in a fun way, perhaps on low-stakes programming tasks) so students realize their programming process is being viewed, and not just their final submission. Dozens of possibilities exist, and we look forward to seeing how instructors use this technology to improve their classes and to conduct future research. Future work may include providing support for sets of labs (like all labs for a week), supporting team projects, and more.

Acknowledgements

This work was supported in part by the National Science Foundation (grant number 1563652).

References

[1] Stephen H. Edwards and Manuel A. Perez-Quinones. 2008. Web-CAT: automatically grading programming assignments. In Proceedings of the 13th annual conference on Innovation and technology in computer science education, 328–328.

[2] zyBooks, www.zybooks.com, August 2020.

[3] Arjun Singh, Sergey Karayev, Kevin Gutowski, and Pieter Abbeel. 2017. Gradescope: a fast, flexible, and fair system for scalable assessment of handwritten work. In Proceedings of the fourth (2017) acm conference on learning@ scale, 81–88.

[4] Mimir, www.mimirhq.com, August 2020.

[5] Vocareum, www.vocareum.com/home/programming-lab, August 2020.

[6] Turing's Craft: CodeLab. https://www.turingscraft.com, August 2020.

[7] Pearson: MyProgrammingLab. www.pearsonmylabandmastering.com, August 2020.

[8] Chelsea Gordon, Roman Lysecky, and Frank Vahid, "The rise of the zyLab program auto-grader in introductory CS courses," zyBook.com, White Paper, 2021. [Online]. Available: https://docs.google.com/document/d/e/2PACX-1vQYwxlY738_9zFFwOer1kKTNGuJx1Qe3IDW8XHf_OOYbaq9Drf_a9ljCqjcHY9Vv4ryPK423W7FmHwZ/pub

[9] Jon Loeliger and Matthew McCullough. 2012. Version Control with Git: Powerful tools and techniques for collaborative software development. O'Reilly Media, Inc.

[10] Louis Glassy. 2006. Using version control to observe student software development processes. Journal of Computing Sciences in Colleges 21, 3 (2006), 99–106.

[11] Keir Mierle, Kevin Laven, Sam Roweis, and Greg Wilson. 2005. Mining student CVS repositories for performance indicators. ACM SIGSOFT Software Engineering Notes 30, 4 (2005), 1–5.

[12] Daniel Rocco and Will Lloyd. 2011. Distributed version control in the classroom. In Proceedings of the 42nd ACM technical symposium on Computer science education, 637–642.

[13] David Robinson and Ken Coar. 2004. The common gateway interface (CGI) version 1.1. Network Working Group, RFC3875, Category: Informational (2004).

[14] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. 2003. Winnowing: local algorithms for document fingerprinting. In Proceedings of the 2003 ACM SIGMOD international conference on Management of data, 76–85.

[15] Renée C. Bryce, Alison Cooley, Amy Hansen, and Nare Hayrapetyan. 2010. A one year empirical study of student programming bugs. In 2010 IEEE Frontiers in Education Conference (FIE), IEEE, F1G-1-F1G-7.

[16] Andrew Ettles, Andrew Luxton-Reilly, and Paul Denny. 2018. Common logic errors made by novice programmers. In Proceedings of the 20th Australasian Computing Education Conference, 83–89.