ASEE 2022 ANNUAL CONFERENCE
Excellence Through Diversity
MINNEAPOLIS, MINNESOTA, JUNE 26TH-29TH, 2022 ASEE
Paper ID #36493

Auto-Awarding Points for Incremental Development in Programming Courses

Frank Vahid (Professor)

Frank Vahid is a Professor of Computer Science and Engineering at the University of California, Riverside, since 1994. He is co-founder and Chief Learning Officer of zyBooks, which creates web-native interactive learning content to replace college textbooks and homework serving 500,000 students anually. His research interests include learning methods to improve college student success especially for CS and STEM freshmen and sophomores, and also embedded systems software and hardware. He is also founder of the non-profit CollegeStudentAdvocates.org.

© American Society for Engineering Education, 2022
Powered by www.slayte.com

Auto-Awarding Points for Incremental Development in Programming Courses

Abstract

Programming textbooks and teachers commonly advise students to develop code incrementally, avoiding writing large amounts of code between test runs, and instead compiling/running frequently. However, without an easy way to give students points for following an incremental development process, students often write large blocks of code and then "hope it works". They may also rush through the development process. Today, many environments capture every student program run, which opens new possibilities. The paper describes a heuristic, incorporated into a tool, that automatically awards points to students for following an incremental development process. The heuristic uses a deplete/replenish approach (similar to "energy level" in video games). Violations of incremental development rules deplete a student's incremental-development points. One kind of violation is exceeding an added-code threshold between runs, such as adding no more than 20 lines, to encourage incremental development. Another violation is an autograder submission-separation threshold, such as 1 minute, to discourage spamming the auto-grader and instead to encourage self-testing. The heuristic scales the depletion proportional to the extent of the violation. To enable students to dig themselves out of a hole, we provide ways for students to replenish incremental-development points, adding points back each time they run code without any violations. We ran the tool on log files from a popular commercial program auto-grader, for 3 labs from our CS1 course, and found the points awarded were computed quickly and within reason of instructor expectations. We hope to make the tool available for the CS community to enable instructors to easily inform students of incremental development goals and to award points automatically.

1. Introduction

Students often write large chunks of code at once, such as 30-40 lines, without testing that code along the way, not even to check if it compiles. Such code may have multiple bugs in the code, which students then have a hard time finding, especially when multiple bugs interact. The approach can result in students: (1) spending excessive time debugging, (2) imposing demands on instructor office hours for help, and (3) getting frustrated and even quitting, especially if the code never ends up working correctly.

Instead, computer science (CS) instructors and textbook authors encourage incremental development, which is an approach of writing only a few lines (perhaps 5-20), testing the code, and repeating. The approach is more likely to catch bugs when introduced, making bugs easier to find (as they are likely found in the new code), and before multiple bugs start interacting.

Unfortunately, because incremental development is usually not rewarded with points, and instead only the final working solution earns points, students continue to write large chunks of code at once, thinking that approach gets them to a working solution faster. Tools have long existed to automatically grade student programs [CaDe13][GoLy21][UlLa18], but the automated grades are

based on correctness (and possibly style) of the final solution, and not on the programming process. Recent emphasis has examined autograders in various ways, such as providing better feedback too [DeSr17], but not the programming process itself. Various CS teachers do give some points for good process, such as giving some points for starting early or earning some points before a given date, for checking code into GitHub regularly, etc.

Today, various programming environments log much of a student's development, such as saves, develop runs (the student self-testing their code), submit runs (the student submitting their code to an auto-grader for points), and more. Some previous work has performed various analyses of student programming logs to better understand student behavior [EdLi16][KaEd17].

In this paper, we aim to use such information to automatically award points to students for following an incremental development process, with students being provided rules to follow to earn those points. We built the tool using the log file downloadable by instructors from a popular commercial auto-grader. The tool can help transform students' mindsets from "I just need to submit a working final solution" to "I need to follow a good process on the way to a working final solution", leading hopefully to more students having less debugging frustration and failure, and a more positive experience in their programming classes. The tool is a first step towards automatically giving points for students following good process (start early, test regularly, spend sufficient time, etc.), complementing points awarded automatically for final-solution correctness.

2. Logged student behavior

Our approach takes as input the log file that instructors can download for any programming assignment in the zyBooks system [Zy21]. We chose this log file format not only because it is the system we use in our classes, but also because it is likely the most widely used program auto-grading system in introductory CS courses, used in 2020 by 2,000 courses and 130,000 students [GoVaLy21]. However, our approach can take input from any other system that logs similar info -- a simple script would convert such logs into the desired input format.

The log file has a row for every run by every student on a particular programming assignment. Key info includes the student name and/or id, timestamp, run type, and a link to the code that was run. The run type can be a "develop" run where the student is just testing the code on their own, or a "submit" run where the student is submitting the code to the auto-grader to earn points for correctness. Figure 1 shows a simplified representation of such a log file.

```
Student, Time, Run type, Code
1001, 8/5/21 09:34:25, dev, , http://... (15)
1001, 8/5/21 09:36:10, sub, 5, http://... (20)
1035, 8/5//21 09:36:42, dev, , http://... (30)
1001, 8/5/21 09:36:50, sub, 6, http://... (21)
```

Figure 1: Simplified log file from an auto-grader.

In the figure, student 1001 did a develop run at 9:34 am (actually at 9:34 and 25 sec). That same student 1001 did a submit run at 9:36:10 that scored 5 points. Student 1035 did a develop run at 9:36 as well. Student 1001 did another submit run at 9:36:50, just 40 sec after their previous submit run, and made some change that earned an additional point, so 6 points total.

Each run has a web link to the source code file that was used in that run, in this case stored on AWS (Amazon Web Services). For our purposes, that source code is useful to enable us to get the lines of code (LOC) for each run. Those LOCs are shown in the figure in italics next to each web link, but don't actually exist in the log file; the LOCs have to be calculated by auto-accessing the code via the web links and calculating the LOC.

3. Incremental development points

The availability of such a log file enables a tool to analyze the file and award each student some additional points for following predefined development process rules. We sought to define an analysis heuristic that:

- Enforces some reasonable incremental process
- Is simple enough to be easily understood by students

We sought a heuristic that, for each student, outputs a score from 0 to 1, what we call the IncDev value, such as 1 (superbly carried out incremental development), 0 (did a terrible job incrementally developing), or 0.5 (did a poor job). That score can be multiplied by a constant to map to any number of points an instructor actually wants to award, like 0-5 points (obtained via 5 * IncDev).

3.1 Depletion: The "Added LOC" rule

A simple but weak rule is an "all or nothing" rule stating that every run's LOC should not exceed the previous run's LOC by more than 20 lines (or some other number chosen by an instructor or via automated means, discussed later). We refer to such a rule generally as the AddedLOC rule, the lines added to a previous run as addedLOC, and the 20 as the addedLOC threshold. In other words, this rule says students should not write large chunks of code without running that code along the way. If all a student's runs obey the rule (addedLOC <= 20), then IncDev = 1, but if any run violates the rule, IncDev = 0.

The rule is weak in being all or nothing: either IncDev = 1, or = 0. Instead, preferably the rule would yield values between 0 and 1, representing the egregiousness of the rule violation(s).

For example, a student who has just one violation with addedLOC = 21 should not be penalized as heavily as a student with one violation with addecLOC = 50. Thus, we want the penalty to be proportional to the size of the addecLOC violation.

Likewise, a student who has just one addedLOC = 30 should not be penalized as heavily as a student who has three addedLOCs of 30 (a repeat offender). Thus, we want the penalty to be

proportional to the number of addedLOC violations.

To achieve the first proportionality goal, we modified the rule to deduct an amount proportional to the excess over the addedLOC threshold -- in other words, depleting the IncDev value. A simple approach is linear, deducting some constant times the excess, such as 0.03(excess). So the student run with addedLOC = 21 yields a tiny deduction of just 0.03(21-20) = 0.03(1) = 0.03, while the student with addedLOC = 50 gets deducted 0.03(50-20) = 0.03(30) = 0.9 points -- nearly all of the 1 point possible. If those violations were the first for each student, the first student IncDev value would become 1-0.03 = 0.97, and the second student's 1-0.9 = 0.1.

To achieve the second proportionality goal, we accumulate deductions. A student with just one run having addedLOC = 30 would be deducted a total of 0.03(30-20) = 0.03(10) = 0.3, so IncDev = 0.7. But a student with three such violations would be deducted 0.3 + 0.3 + 0.3 or 0.9, yielding an IncDev of just 0.1.

Other rules are possible that use a non-linear deduction, so that large violations are penalized even more heavily. One possibility is to multiply a constant by the square of the excess. We leave that for future consideration.

3.2 Replenishment

Introducing the above rule alone could demoralize some students. The reason relates to a principle of auto-grading that students can see their current score, and then improve their code and resubmit for a higher score. But a rule that imposes a penalty with no possibility of recovery runs contrary to that principle.

Thus, we sought a way for students to replenish their IncDev value, what we informally call "digging themselves out of a hole." One possibility is to add a policy for replenishing the IncDev value when runs don't violate rules. This deplete/replenish approach should be designed to be fairly intuitive, especially (but not solely) because such depletion/replenishment of a value (like energy level) is common in video games.

As such, along with the IncDev deductions described earlier (depletion), we also considered adding to IncDev for each run that does not violate the excessive added lines rule (replenishment). One possibility is: For each run that doesn't violate the AddedLOC rule, we add a value like 0.1 to the IncDev value.

3.3 The IncDev heuristic for awarding points for incremental development

Summarizing, for a log file and student, starting with IncDev = 1, for each run from first to last, adjust IncDev as follows:

- **AddedLOC rule**: IF (current run's LOC <= prev run's LOC + 20), THEN IncDev -= 0.03(excess). Excess is lines above 20.
- **Replenishment**: IF AddedLOC rule is not violated, IncDev += 0.1.

The above values are configurable. We found the above values after some tuning for our

programming assignments and students, but for other situations, an instructor might raise the AddedLOC threshold from 20 to another value like 30, or might modify the depletion and replenishment constants of 0.03 and 0.1 to other values as well. Future work may examine automatically adjusting such values too.

We had to consider whether to allow IncDev to temporarily go above 1 or below 0. In one approach, which we call Heuristic 1, we limited IncDev to stay between 0 and 1 at all times, because we felt that concept was easy to understand, and consistent with video game concepts like "energy level" that ranges from 100% to 0%, but not negative.

However, as seen in our experiments, that heuristic performed poorly, due to too easily replenishing egregious AddedLOC violations -- with just 10 subsequent runs, any violation no matter how large would be wiped out and IncDev set back to 1.

Thus, in a second heuristic, which we call Heuristic 2, we allowed IncDev to drop below 0 (but still never above 1). If negative at the end, we then map IncDev to 0.

The above heuristics start with IncDev = 1 for each student. However, this would give a point to students who did no runs or almost no runs. Thus, we can introduce further policies, like requiring at least 5 runs to earn any IncDev points, or like requiring at least some test cases to be passed.

4. Experiments

We ran experiments on historical data, namely the final project lab from our past CS1 course offering in Spring 2021, in a 100-student section. Although the incremental development points had not been created yet and thus students were not told anything about earning points, they were told explicitly that all development had to be done inside the zyBook programming window because teachers would be looking not just at their final submission but the entire history of their development, not to assign points but mostly to ensure students were developing their code (versus copy-pasting a solution written externally by someone else). The project was custom, meaning each student submitted a different program. No template code was provided, but students were allowed to copy code from the book or lecture notes, with attribution.

For each lab, the tool required about 10 minutes to download all the source files from the zyLabs servers, and then just a few seconds to run any of the heuristics.

We visually scanned all 100 students' runs looking for a subset of 20 that seemed to cover the spectrum of different series of runs -- some with large AddedLOC jumps, some very incremental, some between. Table 1 shows those 20 students (column "S") and their runs (LOCs in column "LOC per run"), in random order. We then examined each student's runs in more detail, assigning an incremental development grade (column "G" in Table 1) to each student, with an A grade being deserving of full credit, an F no credit, and B, C, D, falling in between. This gave us a sense of whether our heuristics were matching our "teaching intuition".

Heuristic 1 (column "H1") did poorly, giving too many students high scores when the manual grade suggested lower. The problem was because although egregious jumps in code size would

bring the score to 0, just 10 subsequent runs back up near 1. Student 8 is an example, having submitted 205 lines right away -- an outrageous violation of incremental design, but the subsequent student testing for 8 runs brought their IncDev score back to 0.8, nearly full credit for incremental development. Student 11 is another example, with a 102-line jump, but ultimately receiving 0.7 IncDev points.

Heuristic 2 (column "H2"), which allows negatives, does better. Students 8 and 11 get 0 IncDev points, as they should. For more nuanced cases, H2 usually does well too. Student 19, for example, jumps from 51 to 113, but does 9 more runs ending with 118 -- we'd like to see about half the points awarded (a "C"), and indeed H2 yields 0.64 IncDev points. H2 handles "A" grade cases fine, and most B cases too (close to 1 or at 1).

S	G	H1	H2	LOC per run. To save space in this table, ... replaces 10 runs with no rule violations and all LOCs within those on the left and right.
1	B	1	1	60, 60, 60, 60, 59, 58, 58, 54, ..., 55, 55, 55, 55, 55, 55, 56, 56, 56, 56, 68, 68, 67
2	B	1	1	100, 101, 101, 103, 103, 104, 105, 108, 108, 109, 109, 126, 126, 135, 135, 135, 138, 138, 138, 138, 233, ..., 242, 147, ..., 161, 161, 161, 163, 167, 167, 167, 167, 167, 167, 167, 167, 167, 167
3	A	1	1	16, 15, 16, 16, 21, 21, 23, 24, ..., 27, ..., 26, 29, 29, 29, ..., 35, ..., 41, ..., 41, 41, ..., 43, 59, 66, 66, 67, 67, 67, 67, 67, 67, 67, ..., 75, ..., 92, ..., 95, ..., 101, ..., 102, 99 ,...,. 109, ..., 125, 127, ..., 121, ..., 131, ..., 131, ..., 140,..., 140
4	B	1	1	48, 48, 48, 49, 70, ..., 71, ..., 70, 70, 77, 101, ..., 116, ..., 116, 130, ..., 129, 129, 117, 117, ...4, 134, 133, 133, 133, 133, 133, 133, 133, 133, 133, 133, 133, ..., 145, 150, 151, 155, 155, 156, 156, 156, 156, 156, 156, 156
5	D	.2	0	13, 13, 13, 13, 13, 13,..., 16, 16, 17, 17, 17, 17, 17, 28, 28, 28, 54, 124, 154, 154, 156
6	B	1	1	47, 47, ..., 57, 72, 72, 36, ..., 36, 37, 37, 38, 38, ..., 43, 43, 43, 43, 44, 44, 44
7	B	1	0	42, 42, 43, 44, 44, 55, 55, 55, 61, 61, 74, 74, 74, 74, 78, 77, 79, 82, ..., 82, ..., 89, 89, 89, 89, 95, 95, 94, 94, 90, 90, 91, 91, 91, 140, 136, 138,..., 141, ..., 153, 153, 1, 164, 176, 176, ..., 187, 186, 186, 186, ..., 218, 218, 218
8	F	.8	0	205, 205, 205, 205, 205, 205, 205, 205, 205
9	A	1	1	23, 24, 24, ..., 49, 49, 49, 58, ..., 61, 61,..., 61, 61, 62, 63, 63, 58, 58, 58, 58, 58, 58, 63, 71, 71, 80, 80, 80, 86, 86, 86, 86, 86, 86, 86, 86, 113, 85, 85, 85, 85, 86, 86, 86, 84, 86, 88, 88, 88, 88, 92, 93, 97, 97, 97, 97, 97, 97
10	B	1	1	20, 65, 65, 65, 66, 66, 66, 78, 78, 78, ..., 96, 95, 153, ..., 169, 180, 180, 181, 181, 184, 184, 213, ..., 232, ..., 243, 243, 244, 250, 252, ..., 262, ..., 280, 279, 279, 280, 283, 284, 285, 285, 285, 285, 284, 227, 235, 235, 260, 260, 260, 260, 275, 275, 275, ..., 274, 274, 274, 274, 274, 274, 268, 268, 268, 268, 268, 268
11	F	.7	0	19, 19, 121.....136
12	A	1	1	56, ..., 67, ..., 70,..., 82, 82, 86, 96, ..., 99, 99, 130, 130, 130, 130, 130, 131, 131, 131, 131, 132, 133, 133, 133, 133, 133, 133, 135, 135, 135, 135, 137, 144, 156, 165, 165, 171, 171, 171, 182, 195, 185, 185, 189
13	B	0.4	0	28, 37, 37, 37, 37, 39, 39, 39, 37, 42, 43, 41, 41, 41, 58, ..., 65, 65, 64, 65, 65, 65, 67, 68, 68, 73, 70, 73, 73, 73, 72, 74, 64, 72, 72, 63, 63, 63, 62, 62, 47, 48, 47, 47, 49, 49, 47, 47, 48, 99, 48, 97, 97, 97, 98, 98
14	B	.91	.91	34, 34, 34, 36, 36, 40, 40, 37, 37, 37, 37, 35, 36, 36, 36, 36, 36, 36, 34, 34, 34, 36, 36, 74, 76, 76, 76, 78, 78, 78, 78, 77, 77, 80, 80, 79, 78, 80, 80, 79, 79, 81, 81, 81, 81, 124, 124, 125, 127, 130, 141, 141
15	B	1	.98	10, 84, 84, 89, 89, 89, 93, 93, 93, 89, 89, 89, 89, 88, 88, 88, 91, 107
16	D	.2	0	22, 25, 25, 25, 49, 49, 49, 50, 129, 130, 129
17	A	1	1	27, 30, 30, 35, 40, 40, 42, 42, 46, 47, 49, 49, 51, 53, 59, 61, 62, 62, 60, 66, 66, 67, 85, 92, ..., 98, 120, ..., 123, 123, 164, ..., 168, 168, 168, 168, 172, 172, 180, 181, 181, 170, 170, 170

18	D	1	0	114, 114, 114, 114, 114, 116, 117, 118, 118, 118, 116, 116, 116, 115, 115, 126, 127, 129
19	C	.9	.64	21, 23, 42, 43, 43, 40, 45, 45, 51, 113, 113, 113, 113, 118, 118, 118, 118, 118, 118
20	C	.9	.48	82, 82, 82, 84, 84, 84, 84, 85, 85, 83, 107, 107, 115, 115, 139, 140, 140, 150, 156, 157, 211, 211, 211, 211, 211, 211, 211, 212, 212

Table 1: Lines of code (LOC) for each run for 20 students. AddedLOC violations are highlighted. To help evaluate the heuristics, as teachers we manually assigned an incremental development "grade" (column G) of A-F for each student based on our teaching intuition. Heuristic 2 (H2), which allows IncDev to go negative, was closer to those grades than Heuristic 1.

5. Discussion

A few cases stand out for giving much lower scores than we teachers would have liked. Student 7 is an example. Towards the end of their runs, they have this sequence of LOCs: 153, 153, 1, 164, 176, 176. The 1 to 164 jump causes a large depletion in IncDev, ending greatly negative and hence mapped to 0. But as teachers we would surely realize the student was likely trying something out and would not penalize that temporary 1. Upon investigation, it appears the student was making an external copy of their code, because the run with 1 LOC consisted of this comment: "//I'm making a backup of my code just in case". The student likely used cut-paste rather than copy-paste, (perhaps not knowing how to copy but only to cut -- these are freshmen and their skills differ) and put that comment there because they'd been informed we'd be examining their code history. A similar case occurs near the end of student 13, who has this sequence: 48, 99, 48, 97. Upon examination, we found the student added some code, deleted it, and added it back again, but they were double penalized for the jump from 48 up to 98 and 97. Both Student 7 and 13 were penalized too heavily, and thus further refinement to our approach is desired in the future. However, we also note that this was a historical analysis; students in the future could potentially do additional runs to replenish, even with the same heuristic.

That topic brings us to a potential criticism of the approach: That students could violate the incremental design process, but then just do repeated runs to replenish their IncDev score. This is indeed true. However, one thought is that by giving points for incremental development, we are "encouraging" students towards a better process. In the past, we have found that just a small amount of points can influence student behavior, such as giving just a few points for attending lecture leading to nearly everyone attending. Points are one way students interpret what is important, even if just a few points. Another thought is that having to run code repeatedly to replenish points is a bit of a pain, perhaps taking 5-10 minutes or more. A third thought is that our heuristic could be modified to not easily allow such repeated frequent to earn replenishment points, perhaps by slowing the replenishment rate, or even by depleting the IncDev score. And finally, we'd prefer easy replenishment over no IncDev score at all (the current situation) and over student frustration that would occur if they accidentally dig themselves into an inescapable hole (if no replenishment was possible).

Table 1 shows an interesting feature, namely that nearly every student started with more than 20 lines of code. We noted this was sometimes due to needing to get enough code written to meaningfully run something, and sometimes due to students starting from example code copied from the book or lecture notes (which was allowed). The IncDev depletion due to this initial jump in LOC usually didn't hurt the student since subsequent runs would replenish IncDev, but nevertheless we may wish to avoid penalizing the first run so much, perhaps having a higher AddedLOC threshold for the first run.

Our plan is to introduce the incremental development points later in the term, perhaps after 6 or 8 weeks, so as to avoid "stressing out" the students with this extra rule, as they are just starting up and learning, especially since earlier programs tend to be quite small. In the future though, we might consider adding such points earlier in the term as well, perhaps with a different even-simpler heuristic for smaller programs.

Before introducing the heuristic into a live class, we wanted to hear student feedback on the incremental development points idea. We surveyed our summer CS1 students (16 students) at the end of week 3 of 5 weeks total. The survey explained how the incremental development points would work and rationale, then asked some questions. 56% said they understood the intuition and 25% were neutral (19% disagreed). The same ratios said they understood the details. 46% got right questions asking to calculate the IncDev scores for various run LOCs, and another 39% were close (15% were far off). Interestingly, asked if these points were a good thing, 38% agreed, but 25% disagreed (the rest neutral). In the free response question at the survey's end, concerns included "it's somewhat confusing", "you're adding complication for no reason...sometimes it's hard to test code before it outputs anything", and "sometimes I add print statements when testing, or blank lines to clean the code at the end, it will be frustrating if I have to worry about the length of my code". These latter concerns are important, and may point to more advanced techniques if such points indeed will be awarded. However, many students were positive, with comments like "it's good to give more chances to earn points", "I think it's a great tool", "The AddedLOC rule did get me thinking more", and "it's a good practice".

6. Conclusion

We developed a tool as a step towards auto-awarding points for students following a "good" program development process. "Good" is hard to define of course, but with widely-used tools now recording every student run, making attempts in this direction could prove beneficial in the long run to helping students develop good practice, and may reduce their frustration from buggy code stemming from adding large code chunks all at once. As a first attempt, we defined a heuristic to award points for "incremental development", and showed it to perform reasonably on historical data. Looking ahead, we hope to further improve the heuristic to handle more cases while keeping students happy / unstressed, and to consider more development process factors such as starting early, spending sufficient time, testing thoroughly, and more.

7. Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant No. 2111323.

8. References

[CaDe13] Caiza JC, Del Alamo JM. Programming assignments automatic grading: review of tools and implementations. In7th international technology, education and development conference (INTED2013) 2013 Mar (p. 5691).

[DeSr17] DeNero J, Sridhara S, Pérez-Quiñones M, Nayak A, Leong B. Beyond autograding: Advances in student feedback platforms. InProceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education 2017 Mar 8 (pp. 651-652).

[EdLi16] Edwards S, Li Z. Towards progress indicators for measuring student programming effort during solution development. In Proceedings of the 16th Koli Calling International Conference on Computing Education Research 2016 Nov 24 (pp. 31-40).

[GoLy21] Gordon CL, Lysecky R, Vahid F. The Rise of Program Auto-grading in Introductory CS Courses: A Case Study of zyLabs. In2021 ASEE Virtual Annual Conference Content Access 2021 Jul 26.

[KaEd17] Kazerouni AM, Edwards SH, Shaffer CA. Quantifying incremental development practices and their relationship to procrastination. 2017 ACM Conference on International Computing Education Research 2017 Aug 14 (pp. 191-199).

[UlLa18] Ullah Z, Lajis A, Jamjoom M, Altalhi A, Al‐Ghamdi A, Saleem F. The effect of automatic assessment on novice programming: Strengths and limitations of existing systems. Computer Applications in Engineering Education. 2018 Nov;26(6):2328-41.

[Zy1] zyLabs. zybooks.com, 2021.