2021 ASEE ANNUAL CONFERENCE
Virtual Meeting | July 26–29, 2021 | Pacific Daylight Time
Paper ID #33288

Coding Trails: Concise Representations of Student Behavior on Programming Tasks

Prof. Frank Vahid, zyBooks; University of California, Riverside

Frank Vahid is a professor of Computer Science and Engineering at the Univ. of California, Riverside, and co-founder and chief learning officer of zyBooks. His research interests include CS/engineering education, and embedded systems.

Prof. Roman Lysecky, University of Arizona; zyBooks

Roman Lysecky is VP of Content at zyBooks, A Wiley Brand and a Professor of Electrical and Computer Engineering at the University of Arizona. He received his Ph.D. in Computer Science from the University of California, Riverside in 2005. His research focuses on embedded systems, cybersecurity, and STEM education. He has authored more than 100 research publications, received nine Best Paper Awards, is an inventor on multiple patents, and received multiple awards for Excellence at the Student Interface.

Dr. Bailey Alan Miller, University of California, Riverside

Bailey Miller is the Director of Engineering at zyBooks, a part of John Wiley and Sons. He formerly worked as a software engineer at Space Exploration Technologies Corporation (SpaceX). He received his B.S. in Computer Engineering, and his M.S and Ph.D. in Computer Science, from the University of California, Riverside in 2009, 2011, and 2014, respectively.

Lyssa Vanderbeek, zyBooks

Lyssa Vanderbeek is Sr Director of Product Management at John Wiley, and Sons. Lyssa received her BA in Economics from Whitman College and her MBA from the Haas School of Business, University of California at Berkeley. She has spent over 20 years developing technology products for higher education. After joining Aplia, Inc as an early employee, Lyssa led Product Mangement at Cengage Learning, Macmillan Learning, Inkling, and EduNav.

©American Society for Engineering Education, 2021

Coding Trails: Concise Representations of Student Behavior on Programming Task

ABSTRACT

CS instructors desire visibility into student programming behavior, such as seeing the days a student worked, the time spent, and the number of compiles/runs. Such visibility may help find struggling students, prevent or detect cheating, and provide insight into the effect of new policies like points for earlier starts. Such visibility historically has been severely limited due to student use of external tools. Today, many education-focused program auto-graders provide a cloud-based development environment that records much student behavior. Detailed logs are cumbersome to view, especially for large classes, but conversely, summary statistics like averages and standard deviations lose much useful information. This paper introduces the concept of a "coding trail" as an attempt to visually and concisely summarize a student's coding behavior on a programming assignment. Our visual coding trail displays dates, each develop run, each submit run for auto-grading and score, and dramatic changes in code (often a sign of cheating). The coding trail is textual rather than graphical, allowing easy copy-paste, incorporation into spreadsheet gradebooks, and parsing by tools for further analysis, at the expense of some information loss. A version of our coding trail has been implemented in the zyBooks program auto-grader and appeared for over 2,000 courses and 130,000 students in 2020, with numbers growing. This paper introduces the coding trail, discusses various tradeoffs in its design, and points to a variety of uses.

1 Introduction

Teachers of programming courses have long wanted visibility into their students' programming behavior, such as what dates and times students programmed, how much time they spent, how often they compiled/ran their programs, how much code they wrote between compiles/runs, and so on.

However, in the past, most student programming was done in environments like Eclipse, Visual Studio, or command line tools, that didn't log such activity or make such logs readily available to teachers. Some education-focused environments evolved that logged development behavior, like BlueJ for Java, allowing research into student behavior such as [Jadud05][Jadud06]. Some teachers had students use version control software like Git to get some insight into student behavior [Conner19], modifying Eclipse to integrate with Git [Yan19], or integrating with the Web-CAT auto-grader [Kazerouni17]. The Runestone project logs Python development activity, allowing for research such as [Yeckeh19].

In recent years, cloud-based program auto-grader usage has grown tremendously. Hundreds of universities have switched from manual to auto-grading, reaching hundreds of thousands of students [Gordon21]. Auto-graders can create a log of every submission students make for auto-grading, including date, time, the submitted program, and more. In fact, some such auto-graders provide a development environment too, so potentially can log all compiles/runs also, as depicted in Figure 1.

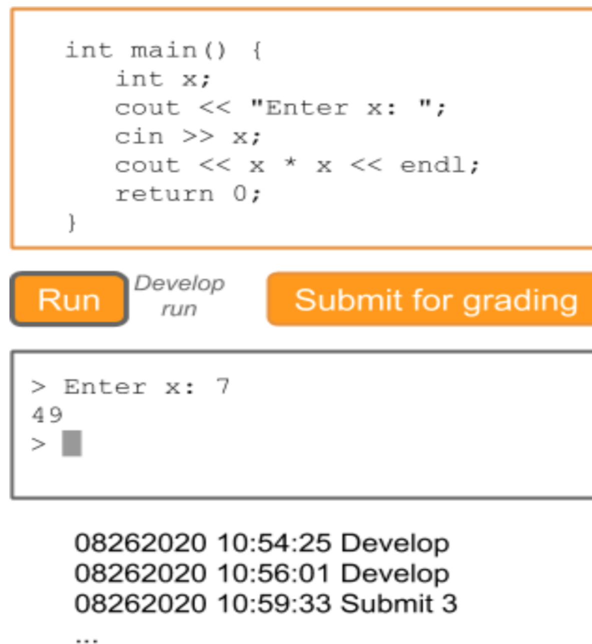

Figure 1: Modern auto-graders may have a development environment and a submit option. Activity may be recorded, including timestamps, scores, and even each run's code itself.

While detailed logs enable researchers to perform analyses, teachers need a way to quickly "see" the behavior of their class or of an individual student. We thus developed a compact visual representation of a student's programming behavior that we call a coding trail. We also want that coding trail to be conspicuously present for students, so they know effort matters, and not the final submission. While researchers have used recorded activity to measure student effort [Edwards16][Li18][Romero20][Toll16], we are not aware of similar attempts to visually represent programming effort.

This paper describes goals of such a visual representation, tradeoffs we considered, our current coding trail, potential uses of such a coding trail, and future work. The coding trail feature has been enabled in the zyBooks program auto-grader used by over 2,000 courses and 130,000 students in 2020.

2 Goals and approach

2.1 Goals

Detailed textual logs are hard for teachers to process. We desired a concise representation that would allow teachers to quickly understand the programming behavior of their class and/or of a particular student. Such a representation ideally would:

- show dates and times worked
- show development runs, when the student tests their own program
- show submission runs for auto-grading, including scores received
- show code size as the student develops and submits
- be visual for quick processing
- be copy-pastable into various documents like emails, spreadsheets, or gradebooks in learning management systems

- be machine readable for further processing
- show anomalous behavior like dramatic code changes between runs or coding style that departs from the class's style

If such a representation were available for a given programming assignment, a teacher for example could quickly see that:

- their class is starting early, or instead is starting too late, in which case a teacher might encourage earlier starts informally or via points, extra credit, etc.
- some students are submitting perfect code with almost no effort, suggesting some students might be copying code from classmates or the web.
- a particular student in office hours has genuinely been working hard, or instead that they have made minimal effort.
- most students are struggling on an assignment, or an assignment was easier than planned.

Furthermore, making such a representation conspicuously visible to students might reduce their temptation to cheat by submitting a copied solution, since students would readily see that a teacher can see not just their final submission but also their effort leading to that submission.

The common approach to concisely represent detailed logs is a statistical summary, which has some benefits. But, statistics hide much information, and providing more detailed statistics can overwhelm the viewer. Furthermore, students may be less likely to notice numerical statistics.

Instead, we desired a visual representation, based on the fact that human brains are optimized to quickly process visual information. Such quick processing is important not only to meet the needs of busy teachers, but also so the representation can be shown to students in a simple comprehensible impactful manner, without cluttering their programming view.

2.2 Approach

Throughout 2019 and 2020, we developed and demonstrated our coding trails to over 50 CS teachers across the U.S., at community colleges, 4-year teaching schools, and 4-year research schools. Using feedback from teachers, we iteratively refined the coding trails with the aim to serve needs.

Teachers indicated much interest in coding trails, with the top interest being coding trails' potential to prevent cheating. Other commonly stated interests were to help teachers notice struggling students so as to proactively intervene, and to quickly see if a particular student was putting in sufficient effort.

3 Attempts and tradeoffs

Although we also investigated and developed graphical representations, such as bar charts on a calendar view showing dates/times and durations spent programming, for this work we wanted to only consider textual representations. The main reason is that text can be pasted directly into most gradebooks, whether in a spreadsheet or in a learning management system (LMS) like Canvas or Blackboard. As such, coding trails can be treated just like scores and statistical data, occupying another column of a

gradebook. And, a textual representation can be parsed by scripts for additional analysis. The main trade off is losing the graphical benefit of showing time simply as a bar with width representing time spent.

3.1 Basic develop and submit runs

As an initial attempt, we tried a compact version of the log file for a given student:

```
dev dev dev dev sub(0) dev dev dev sub(2) sub(5) dev dev sub(10)
```

While enabling a quicker view than a detailed log, we realized the short words weren't needed. Instead, we could use single letters, and eliminate spaces. We used d for each develop run, and s for each submit run followed by the score on that run. The student below did 4 develop runs, then submitted for a score of 0, then 3 more develop runs, then submitted for 2, submitted again for 5, then 2 more develop runs, and finally submitted for a score of 10 (full credit on this assignment).

```
dddds0ddds2s5dds10
```

We found the d's made the submissions and scores, which are important as they indicate correctness, hard to see. We tried using symbols instead of d's, such as asterisks:

```
****s0***s2s5**s10
```

Those were better, but we decided a symbol with less visual clutter was needed, and chose a dash symbol -, which let the submission scores stand out:

```
----s0---s2s4--s10
```

We found the s's distracting, and chose a symbol too. We chose one with minimal visual clutter, in this case a | symbol:

```
----|0---|2|4--|10
```

We eventually eliminated the | unless needed to separate two adjacent submission scores, yielding:

```
----0---2|4--10
```

3.2 Dates and times

Teachers not only want to know how often students run code (develops and submits) and scores, but also to know when students are working and for how long. So we investigated adding dates/times and durations to the coding trail. For dates, we started by showing each date followed by that day's runs:

```
7/13 ----|0  7/14 ---|2|4--|10
```

Feedback from teachers was they thought of weekdays rather than dates. For example, a teacher might every week release assignments on Monday and have them due on Sunday. Thus, we modified the coding trail to only show the start date, and then used standard day-of-the-week abbreviations (M T W R F S U). We also colored those letters to make days stand out even more; the color is optional and may not persist if copy-pasted into places like a gradebook, which is OK. The student below started on Monday 7/13, then worked again on Saturday and finished.

```
7/13 M----|0  S---|2|4--|10
```

In rare cases where the work extends beyond 7 days, we could insert a date again.

4 Examples

Figure 2 shows 10 real coding trails from a programming assignment of a class taught by one of the authors. The assignment asked students to write a function and main program to determine whether an input vector was sorted in ascending order. The instructor's solution was 40 lines. The coding trails are sorted by date, and the assignment was released on Tue 5/12 (2020), due Tue 5/19. We easily see that:

Mia	5/13 W----10
Sha	5/14 R------------0|4|4|4---4|6 U0|0|8
Pi	5/16 S----8|8|8|10
Le	5/17 U0|4|10|10
Sam	5/17 U--4|8|10
Pat	5/17 U-------4|6 M--8-- T-8--8
Al	5/18 M--6|8-10
Mo	5/19 T---6-------------------4--10
Jo	5/19 T0---2|0|0|0|2|2----2|0|0|0|2|2|2-0|8-
Jun	5/19 T10

Figure 2: Selection of real student coding trails for a particular program.

- Mia started early (Wednesday), developed 4 times, and submitted for a perfect 10.
- Sha started Thursday and worked hard but struggled a bit, getting stuck at 6 points, then coming back Sunday and reaching 8 points, still short of 10.
- The next few students look solid.
- Pat looks to be struggling, working Sunday, Monday, and Tuesday and getting stuck at 8.
- Mo started on the due date, but worked hard and eventually reached 10.
- Jo also started on the due date, and seems to have struggled, also reaching but stuck at 8.
- Jun started on the due date and somehow submitted a perfect program in one try.

The key to note is how quickly a teacher can get a sense of how students are doing on the programming assignment – a much better sense than numerical statistics, including days started and worked, and effort. The coding trail lacks some info, like daily time spent, but still yields good insights.

After examining those coding trails, an instructor might:

- Feel satisfied that most students are trying.
- Encourage students to attempt the programs sooner.
- Find out why some students are getting stuck at 8 points. (We found it was due to not correctly handling the test case where all items had the same value).
- Investigate Jun's code further. (We determined Jun had cheated by outsourcing the work to a contract programmer – undetectable by a similar checker).

5 Extensions

5.1 Dramatic change indicator

The coding trail above worked well, with teachers able to quickly understand the coding trails, and indicating they felt a good sense of a student's efforts. Coding trails don't replace statistics like average scores, develop runs, and submit runs, but complement those.

Looking at coding trails, we noticed a pattern where a student struggled, then suddenly got a perfect score. Upon investigating the submissions, sometimes the perfect score was due to finally finding and fixing a bug. But at other times, the sudden perfect score was due to the student resorting to cheating by replacing their own solution with a copied solution. This is precisely the scenario that teachers in Harvard's CS50 class point out: "All too often were students' acts the result of late-night panic, a combination of little sleep, much stress, and one or more deadlines looming" [Malan20]. Teachers could easily see the difference when looking at the source code because the student's program dramatically changed soon before the perfect score.

Thus, we add a "dramatic change" indicator to the coding trail -- the student's code was evolving normally with small changes between runs, and then suddenly the code was dramatically different. One heuristic measures the number of characters that differ after running a standard diff algorithm, but modified to reduce false alarms that might occur if a student simply wrote a lot of code between runs -- requiring a minimum base lines of code, requiring that many lines were deleted in addition to being added, etc. More powerful heuristics can be developed over time, such as detecting differences in coding style too. Whatever heuristic is used, we decided to indicate a dramatic change by inserting a % into the coding trail. Because dramatic changes are rare, we also add a number between 0-100 indicating the amount of change, and make the font red, without fear of obfuscating the rest of the coding trail. Again, the color is optional and may not persist in gradebooks or other places. Below, student B4's coding trail now shows a dramatic change on Saturday at a level of 98 of 100, meaning nearly all the code was changed – perhaps warranting investigation by a teacher.

```
7/17 F-----0--0--0  S%98|10
```

5.2 Time-of-day, time spent, and code size

Teachers would also like to know what times of the day students programmed, to know for example if students were working very late hours. However, we could not find a way to add times that didn't dominate the coding trail and obfuscate the develop and submit info. And, students commonly took

breaks, meaning dozens of time indications (like 8:05pm) would be scattered throughout. We thus abandoned efforts to show when students worked, at least for now.

Total time spent is also something teachers would like to see. We tried putting time spent at the end of each day, plus a total at the end of the coding trail. We found though that teachers were most interested in the total time; the detailed breakdown was interesting but not so important as to clutter the coding trail. Thus, we plan to put the time info at the end of the coding trail, like below where the student spent a total of 42 minutes:

```
7/13 M----0  S---2|4--10 42min
```

Likewise, teachers indicated wanting to know code size each day. Like time-of-day and time spent, such info can quickly clutter the coding trail. Thus, we plan to just put the code size at the end as well, as in:

```
7/13 M----0  S---2|4--10 42m 205L
```

Future work may investigate ways to include time spent and code size throughout the coding trail as well.

We note that coding trails are intended to be concise and visual, thus restricting how much info we can include. However, future work may include a way for teachers to toggle to a more detailed coding trail. And we note coding trails are best complemented by a detailed analytics view; coding trails do not replace such detailed analytics.

Table 1 restates our goals and indicates whether our coding trail satisfies those goals, using a scale of 0-2, with 2 best, based on our subjective analysis.

Goal	Satisfied?	Notes
Dates/times worked	1	Only dates/days, no times
Develop runs	2	Uses dash -
Submit runs	2	Uses score like 6, with | separator like 6|8
Code size	1	Only at end; throughout would be better
Time spent	0.5	Only at end; throughout would be better
Visual	1.5	Text takes some getting used to; doesn't depict time
Copy-pastable	2	Pure text. Even wraps
Machine-readable	2	Pure text, well-formed syntax

Anomalous behavior	1	Shows dramatic change. Doesn’t yet show style departures or other red flags

Table 1: Analysis of coding trail format’s achievement of goals.

The coding trail yielded an unexpected benefit regarding navigation. We found instructors trying to click on the scores or dashes, hoping to view the student's code for that run. Thus, on our webpage showing a student's coding trail, we made every dash and score a link that brings up that run's code for the instructor.

In the future, we hope to develop algorithms that bring “attention-worthy” coding trails to a teacher’s attention. Potential cheating might be given a particular score and highlighted in shades of red. Potential struggle might also be given a score and highlighted in yellow. Ultra-low-effort coding trails might be scores and highlighted. And so on.

6 Applications, concerns, and early results

This paper introduces coding trails as a tool that could yield a variety of benefits to teachers. The paper does not itself conduct research on those benefits, but we encourage future work that does so. However, here we point to various applications that CS teachers have mentioned in response to seeing our coding trails, and some of our class experiences over the past year.

6.1 Applications

Perhaps the most exciting application of coding trails is to prevent cheating. The theory is based on crime/fraud/cheating prevention where three things lead to cheating: opportunity, pressure, and rationalization. Opportunity includes not just the ability to cheat but the belief it will not be detected. For example, stores commonly have customer-facing video monitors at their entrances, to let potential shoplifters know they are more likely to be caught, thus decreasing perceived opportunity. Coding trails serve a similar role; students who are considering cheating (due to pressure, and who have rationalized such cheating) might see decreased opportunity when seeing coding trails and realizing that copy-pasting a solution, even if unique from classmate solutions, may be detected by teachers via little effort or dramatic change indicators. Personally, our goal is less about catching and punishing cheating, and more about reducing the temptation in students, so they instead do the work required to learn. We look forward to future work by others, and plan future work ourselves, to see the impact coding trails have on reducing cheating.

Another application is to quickly see how a particular student is doing, such as when the student comes to office hours. With just a glance, a teacher can quickly see if the student has been starting early each week and trying hard, or instead has been starting late, doing few develop runs, struggling with repeated submissions with low scores, etc. In fact, we have begun doing so in our own classes -- not just for students who come to office hours for help, but in various cases where we want to know how a particular student is doing, such as if we need to decide whether to give a student an extension or whether to give them a grade bump if they are near a cutoff (students who are putting in effort are more likely to receive such breaks when such decisions must be made).

Other applications include quickly gleaning information about how a class is doing, as was done in the earlier example. A related application is to quickly see how a particular programming assignment was -- did it require a lot of work, or was it very easy? Coding trails allow a teacher to quickly see which students are struggling -- in fact, we have pulled up our class' coding trails during class with student names not shown, and then in a fun and supportive way we've discussed various patterns. Students often say "I think that one is mine!" and invite us to open up their code so the class can see where they struggled. The coding trails provide a nice entry point into discussing student code, more so than just a list of statistics per student.

6.2 Concern: Do coding trails stress students?

Upon presenting coding trails to dozens of CS teachers over the past year, common questions included:

- Will students be stressed by seeing they are being watched, like "big brother"?
- Will students even notice coding trails?

Thus, although this paper's purpose is mainly to present the idea of coding trails as a basis for further usage, experimentation, and research, we also surveyed students regarding the above. We surveyed students in a summer CS1 class with 40 students, in which the instructor had not said anything to the students about the coding trails during the class term, and the survey was given in the last week of the term during online lecture time (meaning nearly all students participated).

The survey first pointed students to coding trails so they knew what we were referring to; coding trails had conspicuously been present under their Run/Submit buttons for their entire term. Then, to address the first question above, the survey's first item was: "Please share your thoughts on coding trails." With this open-ended item, we could detect a positive or negative perspective without biasing the students via an agreement question. We examined all 40 responses and found nothing remotely suggesting that coding trails caused stress or any negative reactions. Some comments were neutral, and the only negative ones related to students not fully understanding the format. Instead, most comments were positive, as shown below (comments were copy-pasted verbatim, thus they include the students' typos and English errors).

- It's an interesting way to keep track of student progress
- I think its pretty cool
- I liked it because it showed my progression and gave me a timeline of my efforts
- I like seeing the number go up from, say 20, all the way up to 100.
- It's a good tool to prevent cheating
- I feel like they're just there. Nothing to worry about unless the programmer is cheating.
- The coding trails show me my progress so far in the program and decrease my temptation to cheat.
- They are helpful to see my progress

We then asked several specific questions, summarized in Table 2.

Prior to today, did you notice the coding trails in your lab activities?	90% yes
If you noticed the coding trails, did you understand the format?	63% yes
Did you pay attention to coding trails when you wrote your programs?	15% frequently, 43% sometimes, 27% rarely, 15% never
The existence of coding trails decreased my temptation to cheat (i.e., copying someone else's solution)	45% agreed
I think the existence of coding trails decreased my classmates' temptation to cheat	64% agreed

Table 2: Results of student survey on coding trails, with no mention by instructor of coding trails during the term.

Furthermore, we enabled coding trails in 68 classes across the U.S. in Spring 2020, totaling 3578 students. We looked for any evidence of unhappiness or dissatisfaction. While we received hundreds of pieces of unsolicited feedback from those students, no student or teacher complained about coding trails. Furthermore, we surveyed those students and teachers at the end of term, and in open-ended questions asking for feedback, none said anything negative about coding trails, and we found no evidence that those classes had any less positive of an experience.

In short, we have yet to find even any evidence that coding trails stress students or negatively impact students, and instead that students generally found coding trails to be "cool" or "helpful". And, even without any mention or use of coding trails by the instructor, a majority of students felt coding trails reduced their class' temptation to cheat; we expect numbers will be higher when instructors introduce coding trails to students and use coding trails more directly in classes.

6.3 Time spent

Noting again that this paper's purpose is primarily to introduce the concept of coding trails that have been enabled for several hundred thousand students per year, and not to conduct a specific study around coding trails (as the former is a prerequisite to the latter), nevertheless we wish to share some early data that suggests the usefulness of coding trails. For the spring of 2020, we enabled coding trails in our CS1 class section of 100 students (mostly non-CS majors), and actively introduced coding trails to the students in Week 3 of the quarter. We showed students a coding trail in a lab activity and explained the format. We then used coding trails during class time about once a week to see how the class had done on a particular programming task, using coding trails as a springboard to dive into a particular student's code in front of the class -- always in a positive supportive way, and always with the student's permission. Students enjoyed this part of the class, and many asked us to bring up their coding trail and code for discussion. We did not suggest to students that effort, as indicated by coding trails, would be part of their grade. We then compared the time spent (as measured by the auto-grader system) on four

specific programming assignments from latter weeks in the term, that had also been assigned the previous term without coding trails enabled. In both terms, students were instructed to do all development in the auto-grading system, and no other IDE was introduced or mentioned to the students, who were mostly non-CS majors. The average times (in minutes) are shown in Table 3.

Lab Activity	Fall 2019 (no coding trails)	Spring 2020 (coding trails enabled, discussed, utilized)
Week 8: Password modifier (replace certain characters of input word by symbols)	9	14
Week 8: Word frequencies (read text, output number of times each word appears)	14	33
Week 9: Sort a vector (read a list of numbers into a vector, then sort and output)	10	23
Week 9: Contact list (maintain a list of names and phone numbers)	9	21

Table 3: Time spent on identical programming assignments, without coding trails (Fall 2019), and with coding trails discussed with students and utilized during class time (Spring 2020).

One can see that the time spent went up substantially. The data suggests that the presence of coding trails may have encouraged students to expend more effort, to copy less from classmates or contract programmers, and/or to not use external environments. Even the latter alone would be a useful impact of coding trails, so that teachers can see student efforts for the current class or for future research analysis.

7 Conclusions

We described our efforts over the past year to develop a concise textual visual coding trail of a student's coding efforts. The coding trail can be viewed by teachers to provide quick insights into their class or a particular student, as a way of navigating to view code for any particular run in a student's effort history, to detect some forms of cheating (especially relevant since similarity checking doesn't catch today's increasing use of contract programming), to potentially reduce student temptation to cheat, and to provide a fun in-class way to view student efforts and dive in deeper with the class.

Most teachers we've shown are strongly enthusiastic. Some had concerns that seeing coding trails would stress students, but our surveys, and our spring 2020 enabling in 68 classes, showed no such negative indications. In contrast, students tended to think the coding trails were "cool" or "helpful" and themselves indicated coding trails may reduce cheating (which is one goal of coding trails, but not the only one). Thus, it seems coding trails can be safely introduced as a standard feature, that teachers can

begin utilizing them to gain more insight into their classes, and that researchers can experiment with using coding trails to reduce cheating, detect strugglers, or for other goals. In fact, instructors were so drawn to the coding trails as a means to quickly see a student's work, that we added a navigation feature -- instructors can click on any symbol in the coding trail to be taken directly to that run's code. We enabled coding trails in all classes using our cloud-based system since Fall term 2020, so that coding trails are present for over 2,000 courses and 150,000 students in 2021. We will continue to experiment with how to best utilize coding trails in class and to conduct research on its effectiveness in reducing the temptation to cheat as well.

REFERENCES

[Conner19] Conner, D.C., M. McCarthy, L. Lambert. Integrating Git into CS1/2. Journal of Computing Sciences in Colleges. October 2019.

[Edwards16] Edwards, Stephen, and Zhiyi Li. "Towards progress indicators for measuring student programming effort during solution development." Proceedings of the 16th Koli Calling International Conference on Computing Education Research. 2016.

[Gordon21] Gordon, C., R. Lysecky, and F. Vahid. The rise of the zyLab program auto-grader in introductory CS courses. Accessed March 2021. http://zybooks.com/research/whitepaper_the_rise_of_the_zyLab_program_auto-grader_in_introductory_CS_courses.

[Jadud05] Jadud, M.C. A First Look at Novice Compilation Behaviour Using BlueJ. Computer Science Education, 2005, 15:1, 25-40.

[Jadud06] Jadud, M.C. An Exploration of Novice Compilation Behaviour in BlueJ. Doctor of Philosophy (PhD) thesis, 2006, University of Kent.

[Kazerouni17] Kazerouni, A.M., S.H. Edwards, T.S. Hall, and C.A. Shaffer. DevEventTracker: Tracking development events to assess incremental development and procrastination. In ACM Conference on Innovation and Technology in Computer Science Education (ITiCSE), 2017, pp. 104-109.

[Li18] Li, Zhiyi, and Stephen Edwards. "Applying Recent-Performance Factors Analysis to Explore Student Effort Invested in Programming Assignments." Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS). 2018.

[Malan20] Malan, David J., Brian Yu, and Doug Lloyd. "Teaching academic honesty in CS50." Proceedings of the 51st ACM Technical Symposium on Computer Science Education. 2020.

[Romero20] Romero, Cristobal, and Sebastian Ventura. "Educational data mining and learning analytics: An updated survey." Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 10.3 (2020): e1355.

[Toll16] Toll, Daniel. Measuring Programming Assignment Effort. Diss. Faculty of Technology, Linnaeus University, 2016.

[Yan18] Yan, Lisa, N. McKeown, M. Sahami, C. Piech. TMOSS: Using intermediate assignment work to understand excessive collaboration in large classes. ACM technical symposium on computer science education (SIGCSE), 2018.

[Yeckeh19] YeckehZaare, I. and P. Resnick. Speed and Studying: Gendered Pathways to Success. Proceedings of the 50th ACM Technical Symposium on Computer Science Education (SIGCSE), February 2019, pp. 693–698.