Data-Driven Approaches for Exploring the Effects of Teacher Instruction on Student Programming Behaviors

Nicholas Lytle
NC State University
nalytle@ncsu.edu

Veronica Catete
NC State University
vmcatete@ncsu.edu

Yihuan Dong
NC State University
ydong2@ncsu.edu

ABSTRACT

As programming activities become integrated into K-12 classrooms, teachers with minimal prior programming experience will instruct students of varying backgrounds. This usually takes the form of Live Coding, programming on a display in front of students while they follow along. While effective, Live Coding prohibits teachers from seeing students current progress. Though researchers can observe instruction and can tell teachers when students need the pace altered, this help will not always be available. To improve and scale insights and investigate how teacher instruction influences student programming behavior, we perform several analyses to visualize the influence of instruction on student assignment progression throughout the class period. We present case studies featuring teacher and student programming behaviors. We end on possible means our methodology can be extended to create dashboard and other data-driven tools.

Keywords

block-based programming, instructor influence,Live Coding

1. INTRODUCTION

[1]To address inequities in access to computing education [17], teachers and researchers have begun partnerships to put computing activities in required courses. Teachers new to programming exhibit levels of pedagogical discomfort with instructing programming assignments. Our prior work has demonstrated that teachers new to computing curricula often adopt instructional styles for lessons that are more aligned with their traditional teaching methods, which for middle school grades is often direct instruction[3]. Direct instruction is often done in the form of 'Live-Coding', showing the process of developing the final code. This process leaves teachers without a sense of how the class is doing, whether or not they are following along, or whether or not what they as a teacher are programming is correct. Though observations provide teachers with information on how students are following along with the lesson, this methodology is not scalable and we seek more quantitative data-driven methodologies to investigate student programming behavior. To understand the effects of instructional practices, we examine student and teacher coding artifacts, utilizing the logged interactions of the coding environment. We are interested in *ways to visualize how students follow along with teacher instruction.* We perform exploratory analyses to identify patterns in student behavior concerning completion timing, paths, and rates. Using visualizations, we present case studies of teachers instructional style grounded through classroom observations. We believe these data-driven approaches could lead to both scaleable offline and online analysis of programming instruction, informing the development of intelligent dashboards designed to aid both novice and expert programming teachers during live coding instruction.

2. BACKGROUND

2.1 Computing in STEM Education

If children get experience with programming, it is usually within elective computing classes or outside of school activities [17]. To reach *all* students, computing must be integrated into required K-12 courses. In this age-range, curricula that use block-based programming languages are often employed [8]. Block-based programming languages such as Snap [9] are increasingly popular and have been shown to scaffold novices into learning programming in STEM contexts better than traditional text-based languages [22]. While block-based environments can support novice students, teachers must be trained in computing topics and integrated computing lessons must be developed for their classrooms. These are usually both accomplished during teacher professional development seminars [12], though this might not be accessible for all teachers. Another avenue like our work[3] is design-based implementation research [6], where we collaborate with in-service teachers to iteratively build and refine a computing-integrated activity, actively focusing on professional development and teacher comfort. These iterative designs have made us choose to create lessons where teachers instruct through live coding.

2.2 Instructing Programming in K-12

Teacher's self-efficacy and discomfort are important factors when trying to get core subject teachers to add computing into their classes. Our prior work has demonstrated the success of lessons often hinge on a teachers comfort with instructing, and often their self-efficacy is very low as a novice programmer[15, 3]. Frykholms research backs this by suggesting a teachers self-efficacy filter in combination with tolerance for discomfort will affect their willingness to adopt new materials and curricula [7] such as computing. To mitigate discomfort, teachers tend to employ more instructional-based practices such as focused or guided instruction that contrasts the constructionist and open-ended practices loudly promoted in computing [13]. However, at the middle school level, evidence from controlled studies demonstrate that direct, strong instructional guidance proves more efficient in terms of long-term learning than constructivist-based, minimal guidance [13]. This is further supported by Sweller's cognitive load theory which suggests free exploration of a highly complex learning environment may generate a heavy working memory load that is actually detrimental to learning [21]. These prior general findings apply directly to programming instruction. A study by Lin and Dalbey [14] shows medium and lower performing students do better with explicit instruction which has no negative impacts on high performing students. Husic further argues that beginners lack a repertoire of useful approaches to thinking about and learning programming. To build this, teachers of introductory classes need to be specific when providing integrated information, problem-solving support, and feedback [10]. Direct instruction reduces the working memory load of the students and lets them concentrate on the tasks at hand [14]. Thus, for introductory students, early in the learning curve, teachers should provide a supportive scaffolding to help students carry out a task. Accordingly, when teachers provide scaffolding, they generally carry out parts of the overall task that students cannot yet manage.

This research suggests K-12 teachers might benefit from adopting the pedagogical technique of Live Coding. Live Coding, often found in university programming courses, has an instructor start from scratch (or starter code) and build a program related to the material in front of the students[2]. Unlike code snippits, displaying the construction of code gives students insight into the programming process [2] and can usually be done in a manner that allows students to follow along creating an active learning context. Studies have demonstrated the benefits of live coding for instructing in block-based programming languages [20]. Recent work has focused on intelligent dashboard support for Live Coding [4], though Chen et al. focus on making the process easier rather than analyzing the dynamics between instructor and students. It should be noted that nowhere in definitions for live coding does it specify that students follow along. However, given the research described above, we feel that active, follow-along participation driven by instructors will provide the best platform for students to learn in these contexts. Additionally, those who have some prior experience will be able to independently move on without the instructor, while novices can follow along intently step by step. Given this instructional technique and the classrooms that have already participated in this style, we seek to use additional advances in programming trace data analytics to understand classroom dynamics during live coding sessions.

2.3 Programming Trace Data and Analytics

There are dozens of systems that analyze and collect student programming data (an introductory review can be found here: [11]). In programming data-mining, The sequence of all interactions within the environment is called a *trace*, which comprises a list of all *states* a student environment is in and the interactions that connect them. An important subset of this trace is all interactions that result in differing code (e.g. adding, deleting, and relabeling code), called a *code-trace* and all the different *code-states* (i.e. unique code) that a user progresses through on the way to complete a problem. This code trace is used for intelligent programming environments as input for the creation of data-driven next-step hints [19] as well as other instructional support like worked examples [23]. While useful for data-driven algorithms, the size of the space of all possible code-states is in practice incredibly large even for a small programming problem [24]. As such, methods have been developed to collapse the state-space of programming problems to something more manageable for analysis. Prior work has represented the varying pathways of student attempts using a *feature-state* representation [24, 16]. In this manner, an assignment can be seen as a set of *features*, and a student's current *state* in the assignment can be represented by which features the student has present or absent in their code. Rui et al [24] demonstrated techniques to visualize student pathways through a programming assignment by representing which features students progressively added to to their code until reaching a final solution. Lytle et al extended this methodology to be useful in informing curricula design by looking at common pitfalls in student programming pathways [16]. Additional effort has been placed recently in using student feature completion information to help the *instructor* during classroom implementation [5]. Dashboards have been developed that use programming trace or compilation information to aid instructors in finding students in need of help[5]. Diana et al's dashboard system is able to use features of code-trace data to display to an instructor which students are lagging behind in the lesson (having fewer features complete) in order to aid teachers in selecting which students to help or group together to peer-help [5]. While useful, this requires that the dashboard be visible to the instructor at all times which may take away from the time spent live coding. We wish to extend these methodologies in order to develop ways to provide post-hoc analysis of a classroom implementation as well as in the future, provide unintrusive immediate support to instructors live coding in K-12 environments.

3. METHODS

3.1 Data Collection

The dataset used in this analysis came from several implementations of a 4-day computing-integrated science lesson. Nine instructors across three schools led 19 middle-grade classes in a 'Food Webs' unit using a block-based programming environment, Cellular [1] (example code is shown in figure 2). Cellular is a block-based language to scaffold novices learning to code, and uses an agent-based environment to aid in the modelling of scientific phenomena. Approximately 480 students participated in the integrated activity, however, we are only analyzing trace log data from 287 consenting students with complete participation over the four days. Classroom observations of each initiative were conducted by researchers who sat in each classroom watching the teacher instruct and aiding them as requested if an issue arose.

Table 1: Participants by instructor (* = researcher)

Instructor	School	Sections	Usable N
Teacher A*	I	1	10
Teacher B	I	1	9
Teacher C*	II	2	27
Teacher D	II	4	72
Teacher E	II	3	61
Teacher F*	II	1	11
Teacher G	III	1	16
Teacher H	II	5	62
Teacher J	III	1	19
Total	I-2, II-15, III-2	19	287

A breakdown of participating instructors and schools is available in Table 1. Demographic data for each of the three schools is provided in Table 2 below. Data for each of the science classes mirrors the overall school demographics. Instructors were all trained in the Food Webs course materials before teaching their class. In schools I and II, Research instructors (Teacher A, C, and F) led the first class period each morning. Teacher instructors led the remaining class periods. There were a total of 6 researcher-led (or Non In-Service instructors) class sessions and 13 teacher-led (In-Service Instructors) class sessions.

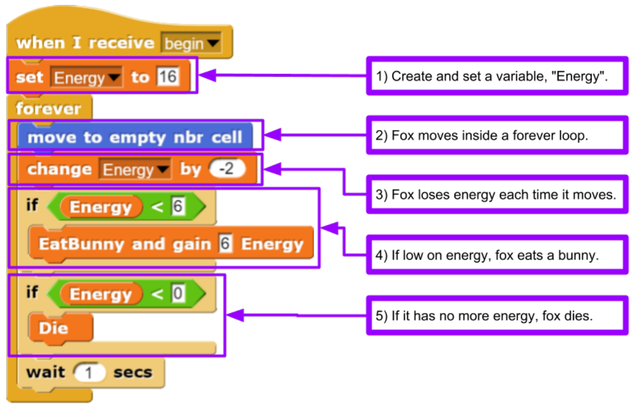
3.2 Curriculum

The Food Webs unit is a 4-day activity designed to meet the state-level education standards for 6th-grade science classrooms on the topic of Energy Transfer in a food web. On the first day of the activity, students use pseudocode to complete scientific worksheets on defining terminology and relationships between organisms in a food chain. Students work as a class to abstract the behaviors that an organism in the food chain might exhibit (eat, grow, move, etc.). On day two, teachers lead students in using their previous pseudocode to model the behavior of sunlight as an abiotic factor in the food web and its effect on producers (plants) using the Cellular programming environment. Using a companion worksheet, students manipulate energy values in the code to see its effects. On day three, teachers help students add in code for a primary consumer (bunny), again exploring the propagation of energy in the food chain. Finally, on day four, teachers instruct students on how to create the basic functionality for a secondary consumer (fox) as shown in Figure 1. Students then set up their own scientific experiment determining an independent and dependent variable, as well as their hypotheses on what will happen. Students are encouraged to run several trials of their experiments before declaring a definitive conclusion.

The Food Webs curriculum was chosen as we have data from the largest set of unique teachers as well as the largest number of student participants. We choose the 4th day rather than the other two programming days for our analysis because as part of our design-based implementation research, we implemented different versions of Days 2 and 3 for different teachers to experiment with different instructional scaffolding techniques. However, every teacher implemented the same final activity. As Day 4 is also the final day of the activity, teachers have had multiple class periods of experience instructing programming assignments and Day 4 serves as a culmination of their final abilities.

3.3 Data Analysis

The primary data used for our analyses is environment trace data from the consenting students working within the assignment. The 19 classroom implementations of the assignment resulted in over 90,000 interactions from 287 unique students. To train the teachers in instructing, instructor guides were developed. These gave the interactions with the environment (code additions) necessary to complete the assignments, included images of example final code, and a suggested order of how to add code in their environment for instruction. Using these instructor guides, we decompose the assignment into five unique, mutually exclusive *features*. These features demonstrate the five additions that teachers led students into creating in the Cellular Environment. These features can be detected using an auto-grader type system and we can determine at any given point in a students programming trace whether or not that feature is present or absent in the code. Figure 1 presents a short description of each feature alongside the corresponding code blocks.



```
when I receive begin
set Energy to 16
forever
    move to empty nbr cell
    change Energy by -2
    if Energy < 6
        EatBunny and gain 6 Energy
    if Energy < 0
        Die
    wait 1 secs
```

1) Create and set a variable, "Energy".
2) Fox moves inside a forever loop.
3) Fox loses energy each time it moves.
4) If low on energy, fox eats a bunny.
5) If it has no more energy, fox dies.

Figure 1: Food Webs Day 4 student coding tasks.

Feature 1 tasks students to create a local variable, Energy, that will represent how much energy a Fox agent has in the Food Web. The instructions explicitly state that the variable **must** be local as it represents an individual fox's current energy. Furthermore it **must** be named "Energy" due to a case-sensitive dependency in custom blocks introduced later in the program. Feature 2 tasks students to make the Fox move within a forever loop, as to simulate a Fox moving around in a biome. The 3rd feature has students decrement the Energy variable inside the forever loop to simulate the Fox losing energy with each move. The 4th feature introduces the first If block - If the energy of the Fox is low (but not 0), the Fox "eats a bunny" in the environment and gains some energy. The final feature, Feature 5, is a second If block where if the Fox's energy is less than 0, the Fox will die. We use code feature detection to determine, for each code state in a student's code-trace, whether or not each feature is present or absent in a student's code. We then use this information to develop the following three analyses:

1. **Feature-Time-Completion** - A histogram that visualizes when students complete a given feature relative

Table 2: Student demographics at each of the studied schools.

School	# Students	White	Asian	Black	Hispanic	Multiracial	Female	Low Income
School I	526	31.4%	6.7%	30.8%	24.3%	6.3%	48.3%	47.7%
School II	815	39.5%	5.9%	22.5%	25.8%	5.8%	48.1%	51.0%
School III	919	22.0%	2.6%	42.3%	28.1%	4.7%	52.2%	65.6%

to the feature completion time of the instructor.

2. **Cumulative-Feature-Completion** - A stair-step graph indicating when students add features to their code during the class relative to the instructor.

3. **Feature-Path-Graph** - A state-space representation outlining what features students add in what order, showing the varying paths they take compared to the path taken by the instructor.

4. RESULTS

4.1 Feature Time Completion

Our first methodology, Feature Time Completion, produces a histogram showing the timing in which students complete a feature relative to when the instructor completes the feature. In the three graphs in Figure 2, we present exemplar cases of the "I do, We do, You do" instructional strategies introduced in the Gradual Release of Responsibility (GRR) model [18]. This is a common instructional strategy taught to teachers, that progresses a teacher from modeling a step to students (I do) to completing a step together with students (We do) to finally allowing students to try on their own (You do), giving the correct answer after their attempts. The histograms shown show the timing of when the students complete the feature relative to when the teacher completes the feature (minute 0). In subfigure 2a we can see evidence of focused instruction, or "I do", where the teacher demonstrates procedures before students attempt to solve problems on their own. The graph is right-skewed and close to the center signifying students completing the task directly after the instructor. The second strategy, guided collaborative learning, or "We do", is characterized by students working as a class to complete the feature while being guided along by the instructor. An exemplar of this strategy is shown in graph '2b'. This has students completing the feature very close (within 5 minutes) of the instructor centering around the instructor's completion time. This suggests an instructional strategy where students are following along and completing the assignment with the instructor in a "we do" fashion. Finally, the last graphs depict independent learning, or a "You do" strategy where students attempt to solve the problem before reviewing as a group. Graph '2c' shows the majority of students completing the feature prior to the instructor demoing, with a subset of students (around 25%) completing the feature after the instructor reviews the solution. The median student (the red-dashed line) is either right, near-center, or left-adjusted for the three cases respectively. Classroom observations confirm that the Teacher scaffolded students by providing students with the blocks they would need to complete the feature (but not giving the full answer or how to combine them). Many students, given the appropriate building blocks, were then able to assemble the answer very quickly (10 minutes before the demonstration). We turn to Figure 4, which provides an alternative representation of the same information for one class period taught by Teacher B. Here, colored points are shown in order to be able to track students across different features. It should be noted that outlier points are removed for scaling purposes (5 total features are completed by 3 students past the 100 second marker). Teacher B completed 4 features in intended sequential order (F1,F2,F3,F4) which is important to note as this was not always the case (as we will see later). Most students complete the same feature within a minute of when Teacher B completes the feature (minus the visibile and non-visible outliers).

4.2 Cumulative Feature Completion

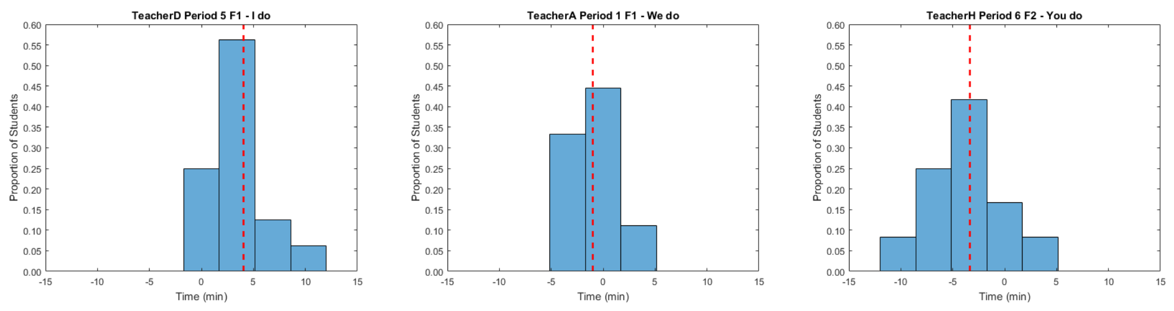
Our second representation shows how students cumulatively add features to their code relative to each other as well as relative to the instructor. Using the timestamps of when students add their next feature to their code, we are able to chart out in a Stair-step fashion students progressing through the assignment, going from having nothing complete to having more and more of the 5 features added. We are also able to see based off the final placement of each line, how many features students ended the lesson with. In the 3 cases and 4 classrooms described below, each student trace is an individual stair-step function. Instructor feature completion is denoted by the dashed vertical lines, and comparing when teachers and students complete features (in combination with the additional classroom observation data) allows us to understand different instructional patterns and their consequences.

4.2.1 Teacher G

Figure 3 describes one teacher doing the Food Web Assignments with their class. After the teacher completes their first feature (with the majority of the class following very closely, within 2 minutes of him completing it) 15 minute without progress pasess. A strand of students make progress independently of the teacher, but the majority are stuck at only completing one feature. From classroom observations, we know that the majority of students had followed the instructor in naming their variable "energy" (though the instructional guide says to name it "Energy" as custom functions expect that name and are case sensitive). A strand of students are able to recognize the error from their student guides and are able to independently make progress. At around 10:00, the teacher fixes the error, and announces it to the class). The wide range of students jumping in progress at varying different times reflects a change in instructional style adopted by the teacher. As it was a simple naming fix that affected every other Feature, fixing it resulted in making progress on multiple features near simultaneously (seen in the tall jumps in each stair step graph).

4.2.2 Teacher E: Period 4 vs 6

We now present a case study of the same instructor teaching two different periods. As part of the faded scaffolding teaching approach, a research member (Teacher C) taught



(a) TeacherD - I do (b) TeacherA - We do (c) TeacherH - You do

Figure 2: Gradual Release of Responsibility as shown by student interaction data

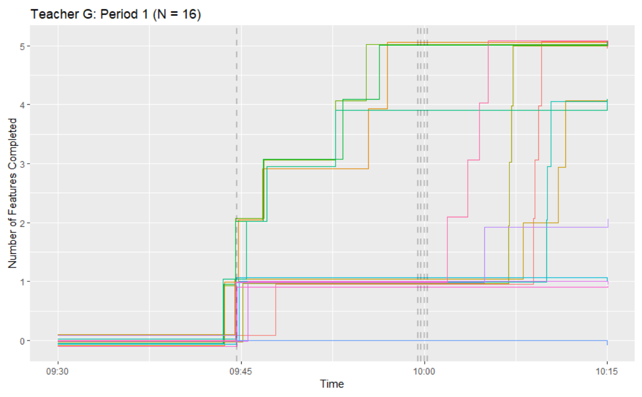


Figure 3: A flat-line after Teacher G's first task - students caught in the same error the Teacher does while live coding.

the first class period (Period 2) where Teacher E observed. Teacher E then taught Period 4. Teacher E requested that Teacher C instruct the next period (Period 5) to observe once more how the lesson should go and then afterwards, the teacher instructed Period 6. From Figure 4, we can see the difficulties that Teacher E had with their first attempt teaching. Teacher E completes all of her features for the class after nearly 45 minutes of instruction with sometimes nearly 8 minute gaps between feature additions. Many students are intently following Teacher E adding a feature when she does, but the number of bands that seem to be moving independently before the teacher suggests that the students were at varying paces not at the instructors. This is backed by the classroom observations with researchers noting that some students were ahead of the teacher and the teacher actually asked students for help on what to add next in the sequence. In the next attempt, Figure 5, Teacher E was able to complete features much faster, adding 4 of the 5 necessary features in an 8 minute span of time. This latter period dramatically changes the pace at which the students add the features, and the closeness of bands suggests that students were more intently following the instructors pace. There are common patterns that can be seen across both graphs. One is the presence of outlier students who complete things at a dramatically faster pace than the teachers. From the two graphs, it also should be noted that the teacher does not demonstrate the last feature to the class (unlike in her previous teaching period), and many of her students therefore do not get all 5 features in their code. We do acknowledge two outliers, one who jumps 4 features (around 11:15) and the other who more gradual progress, but still faster than the group. Classroom observations confirm that that student was copying code that the previous instructor had left on the projector during the transition period.

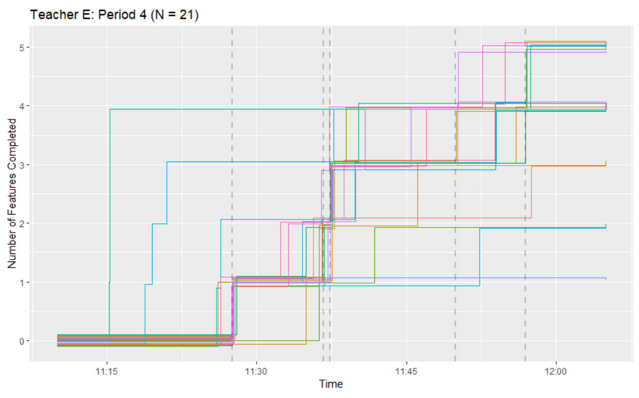


Figure 4: Teacher E's first attempt teaching the lesson.

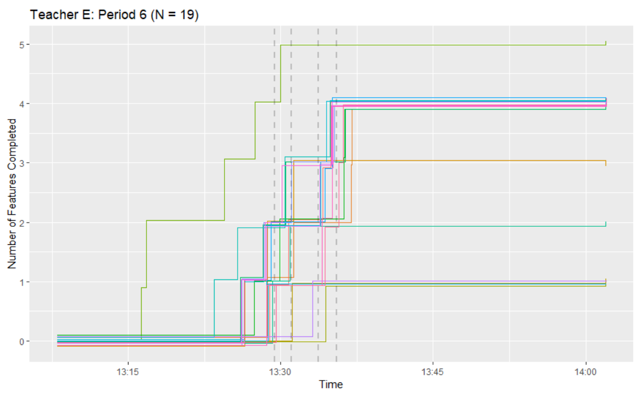


Figure 5: Teacher E's second attempt teaching the lesson.

4.2.3 Teacher F

We now highlight a period taught by Teacher F, a research instructor with a large amount of prior informal computer science teaching experience. Teacher F elected to lead the assignment *without* demonstrating to the students what the code should look like. Instead, she instructed students in what features they should be adding next, facilitating, and helping students by walking around the room. After each feature, she would then recap verbally with the whole class to review what they needed to have done in order to add the feature before moving on to the next one. We can see that student progress is very independent of one another, with the timing of each successive feature addition (i.e. when students go from having 1 to 2 features) varying widely. While students add their first feature within 5 minutes of each other, Students add their 2nd, 3rd, and 4th feature within 10 minutes of each other. This reflects the choice of instruction style, with Teacher F walking around the room independently helping students. In contrast to the end of Teacher G's lecture, progress in Teacher F's class is limited, usually, to *only* adding the next feature and nothing else. This reflects the teaching style of only giving students limited information about what the next tasks are, focusing the instruction on one feature at a time. Teacher F's students that complete their fifth feature (adding death), do it near simultaneously, reflecting the change in instruction style recorded where the teacher instructs the class in completing the feature together.

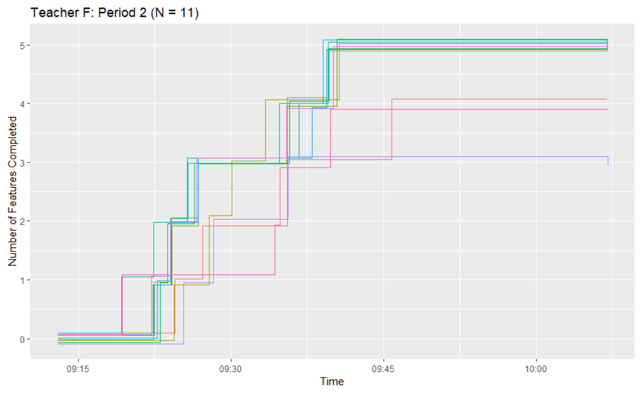


Figure 6: Teacher F instructs without demoing steps.

4.3 Feature-Path-Graph

In our final visualization, we remove the temporal information of the last two graph-types and focus solely on Feature-paths, the order in which students add features to their code. In Figure 7, we highlight three Feature State representations of three classes taught by three different instructors (Teacher C, B, and H). Each state in the graph represents a unique set of features complete by a student. For example, following the path of Teacher B (the states marked 'TB') in the second graph, the states progress from the one with 0 features to the one having only feature 1, to having both feature 1 and 2, to having features 1,2, and 3, and ending having features 1 through 4. Conversely, the state spaces for Teachers H and C show different sequences of feature additions (for C in the order 2,1,3,4,5 and for H in the order 4,2,3,5). Common state transitions, (those where a large proportion of students transitioned) are marked with the Feature that was added during the transition. The size of each state represents the proportion of students in the class that visited that state. As some students do not finish the assignment fully (getting "stuck" at certain points) we represent these as "stuck-states" in which a grayed proportion of the state represents how many students who visited that state did not

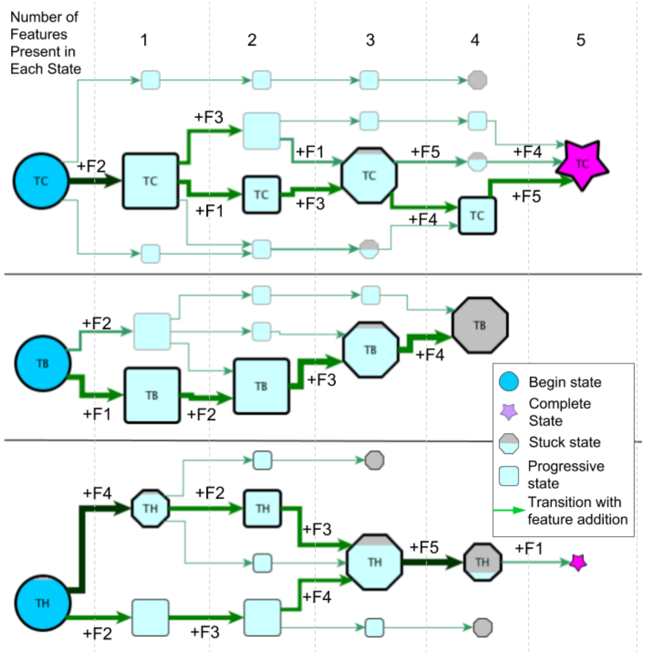


Figure 7: Feature-Path Graphs for Students (and teachers) within sample classes for Teachers C, B, and H.

progress. We also denote the beginning and end state with a special circle state and star-shaped state respectively. This is similar to the representation introduced in Rui et al [24].

The three feature graphs show different instructional strategies at play. The first, TC, show most students following the path of the teacher, adding features as they do. However, it also includes many individual paths taken by students not necessarily following the teachers instruction sequence explicitly, but following the general direction, denoted by the number of 'confluence' points or rejoining back into the Teachers state of various different paths. This classroom had the largest proportion of students actually finish the assignment compared to the other two classrooms. Teacher B's more narrow state-space with the largest states being those that the teacher traversed suggest that most students were following along with the instructors code additions (corroborated by Figure 4 as well as observations), adding in the order that they did. While Teacher H had a subset of students add features in the order they did as well as a subset chosing a separate path, these paths seemed to converge in state 2,3,4 then leading to the addition of Feature 5. While Teacher H and C both show variance in student feature paths, it should be noted that both Teacher H and B did *not* add all 5 features to their code (Teacher H did not add Feature 1, and Teacher B did not add feature 5). As such, all students in Teacher B's classroom and all but one student in Teacher H's class failed to complete all features of the assignment highlighting scenarios in which intent direct mimicry did not fully benefit students.

5. DISCUSSION

5.1 Feature Time Completion

As shown in Figure 2, while there are indeed examples of "I do" sequences, where a teacher demonstrates a task and then students replicate it, we also find examples of “We do” and “You do” sequences rounding out Pearson’s Gradual Release of Responsibility model. Our observations triangulate the “We do” pattern to actually involve the class working together with the teacher to solve a problem, as opposed to Pearson’s more traditional interpretation of the teacher working with just a small group of students. We find “I do” and “We do” strategies often adopted for the first features that students work on together in the assignment with the instructor with “You do” strategies prominent in features *later* in the assignment (the last features students work on). This would corroborate teaching progression strategies pushed for in the GRR model, transitioning students from “I do” to “We do” to “You do” feature completion throughout the course of an assignment.

5.2 Cumulative Feature

By separating out the individual traces of each student into their own function, we can see how a student temporally progresses through the assignment in relation both to the teacher and the other students in the class. From the student perspective, we see many examples of students who work in lock-step with the instructor (or with each other), adding features as they do. But we also see students who trail and lag behind as well as students who rapidly add features before the instructor has even introduced them. This differentiation is to be expected in a core classroom where students have varying access to computing outside of school and therefore different experiences with block-based programming environments. The ability to identify students who are not following along with the instruction, either as the pace is too fast or too slow, can aid instructors in identifying when to create more extension activities to high achievers or when to give more support to students in need.

5.3 Feature-Path

The state-space focuses on *which* features are being added in what order, and can grant information about the way teachers set up instruction (how they choose to pace the students through the task). Very narrow state-space graphs like shown for Teacher B can demonstrate a classroom that is explicitly following along with the instructions as sequenced by the teacher (even if they might be working on the tasks at different times). Conversely, wider state-spaces like those shown for Teachers C and H can be more indicative of classrooms where students had more agency over the order in which they choose tasks. We make no argument that more explicit instruction, or a higher path “follow” rate leads to either more or less students completing the assignment. Teacher B and Teacher H had a high proportion of students following along with their path, however, this might have left students unable to complete the assignment independently as neither teacher added the fifth and final feature.

5.4 Instruction

Looking through the teacher perspective, we can identify patterns indicative of increased teacher discomfort. As mentioned in Teacher G’s case study, as he erred in making the variable, he his following students all flat-lined in their coding progress. This highlighted the loss of the teacher as the “keeper” of knowledge and exposed the vulnerability that they had to problem-solve along with the students, thus incurring pedagogical emotional discomfort. Similar signs of discomfort are suggested in Teacher E’s case study where there are long pauses between teacher actions and student reactions. These graphs contrast Teacher F’s, the experienced computing instructor, which had students clustered on the same features, moving at a steady interval. Teacher F did not feel the need to explicitly direct students with live coding, but instead was able to guide their exploration through facilitation. Conversely, we also discovered evidence of *change* in instructional practice, both within a class period and between them. In many cases, these changes lead to more students following along and getting through the material. As the anxieties that novice teachers tend to have focus on not doing things ’correctly’, showing evidence of positive change between implementations could potentially inspire confidence in novice teachers that they are capable of instructing computing curricula.

5.5 Limitations

The drawn conclusions that we outline in our case studies are supported in part by our classroom observations, corroborating what we find within the data-driven visualizations. Because of the nuanced and varied ways in which classroom behavior can manifest within the data, it is important that we frame these results with our understanding of what happened within the classroom. As such, we understand the reliance our methodology has on observational data, and make no advocacy that our method acts as a complete replacement for it. In the same way that no single one of the three graphs tell the full set of information of a classroom, we believe that these quantitative analysis methods act as an additional tool to act alongside qualitative data collection methods, ones that scale much better though than the observational-based ones. We also understand that other limitations present themselves with aspects of our methodology such as only focusing on one lesson with only 5 possible features. In addition, only having the set of consenting students out of the entire classroom trace data could have created a sampling bias that might alter any and all of our graphs and therefore our analysis. We intend to try and offset these effects by replicating this with the multitude of lessons that are being developed by our teacher partners in the upcoming academic year. Our methodology described is agnostic to the assignment, the features, and how they are tested for (though we acknowledge that the feature state points are only as good as the tests you run on the code).

5.6 Future Work

Our initial results provide a foundation for further analysis into influence of one actor (e.g. teacher) on another (e.g. student) during programming. For example, as more schools begin to adopt pair programming models of instruction, it would be worth investigating these collaborative interaction effects. This current work has practical applications for our context in teacher training. As these graphs, charts, and other figures can be generated rather quickly, teacher-friendly versions can serve as additional means of explaining to teachers how the experience went and how the classroom acted. This data can help teachers inform their practice and provide immediate or post-hoc help. Further, additional processing can translate many of these graphs into *animations* that can give a ’live feel’ of how the instruction played out in real time. Setting times for feature completion targets for an instructor can keep them on pace in tight classroom periods, and windows for instructor-student feature completion can be used to determine if the class is following along intently. Often, the instructor participants in this study looked to researchers for a simple 'Thumbs Up or Thumbs Down' measure to see if they were doing things correctly and kids were following along. If a simple notification such as the one described were implemented into a dashboard system, we can imagine it benefiting instructors of K-12 programming.

6. ADDITIONAL AUTHORS

Mehak Maniktala (mmanikt@ncsu.edu), Danielle Boulden (dboulde@ncsu.edu), Eric Wiebe (wiebe@ncsu.edu) and Tiffany Barnes (tmbarnes@ncsu.edu). This material was supported by the National Science Foundation under grant 1837439. Anything expressed in this material does not necessarily reflect the views of the NSF.

7. REFERENCES

[1] B. M. Aidan Lane and J. Mullins. *Simulation with Cellular A Project Based Introduction to Programming.* BlockBooks Series. Monash University, Melbourne, Australia, first edition, 2012. Online: https://github.com/MonashAlexandria/snapapps.

[2] J. Bennedsen and M. E. Caspersen. Revealing the programming process. In *ACM SIGCSE Bulletin*, volume 37, pages 186–190. ACM, 2005.

[3] V. Cateté, N. Lytle, Y. Dong, D. Boulden, B. Akram, J. Houchins, T. Barnes, E. Wiebe, J. Lester, B. Mott, et al. Infusing computational thinking into middle grade science classrooms: lessons learned. In *Proceedings of the 13th Workshop in Primary and Secondary Computing Education*, page 21. ACM, 2018.

[4] C. Chen and P. J. Guo. Improv: Teaching programming at scale via live coding. In *Proceedings of the Sixth Annual ACM Conference on Learning at Scale*, L@S '19, New York, NY, USA, 2019. ACM.

[5] N. Diana, M. Eagle, J. Stamper, S. Grover, M. Bienkowski, and S. Basu. An instructor dashboard for real-time analytics in interactive programming assignments. In *Proceedings of the Seventh International Learning Analytics & Knowledge Conference*, pages 272–279. ACM, 2017.

[6] B. J. Fishman, W. R. Penuel, A.-R. Allen, B. H. Cheng, and N. Sabelli. Design-based implementation research: An emerging model for transforming the relationship of research and practice. *National society for the study of education*, 112(2):136–156, 2013.

[7] J. Frykholm. Teachers' tolerance for discomfort: Implications for curricular reform in mathematics. *Journal of Curriculum and Supervision*, 2004.

[8] S. Grover, R. Pea, and S. Cooper. Factors influencing computer science learning in middle school. In *Proceedings of the 47th ACM technical symposium on computing science education.* ACM, 2016.

[9] B. Harvey, D. Garcia, J. Paley, and L. Segars. Snap!:(build your own blocks). In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*, pages 662–662. ACM, 2012.

[10] F. T. Husic, M. C. Linn, and K. D. Sloane. Adapting instruction to the cognitive demands of learning to program. *Journal of Educational Psychology*, 1989.

[11] P. Ihantola, A. Vihavainen, A. Ahadi, M. Butler, J. Börstler, S. H. Edwards, E. Isohanni, A. Korhonen, A. Petersen, K. Rivers, et al. Educational data mining and learning analytics in programming: Literature review and case studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports.* ACM, 2015.

[12] R. Jocius, D. Joshi, Y. Dong, R. Robinson, V. Cateté, T. Barnes, J. Albert, A. Andrews, and N. Lytle. Code, connect, create: The 3c professional development model to support computational thinking infusion. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education*, SIGCSE 20. ACM, 2020.

[13] P. A. Kirschner, J. Sweller, and R. E. Clark. Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational psychologist*, 41(2):75–86, 2006.

[14] M. C. Linn and J. Dalbey. Cognitive consequences of programming instruction: Instruction, access, and ability. *Educational Psychologist*, 20(4):191, 1985.

[15] N. Lytle, V. Cateté, D. Boulden, Y. Dong, J. Houchins, A. Milliken, A. Isvik, D. Bounajim, E. Wiebe, and T. Barnes. Use, modify, create: Comparing computational thinking lesson progressions for stem classes. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, pages 395–401. ACM, 2019.

[16] N. Lytle, V. Cateté, Y. Dong, D. Boulden, B. Akram, J. Houchins, T. Barnes, and E. Wiebe. Ceo: A triangulated evaluation of a modeling-based ct-infused cs activity for non-cs middle grade students. In *Proceedings of the ACM Conference on Global Computing Education*, CompEd '19. ACM, 2019.

[17] J. Margolis. *Stuck in the shallow end: Education, race, and computing.* MIT Press, Cambridge, MA, 2010.

[18] P. D. Pearson and M. C. Gallagher. The instruction of reading comprehension. *Contemporary educational psychology*, 8(3):317–344, 1983.

[19] T. W. Price, Y. Dong, and D. Lipovac. isnap: towards intelligent tutoring in novice programming environments. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education.* ACM, 2017.

[20] A. Ruthmann, J. M. Heines, G. R. Greher, P. Laidler, and C. Saulters II. Teaching computational thinking through musical live coding in scratch. In *Proceedings of the 41st ACM technical symposium on Computer science education*, pages 351–355. ACM, 2010.

[21] J. Sweller. Cognitive load during problem solving: Effects on learning. *Cognitive science*, 12(2), 1988.

[22] D. Weintrop and U. Wilensky. Comparing block-based and text-based programming in high school computer science classrooms. *ACM Transactions on Computing Education (TOCE)*, 18(1):3, 2017.

[23] R. Zhi, T. Price, S. Marwan, Y. Dong, N. Lytle, and T. Barnes. Toward data-driven example feedback for novice programming. *International Educational Data Mining Society*, 2019.

[24] R. Zhi, T. W. Price, N. Lytle, Y. Dong, and T. Barnes. Reducing the state space of programming problems through data-driven feature detection.