Visualize Students' Progress To Facilitate Code Communication For In-Class Programming Exercises At Scale

Ashley Zhang [*1], **Yan Chen** [2] **and Steve Oney** [1]

[1] *University of Michigan, Ann Arbor*
[2] *Virginia Tech, Blacksburg*

Abstract

Programming instructors often conduct in-class exercises to help them identify students that are falling behind and surface students' misconceptions. However, monitoring students' progress during exercises is difficult, particularly for large classes. We present VizProg, a system that allows instructors to monitor and inspect students' coding progress in real-time during in-class exercises. VizProg represents students' statuses as a 2D Euclidean spatial map that encodes the students' problem-solving approaches and progress in real-time. VizProg allows instructors to navigate the temporal and structural evolution of students' code, understand relationships between code, and determine when to provide feedback. A comparison experiment showed that VizProg helped to identify more students' problems than a baseline system. VizProg also provides richer and more comprehensive information for identifying important student behavior. However, users also face information overload and have difficulty in reflection in VizProg. We proposed a future study that explores designs to improve VizProg.

Keywords: Programming education at scale. Code visualization.

[1]

1 Introduction

Programming instructors often conduct in-class coding exercises—short programming activities that students perform independently—to give students hands-on practice, assess students' progress, and identify students that are falling behind. By identifying and working with struggling students, instructors can strengthen students' understanding of the material and given them a better intuition for important concepts. However, if left unaddressed, small misunderstandings can escalate to become long-term learning barriers for students. Therefore, instructors should be able to identify struggling students and their misunderstandings during in-class exercises promptly and reliably. However, identifying problems in real time is difficult for several reasons. First, misunderstandings tend to be implicit, abstract, and not readily apparent without carefully reading students' code. However, it is often not possible to read students' code at scale in large classes or for shorter exercises. Second, there are many aspects of students' code (including aspects that the instructor might not anticipate) that instructors need to consider to gain insight into potential learning barriers. This suggests that there needs to be a better way to monitor students' code at scale.

Past research has explored ways to address these challenges. For example, Codeopticon [1] allows instructors to monitor students' code in real-time. However, Codeopticon requires that instructors read students' code individually, making it difficult to assess students' performance as a whole, particularly when needing to scale to large classes. Overcode [2] addresses the scalability issue by clustering and visualizing student code submissions [2]. However, it was designed for post-hoc analyses rather than providing real-time feedback and does not consider the need to monitor students over time. In addition, the time sensitivity and large class sizes make it difficult for instructors to identify learning challenges during in-lecture exercises. Ideally, instructors should be able to easily identify problems among many students' coding activities in real-time.

In this paper, we introduce VizProg, a tool that allows instructors to monitor and inspect large numbers of students' coding submissions over time by presenting students on a 2D map. In VizProg,

PLATEAU
13th Annual Workshop at the Intersection of PL and HCI

DOI: 10.35699/1983-3652.yyyy.nnnnn

Organizers:
Sarah Chasins, Elena Glassman, and Joshua Sunshine

*Author one's government requires funding acknowledegements on the first page.

[1] Portions of this submission are adapted from a concurrent submission to ACM CHI 2023

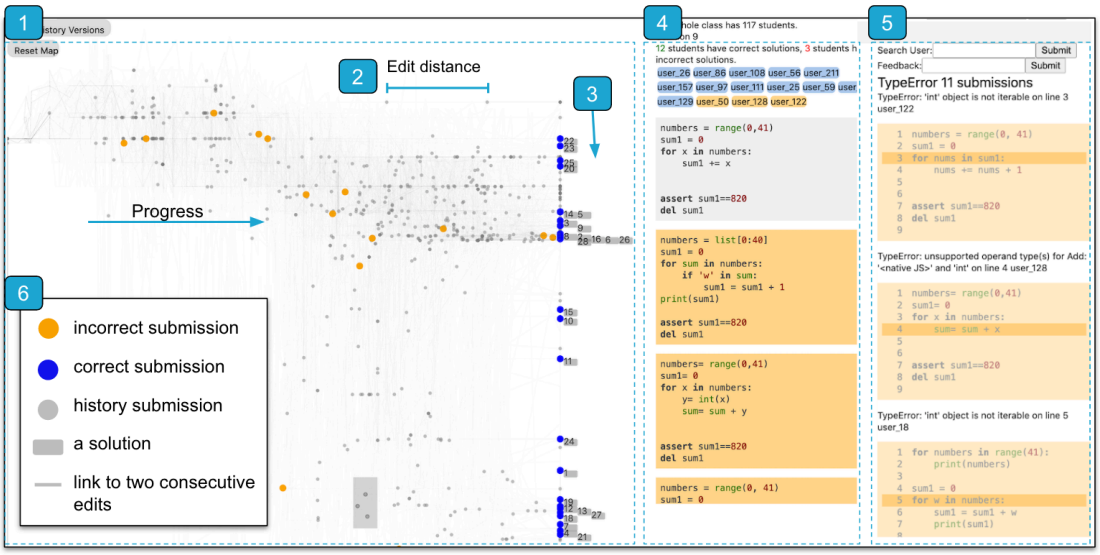

Figure 1. VizProg's User Interface. There are three main view panels: the overall class progress *2D map* view (1), a *solution-centered* view (4), and a *progress detailed* view (5). On the 2D map view, each dot represents a student's submission, each line between two dots indicates the edit movement. The x-axis encodes the size of a code edit to be proportional to the distance (2), and the y-axis represents different kinds of solutions for this exercise (3).

students' status is represented as a position that encodes 1) similarities in students' code (as 2D Euclidean distances); 2) how students *approach* the exercise (using vertical space); 3) students' progress—how close they are to a correct solution (using horizontal space); and 4) how students' status changes over time. This is done by computing the semantic similarity and edit distance between students' code and solution code. Additionally, VizProg allows instructors to navigate the temporal and structural evolution of students' code at different levels of granularity, understand relationships between code, determine when to provide feedback, and assess who might need feedback the most.

A comparison experiment showed that VizProg can help participants to 1) discover more than twice as many student misunderstandings, and 2) find the misunderstandings with less than half of the time and fewer interactions. Furthermore, participants reported that VizProg provides richer and more comprehensive information for identifying important student behaviors.

We also discussed weaknesses of VizProg observed from the comparison experiment. Users face information overload in VizProg. It is challenging for users to accurately 1) understand how code evolves over time, 2) identify most current struggles from all the history data, and 3) reflect on students' history data. We proposed designs to address these challenges from two aspects: reduce visual clutter and reflection. We proposed a future study to evaluate the effectiveness of these designs.

2 System Design

2.1 System Design Goals

Led by prior work and our interviews with instructors, we developed three design goals (DG1-DG3) to guide the design of VizProg to help instructors monitor students' in-class exercise progress in real-time.

- DG1: Easily view students' progress in real-time
- DG2: Easily compare the difference between code
- DG3: Ability to inspect and navigate students' progress at different granularity

2.2 VizProg's User Interface

Figure 1 shows VizProg's user interface which consists of three main panels: the overall class progress ***2D map*** view (1), a ***solution-centered*** view (4), and a ***progress detailed*** view (5). As soon as a user starts VizProg, the system continuously monitors each student's code editor at a keystroke level.

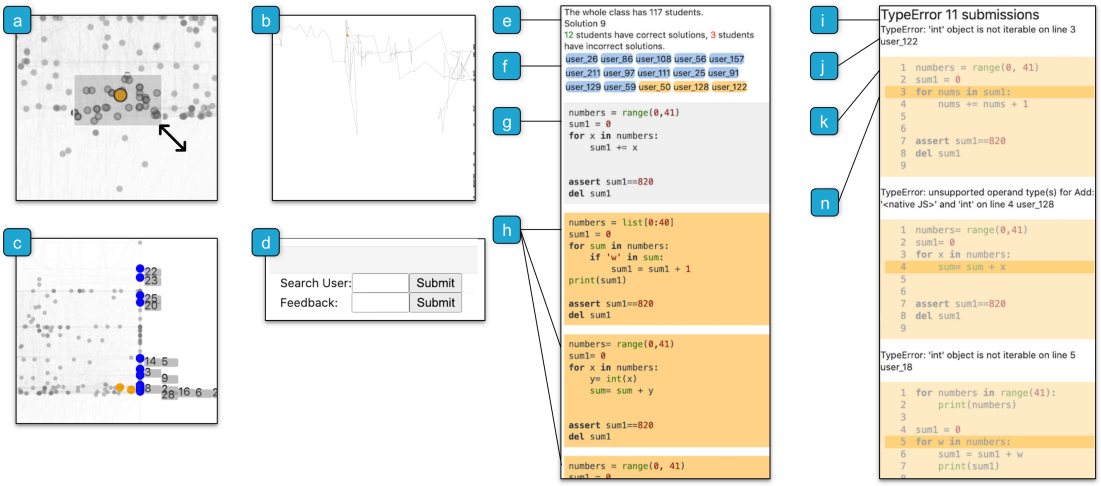

Figure 2. A detailed view of VizProg's user interface. Instructors can crop a region on the 2-D map to view solutions in that area (a). Cropped regions appear as gray rectangles surrounded by dots (a), and the progress detailed view shows statistics for the region (i, j, k, n). Correct solutions are displayed on the right of the map as blue dots with gray labels (c). When the instructor clicks a student name (f), the 2-D map will highlight this student's trajectory (b). The solution-centered view consists of statistics related to the solution (e), a list of students' IDs (f), and correct (g) and incorrect submissions (h) made by these students. Instructors can search for a specific student (d). They can also send feedback to either an individual student or a group of students (d).

On the 2D map view, it uses a color-coded dot to indicate a student's code status (correctness) and a gray line to show how their status changes over time (Figure 1.6). As they progress through the coding exercise, the 2D map updates in real-time to always reflect their current status. Instructors can interact with VizProg during the exercise (Figure 1.4, 5) to track class-wide performance or individual progress. Additionally, VizProg provides a lightweight feedback feature that enables users to send text messages to individual students or to a group of students (Fig. 2.d). Below, we describe the user interface design for VizProg.

2.2.1 2D Map View: Overall Class Progress

To clearly convey when *progress* is being made—when a student's position changed (DG1), VizProg depicts progress by left-to-right motion (Fig. 1.1), since rightward movement is a common representation of progress (and there might be a strong psychological basis for this in other domains [3]). A gray line was used to connect two consecutive edits. Only when a student *submits* their code will a dot appear on the 2D map. Small gray dots represent historical code versions (meaning a student submitted that code but has since moved on). Larger dots represent students' *current* 'location'. Orange dots represent students who have not yet found a correct solution (as determined by the instructor's unit tests). Blue dots represent students who have found a correct solution (Fig. 1.6). We also used a gray rectangle to indicate one type of solution (Fig. 1.3).

To ensure these updates are done in real-time, VizProg computes locations over intermediate code edits rather than submissions. These edits often include syntax errors, which makes many prior techniques that rely on building Abstract Syntax Trees (ASTs) [2], [4] not feasible to apply. VizProg instead uses transformer-based code vectorization [5], which can encode code semantics even when there are syntax errors (as we will describe later). VizProg is designed to make the generated space to *continuous* and *proportional* to the size of code edits—the size of a code edit should be proportional to the distance moved in 2-D space (Fig. 1.2). That is, if the Euclidean distance between dot_a and dot_b is shorter than that between dot_b and dot_c, then student_a and student_b should have a more similar solution than that of student_b and student_c. In addition, instructors should expect a similar coding pattern when inspecting submissions that are close to each other. In this manner, instructors can identify submissions that are far from being correct.

To help instructors understand the variety of students' solutions—to identify which solutions might be common and which might be abnormal (DG2), VizProg uses vertical space to represent different

kinds of solutions (Fig. 1.3). We applied the same clustering algorithm that we used for individual submissions to display similar solutions vertically closer to each other.

2.2.2 Solution-Centered View: Students End With A Solution

To inspect the submissions from all the students who arrived at the same solution (DG3), instructors can click a solution (Fig. 1.3) and see the solution-centered view (Fig. 1.4). The list contains three main sections: statistics related to the solution (Fig. 2.e), a list of IDs of students who are close to (or found) this solution (Fig. 2.f), and a list of submissions made by these students (Fig. 2.h). This view also summarizes the number of correct and incorrect submissions that are approaching this solution. The list of student IDs is color coded to represent correctness. The instructor can click each student ID to see the trajectory of the selected student's submissions on the 2D map view. The progress detailed view also displays all the submission made by this student throughout the history of the exercise (Fig. 1.5).

2.2.3 Progress Detailed View: A Selected Submission(s) View

To allow users to navigate students' progress at different granularities, VizProg lets users examine the code progress of both groups and individuals at the code level (DG3). For group progress, instructors can crop a region on the 2D map to see the submissions only within that region (Fig. 2.a). When a region is selected, the area on the map will be a gray rectangle surrounding multiple dots. As long as the selected region has at least one submission, the 2D map view will hide all the dots outside of the region, and the progress detailed view will also display only the selected submission code (Fig. 1.5). To assist users in identifying common misconceptions, VizProg lists these submissions by error type frequency in descending order (Fig. 2.i)[2]. Furthermore, VizProg color codes each submission by its correctness, where orange indicates an incorrect submission and gray indicates a correct submission (Fig. 2.n). VizProg also displays the error message (Fig. 2.j) and highlights the line of code that caused the error when there is an error in the submission (Fig. 2.k). By resizing the overlay or dragging the overlay on the 2D map, the user can view real-time updates on the progress detailed view of the selected region (Fig. 2.a). For individual progress, users can either search by student ID (Fig. 1.5), or click a student ID on the solution-centered view (Fig. 1.4). This student's progress will also be represented by a trajectory line on the 2D map (Fig. 2.b). After cropping a region or selecting submissions of a student, instructor can use the lightweight feedback feature (Fig. 2.d) to send feedback to the students that are selected.

2.3 VizProg's Algorithm

2.3.1 Normalizing Code

We refer to a given piece of code as c, a string of characters. $c_{s,t}$ refers to the code of student s at time t. We will use $\text{is\_correct}(c) \in \{true, false\}$ to represent if a solution c is correct ($\text{is\_correct}(c) = true$) or incorrect ($\text{is\_correct}(c) = false$), as determined by unit tests. We will use $\text{only\_correct}(C)$, where C is a set of code samples, to represent the subset of C where $\text{is\_correct} = true$. This means that: $\text{only\_correct}(C) \subseteq C$.

We rely on two separate similarity metrics to determine how to represent a student with code c in VizProg: edit distance and vector similarity (both described below). However, neither of these similarity metrics account for differences that have no functional meaning to the Python interpreter, such as differences in variable names, comments, and spacing. For example, both similarity metrics would determine that the following pieces of code are different, even though they are functionally nearly identical (the only difference being that the code sample on the right prints `` `Done!' ``):

2 Failing the unit tests for the problem also produces a runtime error.

```
my_variable = 10
my_dictionary = {}


for key, value in my_dictionary.items():
    other_value = value + 1
    print(key, other_value)
```

```
v = 10
d = {}

# loop over all of the items in d
for k, v in d.items():
    w = v + 1   # add 1 to the value
    print(k, w)
print('Done!')
```

Prior work has accounted for this by computing the Abstract Syntax Trees (ASTs) of both samples and modifying variable names between code samples to match [2]. However, this approach relies on building an AST, which is typically not possible in the presence of syntax errors. As we discuss above, we designed VizProg to work with code that has syntax errors. VizProg instead relies on text-based normalization, which attempts to normalize code by performing string-level operations, using regular expressions. Our normalization performs the following:

- Removes extra spacing (newline characters, `\n`) in the code
- Removes code comments
- Detects variable names, as defined through assignment (e.g., `varname = …`) or implicit declaration (e.g., `for varname in …:`).
- Removes calls to the `print()` function, as these are often used by students to debug their code and are not typically part of the problem definition

For example, VizProg's normalization on the above code samples would produce:

```
v0 = 10
v1 = {}

for v2, v3 in v1.items():
    v4 = v3 + 1
```

We refer to the normalized version of code c as $\text{norm}(c)$. Our normalization method has several drawbacks. First, it could result in small changes producing large semantic changes. For example, if a student s as code at time t $c_{s,t}$ and at time $t+1$, they add a '#' to comment out some portion of code, the distance between $norm(c_{s,t})$ and $norm(c_{s,t+1})$ could be large. Second, there are still several non-functional changes that it does not account for. For example, changing the order of declaration of `v0` and `v1` in the above code makes no functional difference to the code execution but is not accounted for in our normalization technique. Still, we have found that these issues have a small impact on our underlying algorithm. One of the reasons we used short variable names like `v0` is that there is a relatively small cost for naming mistakes; for example, the edit distance between `` `v0' `` and `` `v5' `` is small. However, future work could further improve our normalization method to account for these challenges.

We divide our discussion into our techniques for determining students' *approach* and their *progress*.

2.3.2 Representing Students' Problem-Solving Approaches in VizProg

We represent students' approach on the y-axis and we use the *vector* similarity between a students' solution and existing solutions to determine which approach they are using.

Vector Similarity The first distance metric that VizProg uses is *vector similarity*. VizProg leverages CodeBERT [5], a pre-trained transformer model capable of representing code, to convert code into a vector (with 768 dimensions by default). $\text{vec}(c) \in \mathbb{R}^{768}$ represents the vectorized version of code c, as computed by CodeBERT. We can compute the vector similarity of two different code samples c_1 and c_2 using the cosine similarity, after normalizing the code samples (using the normalization technique described above):

$$\text{vec\_sim}(c_1, c_2) \coloneqq \frac{\text{vec}(\text{norm}(c_1)) \cdot \text{vec}(\text{norm}(c_2))}{|\text{vec}(\text{norm}(c_1))||\text{vec}(\text{norm}(c_2))|}$$

This produces a single number in the range $[-1, 1]$ where higher numbers represent higher similarity. In practice, this vector similarity tends to be very close to 1 when comparing code samples for the same exercise, even when comparing different approaches to the same problem (empirically, in the range $[0.96, 1.0]$).

Building a Solution Space In order to build a Euclidean space for code solutions to a given problem, VizProg first needs a pre-existing set of prior solutions. In practice, these prior solutions might come from previous class sessions, previous semesters, instructor-written solutions, or could be collected after some subset of students has completed the exercise. The source of prior solutions may affect the solution space. Ideally, solution sets should be seeded from a source that contains a diverse and comprehensive set of approaches to solving the problem. We will discuss the problem of seeding VizProg in more detail in section **??**. We denote the set of prior solutions as PAST_CODE $= \{p_1, p_2, \cdots, p_{n_{past}}\}$, where there are n_{past} prior code examples. Ideally, PAST_CODE should contain several examples of correct solutions (is_correct$(p) = true$ for some $p \in$ PAST_CODE) but typically should contain a mixture of correct and incorrect solutions.

We first build a matrix P containing the vector representation of every item p_n in PAST_CODE (after normalizing the code):

$$P = \begin{bmatrix} | & | & & | \\ \text{vec}(\text{norm}(p_1)) & \text{vec}(\text{norm}(p_2)) & \cdots & \text{vec}(\text{norm}(p_{n_{past}})) \\ | & | & & | \end{bmatrix} \in \mathbb{R}^{n_{past} \times 768}$$

We then reduce P from 768 rows to 1 row, first using Principal Component Analysis (PCA) (to reduce from $(n_{past} \times 768)$ to $(n_{past} \times 40)$) and then T-SNE [6] (to reduce from $(n_{past} \times 40)$ to $(n_{past} \times 1)$). This reduces P to a single vector, which we call $\vec{y} =$T-SNE(PCA(P, 40), 1) $\in \mathbb{R}^{n_{past}}$, because we will use it to compute the vertical (y) position of students' code. $\vec{y_p} \in \mathbb{R}$ denotes the position of prior code sample p. We go through this process in order to distinguish between solutions p_i and p_j that are very similar ($\vec{y_{p_i}} \approx \vec{y_{p_j}}$) or different ($\vec{y_{p_i}} \not\approx \vec{y_{p_j}}$).

In addition, we use OverCode [2] to cluster similar correct solutions from PAST_CODE more robustly. A cluster in OverCode [2] is a set of correct solutions that perform the same computation. For a given problem, we get distinct solution clusters, which we use to label correct solutions along the y-axis in VizProg (Fig. 1.3).

Encoding Approach To determine which approach students are attempting to use, we use the vector similarity between students' solutions and prior solutions (all after normalizing the code). For a student's code c we first select the n_{vec_sim} prior solutions in PAST_CODE that are correct and most similar to c and store the result in NEAR_APPROACH. Formally, this is:

$$\text{NEAR\_APPROACH}(c) = \underset{PC \subseteq \text{only\_correct(PAST\_CODE)}, |PC| = n_{vec\_sim}}{\arg\max} \left(\sum_{p \in PC} \text{vec\_sim}(c, p) \right)$$

In Python code, this could be computed as (assuming c is defined as the current code sample):

```
NEAR_APPROACH = sorted(filter(is_correct, PAST_CODE), key=lambda p: vec_sim(c, p))[n_vec_s
```

This produces the subset of PAST_CODE with most semantically similar correct solutions. Smaller values of n_{vec_sim} produce movement that better reflects the solution that a given code sample is closest to but it can result in frequent vertical jumps as the closest solution changes. Larger values of n_{vec_sim} produce movement over time that is smoother but can be less accurate. VizProg uses $n_{vec\_sim} = 10$.

We then compute the y position of code c as the weighted average of these similar solutions:

$$\text{y\_position}(c) := \sum_{n \in \text{NEAR\_APPROACH}(c)} \vec{y_n} \cdot \text{softmax}(\text{vec\_sim}(c, p_n)^3)$$

Where $\vec{y_n} \in \mathbb{R}$ represents the y position of code n (as computed above). We cube the vector similarity to better differentiate between several similarities that are close to 1, while preserving the sign of the vector similarity.

2.3.3 Representing Students' Progress in VizProg

The second distance metric that VizProg uses is *edit distance*. We represent students' progress on the x-axis and we use the edit distance to determine how far they are from a correct solution.

Computing Edit Distance We use the normalized Levenshtein edit distance [7] ('levenshtein(a, b)' denotes the distance between a and b) to determine the edit distance between code samples:

$$\text{edit\_distance}(c_1, c_2) := \frac{\text{levenshtein}(\text{norm}(c_1), \text{norm}(c_2))}{\max(\text{len}(\text{norm}(c_1)), \text{len}(\text{norm}(c_2)))}$$

where $\text{len}(c)$ represents the number of characters in c (a positive integer $\in \mathbb{N}$) and $\max(a, b)$ represents a if $a \geq b$ and b otherwise. We normalize (divide by the maximum length code sequence) in order to avoid disproportionately long or short solutions or submission from overly influencing the edit distance. Thus, $\text{edit\_distance}(c_1, c_2)$ always returns a positive number between $[0, 1]$ where 0 would mean c_1 and c_2 are functionally identical (small edit distance).

Encoding Progress To determine how close students are to a correct solution, we use the edit distance between students' solutions and prior solutions (all after normalizing the code). We first find the n_{edit_sim} closest solutions by edit distance. VizProg uses $n_{edit\_sim} = 10$. Formally:

$$\text{NEAR\_EDIT}(c) = \underset{PC \subseteq \text{only\_correct}(\text{PAST\_CODE}), |PC| = n_{edit\_sim}}{\arg\min} \left(\sum_{p \in PC} \text{edit\_distance}(c, p) \right)$$

In Python code, this could be computed as (assuming c is defined as the current code sample):

```
NEAR_EDIT = sorted(filter(is_correct, PAST_CODE), key=lambda p: edit_distance(c, p))[:n_e
```

If solution c is correct (passes the instructor's unit tests) then we assign its x position to 0. If it is not correct, we compute the x position as the average edit distance for items in NEAR_EDIT:

$$\text{x\_position}(c) := \begin{cases} 0, & \text{if } correct(c) = true \\ \underset{p \in \text{NEAR\_EDIT}(c)}{\text{average}} (-1 \cdot \text{edit\_distance}(c, p)), & \text{otherwise} \end{cases}$$

Where 'average' represents the arithmetic mean. Note that we negate the edit distances, meaning $\text{x\_position}(c)$ is always ≤ 0 and larger numbers signify that c is *more similar* to existing solutions.

3 Discussion

3.1 VizProg's Visualization is Intuitive for Participants

A comparison experiment shows, VizProg helps participants to identify more issues while spending less time analyzing students' submissions. These findings suggest that VizProg's visualization and interactions are intuitive, and can help them identify students' problems at scale. Participants mentioned the movement from left to right in the map tells that students are making progress toward the right answer, whereas conventioanl tools have no indication. This makes sense because the 2D map can off load users' effort of tracking students' history activities to visual information such as the color and the position of the dots. With the encoded information, users can more quickly decide on which students to focus on, shaping their strategies on analyzing students' behaviors.

3.2 Trajectories in VizProg ease the reasoning progress

VizProg offers an innovative way to visualize students' coding progress, which not only reduces instructor's memory load, but also provides visual guide to reasoning about students' behaviors. For instance,

when students are working toward a final solution, instructors can clearly see how a dot moves along a trajectory and easily recall history versions by looking at trajectory's position. In a conventional timeline view, every time students make a new submission, users need to look at previous versions to recall what this student submitted before. Additionally, VizProg also helps participants identify abnormal behaviors. For instance, participants found that some students' trajectories first reached the rightmost side and then wander back to the left side of the map, which means the students had correct submissions and then changed it to incorrect solutions. They reasoned that these students might be exploring other approaches and did not need help. Participants also found that students were wandering in the middle of the map and never reached the right side of it during the whole exercise. Participants then decided to talk to the student and give tailored feedback.

3.3 Weaknesses of VizProg

Users face information overload in VizProg when many students start working on their code. We find these things difficult to understand in VizProg:

- Understanding how code evolves over time. Currently we use paths to represent code evolution, which are displayed in gray lines. When hundreds of students are editing at the same time, it is difficult to follow many of the overlapping gray lines. Therefore, patterns among these trajectories are not clear to users. In addition, VizProg does not provide annotation for the meaning of each area on the map. Users need to read the solutions in the area to understand the semantic meaning.
- Identifying current struggles from all the history data. Currently we use gray lines, colored dots to represent information on students' correctness, progress, and approach. As students come up with a final solution, VizProg does not differentiate data that appear earlier from data that is more recent. It would be challenging for instructors to accurately tell what students' most recent activities are. In addition, some students may struggle at a certain area for a long time while other students just pass the area quickly as an intermediate state. Such behavior is not obvious for instructors to notice without the assistance of visual cues.
- Reflection on students' progress. Users need to manually brush and select areas to view details about students' progress in VizProg, while dots move fast. It is common that dots already move to other areas when instructors try to figure out what struggles students have in a specific area. Therefore, it would be challenging for instructors to reflect on students' struggles and give impromptu assistant to students in real-time.

4 Future Study

In this section, we will discuss different designs to improve VizProg and propose a new study to evaluate these ways.

4.1 Designs

We aim to improve VizProg from two aspects: 1) reduce visual clutter to understand code evolvement and identify current struggles from history data, 2) reflection on students' history submissions to gain insights into their progress. We list potential designs in these two aspects:

4.1.1 Reduce visual clutter

- Clustering. We can cluster dots in VizProg based on the similarity of code. Since the dots are moving at keystroke-level, it may be distracting to render cluster updates at keystroke-level. We can provide a separate view of clusters on the map in addition to the original 2-D map view. The clusters could update every few seconds so that instructors have time to understand what happens within a cluster. In addition to clustering, we can also integrate AI assistance (e.g. ChatGPT) to help instructor understand students' struggles and provide hints more efficiently.
- Increasing granularity (grouping updates). As mentioned in Section 3.3, users need to manually brush and select areas to view progress, which is time-consuming. We can increase granularity of viewing progress in VizProg by grouping students' updates. For instance, there may be common patterns such as editing from "range(0, 40)" to "range(0, 41)" when they need to create a list

ranging from 0 to 40. It would save instructor some time if VizProg can automatically group such common updates.

- Emphasizing students' recent activities. We can also reduce visual clutter in the dots and trajectories. Current user interface of VizProg display all the data in dots and trajectories regardless to the time they appear, which can be very messy at scale and make it difficult for instructors to tell students' most recent struggles. Therefore, we can reduce visual clutter by hiding data that happen earlier so that instructors can focus on current struggles and give useful feedback.

4.1.2 Reflection

- Aggregate past data. VizProg does not aggregate past data compared to geographic map. For instance, students may have different ways to solve a bug except for the ways designed by instructors. In VizProg, these ways are all represented as a bunch of lines between dots, thus being difficult to differentiate. We can aggregate past data in VizProg so that instructors can quickly differentiate the authorized solution given by instructors and the other solutions come up by students.
- Replayability. VizProg does not enable replay when all the students finish their exercises. Therefore, we can add replayability to VizProg so that instructor can go back to a point where students are most struggling. Replayability would be helpful for instructors to reflect on the usefulness of the programming exercises, design impromptu exercises, and improve their instructional content.

4.2 User Study

We propose the following study to evaluate the designs. Our goal is to enhance VizProg to reduce information overload and help instructor understand trajectories. We will conduct lab experiments to examine which design would help instructors understand their progress faster and more accurately. We aim to study the following research questions:

- Which design(s) can effectively reduce visual clutter?
- Which design(s) can effectively help instructors reflect on students' progress?
- What are the pros and cons of these designs?

To evaluate the effectiveness of each design, we will use slide-based mock up with students' data collected from introductory programming courses to compare their pros and cons. We choose slide-based mock up because it could help users focus on the design itself instead of paying too much attention to usability issues.

We will recruit 20 participants with experience teaching Python programming courses. The study will include two sessions. In the first session, participants will use all the designs to view students' progress and spend 8 minutes per design. Participants will then answer interview questions around the pros and cons of each design. We will use their answers to narrow down to three designs that are most useful to instructors. In the second session, participants will be randomly assigned two designs to use. Participants will have 15 minutes to interact with the slide-based mock up, and then answer interview questions to compare the two designs they have tried.

4.2.1 Data collection

The study will be conducted virtually viz Zoom. We will gather the screen recording of how participants interact with the prototype, and the answers to the interview questions where we ask about their experience using the prototypes and feedback on the design. We will also make observational notes during the lab experiments, where researchers collect data on instructors' thoughts and questions.

5 Conclusion

In this work, we explored a design that allows instructors to visualize and understand students' status in real-time for in-class programming exercises. We introduce VizProg, a tool that allows instructors to monitor and inspect large numbers of students' coding submissions over time by presenting students on a 2D map. In VizProg, students' status is represented as a position that encodes similarities in students' code, how students approach the exercise, students' progress—how close they are to a correct solution, and how students' status changes over time. Although VizProg can help participants

to discover more than twice as many student problems, and find these problems with less than half of the time and fewer interactions, we found that users face information overload in VizProg. We proposed designs to reduce information overload from two aspects: reduce visual clutter and reflection. This work illustrates how we can further improve teaching by better understanding students' mental models and providing tailored feedback at scale.

References

[1] P. J. Guo, "Codeopticon: Real-time, one-to-many human tutoring for computer programming," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, 2015, pp. 599–608.

[2] E. L. Glassman, J. Scott, R. Singh, P. J. Guo, and R. C. Miller, "Overcode: Visualizing variation in student solutions to programming problems at scale," *ACM Transactions on Computer-Human Interaction (TOCHI)*, vol. 22, no. 2, pp. 1–35, 2015.

[3] M. L. Egizii, J. Denny, K. A. Neuendorf, P. D. Skalski, and R. Campbell, "Which way did he go? directionality of film character and camera movement and subsequent spectator interpretation," in *International Communication Association conference, Phoenix, AZ*, 2012.

[4] A. Nguyen, C. Piech, J. Huang, and L. Guibas, "Codewebs: Scalable homework search for massive open online programming courses," in *Proceedings of the 23rd international conference on World wide web*, 2014, pp. 491–502.

[5] Z. Feng, D. Guo, D. Tang, *et al.*, "Codebert: A pre-trained model for programming and natural languages," *arXiv preprint arXiv:2002.08155*, 2020.

[6] L. Van der Maaten and G. Hinton, "Visualizing data using t-sne.," *Journal of machine learning research*, vol. 9, no. 11, 2008.

[7] V. I. Levenshtein *et al.*, "Binary codes capable of correcting deletions, insertions, and reversals," in *Soviet physics doklady*, Soviet Union, vol. 10, 1966, pp. 707–710.