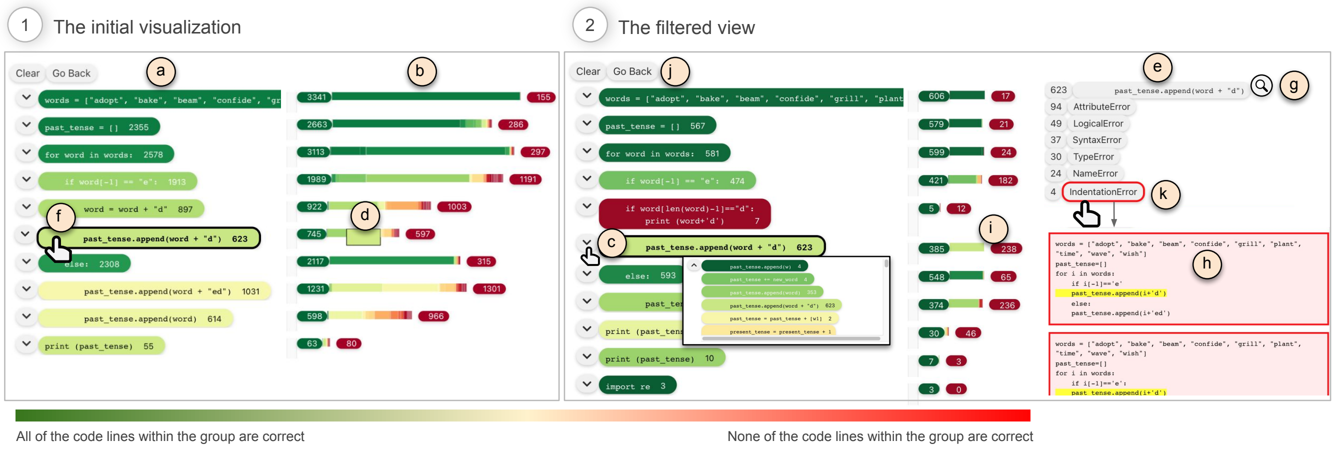
Demonstration of CFlow: Supporting Semantic Flow Analysis of Students' Code in Programming Problems at Scale

Ashley Ge Zhang
University of Michigan
Ann Arbor, Michigan, USA
gezh@umich.edu

Xiaohang Tang
Virginia Tech
Blacksburg, Virginia, USA
xiaohangtang@vt.edu

Steve Oney
University of Michigan
Ann Arbor, Michigan, USA
soney@umich.edu

Yan Chen
Virginia Tech
Blacksburg, Virginia, USA
ych@vt.edu


1 The initial visualization
2 The filtered view
Clear
Go Back
a
b
c
d
e
f
g
h
i
j
k
All of the code lines within the group are correct
None of the code lines within the group are correct

```
words = ["adopt", "bake", "beam", "confide", "gr      3341  155
past_tense = []  2355                                 2663  286
for word in words:  2578                              3113  297
    if word[-1] == "e":  1913                         1989  1191
        word = word + "d"  897                        922   1003
        past_tense.append(word + "d")  623            745   597
    else:  2308                                       2117  315
        past_tense.append(word + "ed")  1031          1231  1301
        past_tense.append(word)  614                  598   966
print (past_tense)  55                                63    80
```

```
words = ["adopt", "bake", "beam", "confide", "grill", "plant     606  17
past_tense = []  567                                           579  21
for word in words:  581                                        599  24
    if word[-1] == "e":  474                                   421  182
    if word[len(word)-1]=="d":
        print (word+'d')      7                                5    12
        past_tense.append(word + "d")  623                     385  238
    else:  593                                                 548  65
        past_te                                                374  236
print (past_tens                                               30   46
print (past_tense)  10                                         7    3
import re  3                                                   3    0
```

```
past_tense.append(w)  4
past_tense += new_word  4
past_tense.append(word)  353
past_tense.append(word + "d")  623
past_tense = past_tense + [w1]  2
present_tense = present_tense + 1
```

```
623  past_tense.append(word + "d")
94   AttributeError
49   LogicalError
37   SyntaxError
30   TypeError
24   NameError
4    IndentationError
```

```
words = ["adopt", "bake", "beam", "confide", "grill", "plant",
"time", "wave", "wish"]
past_tense=[]
for i in words:
    if i[-1]=='e'
    past_tense.append(i+'d')
    else:
    past_tense.append(i+'ed')
```

```
words = ["adopt", "bake", "beam", "confide", "grill", "plant",
"time", "wave", "wish"]
past_tense=[]
for i in words:
    if i[-1]=='e':
    past_tense.append(i+'d')
```

Figure 1: CFlow allows users to explore the semantic flows of student code submissions at a high level through semantic aggregation. First, users are presented with an overview of the entire set of solutions (1), including the SAV (a) and the SHV (b). They can then click on individual code lines (f) to progressively explore the details of specific implementations. The visualization will update to focus on a smaller subset of solutions, displaying the aggregated flow and distributions of the selected set only (2). Users can inspect details of the flow at individual code level in the CDV (e). CFlow offers a breakdown of types of errors within that group (k), and detailed solutions with context-aware highlighting (h).

ABSTRACT

Introductory programming courses have been growing rapidly, now enrolling hundreds or thousands of students. In such large courses, it can be overwhelmingly difficult for instructors to understand class-wide problem-solving patterns or issues, which is crucial for improving instruction and addressing important pedagogical challenges. In this paper, we propose a technique and system, *CFlow*, for creating understandable and navigable representations of code at scale. CFlow is able to represent thousands of code samples in a visualization that resembles a single code sample. CFlow creates scalable code representations by (1) clustering individual statements with similar semantic purposes, (2) presenting clustered statements in a way that maintains semantic relationships between statements, (3) representing the correctness of different variations as a histogram, and (4) allowing users to navigate through solutions interactively using semantic filters. With a multi-level view design, users can navigate high-level patterns, and low-level implementations. This is in contrast to prior tools that either limit their focus on isolated statements (and thus discard the surrounding context of those statements) or cluster entire code samples (which can lead to large numbers of clusters—for example, if there are n code features and m implementations of each, there can be m^n clusters).

CCS CONCEPTS

• **Human-centered computing → Information visualization**; • **Applied computing → Education**; • **Software and its engineering**;

KEYWORDS

Code Visualization, Programming Education, Code Clustering

ACM Reference Format:
Ashley Ge Zhang, Xiaohang Tang, Steve Oney, and Yan Chen. 2024. Demonstration of CFlow: Supporting Semantic Flow Analysis of Students' Code in Programming Problems at Scale. In *Proceedings of the Eleventh ACM Conference on Learning @ Scale (L@S '24), July 18–20, 2024, Atlanta, GA, USA.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3657604.3664717

1 INTRODUCTION

Understanding student code is crucial for instructors to provide personalized feedback and enhance student learning outcomes. When performing these tasks, instructors often need to: 1) identify *issue-relevant code*, and 2) understand the *underlying reasons* for students' struggles. However, with increasing enrollments in CS courses and the diversity of student codes, these tasks become challenging for two reasons. First, students' errors could span multiple lines in code and be correlated. How students put code lines together reflect the student's thoughts on problem solving. An issue might be as simple as a one-line error, or as complex as a mistake spanning multiple lines, such as improper initialization and modification of variable values. Switching between code syntax and semantics on a large scale is cognitively demanding. Second, even with identical syntax structure, two submissions could yield different outputs, and vice versa. For example, students might arrange their if-else statements differently yet produce the same output. This nuanced difference makes it difficult to aggregate a large number of code issues at both the syntax and semantic levels.

Prior work has explored designs to identify code issues using a variety of methods. For example, OverCode addressed scalability concerns by clustering and visualizing student code submissions [3]. However, it targeted primarily on specific issues such as different variable names, and focused on the computations of programs instead of high-level misconceptions between lines of code. To facilitate the discovery of high-level patterns, RunEx enabled instructors to construct runtime and syntax-based search queries with high expressiveness [4]. However, code search tools depend on instructors to generate search queries and demand prior knowledge on students' mistakes. VizProg uses a 2D map view and dynamic dots to represent students' coding progress in a concise view, providing instructors a general sense of mistakes [5]. VizProg's downside is that it lacks semantics meanings between the abstract visual cues and the detailed code content.

In this paper, we introduce a novel method that visualizes the flow of code statements to facilitate analyzing students' submissions. Similar to the narrative flow in an essay, the sequencing of code statements dictates both the execution order within a program and its runtime complexity [1]. In student submissions, this flow sheds light on a student's grasp of the problem, their approaches, thought processes, and misunderstandings. However, visualizing code flow is challenging for two reasons: First, student code submissions often vary significantly in their structure and semantic meaning. The difficulty lies in aggregating and aligning these diverse submissions. Second, designing a visualization that maintains the code's inherent structure while facilitating flow analysis is difficult, as the same code semantic flow might have different structures across submissions. Additionally, aggregating code structures on a large scale could make it more cognitively overwhelming for instructors. For instance, by showing the variation within steps across submissions, it could introduce much more complexity at each step of the approaches than simply reading a single code sample. Therefore, visualizations must present flow information in a way that effectively combines code samples and maintains the readability of the original code samples.

To tackle these challenges, we designed CFlow, a system comprising three distinct views: the *Semantic Aggregation View* (SAV, Figure 1a), the *Semantic Histogram View* (SHV, Figure 1b), and the *Code Detailed View* (CDV, Figure 1e). In essence, CFlow aims to represent semantic flows between distinct value sets and showcases categorical variances. Specifically, CFlow employs a multi-step algorithm where each code line is subjected to a detailed semantic analysis and error check using Large Language Model (LLM). These lines are then vectorized using CodeBert [2] to group them based on similarity. A "common progression" of steps from correct solutions provides a reference structure for mapping all solutions. The resultant structured data is then visualized in a color-coded format, enabling educators to quickly pinpoint student challenges and misconceptions. This method 1) highlights semantic patterns by frequency and accuracy, and 2) simplifies the navigation and comparison of code flows. The core insight of CFlow is to align instructors' analysis of code submissions with the intrinsic characteristics of student code. Details of CFlow's user interface is shown in Figure 1.

2 DEMO EXECUTION

The demo consists of a standalone website, with its front end developed in React and TypeScript, and the back end in Python. It is hosted on the author's computer, which serves as the server. To set up and run the demo, the author's computer needs npm installed and must be connected to the internet.

Visitors will interact with the demo directly on the author's computer or through a network connection to the author's server, e.g. visiting a website through a given URL on their own device, depending on the setup. They will navigate a website displaying the demo, where they can engage with interactive visualizations. These visualizations allow users to explore various students' errors and approaches in a dataset. Users can manipulate the visual data through clicks and other simple interactions, providing a hands-on experience to understand the insights the demo offers. Figure 1 shows the user interface of the demonstration, CFlow. With the demonstration, visitors would be able to click the visual elements in CFlow, through clicking the code labels and the histogram elements.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under DUE 1915515.

REFERENCES

[1] Frances E Allen. 1970. Control flow analysis. *ACM Sigplan Notices* 5, 7 (1970), 1–19.

[2] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[3] Elena L Glassman, Jeremy Scott, Rishabh Singh, Philip J Guo, and Robert C Miller. 2015. OverCode: Visualizing variation in student solutions to programming problems at scale. *ACM Transactions on Computer-Human Interaction (TOCHI)* 22, 2 (2015), 1–35.

[4] Ashley Ge Zhang, Yan Chen, and Steve Oney. 2023. RunEx: Augmenting Regular-Expression Code Search with Runtime Values. In *2023 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 139–147.

[5] Ashley Ge Zhang, Yan Chen, and Steve Oney. 2023. Vizprog: Identifying misunderstandings by visualizing students' coding progress. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*. 1–16.