PathViewer: Visualizing Pathways through Student Data

Yiting Wang, Walker White, and Erik Andersen
Department of Computer Science, Cornell University
{yw428, wmw2, ela63}@cornell.edu

ABSTRACT
Analysis of student data is critical for improving education. In particular, educators need to understand what approaches their students are taking to solve a problem. However, identifying student strategies and discovering areas of confusion is difficult because an educator may not know what queries to ask or what patterns to look for in the data. In this paper, we present a visualization tool, PathViewer, to model the paths that students follow when solving a problem. PathViewer leverages ideas from flow diagrams and natural language processing to visualize the sequences of intermediate steps that students take. Using PathViewer, we analyzed how several students solved a Python assignment, discovering interesting and unexpected patterns. Our results suggest that PathViewer can allow educators to quickly identify areas of interest, drill down into specific areas, and identify student approaches to the problem as well as misconceptions they may have.

Author Keywords
data visualization; programming education

ACM Classification Keywords
H.5.0. Information Interfaces and Presentation: General

INTRODUCTION
Analysis of student data is critical for improving education. Educators need to understand what approaches their students are taking to solve problems. These insights help educators understand the patterns in what students are doing and how they are confused, so they can drill down into areas of interest and modify their approach. There has recently been a lot of interest in using visual data mining to look at student data for programming [14, 10, 13]. These studies analyzed changes in students' programs over time by tracking the addition of conceptual components or modeling the stages of program construction as a finite state machine. However, we still lack tools for rapid visual assessment and comparison of the pathways that large numbers of students are taking.

Ideally, we would have visualizations that allow educators to triage their class and identify the primary ways in which students are confused. This would help them direct their effort towards resolving those specific sources of confusion. In this paper, we present a visualization tool, PathViewer, to model the paths students take when solving a problem. PathViewer visualizes intermediate stages of students' solutions as sequences of discrete states, building on work in education and video games [14, 10, 2, 17]. It innovates by using Sankey diagrams [15] to visualize how students "flow" from one state to the next, and n-grams [6] to visualize specific state sequences.

We evaluated PathViewer by visualizing how a large group of students solved an assignment in a Cornell introductory programming course. By visualizing state transitions and the proportion of students who made each transition, we were able to search for interesting patterns such as loops and common state sequences. We found that students frequently do not move in one direction towards a solution, and instead loop back and forth as they experiment with various ways to solve the problem. We also found that the test cases that were provided to the students did not fully assess whether students had correctly implemented a particular function, since some incorrect implementations also passed all of the test cases. Furthermore, we observed that students are likely to follow patterns they have seen previously when tackling a new problem, and that they often get stuck if this pattern is incorrect.

Our results suggest that PathViewer can allow educators to quickly identify high-level solution patterns that are are particularly interesting and then drill down into specific areas of interest. In doing so, they can identify the key approaches that students are taking and misconceptions that students may have. These techniques are potentially useful not just for education, but also other areas of HCI, since they can help visualize how large groups of participants use an interface.

The main contributions of this paper are:

- A novel technique for visualizing the solution pathways that many students take to solve a problem.
- A case study using this technique in an introductory programming class to identify interesting patterns.

RELATED WORK
Various researchers have tacked the problem of visualizing student data using visual data mining. Hosseini et al. [10] modeled intermediate steps of students' programs by analyzing changes in programming concepts that are reflected in the program and whether these changes increased or decreased the correctness of the program. Although this representation captured changes to the correctness of a student's program, it did not fully capture the number of students in one particular state, nor the paths the students took to solve a problem.

Another approach is finite state machines. Piech et al. [14] took snapshots of students' code during construction, and

CHI 2017, May 06 - 11, 2017, Denver, CO, USA

ACM 978-1-4503-4655-9/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3025453.3025819

used a Hidden Markov Model to discover "sink states", or states that the students had high probability of remaining in for several code updates. Although this kind of visualization captures state transitions between solutions and the number of people that reach a particular state, it does not visualize how many people transition from one state to another. Furthermore, although the state diagram includes transitions to the same state as well as loops, the visualization does not display these loops. Berland et al. [3] also uses a state-based visualization to track sequences of students' programs. They label the edges between states with the percentage of participants who make that transition. In contrast, PathViewer uses these ratios to set the width of the Sankey diagram edges, rapidly drawing visual attention to the most prevalent pathways and facilitating comparison between pathways. In addition, the n-gram mode helps identify common sequences and loops, which conveys useful information about students' problem-solving strategies.

The problem of visualizing pathways in student's data is similar to a problem studied in video games, namely how to visualize the pathways that large groups of players take as they play a game. The Playtracer tool used by Andersen et al. [2] and Liu et al. [12] and the spatiotemporal tool used by Wallner et al. [17] both visualize the states that players go through when playing a level of a game. These approaches produce a state transition model that captures the number of people who reached a particular state, and displays similar states close to each other. However, these approaches do not quite capture the complete paths that people take to reach a particular state, nor the loops that frequently occur.

PATHVIEWER

PathViewer visualizes the pathways that students take through the intermediate stages of solving a problem. The user can provide a set of equivalence classes in order to group similar states together. Paths are visualized using a Sankey diagram [15], a common way to visualize a large amount of data flowing in a particular direction. The visualization was implemented using the d3.js [5] Javascript library, the open source d3.chart.js library, and the d3.chart.sankey.js library.

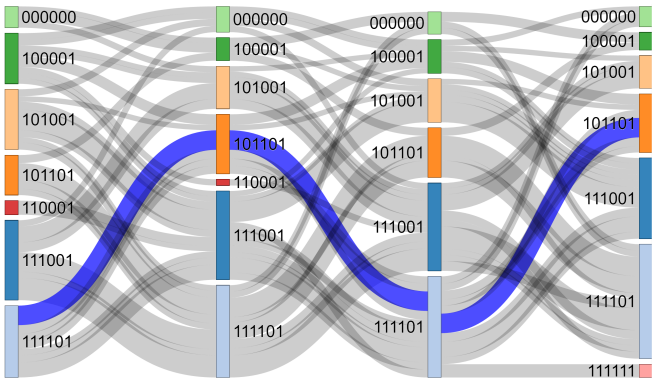

Figure 1. This is a 4-gram visualization of paths students took when solving programming exercises. Paths with fewer than five students are hidden. Each node is a six-character string of "0"s and "1"s that indicates the specific test cases the program passed. For example, the state `111011` holds programs that passed all of the test cases except the fourth one. This visualization displays the proportion of students who take particular paths. For example, the blue path shows a loop in which students move back and forth between states `111101` and `110101`.

A common visual data mining workflow is to identify the most common patterns of pathways through solution states. To visualize and compare pathways, the tool borrows the idea of n-grams from natural language processing to visualize transitions through sequences of a specific length. To visualize this, the tool has an n-gram mode where the path can be displayed in bigrams (Figure 2), trigrams or four-grams (Figure 1). To identify which states and edges are most commonly visited, the weight of both the nodes and the edges is drawn proportionally to the number of students who took this particular path. In this mode, clicking on a particular edge of the graph highlights the path as well as students' solutions that correspond to the states along the path, as depicted in Figure 1. This allows instructors to browse through specific state sequences, and compare these sequences to look for interesting patterns.

Another useful way to analyze student paths is to look at the full path and try to see the bigger picture of where students are headed. Therefore, the tool has another mode, the full path mode, which allows for visualization of the complete paths through all the student solutions. To make visualizing paths easier, the tool supports alternate color schemes and sorting of state nodes within columns. The last three states of the full path are displayed in Figure 3.

EVALUATION

We used PathViewer to visualize how students completed a Python assignment in Cornell's introductory programming course, CS1110.

Data Collection

We first collected data on the intermediate stages that students' programs went through as they completed assignments. To do this, we adapted the Online Python Tutor [9] to record a snapshot of the code every time the student ran it. Students used the tool for multiple short lab exercises. The tool recorded data anonymously, and students could use it multiple times, each time resulting in a new interaction sequence. 545 students were enrolled in the class. Throughout the semester, we gathered about 400,000 snapshots of students' programs.

Data Analysis

We used PathViewer to analyze how students wrote a specific function, *removeDups*. It is a recursive function that returns a copy of the input list with adjacent duplicates removed. We collected 757 program states for this function.

We first defined equivalence classes that would group similar programs together into the same state. Various methods have been used to compare and classify students' code. Glassman et al. [8] reformatted the solutions to remove elements such as extra white space, used an execution trace to rename the variables, then compared the resulting code for identical lines of code. Huang et al. [11] used the Abstract Syntax Trees of students' code and calculated the editing distance [16] between them to compare for similarity between programs. Other equivalence class definitions that have been used for students' programs include program size [7], a combination of code size and frequency [4], and API Call Dissimilarity [14].

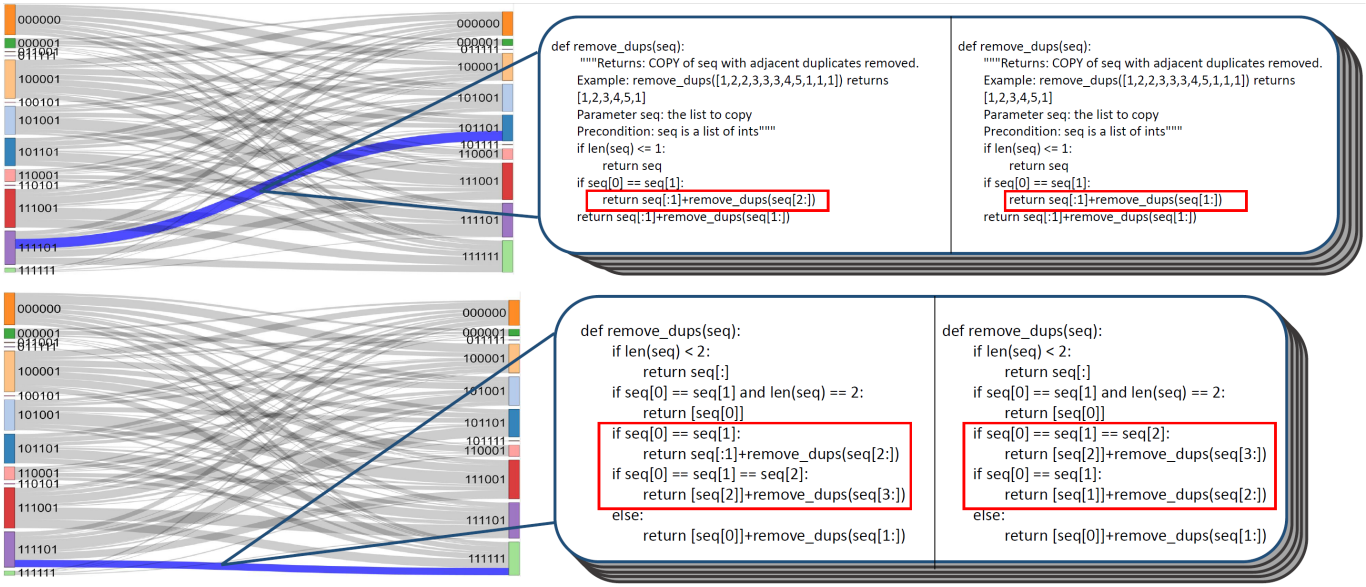

Figure 2. These two images show how the tool can be used to investigate code transitions by displaying code samples when a particular path is clicked. From the code sample, the highlighted line of code is modified from one state to another. In the top image, the particular change in the function is not correct, which resulted in failing a test case that the code shown on the left has originally passed. The bottom image displays a transition that resulted in a piece of code (the code on the right side) that passed all 6 test cases, but is not actually correct as it only works under certain conditions.

For this evaluation, we defined the equivalence classes to be the set of test cases that the students' program passed. This is because test cases are expert-defined equivalence classes that have been used historically to validate a piece of software [1], and are easy to execute on a large data set of student solutions. Six test cases provided feedback to the students while they implemented the function *removeDups*. We grouped programs by success on individual test cases and used binary strings to represent them. The n^{th} bit is "1" if the program passed the n^{th} test case, and zero if not. For example, programs in the state `111011` passed all test cases but the fourth one.

Identifying Sequential Patterns through n-grams

Using PathViewer to view 4-grams, as shown in Figure 1, we noticed that students' solutions often do not progress linearly and revisit certain states. The highlighted blue path is one particular cycle involving the states `101101` and `111101`. The transition from `101101` to `111101` is easy to explain: the program passed an additional test case. However, the transition from `111101` to `101101` was surprising because the student did made a change that failed a previously passed test case.

Figure 2 shows a bigram visualization we used to explore this abnormal but popular pattern further. To compare with successful students, we also visualized the code for the transition `111101` to `111111`, displayed in the bottom of Figure 2. Roughly 60% of students who reached the state `111101` did something wrong and ended up in state `101101`, while roughly 40% then passed all test cases.

We also discovered that of the 155 students who reached state `111101` and moved on to one of the two states above, 76 of them had code similar to the following:

```
if seq[0]==seq[1]:
    return [seq[0]] + remove_dups(seq[2:])
return [seq[0]] + remove_dups(seq[1:])}
```

Of those 76 students, 42 moved on to state `111111`, but 34 moved on to another state `101101`. The second failed test case removed duplicates from a list with two identical elements (e.g. [3,3]). 30 of these students used an approach similar to:

```
if seq[0]==seq[1]:
    return [seq[0]] + remove_dups(seq[1:])
return [seq[0]] + remove_dups(seq[1:])
```

This code is displayed in the top of Figure 2. From this visualization we discovered that the failed test cases were because of a common pattern. In this pattern, students took the head of the list, recursed on the rest, and glued them back together. This pattern was shown for a different problem in the course, but was not correct for this function. This suggests that students are likely to imitate patterns from class, especially if they have just learned a topic that is unfamiliar to them. Therefore, when the instructor is introducing a new problem to students, the instruction should present several ways to solve it.

Identifying Common Causes of Failure

When we used PathViewer to examine the code of students with the transition from `111101` to `111111`, we discovered that some of them passed all the provided test cases, but still did not actually implement the function correctly. The bottom of Figure 2 is one example of this. The code displayed shows that it had been tailored to the six test cases given, and would fail any other test case. This finding reflected that the original six test cases we had for the students did not accurately capture program correctness. To see how many students bypassed our test cases, we added three more test cases so that the equivalence classes were now based on nine test cases.

We used the full path mode in PathViewer with an alternate color scheme to depict two categories of students:

- Green: correctly implemented the function in the end
- Blue: passed all given test cases but program was incorrect

This categorization removes the students who gave up before passing all the test cases.

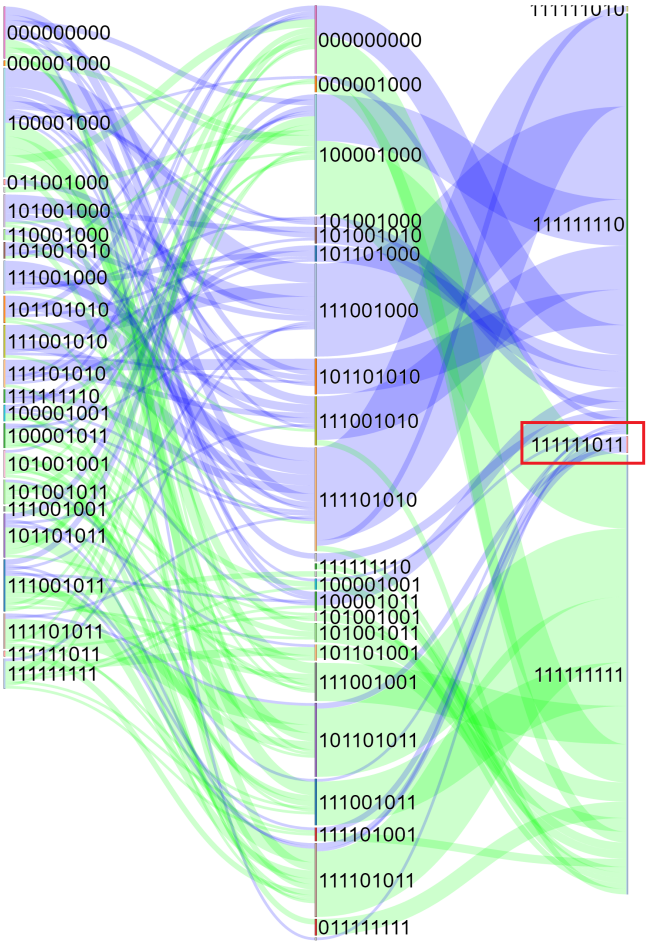

Figure 3. This is a visualization of the last three steps in students' paths, with nodes that have fewer than eight students removed. States are now expressed as binary strings of length nine because we added three more test cases to check for code correctness in addition to the original six. Blue paths indicate programs that passed the original six test cases but did not pass some of the additional three test cases. Green paths show correct solutions. The red box shows the proportion of students who did not pass the seventh test case, suggesting they wrote their code specifically to satisfy the six test cases originally provided.

In Figure 3, we displayed the last three steps of students in the full path mode. From the graph, it is apparent that the number of students ending in state `111111110` and the number of students in the correct state `111111111` was similar. The last test case tested whether students copied the empty list, because the function specification required students to return a copy of the original list. We can see from the pattern of the final transitions in this diagram that success or failure on this test was independent from success or failure on the other tests. Therefore, students' understanding of this step was independent of their understanding of recursion.

In addition, one can see from the state `111111011`, which highlighted by the red box, that 15 students did not pass the seventh test case. This case tested for the correctness of the function using a more complicated input than the previous six test cases. This suggests that these students did not write a correct function, but tailored their code to pass the provided six test cases. An example of one of the programs is shown in the bottom of Figure 2. This suggests (perhaps unsurprisingly) that if test cases are provided to the students, they may write their programs just to satisfy those test cases.

DISCUSSION

PathViewer had an impact on subsequent instruction of the class. The analysis revealed that the original set test of cases for the assignment did not fully capture program correctness and more test cases were added. In the following semester, the instructor continued to use the larger set of test cases. Additionally, the instructor taught an additional recursive programming pattern (divide and conquer) in the following semester because the tool revealed that students were overusing a particular pattern (pull one element off).

However, there are still usability challenges and limitations of PathViewer, due to the preliminary status of the tool. One challenge is working with a large number of states. The user can address this by defining broader equivalence classes to group similar states and pathways together, or by displaying only the most common states and paths. Another challenge is how to define good equivalence classes. This depends somewhat on what the user intends to investigate. For example, we examined program correctness by using the set of passing test cases as an equivalence class. A third challenge is what value of "n" to use for viewing n-grams. We suggest two strategies: 1) start from bigrams and then increase to trigrams to see if there are any interesting loops or other patterns that can be found and 2) use the full graph to identify patterns and then pick "n" to dig down into those patterns. Two additional limitations are that an educator will always need to analyze the graph and find useful patterns, and the quality of analysis will always depend on the choice of equivalence classes.

In the future, we are particularly interested in how an instructor can use PathViewer to improve course materials so that students take more desirable pathways. Ideally, an instructor would use PathViewer to analyze a previous semester's data, identify problematic strategies used by students, improve lesson plans to preempt those strategies, and then use PathViewer to determine if the modifications had the desired effect.

CONCLUSIONS

We presented a way to visualize students' solutions that helps an instructor view how large groups of students progress as they solve a problem. Our tool draws upon ideas such as n-grams and Sankey diagrams to provide a visualization that not only displays the states students solutions are going through, but also the proportion of students that take a particular path. We used this tool to analyze students' programs in introductory computer science class and uncover patterns that would be hard to identify otherwise. There is still a lot of work to be done in creating efficient visualizations of progressions of students' solutions to understand their performance. In future work, we hope to apply this approach to topics other than computer science. We also hope that analysis of intermediate stages of problem solving may make it easier to identify the students who are having the most trouble, and that this can help design early notification systems to alert teachers.

REFERENCES

1. W Richards Adrion, Martha A Branstad, and John C Cherniavsky. 1982. Validation, verification, and testing of computer software. *ACM Computing Surveys (CSUR)* 14, 2 (1982), 159–192.
2. Erik Andersen, Yun-En Liu, Ethan Apter, François Boucher-Genesse, and Zoran Popović. 2010. Gameplay analysis through state projection. In *Proceedings of the fifth international conference on the foundations of digital games*. ACM, 1–8.
3. Matthew Berland, Taylor Martin, Tom Benton, Carmen Petrick Smith, and Don Davis. 2013. Using learning analytics to understand the learning pathways of novice programmers. *Journal of the Learning Sciences* 22, 4 (2013), 564–599.
4. Paulo Blikstein, Marcelo Worsley, Chris Piech, Mehran Sahami, Steven Cooper, and Daphne Koller. 2014. Programming pluralism: Using learning analytics to detect patterns in the learning of computer programming. *Journal of the Learning Sciences* 23, 4 (2014), 561–599.
5. Michael Bostock, Vadim Ogievetsky, and Jeffrey Heer. 2011. D^3 data-driven documents. *IEEE transactions on visualization and computer graphics* 17, 12 (2011), 2301–2309.
6. Peter F Brown, Peter V Desouza, Robert L Mercer, Vincent J Della Pietra, and Jenifer C Lai. 1992. Class-based n-gram models of natural language. *Computational linguistics* 18, 4 (1992), 467–479.
7. Elena L Glassman, Ned Gulley, and Robert C Miller. 2013. Toward facilitating assistance to students attempting engineering design problems. In *Proceedings of the ninth annual international ACM conference on International computing education research*. ACM, 41–46.
8. 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), 7.
9. Philip J Guo. 2013. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM technical symposium on Computer science education*. ACM, 579–584.
10. Roya Hosseini, Arto Vihavainen, and Peter Brusilovsky. 2014. Exploring problem solving paths in a Java programming course. (2014).
11. Jonathan Huang, Chris Piech, Andy Nguyen, and Leonidas Guibas. 2013. Syntactic and functional variability of a million code submissions in a machine learning mooc. In *AIED 2013 Workshops Proceedings Volume*. Citeseer, 25.
12. Yun-En Liu, Erik Andersen, Richard Snider, Seth Cooper, and Zoran Popović. 2011. Feature-based projections for effective playtrace analysis. In *Proceedings of the 6th international conference on foundations of digital games*. ACM, 69–76.
13. Chris Piech, Mehran Sahami, Jonathan Huang, and Leonidas Guibas. 2015. Autonomously generating hints by inferring problem solving policies. In *Proceedings of the Second (2015) ACM Conference on Learning@ Scale*. ACM, 195–204.
14. Chris Piech, Mehran Sahami, Daphne Koller, Steve Cooper, and Paulo Blikstein. 2012. Modeling how students learn to program. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. ACM, 153–160.
15. Patrick Riehmann, Manfred Hanfler, and Bernd Froehlich. 2005. Interactive sankey diagrams. In *IEEE Symposium on Information Visualization, 2005. INFOVIS 2005.* IEEE, 233–240.
16. Dennis Shasha, JT-L Wang, Kaizhong Zhang, and Frank Y Shih. 1994. Exact and approximate algorithms for unordered tree matching. *IEEE Transactions on Systems, Man, and Cybernetics* 24, 4 (1994), 668–678.
17. Günter Wallner and Simone Kriglstein. 2012. A spatiotemporal visualization approach for the analysis of gameplay data. In *Proceedings of the SIGCHI conference on human factors in computing systems*. ACM, 1115–1124.