Programming Pathways: A Technique for Analyzing Novice Programmers' Learning Trajectories

Marcelo Worsley, Paulo Blikstein

Stanford University, Graduate School of Education, Stanford, CA, USA
{mworsley, paulob}@stanford.edu

Abstract. Introductory computer science courses are a valuable resource to students of all disciplines. While we often look at students' end products to judge their proficiency, little analysis is done on the most integral aspect of learning to programming, the process. We also have a hard time quantifying how students' programming changes over the course of a semester. In order to address these we show how a process-oriented analysis can identify meaningful trends in how programmers develop proficiency across various assignments.

Keywords: Machine Learning, Computational Thinking, Programming.

1 Introduction

We are seeing a shift in who is using computers, and in who is doing computer programming. A variety of disciplines are realizing that the skills of computational thinking and debugging, for example, are applicable to nearly every domain. There is something about the process of learning computer programming that facilitates one's ability to think constructively about any number of tasks. Despite the importance of the process, most computer science curricula rely on a final code submission and course examinations in order to validate student learning. In order to get back to processes, this work closely analyzes student learning processes for a class of introductory programming students. Furthermore, the analysis demonstrates how we can use techniques from computer science to automatically identify important changes that take place in the process that students use to complete their assignments.

2 Previous Work

Traditional work in computer programming assessment has focused on learning outcomes (Cooper, Cassell, Cunningham, Moskal 2005; Olds, Moskal and Miller 2005), and designing the right environment for enabling students to achieve those learning outcomes (Moskal, Lurie, Cooper 2004; Goldman 2004). Ironically, initial work in computer science education was heavily centered on process based assessments. For example, from Soloway and Spohrer (1989) we observe that more expert programmers are "planners," who make large, low frequency code updates. The trend towards

process recently re-emerged (Jadud 2005, Blikstein 2011 and Piech et al. 2012). These three studies utilized snapshots of student compilations as the basis of their analysis. This study borrows elements from Jadud, Blikstein and Piech et al., but differs in that we look at changes in students' programming process over a set of assignments, instead of just looking at one assignment.

3 Methods

This work was intended to automatically detect the evolution of student programming strategies and knowledge throughout an introductory programming class. In order to do this, we focused on studying "tinkering," or bricoleur, and "planning" (Turkle and Papert, 1991). We operationalized "tinkerer" and "planner" to be related to the number of characters or lines that a student adds, removes or modifies between snapshots. We are not concerned with absolute labels of tinkering and planning, but are looking for relative changes for each student and to tinkering and planning episodes.

Data comes from four programming assignments that seventy-four students, from a research-1 university, completed during the course of several weeks of their class. These assignments do not represent the entirety of the assignments for this course. Two early assignments were omitted because the nature of the programming environments varied greatly from later assignments.

We first extract the number of lines added, lines removed, lines modified, characters added, characters removed and absolute value of characters modified between successive snapshots, a value that we collectively refer to as the "update characteristics." These values exclude comments and are based on computing the line-by-line difference between snapshots. "Modified" was used for lines that are at least 70% the same as the line in the previous snapshot.

The extracted values are z-transformed across all students for a given assignment. In order to compute the similarity between students' sets of snapshots, we used dynamic time warping, and then scaled all sequences to be of the same length before computing the Euclidean distance between a given pair of snapshots. We then observe whether each student's programming pattern for Assignment 3 was most similar to that of Assignment 1, Assignment 2. Similarly, we record if Assignment 4's updates are more similar to that of Assignment 1, Assignment 2 or Assignment 3. Each student is assigned to a group based on their completion of the last two assignments, with the options: Assignment 1 - Assignment 1, Assignment 1 - Assignment 2, Assignment 1- Assignment 3, Assignment 2 - Assignment 1, Assignment 2-Assignment 3, Assignment 2-Assignment 2. For ease of interpretation we'll give each group a name (Table 1).

Table 1. Proportion of Students in Each Cluster

Cluster	**1- 1**	**1- 2**	**1-3**	**2-1**	**2-2**	**2- 3**
Name	ALPHA	BETA	DELTA	GAMMA	ZETA	OMEGA
Proportion	0.35	0.15	0.22	0.12	0.08	0.08

4 Results and Discussion

Table 1 shows the relative sizes of each group. Comparing clusters across assignment scores, we do not see any significant differences. However, when we compare examination scores (Table 2) we see a clear hierarchy, with OMEGA at the top and ZETA at the bottom[1]. The first thing that we note is that the data is normally distributed with the two smallest groups, ZETA and OMEGA occupying the two extremities. We also present data about help seeking frequency, disaggregated by month, for each group. ZETA, the worst performing group, is the most frequent attender of help during the first two months (Help 1 and Help 2) of the course, but fall to the least frequent attenders during the last month (Help 3). OMEGA, the highest performing group quickly transitions into being frequent help seekers (Help2), and both GAMMA and ALPHA become more frequent help seeking attenders[2].

Table 2. Ranking of Groups Across Variables

Rank	Midterm	Final	Help 1	Help 2	Help 3	Update Vector[3]
1	OMEGA	OMEGA	ZETA	ZETA	BETA	ALPHA
2	GAMMA	GAMMA	GAMMA	OMEGA	OMEGA	ZETA
3	DELTA	ALPHA	BETA	BETA	GAMMA	OMEGA
4	ALPHA	DELTA	DELTA	GAMMA	ALPHA	DELTA
5	BETA	BETA	OMEGA	ALPHA	DELTA	BETA
6	ZETA	ZETA	ALPHA	DELTA	ZETA	GAMMA

In an effort to characterize each groups progress over the course of the class, we present their change in update characteristics between Assignment 1 and Assignment 4 in the "Update Vector" column. ALPHA, ZETA and OMEGA share similar update vectors and DELTA, BETA and GAMMA share similar update vectors. These similarities are startling, given that ALPHA, ZETA and OMEGA occupy different parts of the performance spectrum, and the help seeking spectrum.

Looking closer we saw that students with different levels of expertise get differential benefits from help and differential benefits from their overall update approach. Additionally, we see that students use their code updates differently. Some use their updates as a way for checking syntax. Other students use updates to make their code more efficient. The other important difference is that students change in different ways. Some groups change in terms of average update size, but not in the overall approach. This was largely the case of ALPHA and ZETA. Alternatively some groups: DELTA, GAMMA, BETA and OMEGA; changed in their *sequence* of small

[1] ZETA was outscored on the final exam by ALPHA, GAMMA and OMEGA ($t(30) - 2.6896$ $p < 0.012$, $t(13)= 3.586$ $p < 0.003$, $t(10)=2.1778$ $p<0.04$) and on the midterm ($t(30)=2.5264$ $p<0.02$, $t(13)=2.254$ $p < 0.04$, $t(10) = 2.386$ $p< 0.04$), as well as by DELTA ($t(20) = 2.221$ $p< 0.04$).

[2] GAMMA attended fewer help sessions that ZETA in month 1 and month 2 ($t(20)=2.20$ $p< 0.0049$) and month 2 ($t(20)=2.786$ $p < 0.0114$).

[3] Similarity was computed using the f-statistics across all six items in the update vector.

and large changes, or tinkering and planning episodes, but maybe not in the average size of those updates.

Thus as we consider these types of analysis in future work, and study, in greater depth how different resources and actions impact traditional outcome based measures, we have to consider that students may change in different ways and look to better explain these different processes.

5 Conclusion

In this paper we presented an algorithm for studying changes in programming styles among novice programmers. We showed how using a process-oriented analysis was a meaningful approach. We also showed how looking at changes in students' programming update characteristics, relative to themselves, may provide the most useful lens for studying programming proficiency, as measured through assignment grades and test scores. In future research we will expand this work to a larger population of users and combine this analysis with additional qualitative data to more closely corroborate our interpretation of the data, especially as it relates to planning and tinkering.

References

1. Blikstein, P.: Using learning analytics to assess students' behavior in open-ended programming tasks. In: 2011 Learning Analytics Knowledge Conference (LAK '11). pp. 110-116. ACM, New York, (2011)
2. Cooper, S, Cassel, L., Moskal, B., and Cunningham, S.: Outcomes-based computer science education. In: 36th SIGCSE technical symposium on Computer Science Education (SIGCSE '05). pp. 260-261. ACM, New York (2005).
3. Goldman, K.: A concepts-first introduction to computer science. In: 35th SIGCSE technical symposium on Computer Science Education (SIGCSE '04). pp. 432-436. ACM, New York (2004).
4. Jadud, M.: Methods and tools for exploring novice compilation behaviour. In: 2nd International Workshop on Computing Education Research (ICER '06). pp. 73-84. ACM, New York (2006).
5. Moskal, B., Lurie, D. and Cooper, S.: Evaluating the effectiveness of a new instructional approach. In: 35th SIGCSE technical symposium on Computer Science Education (SIGCSE '04). pp. 75-79. ACM, New York (2004)
6. Soloway E. & Spohrer, J.: Studying the novice programmer. L. Erlbaum Assoc. Inc., Hillsdale (1988)
7. Turkle, S., Papert, S.: Epistemological Pluralism and Revaluation of the Concrete. In: I. H. a. S. Papert (eds.), Constructionism. pp. 161-192. Ablex Publishing Co., Norwood (1991)
8. Piech, C., Sahami, M., Koller, D., Cooper, S., & Blikstein, P.: (2012). Modeling how students learn to program. In: 43rd ACM technical symposium on Computer Science Education (SIGCSE '12). pp. 153-160. ACM, New York (2012)