Exploring Problem Solving Paths in a Java Programming Course

Roya Hosseini
Intelligent Systems Program
University of Pittsburgh
roh38@pitt.edu

Arto Vihavainen
Dep. of Computer Science
University of Helsinki
avihavai@cs.helsinki.fi

Peter Brusilovsky
University of Pittsburgh
135 North Bellefield Ave.
peterb@pitt.edu

Abstract

Assessment of students' programming submissions has been the focus of interest in many studies. Although the final submissions capture the whole program, they often tell very little about how it was developed. In this paper, we are able to look at intermediate programming steps using a unique dataset that captures a series of snapshots showing how students developed their program over time. We assessed each of these intermediate steps and performed a fine-grained concept-based analysis on each step to identify the most common programming paths. Analysis of results showed that most of the students tend to incrementally build the program and improve its correctness. This finding provides us with evidence that intermediate programming steps are important, and need to be taken into account for not only improving user modelling in educational programming systems, but also for providing better feedback to students.

Keywords: POP-II.C. Working Practices; POP-III.B. Java; POP-VI.E. Computer Science Education Research

1. Introduction

Over the last 10 years, the popularity and the use of systems for automated assessment of programming assignments has been growing constantly. From the pioneer Ceilidh system (Benford et al. 1993) to the modern popular Web-CAT (Edwards et al. 2008) system, these systems offer instructors and students some highly valuable functionality - the ability to assess the correctness and other important features of student programs without investing a large volume of manual labour. These systems introduced a number of changes in the way programming is usually taught. They allowed instructors to increase the number of programming assignments in a regular course and offer multiple submission attempts for each assignment (Douce et al. 2005). Additionally, they enabled teaching assistants to focus on more creative ways in assisting the students. As a result of these changes, the automatic assessment systems remarkably increased students' chances to receive feedback on their work.

There is, however, another important impact of automatic assessment systems that is being discussed much more rarely: their ability to increase our understanding of how humans solve programming problems. Course by course, automatic assessment systems accumulate a large number of program submissions. These submissions open a window to the students' problem solving process on both the personal and community level. On the *personal level*, the analysis of single student program submissions could reveal how the student progresses to the correct solution through several incorrect ones and how her knowledge grows from assignment to assignment. On the *community level*, the analysis of submissions for a single problem could reveal multiple correct and incorrect ways to solve the same problem.

Active work on community-level analysis has already started, to a large extent propelled by the availability of a very large volume of problem solutions collected by several programming MOOC courses that apply automatic assessment. Two interesting papers demonstrate how the MOOC data could be used to analyse the landscape of student problem solving solutions for individual problems (Huang et al. 2013, Piech et al. 2012), while another work shows how this data could be used to build an intelligent scaffolding system (Rivers and Koedinger 2013).

In our work we would like to contribute to the analysis of program submission on the *personal level*. This work is enabled by a unique set of data from University of Helsinki programming classes. Unlike other automatic assessment systems, the Helsinki system collected not just final program states submitted for grading, but also many intermediate states. It offered a much broader window to examine the progress of individual students on their way to problem solutions. In addition, we applied a unique concept-based analysis approach to examine and compare student partial and complete solutions.

Armed with a unique dataset and a novel analysis approach, we examined a large body of individual student paths to success. The remaining part of the paper is structured as follows. The next section reviews a few streams of related work. After that we introduce the key assets - the dataset and the concept-based analysis approach. The following sections present the details of our analysis. We conclude with an overview of the performed work and plans for future work.

2. Related Work

Understanding and identifying the various challenges novice programmers face has been in the center of interest of computer science educators for decades. Work on identifying traits that indicate tendency toward being a successful programmer dates back to the 1950s (Rowan 1957), where psychological tests were administered to identify people with an ability to program. Since the 1950s, dozens of different factors have been investigated. This work included the creation of various psychological tests (Evans and Simkin 1989, Tukiainen and Mönkkönen 2002) and investigation of the effect of factors such as mathematical background (White and Sivitanides 2003), spatial and visual reasoning (Fincher et al. 2006), motivation and comfort-level (Bergin and Reilly 2005), learning styles (Thomas et al. 2002), as well as the consistency of students' internal mental models (Bornat and Dehnadi 2008) and abstraction ability (Bennedssen and Caspersen, 2008). While many of these factors correlate with success within the study contexts, more recent research suggests that many of these might be context-specific, and urges researchers to look at data-driven approaches to investigate e.g., the students' ability to solve programming problems and errors from logs gathered from programming environments (Watson et al. 2014). Analysis of submission and snapshot streams in terms of student compilation behaviour and time usage has been investigated in, for instance, Jadud (2005) and Vee (2006).

Perhaps the most known work on exploring novice problem-solving behaviour is by Perkins et al., whose work classifies students as "stoppers," "movers" or "tinkerers" based on the strategy they choose when facing a problem (Perkins et al. 1986). The stoppers freeze when faced with a problem for which they see no solution, movers gradually work towards a correct solution, and tinkerers "try to solve a programming problem by writing some code and then making small changes in the hopes of getting it to work." Some of the reasons behind the different student types are related to the strategies that students apply as they seek to understand the problem; some students approach the programming problems using a "line by line" approach, where they often fail to see larger connections between the programming constructs (Winslow 1996), while others depend heavily on the keywords provided in the assignments that mistakenly led to incorrect or inefficient solutions (Ginat 2003).

The importance of students' constant and meaningful effort has been acknowledged by programming educators, who have invested a considerable amount of research in tools and systems that support students within the context they are learning, providing, e.g,. additional hints based on the students' progress. Such tools include systems that provide feedback for students' as they are programming (Vihavainen et al. 2013), systems that can automatically generate feedback for struggling students (Rivers and Koedinger 2013), and systems that seek to identify the situations when support should be provided (Mitchell et al. 2013).

3. The Dataset

The dataset used in this study comes from a six-week introductory programming course (5 ECTS) held at the University of Helsinki during fall 2012. The introductory programming course is the first

course that students majoring in Computer Science take; no previous programming experience is required. Half of the course is devoted to learning elementary procedural programming (input, output, conditional statements, loops, methods, and working with lists), while the latter part is an introduction to object-oriented programming.

The course is taught using the Extreme Apprenticeship method (Vihavainen et al. 2011) that emphasizes the use of best programming practices, constant practise, and bi-directional feedback between the student and instructor. Constant practice is made possible with the use of a large amount of programming assignments; the students work on over 100 assignments, from which some have been split into multiple steps (a total of 170 tasks) – over 30 tasks are completed during the first week. The bi-directional feedback is facilitated both in aided computer labs, where students work on course assignments under scaffolding from course instructors and peers, as well as by a plugin called Test My Code (TMC) (Vihavainen et al. 2013), which is installed to the NetBeans programming environment that is used throughout the course. TMC provides guidance in the form of textual messages that are shown as students run assignment-specific tests; the detail of the guidance depends on the assignment that the student is working on, as well as on the goal of the assignment, i.e., whether the goal is to introduce new concepts (more guidance) or to increase programming routine (less guidance). In addition to guidance, the TMC-plugin gathers snapshots from the students' programming process, as long as students have given consent. Snapshots (time, code changes) are recorded every time the student saves her code, runs her code, runs tests on the code, or submits the code for grading.

The dataset we used for the study has 101 students (65 male, 36 female), with the median age of 21 for male participants and 22 for female participants. Students who have either not given consent for the use of their data or have disabled the data gathering during some part of the course have been excluded from the dataset, as have students who have not participated in the very first week of the course. Additional filtering has been performed to remove subsequent identical snapshots (typically events where students have first saved their program and then executed it, or executed the program and then submitted it for grading), resulting with 63701 different snapshots (avg. 631 per student). Finally, as snapshots are related to assignment that are all automatically assessed using assessment-specific unit tests, each snapshot has been marked with correctness of the student's solution. The correctness is a value from 0 to 1, which describes the fraction of assignment specific tests this snapshot passes.

When comparing the dataset with other existing datasets such as the dataset from the Blackbox-project (Brown et al. 2014), the dataset at our disposal is unique due to (1) the scaffolding that students received during the course, (2) it provides a six-week lens on the struggles that novices face when learning to program in a course emphasizing programming activities, (3) the assignment that the student is working on is known for each snapshot, making the learning objectives visible for researchers, (4) the test results for each snapshot is available, providing a measure of success, and (5) students have worked within an "industry-strength" programming environment from the start.

4. Approach

The main goal of this paper is to study students' programming behaviour. The approaches we use in this paper are based on two unique aspects. First, we use an enhanced trace of student program construction activity. While existing studies focus on the analysis of students' final submissions, our dataset provides us with many intermediate steps which give us more data on student paths from the provided program skeleton to the correct solutions. Second, when analysing student progress and comparing the state of the code in consecutive steps, we use a deeper-level conceptual analysis of submitted solutions. Existing work has looked at each submitted program as a text and either examined student progress by observing how students add or remove lines of code (Rivers and Koedinger 2013) or attempted to find syntactic and functional similarity between students' submissions (Huang et al. 2013). We, on the other hand, are interested in the conceptual structure of submitted programs. To determine which programming concepts have been used in a specific solution, we use a concept extraction tool called JavaParser (Hosseini and Brusilovsky 2013).

```
import java.util.Scanner;
public class BiggerNumber {
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      System.out.println("Type a number:  ");
      int firstNumber = Integer.parseInt(input.nextLine());
      System.out.println("Type another number:  ");
      int secondNumber = Integer.parseInt(input.nextLine());
      if (firstNumber > secondNumber)
         System.out.println("\nThe bigger number of the
                        two numbers given was: " + firstNumber);
      if (firstNumber < secondNumber)
         System.out.println("\nThe bigger number of the
                        two numbers given was: " + secondNumber);
      else
         System.out.println("\nNumbers were equal: ");
   }
}
```

(a)

```
import java.util.Scanner;
public class BiggerNumber {
   public static void main(String[] args) {
      Scanner input = new Scanner(System.in);
      System.out.println("Type a number:  ");
      int firstNumber = Integer.parseInt(input.nextLine());
      System.out.println("Type another number:  ");
      int secondNumber = Integer.parseInt(input.nextLine());
      if (firstNumber > secondNumber)
         System.out.println("\nThe bigger number of the
                        two numbers given was: " + firstNumber);
      else if (firstNumber < secondNumber)
         System.out.println("\nThe bigger number of the
                        two numbers given was: " + secondNumber);
      else
         System.out.println("\nNumbers were equal: ");
   }
}
```

(b)

Figure 1 - Implementation of the 'Bigger Number' program: (a) first snapshot, (b) second snapshot

The JavaParser tool extracts a list of ontological concepts from source code using a Java ontology developed by PAWS laboratory1; concepts have been extracted for each snapshot.

Table 1 presents the set of concepts that are extracted by Java Parser for an assignment called 'Bigger Number' at two consecutive snapshots of a single user (Figure 1a and Figure 1b). In this assignment, the goal of the student was to create an application that asks for two numbers and then prints out the larger one of the two. More extensive description on this problem is available on the course website2. This program is tested using 3 tests that verify that output is right when the first number is smaller than the second (Test 1); the output is right when the second number is smaller than the first (Test 2); and the student does not print anything unnecessary (Test 3).

Figure 1 shows that the student first wrote a program in Figure 1a that passes Test 1 and Test 2 when the first number is smaller than the second or vice versa. However it does not pass Test 3 since it prints additional information when the second number is smaller than the first one. After receiving this feedback, the student expands the code by adding the 'else if' statement as shown in Figure 1b. Now the program also passes Test 3 since it does not print any unnecessary outputs when the numbers differ.

Snapshot	Extracted Concepts
(a)	ActualMethodParameter, MethodDefinition, ObjectCreationStatement, ObjectMethodInvocation, PublicClassSpecifier, PublicMethodSpecifier, StaticMethodSpecifier, StringAddition, StringDataType, StringLiteral, LessExpression, java.lang.System.out.println, java.lang.System.out.print, ClassDefinition, ConstructorCall, FormalMethodParameter, GreaterExpression, **IfElseStatement**, **IfStatement**, ImportStatement, IntDataType, java.lang.Integer.parseInt, VoidDataType
(b)	ActualMethodParameter, MethodDefinition, ObjectCreationStatement, ObjectMethodInvocation, PublicClassSpecifier, PublicMethodSpecifier, StaticMethodSpecifier, StringAddition, StringDataType, StringLiteral, LessExpression, java.lang.System.out.println, java.lang.System.out.print, ClassDefinition, ConstructorCall, FormalMethodParameter, GreaterExpression, **IfElseIfStatement**, **IfElseStatement**, ImportStatement, IntDataType, java.lang.Integer.parseInt, VoidDataType

Table 1 - Distinct concepts extracted by JavaParser for snapshot (a) and (b) of the program shown in Figure 1. Differences are highlighted in bold.

1 http://www.sis.pitt.edu/~paws/ont/java.owl

2 http://mooc.cs.helsinki.fi/programming-part1/material-2013/week-1#e11

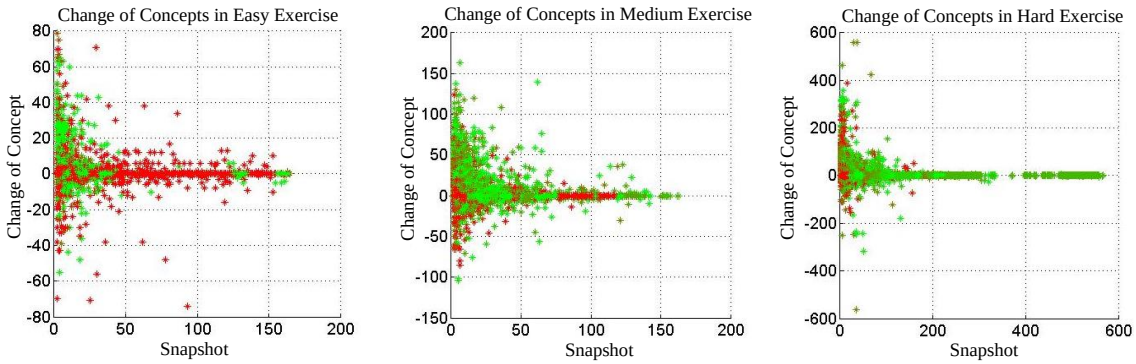
Using this tool we can examine conceptual differences between consecutive submissions – i.e., observe which concepts (rather than lines) were added or removed on each step. We can also examine how these changes were associated with improving or decreasing the correctness of program. Our original assumption about student behaviour is that students seek to develop their programs incrementally. We assume that a student develops a program in a sequence of steps adding components to its functionality on every step while gradually improving the correctness (i.e., passing more and more tests). For example, a student can start with a provided program skeleton which passes one test, then enhance the code so that it passes two tests, add more functionality to have it pass three tests, and so on. To check this assumption we start our problem solving analysis process with the global analysis that looks into common programming patterns and then followed with several other analysis approaches.

5. Analysis of Programming Paths

5.1. Global Analysis

Given the dataset that has all intermediate snapshots of users indexed with sets of concepts, we have everything ready for finding global patterns in students programming steps (snapshots). To examine our incremental building hypothesis, we started by examining the change of concepts in each snapshot from its most recent successful snapshot. We defined a successful snapshot as a program state that has the correctness either greater than or equal to the greatest previous correctness. For simplicity the concept change was measured as numerical difference between the number of concepts in two snapshots. Since student-programming behaviour might depend on problem difficulty, we examined separately the change of concepts per three levels of exercise complexity: easy, medium, and hard. The difficulty level of exercise is determined by averaging students' responses over the question "*How difficult was this exercise on a scale from 1 to 5?"* which was asked after each exercise. An easy exercise has the average difficulty level of [1.00,2.00); a medium exercise has the average difficulty level of [2.00,2.50); and a hard exercise has the average difficulty level of [2.50,5.00].

Figure 2 shows the change of concepts for all easy, medium, and hard exercises, respectively. It should be mentioned that each figure only shows change of concepts from the most recent successful snapshots. Each data point represents one snapshot taken from one user in one exercise. The horizontal position of the point represents the number of this snapshot in the sequence of snapshots produced by this user for this exercise. The vertical position of the point indicates the change in the number of concepts from the previous successful snapshot. If a point is located above the 0 axe, the corresponding snapshot added concepts to the previous snapshot (this is what we expected). If it is located below the 0 axe, concepts were removed. The colour of each point indicates the correctness of the user's program at that snapshot using red-green gradient to visualize [0-1] interval. Light green is showing the highest correctness (1) where all tests are passed and light red showing no correctness (0) where none of the tests are passed. The summary of the same data is presented in Table 2.



(a) (b) (c)

Figure 2 - Change of concepts from the most recent successful snapshot in all (a) easy, (b) medium, and (c) hard exercises for all students.

	1st Quartile			2nd Quartile			3rd Quartile			4th Quartile		
Exercise	**Total (+,-)**	**Avg ±SD**	**Cor ±SD**	**Total (+,-)**	**Avg ±SD**	**Cor ±SD**	**Total (+,-)**	**Avg ±SD**	**Cor ±SD**	**Total (+,-)**	**Avg ±SD**	**Cor ±SD**
Easy	13440 (2482/894)	2.82 ±8.04	0.49 ±0.46	462 (132,60)	2.86 ±5.84	0.09 ±0.26	223 (37,24)	1.69 ±6.04	0.05 ±0.20	107 (21,16)	1.29 ±2.39	0.24 ±0.43
Medium	20449 (6071/1526)	5.74 ±13.62	0.46 ±0.41	1182 (363,210)	8.26 ±19.93	0.45 ±0.38	287 (56,56)	2.47 ±6.70	0.46 ±0.33	74 (24,11)	2.23 ±5.42	0.70 ±0.18
Hard	31229 (10882/3935)	11.26 ±35.18	0.47 ±0.35	636 (319,65)	3.78 ±10.59	0.50 ±0.28	193 (45,44)	2.30 ±6.13	0.65 ±0.17	142 (24,14)	1.05 ±3.73	0.70 ±0.09

Table 2 - The scale of program changes at different stages of program development. Total is the total number of points in each quartile; +/-: the number of positive/negative changes; Avg,SD: the average and standard deviation of change respectively; Cor,SD: is the average and standard deviation of correctness respectively

The analysis of the data presented on Figure 2 and Table 2 leads to two important observations. First, we can see that the amount of conceptual changes from snapshot to snapshot is relatively high at the beginning of students' work on each exercise and then drops rapidly, especially for hard problems. It shows that on average significant changes are introduced to the constructed program relatively early within the first quartile of its development. To confirm this observation, we used Spearman's Rank Order correlation to determine the relationship between number of changed concepts and snapshot number for 567 snapshots, i.e. the maximum number of snapshots in student-exercise pairs. Each snapshot, contained the sum of absolute changes of concepts in the snapshot with this number for all student in all exercises. The analysis confirmed the presence of strong, statistically significant ($r_s(565) = -0.82$, $p < .000$) negative correlation between changed concepts and snapshot number.

The second observation is about the nature of changes. As the data shows, all development changes include large changes in both directions – both adding and removing concepts. It shows that our original hypothesis about incremental program expansion is not correct – there are a lot of cases when concepts (frequently many of them) are removed between snapshots. While the figure provides an impression that snapshots associated with removal tend to be less successful (more "red" in the lower part of the figure), there is still a sufficient number of successful concept-removal steps. To examine the relationship between the scale of removal and the success of result, we ran Spearman's Rank Order correlation between *the number of removed concepts* and *correctness* in 4058 snapshots which have removed concepts. The results confirmed the observation: there is a very weak, negative correlation between removed concepts and correctness, however, it is not statistically significant ($r_s(4056) = -0.03$, $p=.071$). These diagrams, however, integrate the work of many students over many exercises, so it doesn't explain why and when so many reduction steps appeared. Could it be that some specific group of exercises encourages this concept-removal approach while the remaining exercises are generally solved by progressively adding concepts? Could it be that some specific categories of students construct programs by throwing a lot of concepts in and then gradually removing? To check this hypothesis we examined adding/removal patterns for different exercises and groups of students.

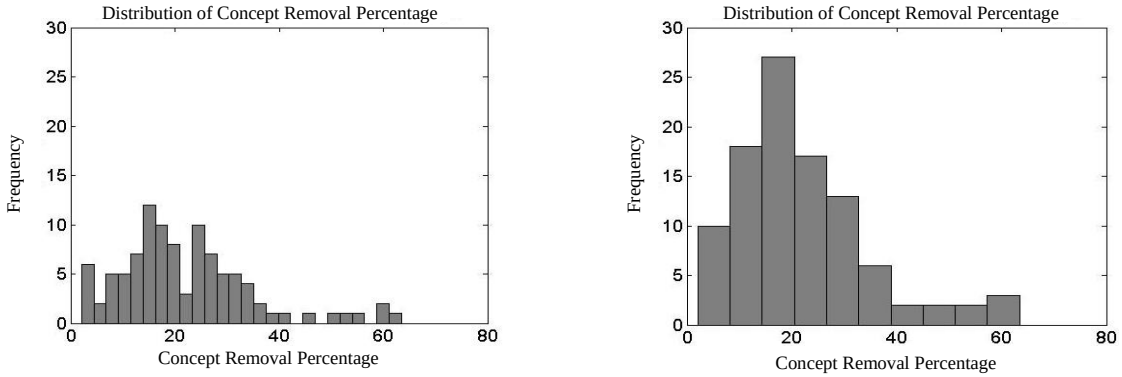


Figure 3 - The distribution of concept-removing steps over exercises. Bins represent group of exercises with same fraction of removal steps, the height of each bar represents the number of exercises in each bin. The left and right diagrams have different number of bins.

Figure 3 shows the distribution of concept-removal behaviour over problems. As the data shows some tasks do provoke larger fraction of removing actions, however, there are very few tasks on the extreme sides (very small or relatively large fraction of removals) and there are no problems with less than 3% or more than 64% of removals. Overall, the distribution shows that the majority of problems have very similar fraction of removal steps. This data hints that the frequency of removal steps is more likely caused by some student-related factor, not by the nature of specific exercises.

Since we have no personal data about students, the only student-related factor we can further examine was the level of programming skills that could be approximated by analysing the individual speed in developing the code. The presence of green dots in the first quartile indicates that some good number of students are able to pass all tests relatively fast, while the presence of the long tail (very long for hard exercises) indicates that a considerable number of students struggle to achieve full correctness for a relatively long time. That is, slower students tend to develop the program in more snapshots while faster students write program in fewer snapshots. To further investigate whether students' programming speed leads to different programming behaviours, we clustered students based on their number of snapshots on their path to success. For each exercise, we separated students into three different groups by comparing the number of their snapshots with the median of students' snapshots in that exercise. We then classified students with snapshots less than the median to be in 'cluster 1' and greater than median to be in 'cluster 2'. We ignored students with the number of snapshots exactly equal to median to improve the cluster separation.

Figure 4 shows changes of concepts per all easy, medium, and hard exercises for all students in the two clusters. Overall, this figure exhibits a similar pattern to the one observed in Figure 2. We see larger-scale changes in the beginning of program development are replaced by small changes in the middle and end of the process. We also see that students on all skill levels working with all problem difficulty levels apply both concept addition and concept reduction when constructing a program. This hints that the patterns of program construction behaviour are not likely to be defined by skill levels and that we need to examine the behaviour of individual students to understand the nature of pattern differences.

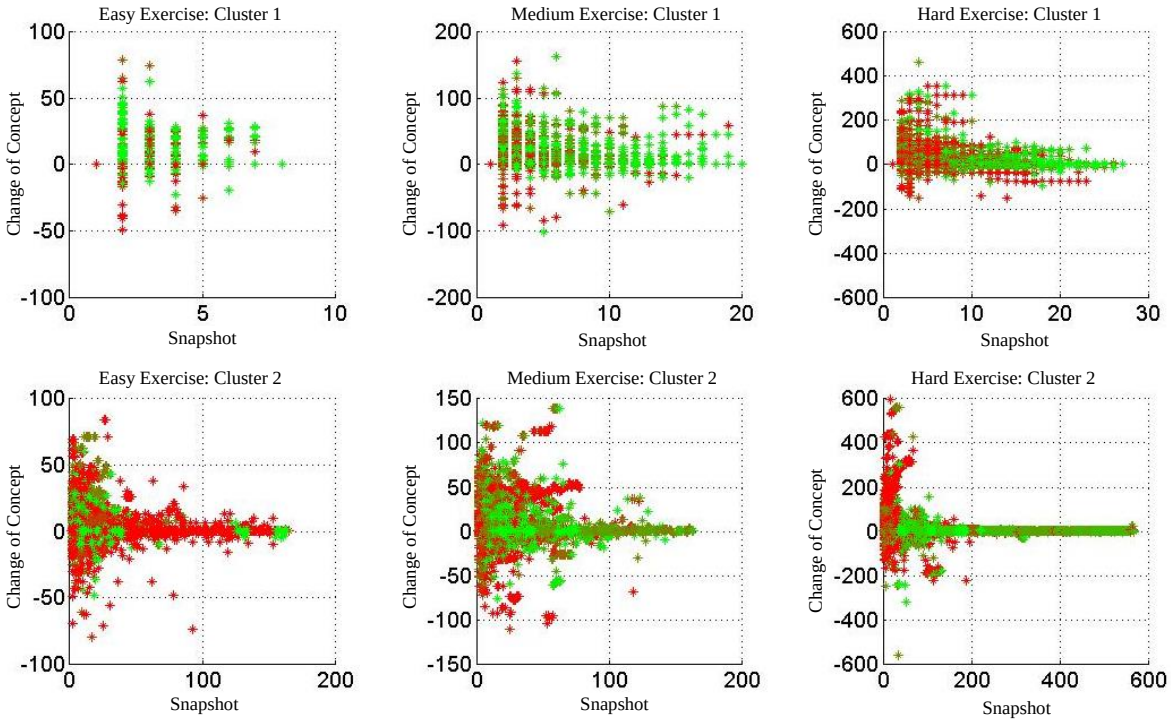


Figure 4 - Change of concepts and correctness for students with different programming speeds. Cluster 1 represents students who need less than the median number of snapshots and Cluster 2 represents students who need more than the median number of snapshots to develop the program.

5. 2. Analysis of programming behaviour on individual level

At the final analysis step, we examined programming behaviour on the finest available level exploring the behaviour of individual students on a set of selected assignments. The analysis confirmed that most interesting differences in problem solving patterns could be observed on the individual level where it likely reflects differences in personal problem solving styles. Figure 5 shows several interesting behaviour patterns we discovered during the analysis. We call the students exhibiting these specific patterns Builders, Massagers, Reducers and Strugglers. This classification is inspired by the work of Perkins et al. (1986), who categorized students as "stoppers", "movers" and "tinkerers;" however, in our dataset, the students rarely "stop" or give up on the assignment, and even the tinkering seems to be goal-oriented. Among these patterns, Builders that roughly correspond to "movers" behave exactly as expected; they gradually add concepts to the solution while increasing of correctness of the solution in each step (Figure 5a). In many cases we observe that students who generally follow a building pattern have long streaks when they are trying to get to the next level of correctness by doing small code changes without adding or removing concepts in the hope of getting it to work (Figure 5b). We call these streaks as code massaging and students who have these streaks as Massagers.

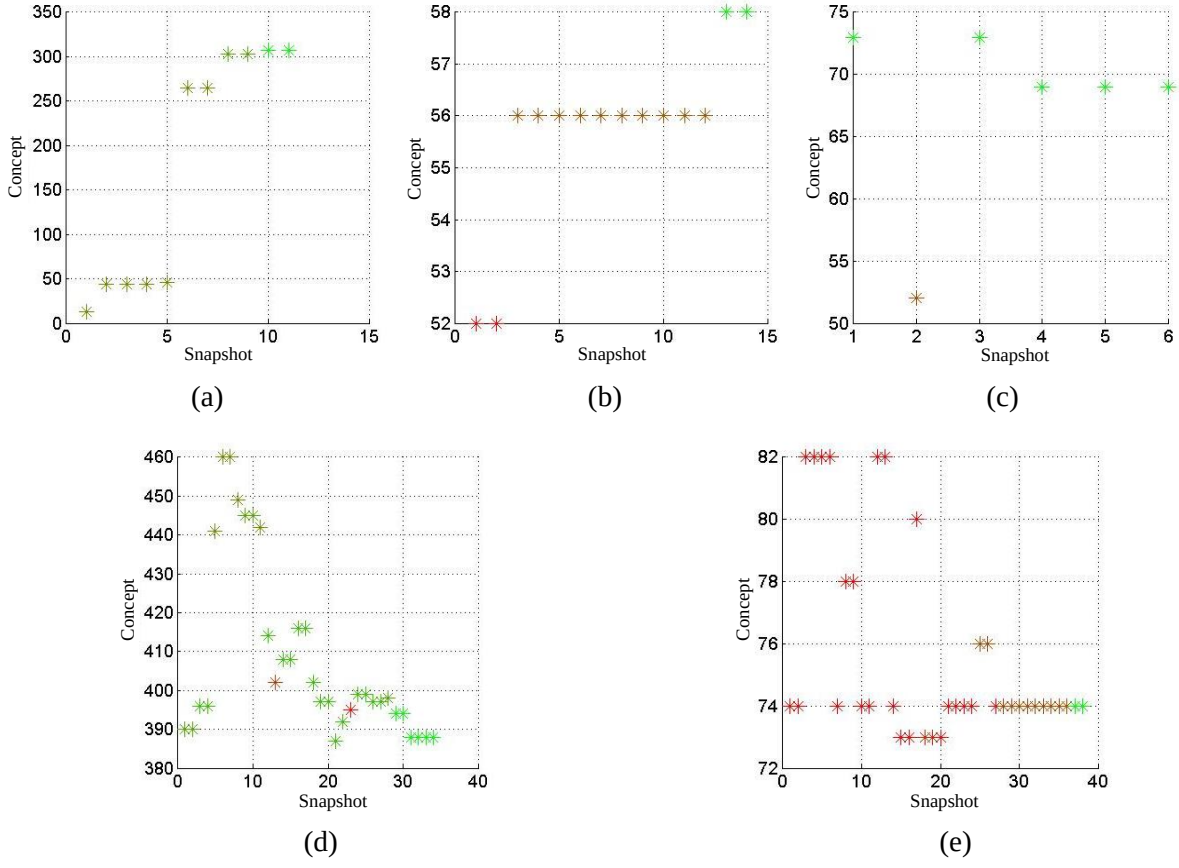


Figure 5 - Patterns of individual programming behaviour for different students. The X-axis represents the snapshot number and the Y-axis represents the number of concepts in the snapshot. The colour of each point ranges from green to red representing increase or decrease in the correctness compared to the previous successful snapshot, respectively. (a) A Builder who solves a hard problem incrementally by adding concepts and increasing the correctness; (b) A Massager who has long streaks of small program changes without changing concepts or correctness levels; (c) A Reducer who tries to reduce the concepts after reaching the complete correctness in a medium exercise; (d) A Reducer who reduces concepts and gradually increases correctness in a hard exercise (e); A Struggler who struggles with the program failing to pass any tests for a long time.

We can also observe behaviour opposite to building when students remove concepts while maintaining or reducing the correctness level. In many cases reducing steps could be found at the end of regular building stage where it apparently represents the attempts to make the code more efficient (Figure 5c). However, we also identified some fraction of students who use concept reduction as the main problem solving approach. These students start by throwing in large amounts of code and then gradually work by reducing the set concepts until reaching the correct solutions (Figure 5d). Massagers and Reducers could be probably considered as a mixture of Perkins' movers and tinkerers. Finally, we identified a considerable fraction of students we call Strugglers (Figure 5e). These students spend considerable amounts of time to pass the first correctness test. They do all kinds of code changes, but probably have too little knowledge to get the code working. We probably could consider these students as a mix of tinkerers and stoppers; they may freeze on a problem (not going towards the "correct" solution) for a long while, but through experimentation and movement typically finally end up getting the solution right.

The observed global patterns provide interesting insight into the variety of student problem behaviour, yet it is very hard to use them in the observed form to classify students in clear-cut categories and analyse how frequently these patterns can be found in student behaviour. To recognize students with different kinds of behaviour we decided to perform micro-analysis of student behaviour. Starting with the main patterns of programming behaviours we observed, we decided to explore the frequency of such patterns inside programming steps of each user in each exercise. Instead of classifying the whole student path as Builder, Massager, Reducer, or Struggler, we started by classifying each problem solving step with these labels based on the change of concept and correctness in each snapshot. Table 3 lists the labels used for the labeling process. Next, we applied sequential pattern mining to identify the most common problem solving patterns among the 8458 students-exercise pairs that we had in the dataset. Specifically, we used software called PEX-SPAM which is an extension of SPAM algorithm (Ayres et al. 2002) that supports pattern mining with user-specified constraints such as minimum and maximum gap for output feature patterns. The tool was originally developed for biological applications but it can be easily employed to extract frequent patterns in other domains, too (Ho et al. 2005). Table 4 summarizes the common patterns obtained by PEX-SPAM with the support (the probability that the pattern occurs in the data) greater than 0.05.

The table shows interesting results related to the frequency of specific patterns. The first important observation is the dominance of the Builder pattern. In particular, the most common retrieved pair is 'BB' with support of 0.304 indicating that this pattern occurs in about ⅓ of the student-exercise pairs. This indicates that our original assumption of regular program building is not really that far from reality. While only a fraction of students could be classified as straightforward Builders, streaks of incremental building represent most typical behaviour. Another frequent kind of pattern is struggling sequences that may or may not end with the building step. It is also interesting to observe that reducing steps are rarely found inside frequent sequences.

To further investigate the most dominant programming pattern inside each of the 8458 students-exercise pairs, we grouped each of these pairs based on their most frequent pattern(s). Table 5 shows the results of the grouping and lists the frequency and percentage of each group. According to this table, about 77.63% of the students are Builders who tend to add concepts and increase the correctness incrementally.

Correctness\Concepts	**Same**	**Increase**	**Decrease**
Zero	Struggler	Struggler	Struggler
Decrease	Struggler	Struggler	Struggler
Increase	Builder	Builder	Reducer
Same	Massager	Builder	Reducer

Table 3 - Labeling sequence of students' snapshots based on change of concept and correctness

Pattern	Support	Pattern	Support
BB	0.304	SSB	0.123
SB	0.281	SSSS	0.088
BS	0.215	BSS	0.083
SS	0.199	SSSB	0.07
SSS	0.128	SSSSS	0.067
BR	0.127	BBBB	0.063
BBB	0.123	BBS	0.058

B: Builder, S: Struggler, R: Reducer

Table 4 - Common programming patterns with support greater than 0.05.

Group	Frequency	Percentage	Group	Frequency	Percentage
B	6566	77.63	BM	60	0.71
BS	1084	12.82	BMS	26	0.31
BR	309	3.65	BRMS	11	0.13
R	173	2.05	RS	11	0.13
BRM	135	1.60	M	11	0.12
BRS	70	0.83	MS	2	0.02

B: Builder, S: Struggler, R: Reducer, M: Massager

Table 5 - Details of dominant group(s) in student-exercises pairs

6. Discussion & Conclusion

To explore behaviour patterns found along student problem paths, we performed macro- and micro-level analysis of student program construction behaviour. Contrary to our original hypothesis of student behaviour as incremental program building, by adding more concepts and passing more tests, the macro view of student behaviour indicated a large fraction of concept reduction steps. Further analysis demonstrated that both building and reduction steps happen in all stages of program development and across levels of problem difficulty and student mastery. The analysis of program building patterns on the micro-level demonstrated that while the original building hypothesis is not strictly correct, it does describe the dominant behaviour of a large fraction of students. Our data indicate that for the majority of students (77.63%) the dominant pattern is building, i.e. incrementally enhancing the program while passing more and more tests. At the same time, both individual pattern analysis and micro-level pattern mining indicated that there are a number of students who do it in a very different way. Few students tend to reduce the concepts and increase the correctness (2.05%) or they might have long streaks of small program changes without changing concepts or correctness level (0.12%). There are also some students who show different programming behaviour, showing two or more patterns (20.2%), hinting that students' behaviour can change over time. Apparently, the original hypothesis does not work for everyone, but it does hold for a considerable number of students. Therefore, we have now enough evidence to conclude that there exists a meaningful expansion programming that implies starting with a small program, and then increasing its correctness by incrementally adding concepts and passing more and more tests in each of the programming steps.

In future work, we would like to see to what extent the introduced programming behaviours are stable over different datasets and even in different programming domains. It would be also interesting to see how information such as age, gender and math performance, as well as many of the traits that have been previously used to predict programming aptitude influences programming behaviour. We believe

having such step-wise analysis of student programming could enable us to have more advanced and accurate user modelling.

8. References

Ayres, J., Flannick, J., Gehrke, J., & Yiu, T. (2002, July). Sequential pattern mining using a bitmap representation. In Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 429-435). ACM.

Benford, S., Burke, E., & Foxley, E. (1993). Learning to construct quality software with the Ceilidh system. Software Quality Journal, 2(3), 177-197.

Bennedssen, J., & Caspersen, M. E. (2008, September). Abstraction ability as an indicator of success for learning computing science?. In Proceedings of the Fourth international Workshop on Computing Education Research (pp. 15-26). ACM.

Bergin, S., & Reilly, R. (2005, June). The influence of motivation and comfort-level on learning to program. In Proc. 17th Workshop of the Psychology of Programming Interest Group (pp. 293-304).

Bornat, R., & Dehnadi, S. (2008, January). Mental models, consistency and programming aptitude. In Proceedings of the tenth conference on Australasian computing education-Volume 78 (pp. 53-61). Australian Computer Society, Inc.

Brown, N. C., Kölling, M., McCall, D., & Utting, I. (2014, March). Blackbox: A Large Scale Repository of Novice Programmers' Activity. In Proceedings of the 45th ACM technical symposium on Computer science education (pp. 223-228). ACM.

Douce, C., Livingstone, D., & Orwell, J. (2005). Automatic test-based assessment of programming: A review. Journal on Educational Resources in Computing (JERIC), 5(3), 4.

Edwards, S. H., & Perez-Quinones, M. A. (2008, June). Web-CAT: automatically grading programming assignments. In ACM SIGCSE Bulletin(Vol. 40, No. 3, pp. 328-328). ACM.

Evans, G. E., & Simkin, M. G. (1989). What best predicts computer proficiency?. Communications of the ACM, 32(11), 1322-1327.

Fincher, S., Robins, A., Baker, B., Box, I., Cutts, Q., de Raadt, M., ... & Tutty, J. (2006, January). Predictors of success in a first programming course. In Proceedings of the 8th Australasian Conference on Computing Education-Volume 52 (pp. 189-196). Australian Computer Society, Inc..

Ginat, D. (2003, June). The novice programmers' syndrome of design-by-keyword. In ACM SIGCSE Bulletin (Vol. 35, No. 3, pp. 154-157). ACM.

Ho, J., Lukov, L., & Chawla, S. (2005). Sequential pattern mining with constraints on large protein databases. In Proceedings of the 12th International Conference on Management of Data (COMAD) (pp. 89-100).

Hosseini, R., & Brusilovsky, P. (2013, June). JavaParser: A Fine-Grain Concept Indexing Tool for Java Problems. In The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013) (p. 60).

Huang, J., Piech, C., Nguyen, A., & Guibas, L. (2013, June). Syntactic and functional variability of a million code submissions in a machine learning mooc. In AIED 2013 Workshops Proceedings Volume (p. 25).

Jadud, M. C. (2005). A first look at novice compilation behaviour using BlueJ. Computer Science Education, 15(1), 25-40.

Mitchell, C. M., Boyer, K. E., & Lester, J. C. (2013, June). When to Intervene: Toward a Markov Decision Process Dialogue Policy for Computer Science Tutoring. In The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013) (p. 40).

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1986). Conditions of learning in novice programmers. Journal of Educational Computing Research, 2(1), 37-55.

Piech, C., Sahami, M., Koller, D., Cooper, S., & Blikstein, P. (2012, February). Modeling how students learn to program. In Proceedings of the 43rd ACM technical symposium on Computer Science Education (pp. 153-160). ACM.

Rivers, K., & Koedinger, K. R. (2013, June). Automatic generation of programming feedback: A data-driven approach. In The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013) (p. 50).

Rowan, T. C. (1957). Psychological Tests and Selection of Computer Programmers. Journal of the Association for Computing Machinery, 4, 348-353.

Thomas, L., Ratcliffe, M., Woodbury, J., & Jarman, E. (2002, February). Learning styles and performance in the introductory programming sequence. In ACM SIGCSE Bulletin (Vol. 34, No. 1, pp. 33-37). ACM.

Tukiainen, M., & Mönkkönen, E. (2002). Programming aptitude testing as a prediction of learning to program. In Proc. 14th Workshop of the Psychology of Programming Interest Group (pp. 45-57).

Vee, M. H. N. C., Meyer, B., & Mannock, K. L. (2006). Understanding novice errors and error paths in object-oriented programming through log analysis. In Proceedings of Workshop on Educational Data Mining at the 8th International Conference on Intelligent Tutoring Systems (ITS 2006) (pp. 13-20).

Vihavainen, A., Paksula, M., & Luukkainen, M. (2011). Extreme apprenticeship method in teaching programming for beginners. In Proceedings of the 42nd ACM technical symposium on Computer science education (pp. 93-98). ACM.

Vihavainen, A., Vikberg, T., Luukkainen, M., & Pärtel, M. (2013, July). Scaffolding students' learning using test my code. In Proceedings of the 18th ACM conference on Innovation and technology in computer science education(pp. 117-122). ACM.

Watson, C., Li, F. W., & Godwin, J. L. (2014, March). No tests required: comparing traditional and dynamic predictors of programming success. In Proceedings of the 45th ACM technical symposium on Computer science education (pp. 469-474). ACM.

White, G., & Sivitanides, M. (2003). An empirical investigation of the relationship between success in mathematics and visual programming courses. Journal of Information Systems Education, 14(4), 409-416.

Winslow, L. E. (1996). Programming pedagogy—a psychological overview. ACM SIGCSE Bulletin, 28(3), 17-22.