Control-Flow-Only Abstract Syntax Trees for Analyzing Students' Programming Progress

David Hovemeyer
York College of Pennsylvania
dhovemey@ycp.edu

Arto Hellas
University of Helsinki
avihavai@cs.helsinki.fi

Andrew Petersen
University of Toronto
Mississauga
petersen@cs.toronto.edu

Jaime Spacco
Knox College
jspacco@knox.edu

ABSTRACT

The abstraction of student code for use in automated analysis is a key challenge. The code must be processed in a manner that reveals interesting properties while reducing the "noise" introduced by less important details. In this work, we investigate the importance of control flow as a property in the analysis of students' programming processes.

We introduce the Control-Flow Abstract Syntax Tree (CFAST), a representation of source code that focuses on control structures, and apply it in an analysis of a large, multinational dataset of introductory programming exercises. We provide data on how many different control-flow designs are observed, how quickly students identify a "correct" control flow structure, and how much additional work is required to convert a correct structure to a correct program.

Our results indicate that while even simple problems yield a surprising number of CFASTs, the work of most students can be mapped to a small number of CFASTs. CFASTs that map to fewer submissions tend to be larger, and more complex, but they can still correspond to "correct" solutions to the problem.

1. INTRODUCTION

Modern learning environments are recording an increasingly detailed view of student working behaviours, resulting in a veritable gold mine for researchers. This is particularly true in courses such as introductory programming, where the emergence of easy-to-adopt automatic assessment systems and the need for students to spend considerable time working on practice assignments have combined to produce significant amounts of data about novice programmers. As a result, educational data mining and learning analytics in the context of computer science education has grown significantly over the last decade [19].

With the increase of both data granularity and quantity, researchers face a question about what level of detail should be retained during analysis. Snapshots of programming exercises can, for example, be represented simply with numeric information (a timestamp and a score), as a string representing the submitted code, or in a format that encodes information about the structure of the solution, like an abstract syntax tree. In any case, the chosen granularity plays a large role in the quantity of explorable states [50], and it may even be the case that too fine-grained data simply introduces noise to the analysis process.

In this work, we explore the use of control-flow focused representations in analysis of novice programmer data. Our intuition is that control flow — conditions, decisions, and loops — forms the essential structure of any algorithm expressed in an imperative programming language. Selecting and interleaving appropriate plans to produce a workable control structure is a challenging task [43, 44] and is necessary (although not sufficient) to solve any nontrivial programming problem [16, 26].

We introduce a variant of abstract syntax trees called a Control-Flow Abstract Syntax Tree (CFAST). A CFAST only includes control flow elements such as `while`, `if` and `for` and enclosing contexts sufficient to maintain sequence and nesting relationships between the control elements. Our goal, then, is to assess the usefulness of control flow structure in categorizing novice programmer submissions to programming exercises. In particular:

RQ 1. Do CFASTs encode useful information about student programming behaviour?
- 1.1 Do the solutions map to a small number of CFASTs?
- 1.2 How do students add control elements to code?
- 1.3 If the control structure in a submission is correct, does it remain correct in subsequent submissions?

RQ 2. Can CFASTs be used to identify students in difficulty?
- 2.1 At what point in their programming process do students reach a CFAST that corresponds with a correct solution?
- 2.2 Do sequences of CFASTs encode evidence of ineffective programming behaviours?

This article is organized as follows. In Section 2 we review the body of work related to how students construct computer programs and focus on work related to schemas, plans and

ICER '16, September 08-12, 2016, Melbourne, VIC, Australia
© 2016 ACM. ISBN 978-1-4503-4449-4/16/09…$15.00
DOI: http://dx.doi.org/10.1145/2960310.2960326

plan composition. In Section 3, we describe the data sources used in our analysis and discuss the CFAST structure in more detail. We outline the results from our evaluation of CFAST structures in Section 4 and highlight key findings in more detail in Section 5. Finally, in Section 6, we conclude the article and outline potential future work.

2. RELATED WORK

In their influential review of computer science education research, Robins et al. suggest that "the distinction between effective and ineffective novices" is of central importance [34]. Our work adds to the body of knowledge in this area by exploring whether control-flow representations of novice programmer code provide insights into programming ability or evidence that a student is in difficulty.

2.1 Schemas and Plans

In the mid 1980's, Spohrer and Soloway identified several common errors, several of which involved control structures, and argued that these errors were caused by issues with selecting appropriate *plans* (or *schemas*) for solving problems and interleaving such plans [43, 44]. Bellamy and Gilmore, in contrast, argue that programmers frequently use other representations, including control-flow based representations, when solving problems [3]. Regardless, work on schema acquisition suggests that programmers demonstrate "backwards development" as plans are being formulated and "forward development" when previously stored plan schema are accessed and implemented [12, 29]. More recent work on problem solving techniques used by a small number of high school aged students found a similar preference for bottom-up strategies of problem solving [21]. This work suggests that an analysis of a sequence of control flow representations of programs, if the data is sufficiently fine-grained, may be able to identify when a student is formulating a new plan, rather than implementing a previously observed plan.

Soloway introduced the Rainfall problem as an example of a simple problem that required interleaving of plans [40]. A successful solution to Rainfall reads integers from standard input and, after all integers have been entered, outputs the average, with some variants also requiring that negative numbers be identified as errors or that additional output be generated. In particular, since the number of integers to be entered is not pre-defined, the problem requires interleaving plans that involve control (sentries and repetition).

There has been a recent resurgence of research on Rainfall and, more generally, in plan composition [14, 15, 23, 37, 38]. In 2013, Simon noted that the Rainfall problem appeared to have become more difficult and suggested that unfamiliarity with input-based loop control patterns may be the cause [38]. Some later researchers found that the problem was no more difficult, with Fisler arguing that the method of instruction, including teaching certain control-based patterns, might explain the difference in results [14]. Seppala et al. argued that differences in observed difficulty could be attributed to varying formulations of the problem and levels of experience in participants [37]. Lakanen et al. did not comment on difficulty but found that most students could design a program with a viable high-level control structure — one that reflected the control flow taught in class to deal with data from an array — but often failed to deal with edge cases [23].

Other modern research on plans and plan composition have proposed a shift away from Rainfall and other IO-driven problems [15], but control flow structures remain an important feature of the problems being studied. Many errors in plan composition resulting in malformed or inappropriately structured control [16, 26], and analysis of control flow will be able to some, but not all of these, invalid programs. For example, errors that introduce (or omit) control structures could be identified with an analysis of control structures in a program, but errors that result in a different condition (but the same structure) require a more precise analysis.

Building on this potential to differentiate between correct and invalid program designs, Luxton-Reilly et al. proposed using control flow as the first level of a taxonomy designed to categorize student solutions to coding problems [24]. They used control flow graphs (CFGs) to represent control in programs. The taxonomy was evaluated by asking CS1 students to provide solutions to a set of ten problems (five with conditionals and five with loops). They found that even submissions to the simplest problems exhibited large numbers of CFGs (6-58, with half containing more than 50), including a large number of "correct" CFGs. For many problems, there was a clear "most popular answer", but several exercises featured a number of CFGs with comparable numbers of correct submissions, and several bins contained only a single, "correct" submission. These observations demonstrate significant variation in the complexity of student submissions, even for simple problems, and suggest that there might be value in evaluating the "quality" of a control flow representation, such as a CFG, by comparing it to a reference solution.

2.2 Novice Programmers and Control

Researchers have also examined control structures separate from schemas and plans to identify the roots of misconceptions commonly held by novice programmers. In the early 1980's, Soloway et al. identified student strategies for solving loop problems [41], and related studies by Bonar and Soloway suggested that misconceptions of the "while" loop were based on natural language interpretations of the word "while" [4, 5]. They suggested that novice programmers have less problems with post-test loops, since the natural interpretation of "until" better matches the actual execution of such loops.

This result has been confirmed by several independent groups. As part of the "commonsense computing" series, Simon et al. investigated what students with no experience programming know about programming concepts and found that they prefer post-test loops [39], and VanDeGrift et al. found that students performed better when both sides of a conditional were explicitly defined [49]. Separately, Craig et al. looked for evidence of "natural" representations of programming concepts in knitting patterns [9]. They confirmed a preference for post-test loops and identified a preference for terminating ("stop when"), rather than continuing ("continue while"), conditions in loops. Stefik et al. focused on word choice and noted that non-programmers rated the most common keywords for loops (including "for" and "while") as the most unintuitive and the word "repeat" as the most intuitive [45, 46].

Loop problems feature prominently in research on student understanding of programming concepts, since they are a major, challenging component of CS1 courses [18, 48] and are a common source of errors [36]. Robins et al. reviewed questions asked by students during a lab and noted that while conditionals and loops generated many questions, students improved in their ability to solve such problems over time [33]. This contrasts with questions about reference types and scope, which remained at a high volume throughout the course. In contrast, later work by Cherenkova et al. mined student submissions to programming exercises and found control structures — both conditionals and loops — to be particularly challenging and to remain challenging relative to other course topics [8]. However, since the exercises were featured in a Python course that taught objects late, it is possible that the topics identified by Robins et al. as being more difficult were simply not encountered.

2.3 Code Mining and Analytics

Ihantola et al., an ITiCSE working group, surveyed the state of educational data mining in 2015 and concluded that there has been a significant increase, over the past decade, in the number of papers that use data mining techniques to study computer science education [19]. Their work contains a comprehensive survey of papers from 2005–2015 that, like this paper, mine large sets of programming data to better understand novice programmers.

One thread of research in this area is focused on finding heuristics to identify students who are frustrated [35] or to predict novice programmer performance [20, 27, 47, 52]. Watson et al. claimed in 2014 that programming behaviour is an effective predictor of student performance in CS1 and showed that their algorithm outperformed various traditional predictors of performance [51].

Much of the early work in this area relies on the insight that syntactic errors are a significant issue for novice programmers [10, 11, 25, 46]. The most prominent early work in this area was performed by Jadud, who suggested that sequential submissions featuring syntactic errors was an indicator of a student in need of assistance [20]. Later work, by Spacco et al., examined submission traces and identified evidence of students *flailing* and increased ability, over the course of the term, in producing syntactically correct programs [42].

However, other recent work has expanded the set of features in an attempt to model the student programming process. Carter et al. explicitly extended previous work to model programming process as a state machine [6, 7]. Others have started to incorporate semantic correctness, as evaluated by tests: Koprinska et al. used submission times as well as improvement over a sequence of submissions [22], and Ahadi et al. identified students at risk after just one week of the term using the number of steps required to reach a correct solution and the the number of tests passed, among other non-programming exercise factors [1]. Pettit et al. used traditional software metrics like cyclomatic complexity to describe how student submissions to coding exercises evolved over time [28]. In total, these projects suggest that other features, like control-flow structure, must be integrated into our models of student behaviour and performance.

Another thread of research examines methods for using student submissions to provide feedback. Glassman et al. clusters submissions and proposes providing examples from other clusters [17]. The proposed clustering relies on a number of features, including structural (control-based) features. Other groups provide hints by comparing a novice's current submission to a reference solution [2], by transforming submissions into a canonical abstract syntax tree (AST) form and comparing to other student submissions [30–32], and finding related questions and study material based on patterns in submitted problems [13]. These efforts are examples of potential applications of control-focused mining of student submissions.

	Total #	Concepts addressed		
Course	activities	if	loops	both
1	9	0	4	5
2	9	5	2	2
3	9	4	5	0

Table 1: Summary of activities analyzed in each course

Function template and PyDoc

```
def insert(lst, v):
  """(list of int, int) -> NoneType
  Insert v into lst just before the rightmost
  item greater than v, or at index 0 if no
  items are greater than v.

  >>> my_list = [3, 10, 4, 2]
  >>> insert(my_list, 5)
  >>> my_list
  [3, 5, 10, 4, 2]
  >>> my_list = [5, 4, 2, 10]
  >>> insert(my_list, 20)
  >>> my_list
  [20, 5, 4, 2, 10]
  """
```

Python source code

```
def insert(lst, v):
  if v > max(lst):
    lst.insert(0, v)
  else:
    lst.reverse()
    for i in range(len(lst)):
      if (v < lst[i]):
        lst.insert(i, v)
        break
```

CFAST

```
FunctionDef
  If
  Else
    For
      If
        Break
```

Figure 1: Example exercise consisting of a Python function template and its PyDoc, source code from a sample student submission, and the corresponding CFAST

3. METHODOLOGY

3.1 Data Sources

We have prepared data from CS1 courses taught at three very different institutions. Course 1 was taught at the University of Toronto and Course 2 was taught at the University of Helsinki, both large research universities. Course 3 was taught at York College of Pennsylvania, a small, undergraduate focused college. The first course was taught in Python, the second in Java, and the third in C. All three courses assumed students were entering with little to no programming experience and explicitly taught conditionals and loops in the first half of the course.

Each data set consists of student submissions to a number of programming activities, which range from short exercises

Course/ Activity	Avg. # lines	# distinct (w/ correct)	Avg. subs per student	% in top 20% CFASTs	% in CFAST w/ most correct subs	# CFASTs for 90%	Total correct submissions
1/37	6.9	341 (101)	4.6	89.3	47.7	24	960
1/39	5.6	40 (9)	3.1	98.3	86.1	2	826
1/45	6.8	190 (43)	5.9	95.0	79.9	4	927
1/47	11.5	979 (207)	6.3	75.4	07.9	121	866
1/48	5.9	86 (14)	3.8	97.1	82.1	1	1044
1/59	9.7	239 (45)	5.2	95.1	63.9	6	787
1/63	8.4	491 (143)	8.0	83.2	26.1	73	704
1/64	5.8	180 (69)	3.9	90.2	35.9	16	850
1/84	21.4	232 (97)	2.8	88.1	10.8	35	876
2/018	13.3	7 (3)	6.8	98.9	93.1	4	414
2/021	20.8	12 (5)	6.3	96.6	49.4	6	402
2/023	12.7	5 (3)	6.5	95.6	95.6	3	399
2/024	15.5	17 (8)	16.9	85.7	36.9	9	405
2/026	18.1	15 (7)	7	94.1	57.6	7	376
2/027	17.5	36 (7)	10	93.4	86.8	8	293
2/029	44.5	142 (25)	29.5	78.6	33.9	25	356
2/035	15.0	27 (6)	13.8	94.4	88.9	6	149
2/041	22.7	96 (26)	29.7	69.6	17.4	28	226
3/111222333444	13.0	39 (5)	8.5	93.9	78.8	2	76
3/bananana	9.3	9 (3)	4.3	98.6	98.6	1	87
3/checkinput	12.2	32 (8)	7.4	90.9	78.8	2	72
3/doublecoupon	11.7	9 (4)	3.9	36.0	53.5	2	110
3/keepdoubling	12.5	22 (10)	10.6	89.2	74.3	4	87
3/memberdiscount	19.4	31 (12)	5.4	84.4	37.7	6	92
3/restaurant	12.3	18 (7)	3.6	80.7	74.7	4	101
3/squares	13.3	22 (9)	11.8	87.3	69.8	4	70
3/triplecoupon	16.0	12 (7)	4.2	71.8	47.1	4	105

Table 2: Summary statistics for CFASTs in each activity: average number of lines of non-blank/non-comment lines of code submitted by students for each exercise, number of distinct CFASTs (and size of subset containing any correct submissions), avg. number of submissions per student, percentage of students whose final CFAST was in the top 20% most frequently observed CFASTs, percentage of students whose final submission was in the single CFAST with the most correct submissions, number of CFASTs needed to account for 90% of all submissions, and total number of correct submissions

to more substantial programming problems. All of the activities were presented to students using integrated programming environments in which students are able to submit answers to coding problems an unlimited number of times and receive feedback from a suite of automated tests after each submission. Therefore, each submission represents an explicit decision by the student to submit her code for automated testing, and encodes the time of submission, code submitted, and results of the tests.

The data sets do not include all of the problems presented to students. We selected the required exercises that focus on control-flow topics, omitting optional exercises, exercises with no control flow, and exercises that focused on more advanced topics such as classes or reference parameters. Table 1 summarizes the activities that were selected.

It's clear, from the topics addressed, that the programming exercises were very different, and these differences reflect the intended use of the exercises. In Course 1, the activities were intended to be challenging, formative feedback, with integration of concepts highlighted. In contrast, in Course 2 assignments were typically first introduced as isolated practice assignments, and then integrated together, and in Course 3, the exercises were designed to evaluate basic proficiency in individual concepts in the course, without integration.

3.2 Constructing CFASTs

For each student code submission, we construct a *CFAST*, meaning "Control-Flow only Abstract Syntax Tree". Prior work has focused on control flow graphs (CFGs) [24], which are typically constructed for individual functions and encode all possible control paths through the function. In contrast, CFASTs can be easily constructed over entire programs, and rather than encoding control paths, they reflect *control structure*: the specific syntactic elements chosen to implement the control and how they are sequenced and nested.

To construct a CFAST, we use a standard library to parse a program and generate a full abstract syntax tree (AST). Submissions which cannot be parsed (for example, because of a syntax error) cannot be analyzed. Then, we walk the resulting AST, pruning nodes that do not represent control-flow structures. Note that, for this analysis, the conditions used to direct control are also discarded. They could be retained in future work, but for the purposes of this study, we chose to focus solely on control structure.

For example, when constructing a CFAST for a C language submission, the only AST nodes retained are if and else statements; loop statements; and break, continue, and return. Block nodes are used to contain a sequence of statements nested in a function definition, loop, or if, and are retained only when they contain control structures. The resulting structure maintains structural relationships (sequences and nesting) as well as the specific keywords chosen to implement control in the program. Figure 1 shows an example of a Python program and its corresponding CFAST. Indentation is used to indicate the tree structure of the CFAST.

Table 2 lists summary statistics about the CFASTs of student submissions in each activity.

3.3 Threats to Validity

This study is necessarily limited by the type of data used and the methods applied. We selected data from a range of institutions, but the topics and formats of questions that generated the data are limited. Our programming assignments are typically small exercises from the early weeks of CS1 courses that feature an imperative programming approach. In addition, the exercises selected were built to focus on control topics and do not contain structures introduced later, like classes.

As we rely on quantitative methods applied to anonymous submissions, our analyses are blind to the individual contexts of the students represented in the data set. In particular, we cannot know the thought processes that lead to the events we have observed. Our hypotheses on individuals struggling with specific constructs are *interpretations* based on experience, rather than evidence-based observations. Similarly, we do not know why students submitted their code. While we expect that the majority of submissions were attempts to obtain feedback or to submit a potentially complete solution, it's possible that the student was choosing, for example, to store the solution on the server for future work or unknowingly submitted several times due to an unresponsive network connection.

Finally, when contrasting the CFAST path lengths to course outcomes, we chose to use exam scores as a proxy of knowledge or skill despite knowing that it is a rather poor metric. It was selected as being the only supervised evaluation of student ability that was available from all three institutions.

4. RESULTS

4.1 Evaluating RQ 1

In RQ 1, we were interested in finding out whether CFASTs were a useful lens through which to understand student code, both statically and over time. To make this question more concrete, we identified three specific components, 1.1–1.3.

4.1.1 RQ 1.1

RQ 1.1 asks "Do student solutions converge on similar or identical CFASTs?" We approached this question in two ways.

For each activity, we counted the percentage of students whose last submission was (a) in the top 20% of most frequently observed CFASTs, and (b) in the single CFAST with the most correct submissions. The fifth and sixth columns of Table 2 show the results.

From this data, it is clear that for all activities, a large majority of all submissions fall under a relatively small number of CFASTs. This confirms the hypothesis that most students will use similar or identical control flow structures when solving the same problem. We also see that for *many* activities, the CFAST with the most correct submissions contains a substantial percentage of all correct submissions. Anecdotally, we found that the single CFAST with the most correct submissions tended to correspond to the "obvious" control structure. However, we do see some notable exceptions: for example, activities 1/47, 2/041, and 3/memberdiscount all had a significant degree of variation in the CFASTs of correct solutions. Significantly, all of these exercises included chained or independent decisions.

To further investigate the amount of variation in correct solutions, we counted how many CFASTs were required to account for 90% of all correct solutions for each programming activity. This data is shown in the seventh column of Table 2. Again, in general a relatively small number of control-flow structures accounts for most correct submissions, but we do see certain activities leading to a greater than usual variety of correct solutions, such as 1/47.

Course/Activity	(a)	(b)	(c)	(d)	(e)
1/37	54.1	43.5	43.5	45.9	82.2
1/39	76.2	52.5	52.5	23.8	88.5
1/45	79.6	42.9	42.9	20.4	83.0
1/47	38.1	27.1	27.1	61.9	82.0
1/48	81.4	56.1	56.1	18.6	80.5
1/59	57.7	45.0	45.0	42.3	88.3
1/63	57.6	31.3	31.3	42.4	89.8
1/64	34.2	25.4	25.4	65.8	87.9
1/84	44.7	40.7	40.7	55.3	89.0
2/018	99.7	99.7	38.8	0.3	61.4
2/021	80.0	79.4	28.7	20.0	63.2
2/023	91.9	91.9	28.9	8.1	77.8
2/024	82.4	80.0	9.1	17.6	40.0
2/026	81.3	76.2	29.3	18.7	68.5
2/027	79.6	78.6	25.5	20.4	22.5
2/029	82.4	80.2	6.2	17.6	9.2
2/035	95.5	92.6	17.2	4.5	5.7
2/041	87.7	84.6	7.9	12.3	2.5
3/111222333444	79.5	76.4	12.1	20.5	16.7
3/bananana	91.4	80.2	42.0	08.6	56.2
3/checkinput	82.1	68.5	15.3	17.9	34.8
3/doublecoupon	80.6	75.6	32.5	19.4	41.9
3/keepdoubling	86.5	71.6	07.6	13.5	35.1
3/memberdiscount	75.7	73.3	19.8	24.3	40.3
3/restaurant	80.6	77.4	23.9	19.4	45.8
3/squares	90.1	67.6	11.0	09.9	30.2
3/triplecoupon	79.8	78.8	23.2	20.2	44.7

Table 3: For all partially-correct submissions, the percentage of the time the next submission was observed to be (a) in the same CFAST, (b) at least partially correct and in the same CFAST, (c) fully correct and in the same CFAST, and (d) in a different CFAST. In addition, (e) indicates the percentage of students without successive submissions from a partially correct submission.

4.1.2 RQ 1.2

RQ 1.2 asks, "How do students add control flow elements to their code?" We conducted two analyses to address this question.

As a way of tracking changes in control flow over student work histories, we computed the tree edit distance [53] between the CFASTs of successive pairs of submissions in each

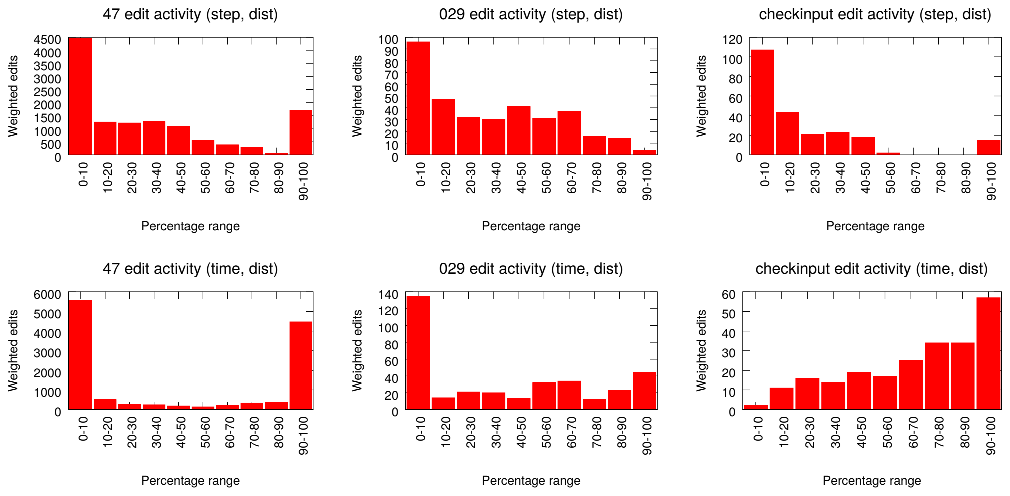

Figure 2: Step- and time- accumulated edit distance for three activities (1/47, 2/029, and 3/checkinput)

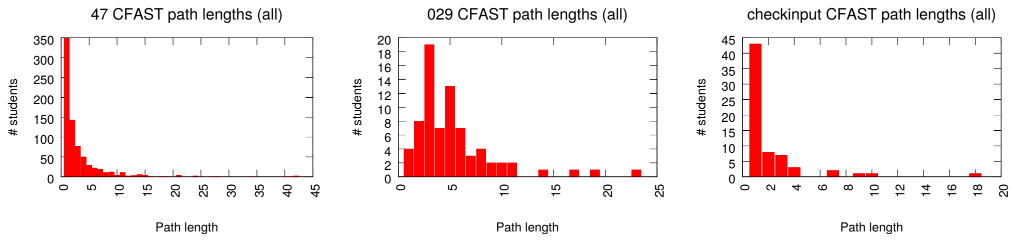

Figure 3: Histograms of CFAST path lengths for three selected activities (1/47, 2/029, and 3/checkinput)

student's work history, assigning a weight of 1 to each deletion, insertion, and replacement. Note that this metric takes only changes to the CFAST into account, and ignores code changes which do not affect the CFAST. To visualize the resulting data, we constructed histograms showing the accumulated edit distance over all adjacent pairs of submissions. We analyzed both step-wise and time-wise[1] chronology, using percentages to represent step or time, with 0% being the first step or the start of work, and 100% being the last step or the end of work. Figure 2 shows examples of accumulated edit distance for three activities.

We can see some interesting patterns in these histograms. The CFAST edits (which represent changes in the control structure) tend to occur early in the sequence of submissions (steps), which we interpret as indicating that students tend not to submit their work for testing until they arrive at something approximating their final control structure, although subsequent fine-tuning does occur at least some of the time. The time data for courses 1 and 2 is somewhat similar, showing changes occurring early. (The surge in edits at the end of the steps in activity 1/47 is largely due to students who completed the activity with a single submission, which we treated as occurring at the end of the timeline.) The time-wise histogram for 3/checkinput looks quite different because we were able to track student work sessions more precisely, and could account for the student work which preceded the first submission.

Another way to understand control-flow changes is to examine the paths — sequences of CFASTs — by which students arrive at either a correct solution or their last submission. A path whose length is 1 indicates a student who did not make any changes to the control-flow structure of her program over time. A longer path indicates a student who tried different control-flow structures over time. Figure 3 summarizes the CFAST paths as a histogram of path lengths for three selected activities. One interesting phenomenon we can observe from the path histograms is that most paths are fairly short, meaning few or no changes in control structure over time. We also observe that there is a long tail in each histogram showing a few students who tried a variety of control structures, possibly returning to previously-tried structures. We will have more to say about this phenomenon in Section 4.2.2.

4.1.3 RQ 1.3

RQ 1.3 asks, "Once a student has control flow correct in code, does it remain correct?"

[1]Note that the time-based chronology is computed differently for the three courses. For courses 1 and 2, time is based on absolute submission timestamps. For course 3, in which fine-grained edit events are available, time is relative to the time the student actually spent working, under the assumption that any interval of 5 minutes or more between edit events represents a break in the student's work.

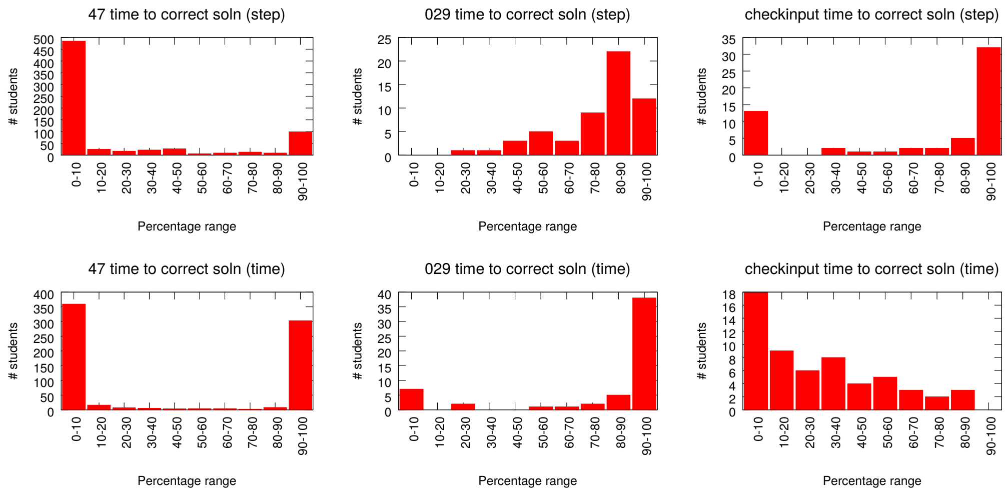

Figure 4: Histogram of time required to complete a correct solution after first reaching the final CFAST form, expressed as a percentage of total steps/duration, for three selected activities (1/47, 2/029, and 3/checkinput)

To address this question, we examined all consecutive pairs of submissions in all student work histories where the first submission was at least partially correct, and counted how often the next submission (a) had the same CFAST, (b) had the same CFAST and was at least partially correct, (c) had the same CFAST and was completely correct, and (d) used a different CFAST. In addition, (e) indicates the percentage of students *without* consecutive submissions, which is important for interpreting the other columns. Table 3 presents the results of this analysis.

One observation regarding the data in Table 3 is that there are significant variations by course. Courses 2 and 3 both exhibit a fairly strong tendency for subsequent submissions to stay in the same CFAST, and a fairly low percentage of students who have only a single submission. These courses tend to support the hypothesis that students arrive at an intended control flow pattern earlier rather than later, and tend to stick with it. In contrast, in Course 1, the probability of a subsequent submission remaining in the same CFAST is lower, and the probability of students not having subsequent submissions is higher. We believe that because the activities in Course 1 are, in general, harder than those in Courses 2 and 3, there is a higher percentage of students in course 1 who give up, and also a higher incidence of plan changes for the students who persist.

4.2 RQ 2

In RQ 2, we were interested in whether CFASTs would shed any light on whether students are working productively or struggling. We identified two concrete component questions, RQ 2.1 and RQ 2.2.

4.2.1 RQ 2.1

RQ 2.1 asks, "At what point in their programming process do students reach a CFAST that corresponds with a correct solution?"

We assessed this question somewhat indirectly. Figure 4 shows step-wise and time-wise histograms of time to reach a correct solution as a percentage of the total time (number of steps or elapsed time), for three selected activities. Shorter time to correct solution corresponds to later emergence of control structure. We observe that there is some tendency for the "final" CFAST form to appear either early or late in a student's work history.[2] Activity 2/029 is an interesting exception, with different students' final control structures appearing at many different points in the students' work histories. One explanation for this difference is that this activity included a visualization, encouraging experimentation with different control structures, and also included intermediate milestones, encouraging submissions with partial functionality to a greater degree than most of the other activities.

4.2.2 RQ 2.2

RQ 2.2 asks, "Do sequences of CFASTs encode evidence of ineffective programming behaviours?" We identified one metric which we supposed might be correlated with academic performance: CFAST path length, meaning the extent to which student work histories were observed to exhibit changes in control structure from submission to submission. Specifically, we surmised that students who had greater mastery of control structures would change them less often, and thus we would see a negative correlation between path lengths and academic performance.

To test this hypothesis, we computed the sum of the lengths of the observed CFAST paths for each student, and computed the correlation with exam scores. For Courses 1 and 2, we used the course's written final exam score, and for Course 3, we used the programming question scores for the second midterm exam (which was the exam focusing the most on control structures.) Table 4 shows the results.

For Course 1, there was a significant but weak negative correlation between cumulative path lengths and exam scores. For Course 2, we see no statistically significant correlation. For Course 3, we see a significant moderate negative correla-

[2]Note that because students who have a single submission (of which there are a substantial number in activities 1/47 and 3/checkinput) are considered to reach the final CFAST form at the end of the history, these plots may exaggerate the "late" appearance of control structure to some degree.

		Correlation	
Course	Exam type	rho	p-value
1	Final written exam	-0.11	0.008
2	Final written exam	-0.17	0.34
3	*Programming, 2nd midterm*	*-0.40*	*0.003*

Table 4: Correlations between accumulated CFAST path lengths and exam scores

tion. One possible explanation for the significant result for Course 3 but not Courses 1 and 2 is the type of activities used. The activities in Course 3 cover control flow concepts at a relatively basic "application" level, so students who have difficulty constructing a correct control-flow form for such exercises are likely to have important deficits in their concept knowledge. In contrast, Courses 1 and 2 include some more sophisticated "synthesis" activities, where some amount of exploration is expected. In addition, there are other possible confounding factors. For example, students who plagiarize their solutions will have low path length, but may fare poorly on an exam. Another issue is that students who give up early will have low path lengths.

5. DISCUSSION AND CONCLUSIONS

In this study, we explored the applicability of Control-Flow only Abstract Syntax Trees (CFAST) for source code snapshot analysis. We studied whether students' solutions converge to specific CFASTs and how students work with their code through the CFAST representations. We also explored the extent to which CFASTs can be used to identify students in difficulty.

Our findings agree with those of Luxton-Reilly et al. [24]. We confirmed that for *most* exercises, most of the student submissions could be categorized under a relatively small number of control structures. We did notice some interesting exceptions to their findings, however. For a few activities in our study, for example 1/47, there were a very large number of attempted (979) and correct (207) control forms. We posit that additional requirements for conditional logic (if/else) can lead to exponential increase in the number of reasonable solutions to a programming problem.

Here, the order of the assignments and practice likely plays a large role. If the students have already practiced the individual constructs before attempting to construct larger programs – by merging existing plans together — they are likely more successful in the process. At the same time, if the students have not practiced sufficiently the individual constructs beforehand, students may enter a trial-and-error problem solving mode, where their attempts in constructing working solutions to the problems may be rather wild. The data showed a handful of students who show a high degree of trial-and-error behavior, which merits further study. Such behavior — when encoded through the accumulated distinct CFAST path lengths — was not indicative of the performance in a written exam, but had a modest negative correlation with a programming-related midterm.

Overall, changes in control structures, as indicated by CFAST changes, provide some useful insights into students' programming strategies. However, we noticed some phenomena in our data that impeded analysis. For example, it was more common than we anticipated for students to only submit one potential solution for an activity. Often, this was the correct solution for the problem, which means that the students work through the problem (or plagiarize it) on their own, and do not need the feedback from the submission system. It would be meaningful to have access to CFAST changes on the client system even if the students choose to not submit their solutions at every step.

We have several ideas for future work. Use of fine-grained edit information can help to overcome the issue of infrequent student submissions. We would like to conduct more detailed analyses of *how* student code changes over time: for example, do we see implementation of control structures bottom-up or top-down? We think that it may be helpful to introduce some details about non-control-flow constructs into our analysis: for example, upper and lower bounds might be interesting as a way of characterizing loop structures, and might help tease out important correctness requirements within the set of submissions in a single CFAST. Based on our observations that frequently vs. infrequently observed CFASTs tend to correspond to "obvious" vs. "idiosyncratic" control structures, CFASTs may be useful in selecting example solutions (as suggested by Luxton-Reilly et al. [24]) or to automate construction of peer instruction questions.

6. REFERENCES

[1] A. Ahadi, R. Lister, H. Haapala, and A. Vihavainen. Exploring machine learning methods to automatically identify students in need of assistance. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, pages 121–130, 2015.

[2] P. Antonucci, C. Estler, D. Nikolić, M. Piccioni, and B. Meyer. An incremental hint system for automated programming assignments. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, pages 320–325, 2015.

[3] R. Bellamy and D. Gilmore. Programming plans: Internal or external structures. *Lines of thinking: Reflections on the psychology of thought*, 2:59–72, 1990.

[4] J. Bonar and E. Soloway. Uncovering principles of novice programming. In *Proc. 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 10–13, 1983.

[5] J. Bonar and E. Soloway. Preprogramming knowledge: a major source of misconceptions in novice programmers. *Human-Computer Interaction*, 1:133–161, June 1985.

[6] A. S. Carter, C. D. Hundhausen, and O. Adesope. The normalized programming state model: Predicting student performance in computing courses based on programming behavior. In *Proceedings of the Eleventh Annual International Conference on International Computing Education Research*, pages 141–150, 2015.

[7] J. Carter, P. Dewan, and M. Pichiliani. Towards incremental separation of surmountable and insurmountable programming difficulties. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 241–246, 2015.

[8] Y. Cherenkova, D. Zingaro, and A. Petersen. Identifying challenging cs1 concepts in a large problem dataset. In *Proceedings of the 45th ACM Technical*

Symposium on Computer Science Education, pages 695–700, 2014.

[9] M. Craig, S. Petersen, and A. Petersen. Following a thread: Knitting patterns and program tracing. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, pages 233–238, 2012.

[10] P. Denny, A. Luxton-Reilly, and E. Tempero. All syntax errors are not equal. In *Proceedings of the 17th Conference on Innovation and Technology in Computer Science Education*, pages 75–80, 2012.

[11] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx. Understanding the syntax barrier for novices. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, pages 208–212, 2011.

[12] F. Détienne. Design strategies and knowledge in object-oriented programming: Effects of experience. *Hum.-Comput. Interact.*, 10(2):129–169, Sept. 1995.

[13] A. K. Dominguez, K. Yacef, and J. R. Curran. Data Mining for Individualised Hints in e-Learning. In *Proceedings of the International Conference on Educational Data Mining*, pages 91–100, 2010.

[14] K. Fisler. The recurring rainfall problem. In *Proceedings of the Tenth Annual Conference on International Computing Education Research*, pages 35–42, 2014.

[15] K. Fisler, S. Krishnamurthi, and J. Siegmund. Modernizing plan-composition studies. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*, pages 211–216, 2016.

[16] D. Ginat, E. Menashe, and A. Taya. Novice difficulties with interleaved pattern composition. In *Proceedings of the 6th International Conference on Informatics in Schools: Situation, Evolution, and Perspectives*, pages 57–67, 2013.

[17] 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 Trans. Comput.-Hum. Interact.*, 22(2):7:1–7:35, Mar. 2015.

[18] K. Goldman, P. Gross, C. Heeren, G. Herman, L. Kaczmarczyk, M. C. Loui, and C. Zilles. Identifying important and difficult concepts in introductory computing courses using a delphi process. *SIGCSE Bull.*, 40(1):256–260, Mar. 2008.

[19] P. Ihantola, A. Vihavainen, A. Ahadi, M. Butler, J. Börstler, S. H. Edwards, E. Isohanni, A. Korhonen, A. Petersen, K. Rivers, M. A. Rubio, J. Sheard, B. Skupas, J. Spacco, C. Szabo, and D. Toll. Educational data mining and learning analytics in programming: Literature review and case studies. In *Proceedings of the 2015 ITiCSE on Working Group Reports*, pages 41–63, 2015.

[20] M. C. Jadud. Methods and Tools for Exploring Novice Compilation Behaviour. In *Proc. of the 2nd International Workshop on Computing Education Research*, pages 73–84, 2006.

[21] U. Kiesmueller, S. Sossalla, T. Brinda, and K. Riedhammer. Online Identification of Learner Problem Solving Strategies Using Pattern Recognition Methods. In *Proceedings of the Fifteenth Annual Conference on Innovation and Technology in Computer Science Education*, pages 274–278, New York, NY, USA, 2010.

[22] I. Koprinska, J. Stretton, and K. Yacef. Students at Risk: Detection and Remediation. In *Educational Data Mining*, 2015.

[23] A.-J. Lakanen, V. Lappalainen, and V. Isomöttönen. Revisiting rainfall to explore exam questions and performance on cs1. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, Koli Calling '15, pages 40–49, New York, NY, USA, 2015. ACM.

[24] A. Luxton-Reilly, P. Denny, D. Kirk, E. Tempero, and S.-Y. Yu. On the differences between correct student solutions. In *Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, pages 177–182, 2013.

[25] D. Moore, A. Parrish, and D. Cordes. Analyzing syntax error patterns among novice programmers. In *Proceedings of the 35th Annual Southeast Regional Conference*, pages 188–190, 1997.

[26] O. Muller, D. Ginat, and B. Haberman. Pattern-oriented instruction and its influence on problem decomposition and solution construction. *SIGCSE Bull.*, 39(3):151–155, June 2007.

[27] A. Petersen, J. Spacco, and A. Vihavainen. An exploration of error quotient in multiple contexts. In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, pages 77–86, 2015.

[28] R. Pettit, J. Homer, R. Gee, S. Mengel, and A. Starbuck. An empirical study of iterative improvement in programming assignments. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, pages 410–415, 2015.

[29] R. S. Rist. Schema creation in programming. *Cognitive Science*, 13:389–414, 1989.

[30] K. Rivers and K. R. Koedinger. Automatic generation of programming feedback: A data-driven approach. In *The First Workshop on AI-supported Education for Computer Science (AIEDCS 2013)*, page 50, 2013.

[31] K. Rivers and K. R. Koedinger. Automating hint generation with solution space path construction. In *Intelligent Tutoring Systems*, pages 329–339. Springer, 2014.

[32] K. Rivers and K. R. Koedinger. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *International Journal of Artificial Intelligence in Education*, pages 1–28, 2015.

[33] A. Robins, P. Haden, and S. Garner. Problem distributions in a CS1 course. In *Proc. 8th Australian Conference on Computing Education*, pages 165–173, 2006.

[34] A. Robins, J. Rountree, and N. Rountree. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13:137–172, 2003.

[35] M. M. T. Rodrigo and R. S. Baker. Coarse-grained detection of student frustration in an introductory programming course. In *Proceedings of the Fifth International Workshop on Computing Education Research Workshop*, pages 75–80, 2009.

[36] M. Satratzemi, V. Dagdilelis, and G. Evagelidis. A System for Program Visualization and Problem-solving Path Assessment of Novice Programmers. In

Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education, pages 137–140, New York, NY, USA, 2001.

[37] O. Seppälä, P. Ihantola, E. Isohanni, J. Sorva, and A. Vihavainen. Do we know how difficult the rainfall problem is? In *Proceedings of the 15th Koli Calling Conference on Computing Education Research*, pages 87–96, 2015.

[38] Simon. Soloway's rainfall problem has become harder. In *Proceedings of the 2013 Learning and Teaching in Computing and Engineering*, LATICE '13, pages 130–135, Washington, DC, USA, 2013.

[39] B. Simon, T.-Y. Chen, G. Lewandowski, R. McCartney, and K. Sanders. Commonsense computing: what students know before we teach (episode 1: sorting). In *Proc. 2nd International Workshop on Computing Education Research*, pages 29–40, 2006.

[40] E. Soloway. Learning to program = learning to construct mechanisms and explanations. *Commun. ACM*, 29(9):850–858, Sept. 1986.

[41] E. Soloway, J. Bonar, and K. Ehrlich. Cognitive strategies and looping constructs: an empirical study. *Comm. of the ACM*, 26:853–860, November 1983.

[42] J. Spacco, P. Denny, B. Richards, D. Babcock, D. Hovemeyer, J. Moscola, and R. Duvall. Analyzing student work patterns using programming exercise data. In *Proc. of the 46th ACM Technical Symposium on Computer Science Education*, pages 18–23, 2015.

[43] J. C. Spohrer and E. Soloway. Novice mistakes: Are the folk wisdoms correct? *Commun. ACM*, 29(7):624–632, July 1986.

[44] J. C. Spohrer, E. Soloway, and E. Pope. A goal/plan analysis of buggy pascal programs. *Hum.-Comput. Interact.*, 1(2):163–207, June 1985.

[45] A. Stefik and E. Gellenbeck. Empirical studies on programming language stimuli. *Software Quality Journal*, 19(1):65–99, 2011.

[46] A. Stefik and S. Siebert. An empirical investigation into programming language syntax. *Trans. Comput. Educ.*, 13(4):19:1–19:40, Nov. 2013.

[47] E. S. Tabanao, Ma, and M. C. Jadud. Predicting At-risk Novice Java Programmers Through the Analysis of Online Protocols. In *Proceedings of the Seventh International Workshop on Computing Education Research*, pages 85–92, 2011.

[48] A. E. Tew and M. Guzdial. Developing a validated assessment of fundamental cs1 concepts. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, pages 97–101, 2010.

[49] T. VanDeGrift, D. Bouvier, T.-Y. Chen, G. Lewandowski, R. McCartney, and B. Simon. Commonsense computing (episode 6): logic is harder than pie. In *Proc. 10th Koli Calling International Conference on Computing Education Research*, pages 76–85, 2010.

[50] A. Vihavainen, M. Luukkainen, and P. Ihantola. Analysis of source code snapshot granularity levels. In *Proceedings of the 15th Annual Conference on Information Technology Education*, SIGITE '14, pages 21–26, New York, NY, USA, 2014. ACM.

[51] C. Watson, F. W. Li, and J. L. Godwin. No tests required: Comparing traditional and dynamic predictors of programming success. In *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, pages 469–474, 2014.

[52] C. Watson, F. W. B. Li, and J. L. Godwin. Predicting performance in an introductory programming course by logging and analyzing student programming behavior. In *Proceedings of the 2013 IEEE 13th International Conference on Advanced Learning Technologies*, pages 319–323, Washington, DC, USA, 2013.

[53] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, Dec. 1989.