Is there Method in Your Mistakes? Capturing Error Contexts by Graph Mining for Targeted Feedback

Maximilian Jahnke
Ostfalia University of Applied Sciences
maxi.jahnke@ostfalia.de

Frank Höppner
Ostfalia University of Applied Sciences
f.hoeppner@ostfalia.de

ABSTRACT

The value of an instructor is that she exactly recognizes what the learner is struggling with and provides constructive feedback straight to the point. This work aims at a step towards this type of feedback in the context of an introductory programming course, where students perform program execution tracing to align their understanding of Java instructions with reality. The students' submissions are analyzed for repeating mistakes across different exercises by representing the context surrounding the error by a graph and applying graph mining techniques to discover their common grounds. The patterns need to be annotated only once and help to address misconceptions of individual students. They may also be used to select follow-up exercises automatically, that contain the same intricacy.

Keywords

graph mining, formative feedback, teaching programming, misconception

1. INTRODUCTION

Automatized grading is integrated (to varying extent) in many educational systems. It is often limited to comparing a correct answer against the submission (or applying unit tests), but is nevertheless welcomed by both, students and instructors, for different reasons: While students appreciate immediate feedback, instructors are relieved that the burden of manual grading is lifted. But a student who has missed the point of some problem and thus repeatedly gets the answers wrong would benefit much more from personal feedback from an instructor who is capable of recognizing and directly addressing the student's problem. And if none of the submissions is inspected manually any more, the instructor withholds his diagnostic skills that may otherwise have been proven useful to identify misconceptions of several course members. In this work we investigate – for a particular type of exercises in programming courses – how this situation may be improved.

M. Jahnke and F. Höppner. Is there method in your mistakes? Capturing error contexts by graph mining for targeted feedback. In A. Mitrovic and N. Bosch, editors, *Proceedings of the 15th International Conference on Educational Data Mining*, pages 422–429, Durham, United Kingdom, July 2022. International Educational Data Mining Society.

© 2022 Copyright is held by the author(s). This work is distributed under the Creative Commons Attribution NonCommercial NoDerivatives 4.0 International (CC BY-NC-ND 4.0) license.
https://doi.org/10.5281/zenodo.6853193

We consider an introductory programming course in Java for students who have no programming experience yet. While students may struggle with programming for a large number of reasons (e.g. fighting with the development environment, getting a grip on computational thinking, etc.), in this work we concentrate on a single aspect only: instruction comprehension. Especially for students who have no experience in programming, we observe in lab discussions that their perception of an instruction often deviates from reality. Sticking to a wrong mental model of program execution makes it difficult to write correct programs. A human advisor can (1) discover the misconception (even if it did not occur before), (2) explain the problem (and link it to the course material), and (3) challenge the student afterwards (to monitor progress). The research question of this paper is how such an ideal instructor might be mimicked (in the given context). In particular, is it possible to support the instructor in the discovery of new misconceptions without falling back to time-consuming manual inspection of all submissions?

2. RELATED WORK

The importance of dedicated feedback has been acknowledged by many researchers ([5] gives a review). In the context of programming, several approaches derive precise commands to fix the mistakes of a submitted solution (e.g. [9, 11]). While being helpful to fix a technical problem, applying detailed instructions mechanically will unlikely trigger a change of the students' mental model – the same mistake might be made again if nothing can be learned from the feedback (e.g. because it is lacking explanations). Different hint levels are offered as a solution in [10]: On the first level, a hint points to the problem but still requires the *right student action*, while a hint on the second level enables them to fix the problem mechanically. When misconceptions are known beforehand, their discovery in a submission turns into a classification problem. Recently a number of deep learning approaches have been proposed to predict whether a student will complete an exercise [14], the code exhibits certain logic errors [6], or some free-text answer hints towards a common misconception [7]. For all those misconceptions a dedicated feedback may be prepared only once (manually by the instructor) and whenever it is predicted in a new submission the feedback can be provided automatically. This is effective for *known misconceptions*, but such approaches do not help to discover *new misconceptions*.

One of the few approaches that may also help to discover *new problems* (and is therefore close to what we have in

mind) is suggested in [8], where the focus is on programming bugs. Common coding errors (for the same exercise) are discovered by finding subtrees (in the submissions' abstract syntax trees[1] (AST)) that correlate with failed unit tests, hinting at *typically wrong* code fragments for the given exercise. As with other approaches, an instructor provides feedback for a buggy subtree that will be delivered later whenever a similar subtree occurs again. The focus of these explanations is, however, restricted to the exercise at hand – a buggy subtree in one exercise may be correct in another.

3. PROGRAM EXECUTION TRACES

When an instructor explains some concepts in class it is difficult to verify that the students perceive the concepts in the intended way. This holds in general and for instructions of a programming language in particular. If a student has only vague ideas about stack and heap, call-by-value/reference, array organization, etc., programming errors will happen sooner or later. But since code will be executed by *man-made* machines, we know the ground truth (just find a debugger) and can ask students to simulate program execution (to some extent). Paper-based analysis of variable tracing exercises have been used to investigate misconceptions in the literature [1, 2, 4, 12]. Gaining an understanding of the mechanics how the language implementation worked, was also identified as a key issue in [13]. Employing tracing exercises in class has been reported to increase the students' performance in [3]. We thus consider execution traces as a helpful tool to align mental models and encourage students to verify their understanding by filling out traces: for every executed code line the content of all variables has to be documented.

Fig. 1 shows a Java code snippet on the right that uses a mixture of language elements for demonstration purposes. We ask the students to fill out an execution trace as shown on the left, which is basically a large table where each line corresponds to a complete memory snapshot – ordered in time from top to bottom. For each row, the student has to provide the line number that is executed next (e.g. execution starts with the first line 11 of the main function). The state of all variables must be entered for each row after the corresponding code line has been executed. For instance, when line 12 got executed there is a new variable `x` on the stack which points to the heap address `0x0`, where an array of length 2 has been instantiated and initialized with `null` references. Students enter such traces in a web application, which marks erroneous lines but not the exact error position to make the students think about their input and make *trial and error* strategies less attractive. The downside of this practice is that students may get stuck at some point because they are simply not aware of what they are doing wrong. A lab advisor, pointing them to the nature of their problem, would be appreciated by them.

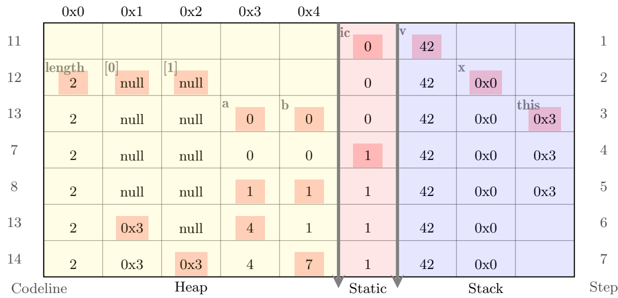
4. OUTLINE OF THE APPROACH

Although a full trace may consist of a considerable number of inputs (cf. Fig. 1), most of the student's input is not very informative because it (hopefully) corresponds to the already known solution. The valuable sources of information are the deviations alone, because they may hint to misconceptions. A wrong value in the trace alone does not tell us anything about possible reasons *why* this error was made. This can only be judged if more context is given, that is, the situation in which the error occured. If errors re-occur in *similar situations*, the likelihood of a misconception rises – otherwise it just might be a typo. The *context* is the Java program itself. However, similar situations do not necessarily correspond to identical source lines. For the sake of simplicity, let us consider the case of a student who is confused by assignments where the same variable occurs on the left and the right hand side of the assignment. (Sometimes beginners read instructions with `=` as a mathematical equation rather than an assignment.) Fig. 2 shows a simplistic code example together with the correct trace. Can we tell from the student's input if he is affected by this misconception? There are two places in the trace (marked in yellow) which we can associate with this misconception: the value of variable `b` after executing line 7 and the value of variable `a` after executing line 9 as both lines exhibit the discussed kind of assignment. If both places – or *similar* places in other traces – are wrong, chances rise that assignments have not yet been understood correctly. We have identified these two places in the trace because we *knew* what kind of misconception we were looking for. At these places the probability $P(\text{error}|\text{misconception})$ of an faulty input rises, given the misconception holds. As it may be difficult to come up with an operational definition of *misconception*, we retreat to $P(\text{error}|\text{context})$ and will define *context* as a situation or pattern that can be matched against the traces and the source code.

Recalling the research question, we seek a mechanism that identifies both, the context (of a potential misconception) as well as the positions affected by it. If the context is easily understandable, such a mechanism would enable us to mimick the behaviour of a valuable instructor with only little manual intervention: (1) discovery: if errors in an (automatically derived) context accumulate substantially (across all students), it is likely the context captures a problematic situation. The instructor inspects the context once and decides what a likely root cause might be. Once it got accepted as a misconception in this way, it can be discovered in subsequent submissions automatically. If the errors accumulate for some individual student, we conclude that he suffers from this misconception. (2) feedback: The instructor writes a short explanation when the problem was discovered for the first time (or may link it to existing material). As with other approaches, this feedback can be delivered in subsequent occurrences. (3) challenge: Beyond the textual feedback, we need a challenge to check for an improved understanding. We characterize a student by a set of *pending misconceptions* and, in the same way, we characterize exercises by the contexts they contain. A suitable challenge is thus a challenge that matches the students profile best (in the fashion of an adaptive recommendation system).

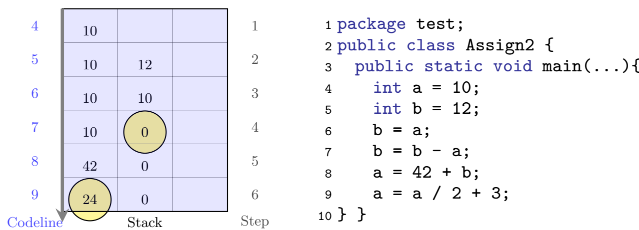
A *context* must address properties of the source code **and** the trace: We want to align errors in the trace to properties of the source code. The plain code (Fig. 2(right)) is therefore transformed into a graph as shown in Fig. 3 as follows: The first step is the creation of an abstract syntax tree, which already provides most of the nodes in the graph. Starting from the top node we can see the method declaration (of

[1] An AST is build by a parser and represents the syntactical structure of code while omitting some less relevant details.




```
1 package test;
2 public class Demo {
3   static int ic;
4   int a,b;
5   public Demo() {
6     ic++;
7     a=ic; b=ic*ic;
8   }
9   public static void main(String[] args) {
10    int v = 42;
11    Demo[] x = new Demo[2];
12    x[0] = new Demo(); x[0].a=4;
13    x[1] = x[0]; x[1].b=7;
14 } }
```

Figure 1: For the example code on the right, the corresponding execution trace is shown on the left. The different memory areas stack, heap, and static data are shown in blue, yellow, and red, resp.




```
1 package test;
2 public class Assign2 {
3   public static void main(...){
4     int a = 10;
5     int b = 12;
6     b = a;
7     b = b - a;
8     a = 42 + b;
9     a = a / 2 + 3;
10 } }
```

Figure 2: Code (right) and corresponding trace (left). Yellow marks indicate where the misconception may be observed.

main), which contains a body (instruction block {}) with 6 statement nodes, two declarations and four assignments. But the AST does not yet suit the trace, it represents the static structure of the code, but the trace is concerned with the dynamic evolution of the variables during code execution. We bridge this gap by two changes:

1. Every occurrence of the *same variable* in the code introduces a new node in the AST, but it has a unique position in memory (and thus a unique position in the trace). Therefore all AST nodes that refer to the same variable are united to a single node, turning the tree into a graph. In Fig. 3 both nodes for the variables `a` and `b` are shaded in gray. Every incoming edge into one of these variable nodes correspond to the use of the variable in the code.

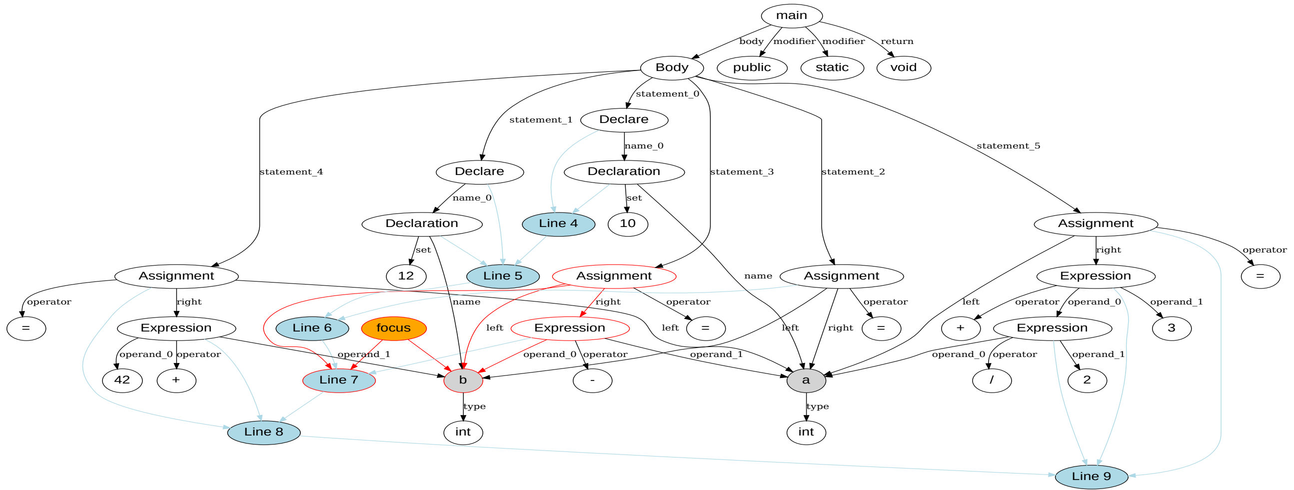
2. The content of the trace is aligned with the code lines, but by looking at an AST graph one cannot tell which instruction comes from which code line. So we introduce a new node for each code line (shown in blue in Fig. 3) and link them with all instructions from this code line. (For instance, the rightmost assignment node in Fig. 3 occurs in line 9 of Fig. 2.) We capture the program flow by connecting code line nodes that may get executed in sequence. In the example we have no loops or conditional statements, so the nodes connect linearly (*Line 5* to *Line 6*, *Line 6* to *7*, etc).

This graph combines information from code and trace, but does not yet encode where the student entered a wrong value. The error position in the trace is identified by row and column (cf. yellow circles in Fig. 2): The column corresponds to the variable and the row corresponds to a time step during execution, which always refers to a specific code line. An additional *focus* node (orange in Fig. 3) thus encodes the error position by connecting to both, the variable for which an erroneous value has been entered in the trace[2], and the code line node whose execution has been traced.

We denote such a graph G as a **code graph**. A *context* may now be represented as a subgraph C of G. In Fig. 3 such a subgraph has been highlighted with red lines. We read the subgraph as follows: We focus on a code line (here: line 7) that contains an assignment where the variable (here: `b`) on the left hand side also occurs in the expression on the right hand side of the assignment. The context graph is also shown isolated in Fig. 4(left). If this context accumulates a high number of student errors, it can be shown to the instructor and should be reasonably simple to interpret. The instructor may then decide that this context represents a misconception and writes a short explanation for the context pattern as indicated in Fig. 4(right). The text template may refer directly to nodes in the context graph (e.g. `[varname_2]`). These references are replaced with the true line numbers or variable names from the real exercise the student is currently working on. This links the feedback text very tight to the exercise just submitted by the student.

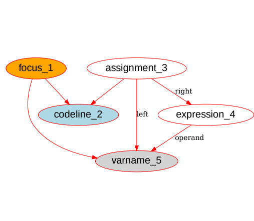
To match such a subgraph to other occurrences, it has to be *anonymized*, for instance, the variable names have to be replaced by placeholders 'varname' (as we have already seen in Fig. 4). Variables have unique nodes in the graph and we no longer need the variable's name for disambiguation. Other graph transformations may be applied to compensate structural differences in the AST that are not helpful for our purposes. For instance, we want the context to match twice in the example code of Fig. 2, but the subtree for the expression on the right hand side of b=b−a is less complex than that of a=a/2+3, so the code graph is structurally different. To ensure that a context can still match both occurrences, we apply two measures: Firstly, the graph is transformed to simplify some technical details (we collapse a tree of expression nodes into a single expression node). Secondly, we extend the expressiveness of a context: Rather than only simple edges we allow, e.g., for transitive connec-

[2] only if there is one; line number errors do not have an associated variable



;

Figure 3: A graph that fuses information from the abstract syntax tree (black), the execution trace (blue) and a potential error position in the trace (orange). A potential misconception can be characterized as a subgraph (red).



"You seem to have difficulties with assignments that use the same variable on both sides of the assignment, as it happens in line [codeline_2] with variable [varname_5]. In Java the '=' sign does not indicate a mathematical equation but..."

Figure 4: Context graph and excerpt from a text template to deliver meaningful feedback to a student with a potential misconception.

tions (a path of arbitrary length from the expression node to the variable node instead of a direct neighbour).

5. ERROR CONTEXT DISCOVERY

In this section we briefly sketch a graph mining approach to discover context patterns automatically from data.

5.1 Code and Context Graphs

Our input consists of various Java sources, the correct solution trace and the student's input. The example graph shown in Fig. 3 *focusses* on a specific position in the trace (row/column as encoded by the `codeline` and `varname` nodes linked to the focus node). We create corresponding graphs for every position (row/column) in a trace. At first glance, this affects only the edges of the focus node while the rest of the graph remains unaltered (for a given tracing exercise). However, the relevant context for entering a wrong value in the trace can be narrowed down to the *already passed code lines*: The reason for a mistake cannot be found in source code lines yet to come. Knowing which code line we are focussing at, we remove all AST nodes that correspond to code beyond the focus line.

Apart from the transformations already mentioned in Sect. 4 (renaming variable nodes, flattening expressions), we introduce extra edges from an instruction block to the very first and last statement of the block (labelled `first` and `last`). Other transformations replace numbers by a `literal` label, or re-insert information about variable names: whenever two variables have the same name (before replacing them) we add a `samename`-edge between them. A code graph is then defined as follows: A triple $G = (V, E, \lambda)$ is called **code graph**, iff V is a non-empty set of nodes, $E \subseteq V \times V$ is a set of directed edges between nodes, $\lambda : (V \cup V \times V) \to L$ is a function that assigns labels (from a set of labels L) to nodes and edges, and there is exactly one $v \in V$ with $\lambda(v) = \texttt{focus} \in L$.

We assume that a set $\mathcal{G}$ of all code graphs is given. Having a graph $G \in \mathcal{G}$ for every individual cell (of every trace), we associate counters with them to track how often the respective cell has been entered correctly or not in the submissions. We denote the (absolute) frequencies of correct and incorrect entries by $\text{pos}(G)$ and $\text{neg}(G)$, resp. For a given context C the core operation is to decide whether C can be found in a given code graph G, which is abbreviated as $C \sqsubseteq G$. Then we define

$$P(\text{error}|\text{context } C) = \frac{\sum_{G \in \mathcal{G}, C \sqsubseteq G} \text{neg}(G)}{\sum_{G \in \mathcal{G}, C \sqsubseteq G} \text{pos}(G) + \text{neg}(G)} \quad (1)$$

The graph mining algorithm then searches for a context C that maximizes $P(\text{error}|\text{context } C)$. Formally, we define a context pattern as follows: A tuple $P = (V, E, \lambda, \tau)$ is called **context graph**, iff V is a non-empty set of nodes, $E \subseteq V \times V$ is a set of directed edges between nodes, $\lambda : (V \cup V \times V) \to L \cup \{*\}$ assigns labels to nodes and edges (where $*$ is a wildcard label), there is exactly one $v \in V$ with $\lambda(v) = \texttt{focus}$, and $\tau : E \to \{N, X, T, W\}$ classifies edges into one of four types: normal (N), excluding (X), transitive (T) or wildcard (W).

The semantics of the edge types and the wildcard label (which matches any other label) is defined along with the definition of graph inclusion $\sqsubseteq$ as follows: A context graph $C = (V_C, E_C, \lambda_C, \tau_C)$ is contained in a code graph $G =$

(V_G, E_G, λ_G), $\mathbf{C} \sqsubseteq \mathbf{G}$ for short, iff there is a bijective function $\sigma : V_C \rightarrow V_G$ such that all node labels match ($\forall v \in V_C : \lambda_C(v) = \lambda_G(\sigma(v))$) or the context label is a wildcard ($\lambda_C(v) = *$), and for all edges in C there are *corresponding edges* in G. An edge $(u,v) \in E_C$ has a corresponding edge if: (1) for normal edges ($\tau((u,v)) = N$) we have $(\sigma(u), \sigma(v)) \in E_G \wedge \lambda_C((u,v)) = \lambda_G((\sigma(u), \sigma(v)))$, (2) for transitive edges ($\tau((u,v)) = T$) there is a path $(w_1, w_2, \ldots, w_k)$ in G with $\sigma(u) = w_1$, $\sigma(v) = w_k$, and $\forall i : \lambda_C((u,v)) = \lambda_G((w_{i-1}, w_i))$, (3) for wildcard edges ($\tau((u,v)) = W$) we have $(\sigma(u), \sigma(v)) \in E_G$, (4) for excluding edges ($\tau((u,v)) = X$) we have $(\sigma(u), \sigma(v)) \in E_G \wedge \lambda_C((u,v)) \neq \lambda_G((\sigma(u), \sigma(v)))$.

5.2 Mining Algorithm

In contrast to other graph mining algorithms, our notion of subgraph inclusion ($\sqsubseteq$) is more complicated due to the different edge types. But we benefit from the fact that we have a clear starting point for inclusion tests because a unique `focus` node is required in both, code graphs and context graphs. The out-degree for many nodes is limited by the fixed syntax of Java instructions, but this does not hold for nodes like instruction blocks or expressions.

From a context graph that may serve as an indicator for a misconception we expect a higher rate of faulty student inputs than on average. A context graph C matches various cells in the traces, but the total number of correct inputs is usually much larger than the total number of incorrect answers. To account for this imbalance, we normalize the correct (and incorrect) number of answers in context C:

$$p_C = \frac{\sum_{G \in \mathcal{G}, C \sqsubseteq G} \text{pos}(G)}{\sum_{G \in \mathcal{G}} \text{pos}(G)}, \quad n_C = \frac{\sum_{G \in \mathcal{G}, C \sqsubseteq G} \text{neg}(G)}{\sum_{G \in \mathcal{G}} \text{neg}(G)}$$

We define an objective function f to rate context C as:

$$f(C) \stackrel{\text{max!}}{=} \log_2 \left(\frac{n_C}{p_C} \right) \tag{2}$$

If the fractions n_C and p_C are about the same, the answers in context C do not distinguish from the average rate over all cells and we obtain $f(x) = \log_2(1) = 0$. If we manage to find a context C that covers twice as many erroneous cases (than on average) and half as many correct cases (than on average), it evaluates to $\log_2 \left(\frac{2}{\frac{1}{2}} \right) = 2$. We perform a beam search in the space of context graphs to identify the context that maximizes (2). Details about the search algorithm are given in the appendix A.

6. EXPERIMENTAL EVALUATION

When the students filled out the execution traces, the submitted solution may contain multiple errors. It is likely that the first error in the trace causes subsequent faults: If the student enters a wrong value in some cell, but this variable is not altered in subsequent code lines, the wrong value remains and invalidates multiple trace lines. Therefore we consider only the first error from each submission, which most likely contains its root cause. In total, there were roughly 100 accounts (some were used only a few times) and all traces together consisted of $|\mathcal{G}| = 7128$ different input cells. All students together pressed the 'evaluation' button almost 92000 times. The total number of correct and incorrect entries in the submissions was 2.4M and 64k, resp.

6.1 Discovered Contexts

In this section we present and discuss a few of the context graphs that were automatically discovered by the algorithm from Sect. 5. The objective is to investigate meaningfulness and interpretability of the discovered contexts. In the figures, the node labels carry unique numbers for easier reference (e.g. `focus_0` denotes node #0) and the edges have a suffix indicating their edge type (e.g. N for normal edge) and a varying line style (N : solid, W : dotted, X : red).

The context in Fig. 5 addresses the lifetime of local variables. Many students have a superfluous entry on the stack in this context. The variable #2 has been declared #3 in an instruction block #6, whose last statement is an assignment (#5, connects to #6 with edge labelled `last`). This is the last statement of the block #6 and the superfluous variable has been observed in the next executed line (#1, connected to #4 with an edge labelled `prevex` (previously executed)). This code line is thus the first after the end of block #6 and variable #2 should have vanished because its lifetime ended.

A somewhat surprising context is shown in Fig. 6, which describes a code line #1 with an if-statement #5, where the focussed variable #2 is somehow involved in the control-expression #6 of the if-statement. An inspection of the submitted traces reveals that some students stored the boolean result of the control expression on the stack. (Probably the students thought that the evaluation result must be traced *somehow*, but as it does not affect any variable it is not.)[3]

Fig. 7 captures a context where local variables and attributes are mixed up. The focus is on a local variable #2 (local because of its path to #6) in a function #6 of a class #7 that also contains a field declaration #8. The local variable has a `samename`-edge to another variable, which causes the confusion. Although the equally named variable #4 is not connected to the field declaration #8, it corresponds to it in all matching code graphs. So the student was not aware which variable was addressed in the code and picked the wrong one.

Some contexts may not really relate to code understanding, but point at pitfalls when starting to use the web application. Programs usually start with some variable declaration, so the attention is initially on the stack – and filling out the line number is easily forgotten: In Fig. 8 the expected code line (whose line number was not entered correctly) contains a declaration #2, which is the first instruction in the body #3 of a function #8 – it thus most likely corresponds to the very first code line that has to be traced.

Discussion: The graph mining approach delivered various meaningful context patterns that correspond well to our expectations as well as some unexpected patterns. Once an instructor gets used to them, the graphs are comparatively easy to interpret, which was a requirement for the envisaged approach to solve the research question in Sect. 4 – although sometimes it may be necessary to have a look at the submissions to get an idea what types of error were made. Not all of the discovered patterns were useful, some are too general in nature or mix up several problems. But compared to the

[3]We consider it unlikely that they tried to mimick the JVM-internal evaluation of expressions.

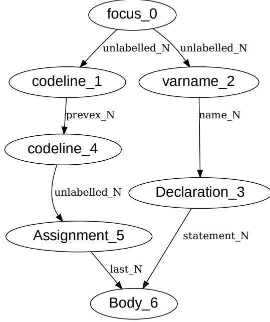


Figure 5: Error context for the lifetime of variables.

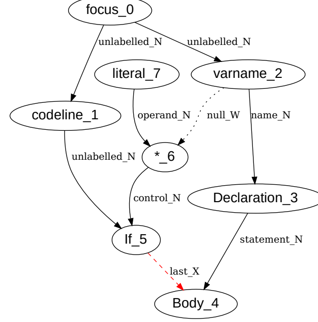


Figure 6: Effect of evaluating expressions.

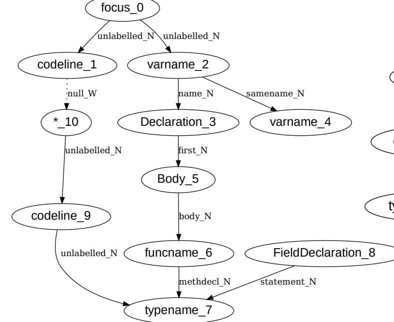


Figure 7: Confusing equally named variable and attribute.

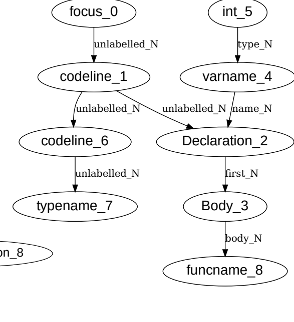


Figure 8: Line number is missing at start of an exercise.

Table 1: Based on the performance over a series of exercises, the students are classified in three groups S_+, $S_\uparrow$, S_-. The table shows how many students belong to which group (percentage) and the median number of exercises per group.

	group size %			practise #		
context	$S_\uparrow$	S_+	S_-	$S_\uparrow$	S_+	S_-
line number	39.6	27.9	32.4	52	26	25
missing stack	29.8	51.8	18.2	26	19	19
wrong heap value	18.8	64.1	16.9	48	27	20
expected reference	50.2	36.1	13.6	44	31	23
Fig. 5	33.3	44.4	22.2	16	12	13
Fig. 7	20.0	40.0	40.0	23	16	19

time spent on inspecting many raw submissions manually, it takes an instructor much less time to select useful patterns and write an accompanying explanation.

6.2 Evaluating Demand and Progress

Whenever a new concept or instruction has been introduced during the lecture, one group of students (S_+) will have no problems at all with the new instruction type and will enter the traces flawlessly. The second group, which experiences problems, splits up further: the group of reflective students ($S_\uparrow$) will notice the knowledge gap, may read the course material or offered help text again, and will at some point close the gap – in the future this type of mistake will happen only occassionally and incidentally. We expect a significant reduction in the error rate for this group of students. The remaining group (S_-) does, for some reason, not improve. We can distinguish these three groups on the level of individual misconceptions as follows: For a given error context, we collect all submissions of a student and order them in time. For any given point in time t we may characterize the students performance *before* and *after* t by counting how often the student got the trace in this context right or not. If we can find a point in time t such that there is a *significant improvement* in the distribution of mistakes before and after t, we say that this student belongs to group $S_\uparrow$. We assign students with a rather low error rate (say, consistently $\leq 10\%$) to group S_+ (at such a low error rate it is very difficult to improve significantly). The remainder belongs to group S_-.

Table 1 shows the size of the groups (percentage) and the median number of exercises per group. The rows aggregate a few different context graphs related to line number errors, missing values on the stack, wrong values on the heap, and situations in which a reference was expected but a number was entered. The numbers for the context patterns of Fig. 5 and 7 are also shown. The table shows that members of $S_\uparrow$ have traced more exercises than any other group (sometimes more than twice as many as group S_-), so the success is correlated with the gained practice. It is not surprising that the median number of practised exercises per context in group S_+ is more similar to group S_- than to $S_\uparrow$: If they do it right from the beginning, there is less need for further practising.

The experiments show that we can measure (by means of the statistical test) whether an individual student performs significantly worse on a context patterm than the course on average and whether he has improved over time. Similar to feedback from a lab advisor, which is rare but to the point, we may provide feedback to detect misconceptions only if the students make mistakes in the error context significantly more often than the course average (rather than annoying a student with hints at the first incidental mistake). The numbers from Tab. 1 also show that there is demand for support, a relevant fraction of the course did not manage to significantly improve on their own.

7. CONCLUSION

A considerable fraction of the participants did not manage to overcome tracing difficulties on their own, which shows the demand for constructive feedback that goes beyond a right/wrong classification. The automatically discovered context graphs can be assessed by an instructor with comparatively low effort, because they are quite easy to interpret and meaningful. This paves the way for identifying even new misconceptions in reasonable time. The context graphs allow us to analyse the students' state of knowledge and consequently deliver purposeful feedback when needed. It also enables us to recommend suitable exercises to check an improved understanding afterwards. Just like a human advisor, we may congratulate the student if the error rate decreased significantly. This will bring us, in response to the research question, quite close to a human advisor.

8. REFERENCES

[1] P. Baffes and R. Mooney. Refinement-based student modeling and automated bug library construction. *Journal of Artificial Intelligence in Education*, 7(1):75–116, 1996.

[2] R. Bornat, S. Dehnadi, and Simon. Mental models, consistency and programming aptitude. *Conferences in Research and Practice in Information Technology Series*, 78(1986):53–61, 2008.

[3] M. Hertz and M. Jump. Trace-Based Teaching in Early Programming Courses. *Proc. Techn. Symp. Computer Science Education*, pages 561–566, 2013.

[4] A. N. Kumar. Explanation of step-by-step execution as feedback for problems on program analysis, and its generation in model-based problem-solving tutors. *Technology, Instruction, Cognition and Learning*, 4(1):65–107, 2006.

[5] N.-T. Le. A classification of adaptive feedback in educational systems for programming. *Systems*, 4(2):22, 2016.

[6] D. Miao, Y. Dong, and X. Lu. PIPE: predicting logical programming errors in programming exercises. In *Proc. Int. Conf. Educational Data Mining*, 2020.

[7] J. J. Michalenko, A. S. Lan, A. E. Waters, P. Grimaldi, and R. G. Baraniuk. Data-mining textual responses to uncover misconception patterns. In *Proc. Int. Conf. Educational Data Mining*, 2017.

[8] A. Nguyen, C. Piech, J. Huang, and L. Guibas. Codewebs: scalable homework search for massive open online programming courses. In *Proc. Int. Conf. on World Wide Web*, pages 491–502, 2014.

[9] T. W. Price, R. Zhi, and T. Barnes. Evaluation of a data-driven feedback algorithm for open-ended programming. In *Proc. Int. Conf. Educational Data Mining*, 2017.

[10] K. Rivers and K. R. Koedinger. Data-driven hint generation in vast solution spaces: a self-improving python programming tutor. *Int. J. Artif. Intell. Educ.*, 27(1):37–64, 2017.

[11] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proc. Conf. Programming Language Design and Implementation*, pages 15–26, 2013.

[12] K. Stephens-Martinez, A. Ju, K. Parashar, R. Ongowarsito, N. Jain, S. Vekat, and A. Fox. Taking advantage of scale by analyzing frequent constructed-response, code tracing wrong answers. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, pages 56–64. ACM, 2017.

[13] M. W. van Someren. What's wrong? understanding beginners' problems with prolog. *Instructional Science*, 19(4-5):257–282, 1990.

[14] L. Wang, A. Sy, L. Liu, and C. Piech. Learning to represent student knowledge on programming exercises using deep learning. In *Proc. Int. Conf. Educational Data Mining*, 2017.

[15] B. Weisfeiler and A. Lehman. A reduction of a graph to a canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9):12–16, 1968.

APPENDIX

A. MINING ALGORITHM

We incrementally explore the space of all possible context graphs by extending it step by step (add an edge and/or node). To evaluate which extension improves the objective function most, we need to match a new context C' to the graphs $G \in \mathcal{G}$, that is, we have to make sure that a bijective mapping σ exists. Finding the possible mappings σ each time from scratch would imply substantial computational cost (especially as we have matched the first $n-1$ of the n nodes in the previous expansion already), so we keep track of the mappings σ that have been used for the current context so far. The bookkeeping is illustrated in Fig. 9. On the top left a context graph and on the top right excerpts from a code graph are shown. The node sets V consist of natural numbers, which are shown in subscript next to the assigned labels. Initially we start with a context graph C_0 that consists of a `focus` node alone ($V = \{1\}$, $\lambda(1) = \texttt{focus}$). At that time, the mapping σ_{C_0} needs to map only node #1 to node #3. Below the two graphs, the row σ_{C_0} shows the mapping ($1 \to 3$ or $\sigma(1) = 3$). Suppose C_0 is extended by a `codeline` node to context C_1. Now σ_{C_1} needs to map two nodes, but rather than starting from scratch, we assume that the previous $n-1$ nodes have been assigned already and only the new node #2 has to be assigned. There is only one possibility in the code graph of Fig. 9, we map $2 \to 5$. In the figure σ_{C_1} corresponds to a list of length 1, with a single entry ($2 \to 5$). This entry, however, also points (vertically) to the previous assignment $1 \to 3$: Following the pointers in vertical direction reconstructs the full mapping σ_{C_1} ($1 \to 3$, $2 \to 5$). So the σ-mapping of a new context is based on an existing successor node mapping and we only supplement the assignment of the latest node to complete it.

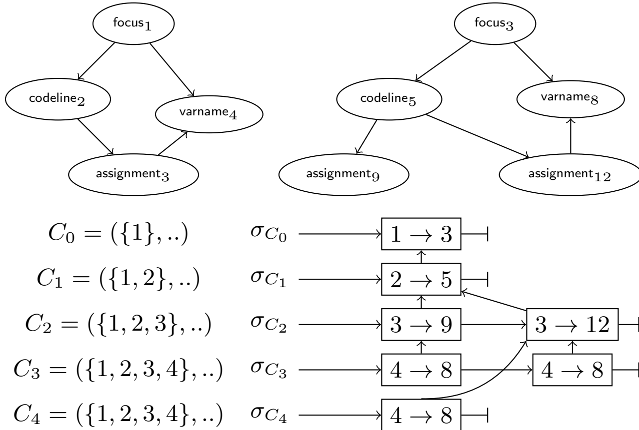


Figure 9: Efficient data structure to keep track of node assignments during the search. As a context pattern evolves (node by node) we re-use all mappings σ of successor graphs.

The main algorithm is shown in Alg. 1, which takes the set $\mathcal{G}$ of code graphs and a start context C_0 (usually consisting of the `focus` node alone). In line 1 we initialize the search front S, which is a priority queue of limited size (we used 1000) ordered by (2). By Σ_0 we denote the set of all valid initial mappings $\Sigma_0 = \{\{\sigma_0^G\} \mid G \in \mathcal{G}\}$, where σ_0^G maps the unique `focus` node in the context to the unique `focus` node in the code graph G.

In line 2 we initialize a set X of explored nodes: it may happen at some point of the search that we re-consider a context graph we have visited earlier, simply because the nodes and edges of the same context graph could have been inserted in a different order. To avoid wasting time on examining graphs that were already explored, we store a hash code of all explored context graphs in X (set of explored graphs). In our implementation, we use a hash of the Weisfeiler-Lehman kernel [15] for this purpose. The remainder of Alg. 1 represents the search loop: we pick the best-so-far context graph C, expand it, check for an improvement and return the best found context C^* in the end.

Algorithm 1 ContextMining($\mathcal{G}$,C_0)

1: $S = \{(C_0, \Sigma_0)\}$ ▷ search front as priority queue
2: $X = \emptyset$; $(C^\star, \Sigma^\star) = (\perp, \perp)$ ▷ visited nodes, best context
3: **while** $S \neq \emptyset$ **do** ▷ queue not empty
4: $\quad (C, \Sigma) = \text{top}(S)$ ▷ pick best context from queue
5: $\quad S = \text{expand}(C, \Sigma, S, X)$ ▷ expand context
6: $\quad$ **if** $C^\star = \perp \vee f(C) > f(C^\star)$ **then** ▷ improvement?
7: $\quad\quad C^\star = C$; $\Sigma^\star = \Sigma$ ▷ update best-so-far
8: $\quad$ **end if**
9: $\quad X = X \cup \{h(C)\}$ ▷ save hash of explored graph
10: **end while**
11: **return** $(C^\star, \Sigma^\star)$ ▷ return best context graph

Algorithm 2 expand(C,Σ,S,X)

1: $Q = \text{explore}(C, \Sigma)$; $E = \emptyset$
2: **for** $e \in Q$ **do** ▷ for all possible extensions
3: $\quad$ apply extension e to context C and obtain C'
4: $\quad$ **if** $h(C') \notin X \wedge \text{size}(C') < \text{limit}$ **then**
5: $\quad\quad \Sigma' = \text{prolong-and-filter}(C', \Sigma)$
6: $\quad\quad$ **if** C' is substantial improvement of C **then**
7: $\quad\quad\quad C'.\text{momentum} = 2$ ▷ restore to max
8: $\quad\quad$ **else**
9: $\quad\quad\quad C'.\text{momentum} = C.\text{momentum} - 1$
10: $\quad\quad$ **end if**
11: $\quad\quad$ **if** $f(C') > \text{worst}(S) \wedge C'.\text{momentum} > 0$ **then**
12: $\quad\quad\quad S = S \cup \{(C', \Sigma')\}$
13: $\quad\quad$ **end if**
14: $\quad$ **end if**
15: **end for**
16: **return** S

The node expansion is shown in Alg. 2. It calls a function explore(.) in line 1, which provides a priority queue (size:60) of the best possible extensions of the current context C (which includes all mentioned types of edges). This allows us to concentrate on the best extensions before we actually expand the context in the search front S. We exclude a new context if it has been explored earlier (line 4) or it becomes too large (max. 12 nodes per context). If the extended context passed all checks, we prolong or adapt the σ-mappings (line 5) to reflect newly inserted nodes. We assign a *momentum* to each context, which is initialized to a small number (here: 2) and reduced by 1 for each extension that *did not* lead to a *substantial* improvement in the object function (2). If the momentum has reached 0, the context graph will not be considered further in the search front. We use a statistical test (G-test) on the 2x2 contingency table of positive and negative cases before and after the extension to decide whether the extension was substantial.