Automated Personalized Feedback in Introductory Java Programming MOOCs

Victor J. Marin, Tobin Pereira
Rochester Institute of Technology, USA
vxm4964@rit.edu, ttp5232@rit.edu

Srinivas Sridharan
Stevens Institute of Technology, USA
ssridha3@stevens.edu

Carlos R. Rivero
Rochester Institute of Technology, USA
crr@cs.rit.edu

***Abstract*—Currently, there is a "boom" in introductory programming courses to help students develop their computational thinking skills. Providing timely, personalized feedback that makes students reflect about what and why they did correctly or incorrectly is critical in such courses. However, the limited number of instructors and the great volume of submissions instructors need to assess, especially in Massive Open Online Courses (MOOCs), prove this task a challenge. One solution is to hire graders or create peer discussions among students, however, feedback may be too general, incomplete or even incorrect. Automatic techniques focus on: a) Functional testing, in which feedback usually does not sufficiently guide novices; b) Software verification to find code bugs, which may confuse novices since these tools usually skip true errors or produce false errors; and c) Comparing using reference solutions, in which a large amount of reference solutions or pre-existing correct submissions are usually required. This paper presents a semantic-aware technique to provide personalized feedback that aims to mimic an instructor looking for code snippets in student submissions. These snippets are modeled as subgraph patterns with natural language feedback attached to them. Submissions are transformed into extended program dependence graphs combining control and data flows. We leverage subgraph matching techniques to compute the adequate personalized feedback. Also, constraints correlating patterns allow performing fine-grained assessments. We have evaluated our method on several introductory programming assignments and a large number of submissions. Our technique delivered personalized feedback in milliseconds using a small set of patterns, which makes it appealing in real-world settings.**

I. Introduction

University enrollment in computing courses is increasing worldwide [10]. Several universities reported 1,000+ students taking an introductory programming course in 2015, e.g., Universities of Michigan, Toronto, and Washington in North America [10]. The Computing Research Association 2015 Taulbee Survey [41] reports that enrollment in computing majors in North America has increased 22.5% from 2014 to 2015, the eighth year in a row with steady increasing enrollments. As a result, the number of novice students is starting to become unmanageable, and there is a need to deal with this current "boom" in computing courses [4].

Introductory programming courses rely on computational thinking that consists of providing a description of a program's functionality and, then, the students must provide a suitable solution [39]. Students taking such courses tend to come from different backgrounds and require strong guidance in the form of personalized feedback, i.e., an explanation of what and why they did correctly and/or incorrectly in their submissions [22]. As a result, there is a need for providing timely and personalized feedback to novices [15], [28], [33].

Providing personalized feedback can be challenging when dealing with hundreds of student submissions for a single assignment. Universities have traditionally addressed the problem by using graders, who are usually students that passed the course in previous years. The problem is that, if the number of assignments and submissions is large, many graders need to be hired, who have to go through each submission manually. Furthermore, some graders may provide more personalized feedback than others, or they may have different criteria when grading the same assignment [5]. Another approach is using peer discussions with other students taking the course and/or graders [10]. Unfortunately, students may need to wait hours to get feedback that may be too general, incomplete or even incorrect, especially if it comes from other students that are struggling with the course [32], [33].

The challenge of providing personalized feedback is exacerbated in Massive Open Online Courses (MOOCs), where an assignment may have hundreds of thousands of submissions [6]. Several companies like *edX.org* or *Coursera.org* are currently offering these MOOCs, in which two methods have been exploited to overcome the challenge of providing personalized feedback [33]: a) Discussions with peers; and b) Functional testing. On one hand, students can post answers and discuss solutions in forums that are moderated by instructors and/or graders. However, this approach has the same problems as described above for graders and peer discussions. On the other hand, student submissions are assessed using predefined tests that aim to trigger as many errors as possible. Unfortunately, personalized feedback provided by functional tests usually does not sufficiently guide novice students, and devising such tests to trigger all possible errors is challenging [7], [33].

There is a new trend of using automatic approaches more sophisticated than functional testing: a) Software verification tools; and b) Assessment based on reference solutions. Automated software verification aims to find software bugs like buffer overflow or division by zero, and it is also possible to analyze the functional correctness of the code using a functional specification [37]. However, personalized feedback provided by these tools may confuse novice students since they usually skip true errors or produce false errors, that is, reported bugs that are not real bugs [37]. Other approaches rely on one or more reference solutions to a given assignment that are compared over student submissions [14], [15], [27], [28], [33], [36], [40]. Their main drawback is that multiple reference

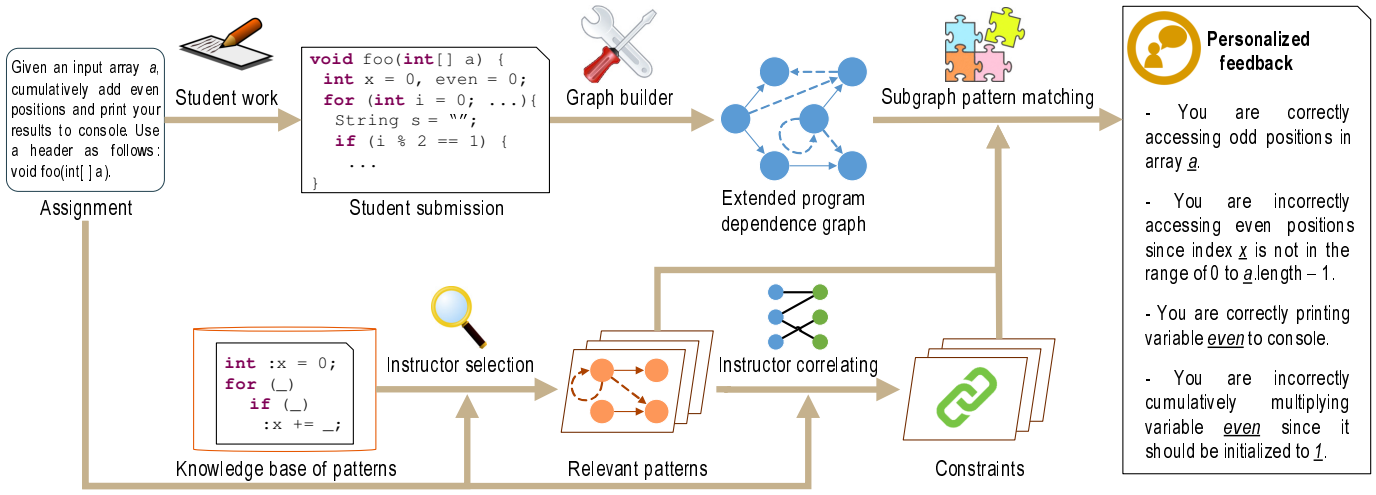


Fig. 1. Overview of our approach: for a given assignment, instructors select relevant patterns they expect to find in student submissions from a provided knowledge base. Also, instructors can set different types of constraints among such patterns to correlate them and provide more fine-grained feedback. Given a submission, our technique transforms it into an extended program dependence graph and performs subgraph pattern matching over it to provide personalized feedback associated with the patterns and constraints indicated by the instructor.

solutions are usually required for the same assignment since student submissions and reference solutions are compared as a whole [20]. Furthermore, some of them also have scalability problems that prevent them from being used in real-world settings [15], or require a large amount of pre-existing correct submissions [15], [27], [28].

In this paper, we propose a novel, semantic-aware technique to providing personalized feedback for Java assignments. It is based on the grading approach of an instructor looking for code snippets that she expects to find in student submissions [9], e.g., accessing odd positions in an array a. Such code snippets are modeled as subgraph patterns with feedback attached to them, e.g., "you are incorrectly accessing odd positions since the index is not in the range of 0 to $a.length - 1$". Patterns are stored in a knowledge base and instructors are responsible for selecting relevant patterns to a given assignment (see Figure 1). Our technique provides personalized feedback by matching patterns independently from each other over a submission. Patterns are generic enough to be reused in multiple assignments. However, some assignments may need more fine-grained assessment, so we introduce constraints that correlate patterns to perform this assessment. As far as we know, this is the first approach that focuses on static analysis and discovering semantic patterns to provide personalized feedback in introductory programming MOOCs.

The contributions of this paper are as follows:

- We devise a framework in which submissions are modeled using extended program dependence graphs [8] that combine control and data flows and include information on the operations that are being performed, e.g., accessing an array. Patterns are modeled as subgraph queries, and feedback as natural language templates that are instantiated with variable names and constants from student submissions. We provide three types of constraints as follows: a) Ensuring that two nodes from different patterns match the same node in a submission; b) There exists an edge between two nodes in a submission that comes from two different patterns; and c) A given node in a submission contains variables coming from several patterns.
- We leverage subgraph matching techniques borrowed from the graph database field to perform the matching and to provide personalized feedback [24]. We extend the notion of embeddings [17] (solutions) of a subgraph query in a data graph to include, not only the matching nodes, but also the matching variables. We also allow approximations in these embeddings to provide personalized feedback on errors. We develop an algorithm that takes embeddings from multiple patterns as input, and enforce constraints correlating patterns in a best-effort approach capable of dealing with multiple expected methods.
- We devise a publicly-available knowledge base of patterns and constraints that can be used for providing personalized feedback for twelve real-world assignments that are used in the "Introduction to Programming" course offered by the Massachusetts Institute of Technology (MIT) in *edX.org*, the "Fundamentals of Computing" course offered by Indian Institute of Technology Kanpur (IIT Kanpur), and "Computer Science I" offered by the Rochester Institute of Technology (RIT) [15], [33]. Our knowledge base contains twenty four unique patterns and eighty six constraints.
- We provide a publicly-available Java implementation and evaluate it with 1.6M submissions per assignment on average. Such submissions were synthetically generated following the same hypothesis as Singh et al. [33] and Pu et al. [28] that errors in student submissions in introductory programming courses are predictable. We used rules encoding common student errors, making the search space of correct and incorrect student submissions explicit. Submissions are only available by request to avoid plagiarism. Our implementation took milliseconds on average to

provide personalized feedback for each submission, which entails that our approach is appealing for real-world settings. We used functional testing to ensure the accuracy of our personalized feedback.

- We discuss benefits, drawbacks, and limitations of our technique. We compare our technique with current state-of-the-art techniques to providing personalized feedback, i.e., AutoGrader [33] and CLARA [15].

This paper is organized as follows: Section II presents the related work. Section III describes our framework that comprises extended program dependence graphs, patterns, and constraints with feedback attached to them. Sections IV and V describe our pattern and submission matching algorithms, respectively. In Section VI, we present our experimental results. Finally, Section VII recaps our main conclusions.

II. Related work

Efforts to providing personalized feedback have mainly focused on functional testing (see [21] for a survey), which consists of running predefined tests over student submissions that aim to trigger as many errors as possible. Unfortunately, personalized feedback usually comes as "Test *i* was [not] passed", which does not sufficiently guide novice students [7]. Additionally, devising functional tests to trigger all possible errors is not a trivial task. Comparing expected and actual outputs is challenging, especially if the assignment requires printing to console [18].

Vujosevic-Janicic et al. [37] proposed to use a combination of automated software verification, control flow graph similarity measure, and automated functional testing to grade student submissions. Specifically, automated software verification tools can be used to provide personalized feedback to students. These tools can find software bugs like buffer overflow or division by zero, however, the following drawbacks may confuse novice students [37]: a) All of these tools rely on approximations, so they cannot report all the errors in a program without skipping true or producing false errors; and b) Such tools detect more errors than needed, e.g., an integer overflow in memory allocation will be likely ignored in an introductory programming course. Additionally, automated software verification tools can analyze the functional correctness of the code using a functional specification. However, defining specifications requires a high level of expertise. Such functional specifications can be used to evaluate pre- and post-conditions of a method based on its returned values [19]. However, when the assignment consists of printing results to console (a typical scenario in introductory programming courses [15], [33]), these tools are not able to verify them. In practice, a loop or condition that we wish to verify must be annotated with a functional specification of its expected behavior [23]. Therefore, these annotations are based on the exact same variables used in the code. In our setting, this will require an instructor to annotate each student submission, which is prohibiting, or to force students to use the same template for a given assignment, which will not help them developing their computational problem-solving skills [39].

Another approach to providing personalized feedback consists of comparing student submissions with one or more reference solutions [14], [15], [27], [28], [33], [36], [40]. The main problem of this approach is that multiple reference solutions are usually required for the same assignment since submissions and reference solutions are compared as a whole, i.e., if an assignment consists of multiple parts that can be performed in several ways, we need to assemble a reference solution per combination [20]. Xu and Chee [40] and Truong et al. [36] use several transformation rules to normalize submissions and reference solutions, which are compared between them after normalization by translating them into program dependence graphs. Gulwani et al. [14] focus on providing personalized feedback on performance and problem-solving strategies. Piech et al. [27] and Pu et al. [28] exploit machine learning to provide personalized feedback that requires the existence of a large amount of previous student submissions, which is not always the case [27]. Furthermore, Piech et al. [27] require the number of variables to be fixed beforehand, which may be not beneficial for novice students developing their problem-solving skills [12]. Pu et al. [28] require functional tests to trigger errors in the submissions.

The most related techniques to our work are [15], [33], which are both based on reference solutions. Singh et al. [33] propose an error model language to define rules describing possible errors students may produce, e.g., initializing an array index in 1 instead of 0. The technique transforms each student submission using these rules into a program sketch, i.e., a program that contains multiple choices [34]. Then, a correct program is assembled from the program sketch by ensuring functional equivalence with a single reference solution. Personalized feedback is computed by analyzing the repairs to transform the original student submission into the correct program. One problem with this technique is that error rules are difficult to specify and depend completely on each assignment [28]. Additionally, its performance degrades considerably after four or more repairs [15]. It cannot deal with printing to console assignments and infinite loops, a major drawback in introductory programming courses [38].

Gulwani et al. [15] cluster correct submissions using variable traces computed for different inputs. Then, a representative submission from each cluster is selected as a reference solution. An incorrect submission is compared to each reference solution using variables traces, and some repairs are computed. The technique provides personalized feedback using the reference solution with the least number of repairs. A problem of this technique is that it requires inputs that are not easy to provide to trigger all possible errors. Variable traces are compared as a whole, so it needs a reference solution per any possible variation of a given assignment. Furthermore, the technique is not able to deal with infinite loops and submissions with multiple methods.

Different than Singh et al. [33] and Gulwani et al. [15], our method relies on understanding the desired semantics an instructor is expected to find, i.e., the repairs that both techniques compute are the result of analyzing the functional equivalence of a student submission over one or more reference solutions. As a result, personalized feedback generated using these techniques is usually very low level, and more abstract personalized feedback is required in the context of introductory programming courses [15]. In addition, the personalized feedback generated by these techniques lacks of context since they suggest to add/delete/replace one line of code without

any further reason, which does not help students developing their computational thinking skills. Our technique is based on static analysis of the code and, therefore, it does not have the drawbacks of dynamic analysis like infinite loops. We aim to capture the intent of the student by finding relevant patterns in their code. Thus, we can provide feedback according to the goal that the student was attempting to accomplish when typing a particular snippet of code. With the help of this feedback, students can understand what pieces of their whole composition require modification and guide them to perform these changes by themselves, thereby developing their computational thinking skills.

III. Framework

Our framework comprises extended program dependence graphs (Section III-A), patterns with feedback attached to them that are matched over an extended program dependence graph (Section III-B), and constraints correlating several patterns for fine-grained assessment (Section III-C).

A. Extended program dependence graphs

We extend the concept of program dependence graph [25] to incorporate additional information that will help us with the matching process. We define nodes and edges in such a graph.

Definition 1: A graph node $v = (t_v, c)$ is a tuple where $t_v \in \{Assign, Break, Call, Cond, Decl, Return\}$ is its type, and c is a Java expression denoting the operation the node is performing. The graph node types are as follows:

- *Assign* entails an assignment expression like $x \mathrel{+}= y$.
- *Break* expression.
- *Call* entails expressions like $System.out.print(x)$.
- *Cond* entails loop, if or switch expressions.
- *Decl* entails a declaration of a parameter in a method.
- *Return* entails a return expression like *return* $x + y$.

Definition 2: A graph edge $e = (v_s, v_t, t_e)$ is a tuple where v_s and v_t correspond to the source and target nodes, respectively, and $t_e \in \{Ctrl, Data\}$ is its type. The graph edge types are as follows:

- There is a *Ctrl* graph edge from a graph node v_s to another graph node v_t when v_s has type *Cond* and the truth of its expression controls the execution of v_t.
- There is a *Data* graph edge from a graph node v_s to another graph node v_t when: 1) A variable x is assigned or declared in v_s, 2) v_t reads x, and 3) There is an execution path in the student submission from v_s to v_t in which x is never reassigned.

Definition 3: An extended program dependence graph $g = (V, E)$ of a student submission s is a tuple where V is a set of graph nodes such that $v = (t_v, c) \in V$ is an expression in s. Furthermore, E is a set of graph edges such that $e = (v_s, v_t, t_e) \in E$ and $\{v_s, v_t\} \subseteq V$.

We use the following assignment to illustrate our technique:

Assignment 1: (Adapted from MIT's "Introduction to Computer Science and Programming Using Python" at *edX.org* [33].) Given an input array, devise a Java method that adds odd positions and multiplies even positions in the array. Print to console your results. Use a header as follows: *void assignment*$1(int[\ ]\ a)$.

Figure 2 presents three sample student submissions to the previous assignment. Figure 2a shows a submission that is incorrect because variable *even* should be initialized to 1, i in the *for* loop should not go beyond *a.length*, we need to use $i \ \% \ 2 == 0$ to access even positions, and variable *even* needs to be printed to console. The submission in Figure 2b is correct. Figure 2c presents a student submission that is incorrect because variables x and y should be initialized to 1 and 0, respectively.

Figure 3a presents the extended program dependence graph generated from the submission in Figure 2a, in which the node types are represented as labels and the Java expressions corresponding to each node are depicted in Figure 3b. Additionally, we represent *Data* and *Ctrl* graph edges as solid and dashed arrows, respectively.

Note that Definition 2 allows transitive *Ctrl* graph edges, e.g., the execution of node v_7 depends on the truth of nodes v_6 and v_4. However, the execution of v_6 also depends on the truth of node v_4. If we take all of these transitive *Ctrl* graph edges into account, the resulting graph can be overloaded with redundant relationships that will hinder the matching process. As a result, we remove them from each final extended program dependence graph.

Another concern is whether or not taking into account that loop or if conditions may not be fulfilled, e.g., if the *for* loop in Figure 2a is not satisfied, like when receiving an empty array, the last time variable *odd* is assigned is $odd = 0$ and not $odd \mathrel{+}= a[i]$. As a result, there may be a *Data* graph edge from v_1 to v_{10}. In addition, if we consider that the same loop may iterate multiple times, there will be a *Data* graph edge from v_5 to v_6, v_7, v_8 and v_9. In the literature, some approaches generate some of these edges, e.g., Baah et al. [1], while others do not take them into account, e.g., Bhattacharjee and Jamil [2]. The reason behind these discrepancies is that program dependence graphs are informally defined and, as a result, there may be several correct graphs for a single given submission. In our case, we follow the same approach as Bhattacharjee and Jamil [2], and do not generate *Data* graph edges considering that loop or if conditions may not be fulfilled, or that loops may iterate multiple times.

B. Patterns and feedback

We use patterns to mimic the process of an instructor looking for code snippets in submissions. We extend the notion of graph nodes previously defined for extended program dependence graphs allowing incomplete Java expressions.

Definition 4: A pattern node $u = (t_u, r, \hat{r}, f_c, f_i)$ is a tuple where $t_u \in \{Assign, Break, Call, Cond, Decl, Return, Untyped\}$ is its type, r is an incomplete Java expression, $\hat{r}$ is an approximate incomplete Java expression, and f_c and f_i are natural language templates that encode feedback for when the submission is correct or incorrect, respectively. Let X and Y be the set of variables in r and $\hat{r}$, respectively, then $Y \subseteq X$.

```
void assignment1(int[] a) {
 int even = 0;
 int odd = 0;
 for (int i = 0;
    i <= a.length ; i++) {
   if (i % 2 == 1)
    odd += a[i];
   if (i % 2 == 1)
    even *= a[i];
 }
 System.out.println(odd);
 System.out.println(even);
}
```

(a) Incorrect submission.

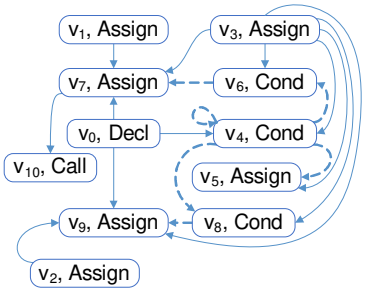
```
void assignment1(int[] a) {
 int o = 0, e = 1;
 int i = 0;
 while (i < a.length ) {
   if (i % 2 == 1)
    o += a[i];
   if (i % 2 == 0)
    e *= a[i];
   i++;
 }
 System.out.print(o +
     ", " + e);
}
```

(b) Correct submission.

```
void assignment1(int[] a) {
 int x = 0, y = 1;
 for (int i = 0;
   i < a.length ; i++)
  if (i % 2 == 1)
   x *= a[i];
 for (int i = 0;
   i < a.length ; i++)
  if (i % 2 == 0)
   y += a[i];
 System.out.print(
   "O: " + x + ", E: " +y);
}
```

(c) Incorrect submission.

Fig. 2. Sample student submissions for Assignment 1.

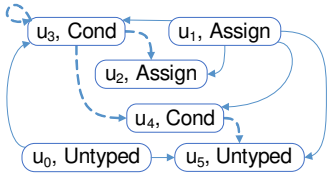


(a) Program dependence graph.

v0: a
v1: odd = 0
v2: even = 0
v3: i = 0
v4: i <= a.length
v5: i++
v6: i % 2 == 1
v7: odd += a[i]
v8: i % 2 == 1
v9: even *= a[i]
v10: System.out.println(odd)

(b) Java expressions.

Fig. 3. Extended program dependence graph of Figure 2a.



(a) Program dependence graph.

u0: s	
u1: x = 0	(.*x.*)
u2: x++	(x.*)
u3: x < s.length	(x <= s.length)
u4: x % 2 == 1	
u5: s[x]	(.*s\[.*x.*\].*)

(b) Java expressions.

Fig. 4. Pattern p_o: Accessing odd positions sequentially in an array.

Note that we add *Untyped* to the set of possible pattern node types since we may allow a pattern node to match with every graph node independent from its type. We refer to incomplete Java expressions as partial Java expressions that are matched over Java expressions in the submissions. An approximate incomplete Java expression is incomplete and allows us to loosely match over a Java expression.

Definition 5: A pattern $p = (U, F, f_p, f_m)$ is a tuple where U is a set of pattern nodes, F is a set of graph edges such that $(u_s, u_t, t_e) \in F$ and $\{u_s, u_t\} \subseteq U$, and f_p and f_m are feedback messages we wish to deliver when this pattern is present or missing in a student submission, respectively.

Figure 4 presents pattern p_o that encodes the code snippet of accessing odd positions sequentially in an array. Nodes u_0 and u_5 in the program dependence graph in Figure 4a are *Untyped*. The Java expressions corresponding to each node are shown in Figure 4b, in which some of them are incomplete, e.g., u_5. We also show the approximate expression of each node in which we use regular expressions to represent some of them, e.g., node u_3 is enforcing that we may approximately match $x <= s.length$ (a common error of novice students), while u_1 is encoding any Java expression containing variable x. The feedback messages if p_o is present or missing are "You are correctly accessing odd positions sequentially in an array" and "You are not accessing odd positions sequentially in an array, please, consider using a loop and a condition; recall that odd is computed by $i \% 2 == 1$, where i is an index variable", respectively. The following feedback is attached to each pattern node in p_o:

- u_1 Correct: "x is initialized to 0". Incorrect: "x should be initialized to 0".
- u_2 Correct: "x is incremented by 1". Incorrect: "x should be incremented by 1".
- u_3 Correct: "x does not go beyond $s.length-1$". Incorrect: "x is out of bounds going beyond $s.length - 1$".
- u_4 Correct: "You are using $x \% 2 == 1$ to control that x is odd".
- u_5 Correct: "x is used exactly to access s". Incorrect: "You should access s by using x exactly".

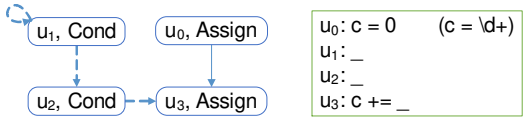
Note that we do not provide incorrect feedback for node u_4. These are crucial nodes since, if they do not match with the submission, we are not able to recognize such pattern.

We match patterns over extended program dependence graphs. The matching task is divided into a structural matching of nodes and a variable matching within the nodes. We first define a variable matching.

Definition 6: Let $\gamma : X \mapsto Y$ be a mapping between two sets of variables X and Y. Let r and c be two Java expressions such that X and Y are sets of variables in r and c, respectively. We denote that r matches c using γ as $r \preceq_\gamma c$, in which we substitute X by Y in r using γ, and match it over c.

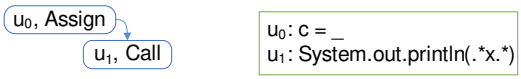
The matching task of the node contents can be addressed in different ways. We rely on regular expressions in this paper, e.g., pattern p_o in Figure 4. However, it is also possible to use other methods like tree or subgraph matching [31], [35], graph-edit distance [13], and/or variable traces [15].

Definition 7: An embedding $m = (\iota : U \mapsto V, \gamma : X \mapsto Y)$ of a pattern $p = (U, F, f_p, f_m)$ in an extended program dependence graph $g = (V, E)$ is a tuple, where ι maps pattern nodes



(a) Program dependence graph. (b) Java expressions.

Fig. 5. Pattern p_a: Conditional cumulatively adding.



(a) Program dependence graph. (b) Java expressions.

Fig. 6. Pattern p_p: Assign and print to console.

in p (U) to graph nodes in g (V), X and Y denote variables in p and g, respectively, and γ maps variables in X to variables in Y. Also, the following conditions hold:

$$\forall u = (t_u, r, \hat{r}, f_c, f_i) \in U \Rightarrow (t_v, c) = \iota(u) \wedge (t_u = \mathit{Untyped} \vee t_u = t_v) \wedge (r \preceq_\gamma c \vee \hat{r} \preceq_\gamma c) \quad (1)$$

$$\forall (u_s, u_t, t_e) \in F \Rightarrow (\iota(u_s), \iota(u_t), t_e) \in E \quad (2)$$

A sample embedding of pattern p_o in Figure 4 in the extended program dependence graph in Figure 3 is as follows: $(\{\iota(u_0) = v_0, \iota(u_1) = v_3, \iota(u_2) = v_5, \iota(u_3) = v_4, \iota(u_4) = v_6, \iota(u_5) = v_7\}, \{\gamma(s) = a, \gamma(x) = i\})$. Note that u_3 approximately matches with v_4.

C. Constraints

Patterns are reusable and can be used to assess several assignments since they are checked in isolation from each other. However, we usually wish to enforce additional constraints that are specifically tailored to a given assignment. We use three types of constraints to perform this fine-grained assessment as follows: equality, edge existence, and containment.

Definition 8: An equality constraint is a tuple (p_i, u_i, p_j, u_j), where p_i and p_j are patterns, and u_i and u_j are pattern nodes in p_i and p_j, respectively. Let (ι_i, γ_i) and (ι_j, γ_j) be two embeddings of p_i and p_j in an extended program dependence graph g, respectively. Then, the following condition holds: $u_i \in \mathit{dom}\ \iota_i \wedge u_j \in \mathit{dom}\ \iota_j \wedge \iota_i(u_i) = \iota_j(u_j)$.

Equality constraints enforce that two pattern nodes from different patterns match the same graph node in a student submission. For instance, pattern p_o helps us identify if we are accessing odd positions sequentially in an array. Pattern p_a in Figure 5 is used to ensure that a variable is cumulatively added under a certain condition. However, in Assignment 1 we need to enforce that the same variable is conditional cumulatively added using odd positions of the input array. We use equality constraint (p_o, u_5, p_a, u_3) to enforce such behavior, that is, node u_3 in p_a ensures that a variable is cumulatively added, and node u_5 in p_o that odd positions are accessed.

Definition 9: An edge existence constraint is a tuple $(p_i, u_i, p_j, u_j, t_e)$, where p_i and p_j are patterns, u_i and u_j are pattern nodes in p_i and p_j, respectively, and $t_e \in \{\mathit{Ctrl}, \mathit{Data}\}$ is an edge type. Let (ι_i, γ_i) and (ι_j, γ_j) be two embeddings of p_i and p_j in an extended program dependence graph $g = (V, E)$, respectively. Then, the following condition holds: $u_i \in \mathit{dom}\ \iota_i \wedge u_j \in \mathit{dom}\ \iota_j \wedge (\iota_i(u_i), \iota_j(u_j), t_e) \in E$.

Edge existence constraints enforce that there exists an edge of an specific edge type between two pattern nodes from different patterns in a student submission. An example is $(p_a, u_3, p_c, u_1, \mathit{Data})$ that enforces that the variable that is printing in pattern p_p in Figure 6 is the same variable that was cumulatively added in pattern p_a.

Definition 10: A containment constraint is a tuple (p, u, r, P), where p is a main pattern, u is a pattern node in p, r is an incomplete Java expression, and P is a set of supporting patterns. Let X and X_i be the set of variables in p and $p_i \in P$, respectively, such that $X \cap X_i = \emptyset, i = 1..|P|$ and $\bigcup_{i=1}^{|P|} X_i = \emptyset$. Let (ι, γ) be an embedding of p in an extended program dependence graph g, M a set of embeddings of P in g such that for each pattern $p_i \in P$ there is an embedding $(\iota_i, \gamma_i) \in M$, and $\gamma' = \gamma \cup \{\gamma_i \mid (\iota_i, \gamma_i) \in M\}$. Then, the following condition is fulfilled: $u \in \mathit{dom}\ \iota \wedge \iota(u) = (t_v, c) \wedge r \preceq_{\gamma'} c$.

A containment constraint enforces incomplete Java expressions in the contents of a specific pattern node using other supporting patterns. Containment constraint $(p_o, u_5, c \mathrel{+}= s[x], \{p_a\})$ checks that the node that matches u_5 in p_o contains a expression $c \mathrel{+}= s[x]$, in which c is a variable from pattern p_a.

IV. Pattern Matching

The core of our approach consists of performing subgraph matching of a pattern over an extended program dependence graph. Recall that patterns contain nodes and edges, and the nodes also contain expressions with variables that we need to match. As a result, we rely on subgraph matching techniques to match nodes and edges at the structural level [24], and we extend them to take variables into account as well.

Algorithm 1 presents how we compute the embeddings M that result from matching a pattern p over an extended program dependence graph g. We initially compute the search space Φ (see lines 2–4), which is the set of graph nodes in g that may match with each pattern node in p according to their types. For instance, the search space of p_a in Figure 5 over the extended program dependence graph in Figure 3 is: $\Phi(u_0) = \Phi(u_3) = \{v_1, v_2, v_3, v_5, v_7, v_9\}$, $\Phi(u_1) = \Phi(u_2) = \{v_4, v_6, v_8\}$.

We then start a backtracking search in which the initial call uses an empty embedding $(\emptyset, \emptyset)$ (see line 6). In the base case, we store the current embedding (ι, γ) in M when all the pattern nodes are present in it (see lines 8–9). Otherwise, we take a pattern node u that is still not present in the embedding (line 11), and iterate over the search space of u but only over graph nodes that are not in the embedding yet, since we only allow a graph node to be matched to a single pattern node (line 12). Pattern node u matches graph node v if, for each outgoing neighbor u_i in the current embedding, there is a graph node in the embedding $\iota(u_i)$ that is an outgoing neighbor of v, and the types of the graph edges are the same (see lines 13–14). If that is the case, we add $u = v$ to the current embedding (line 15) and remove it to continue with the backtracking process (line 26). For instance, assume $\iota(u_1) = v_4$ for pattern p_a over the extended program dependence graph in Figure 3.

Algorithm 1: PatternMatching

Input: A pattern $p = (U, F, f_p, f_m)$, an extended program dependence graph $g = (V, E)$.
Output: A set of embeddings M

```
1  // 1: Compute search space Φ.
2  foreach u = (t_u, r, r̂, f_c, f_i) ∈ U do
3      Φ(u) := {v | v = (t_v, c) ∈ V ∧
4                   (t_u = Untyped ∨ t_u = t_v)}
5  // 2: Backtracking search.
6  Search(0, p, g, Φ, (∅, ∅), M)

7  void Search(j, (U, F, f_p, f_m), (V, E), Φ, (ι, γ), M)
8      if dom ι = U then
9          M := M ∪ {(ι, γ)}
10     else
11         u := (t_u, r, r̂, f_c, f_i) ∈ U \ dom ι
12         foreach v := (t_v, c) ∈ Φ(u) | v ∉ ran ι do
13             if ∀(u, u_i, t_e) ∈ F | u_i ∈ dom ι ⇒
14                                 (v, ι(u_i), t_e) ∈ E  then
15                 ι := ι ∪ {u = v}
16                 X := Variables(r) \ dom γ
17                 Y := Variables(c) \ ran γ
18                 if |X| = |Y| then
19                     foreach Z ∈ Combinations(X, Y) do
20                         γ := γ ∪ Z
21                         if r ⪯_γ c ∨ r̂ ⪯_γ c then
22                             Mark u as correct/incorrect in ι
23                             Search(j + 1, (U, F, f_p, f_m),
24                                    (V, E), Φ, (ι, γ), M)
25                         γ := γ \ Z
26                 ι := ι \ {u = v}
```

When performing the recursive call, we have two potential matches: $\iota(u_2) = v_6$ and $\iota(u_2) = v_8$. Since there is an edge $(u_1, u_2, \mathit{Ctrl})$ in the pattern, we look for edges $(v_4, v_6, \mathit{Ctrl})$ and $(v_4, v_8, \mathit{Ctrl})$, respectively; however, the former exists while the latter is not present in the extended program dependence graph, which means that we will discard $\iota(u_2) = v_8$ as a feasible embedding.

In the next step, we need to match the variables in the current node. To perform this, we take only into account the variables in r and c that are not present in the current embedding (lines 16 and 17). We only proceed to the next step if the number of variables is the same in both sets (line 18), in which we compute all the combinations of the variables using *Combinations* and iterate over them (line 19). Note that $\mathit{Combinations}(\emptyset, \emptyset) = \{\emptyset\}$. We add the new matchings to the embedding (line 20), and remove them to continue the backtracking process (line 25). We check if r or $\hat{r}$ in the current node match with c in the current node using the current embedding (line 21). We mark the current node as correct or incorrect if $r \preceq_\gamma c$ or $\hat{r} \preceq_\gamma c$, respectively. If we are not able to match the variables, we will discard the current embedding and continue with our search. Otherwise, we mark u as correct or incorrect, and perform a recursive call (lines 22–24). Assume that we are matching node u_3 in pattern p_o in Figure 4 over node v_4 in the extended program dependence graph in Figure 3. We will try the following combinations of variables: $\{\gamma(s) = i, \gamma(x) = a\}$, and $\{\gamma(s) = a, \gamma(x) = i\}$. The former will never produce a match while the latter will produce one using the approximate incomplete Java expression ($i <= a.length$), which means that we mark it as incorrect since we have been able to detect the pattern but the submission is not completely correct (it should be $i < a.length$).

In the worst case, the time complexity of this algorithm is $O(n^m)$, where n and m represent the sizes of the extended program dependence graph and pattern, respectively. This complexity is due to the NP-hard nature of the subgraph matching problem [11]. However, in practice, the performance depends on the size of the search space and the processing order of the pattern nodes. Note that subgraph matching techniques [16], [17], [29], [30], and their optimizations [3], can be used to optimize our main algorithm.

V. Submission Matching

Algorithm 2: SubmissionMatching

Input: A student submission s, a mapping $\bar{p}$ from a set of expected methods Q to a set of patterns $\mathcal{P}$, a mapping $\bar{t}$ from $Q \times \mathcal{P}$ to $\mathbb{N}$, a mapping $\bar{c}$ from Q to a set of constraints $\mathbb{C}$.
Output: A set of feedback comments B

```
1  // 1: Extract extended program dependence graphs.
2  ḡ := ∅
3  foreach h ∈ GetMethods(s) do
4      ḡ := ḡ ∪ {h = ExtractEPDG(s, h)}
5  // 2: Providing feedback.
6  B := ∅
7  foreach H ∈ Combinations(Q, dom ḡ) do
8      B' := ∅
9      foreach (q, h) ∈ H do
10         m̄ := ∅
11         // 2.1: Matching patterns.
12         foreach p ∈ p̄(q) do
13             M := PatternMatching(p, ḡ(h))
14             m̄ := m̄ ∪ {(p = M)}
15             B' := B' ∪ {ProvideFeedback(M, p, t̄(q, p))}
16         // 2.2: Matching constraints.
17         foreach c ∈ c̄(q) do
18             B' := B' ∪ {ConstraintMatching(c, ḡ(h), m̄)}
19     // 2.3: Store if there is some improvement.
20     if B = ∅ ∨ Λ(B') > Λ(B) then
21         B := B'
```

Algorithm 2 presents how we process a given student submission s, in which we are able to deal with submissions where multiple methods are expected, e.g., *main*, *factorial*, or *Fibonacci*. Our algorithm takes as input s, a mapping $\bar{p}$ indicating the set of patterns that apply to each expected method, a mapping $\bar{t}$ indicating the number of occurrences of a given pattern in a given expected method, and a mapping $\bar{c}$ indicating the set of constraints that apply to each expected

method. It outputs a set of feedback comments B over the student submission. To compute such feedback, we first use *ExtractEPDG* to calculate the extended program dependence graph related to each method in the submission (lines 2–4).

In the second step, we initialize the feedback comments to be delivered (line 6) and start the matching process. Note that we take a number of expected methods as input that we wish to match with the student submission that may contain several methods. Our goal is to try different combinations and returning the most promising one, i.e., the one with the greatest number of positive feedback. Our algorithm computes all possible combinations of expected and existing methods (line 7), e.g., we expect m and f methods and a submission provides r and g methods, $Combinations(\{m,f\},\{r,g\}) = \{\{(m,r),(f,g)\},\{(m,g),(f,r)\}\}$. The number of combinations can be limited if we enforce the students to use one or more method headers, which is a common practice in introductory programming assignments. However, we will not provide feedback to those submissions that do not adhere to the specification.

We will study the feedback provided with the current combination of methods, so we initialize the comments to the empty set (line 8). We iterate over each combination (q,h), in which q is the expected method and h is the existing method in the submission (line 9). The enforcement of constraints is performed by taking into account embeddings of patterns, so we store such embeddings for each pattern in $\bar{m}$ (line 10). For each pattern related to method q (line 12), we apply Algorithm *PatternMatching* to compute embeddings M that are added to $\bar{m}$ (lines 13 and 14).

ProvideFeedback in line 15 is responsible for providing the feedback. $|M| \neq \bar{t}(q,p)$ entails that we have found a different number of embeddings than expected, so we provide *NotExpected* feedback; otherwise, we check whether or not an embedding has pattern nodes marked as incorrect (see Section IV). If all of them are correct, then we output *Correct* and, if not, we output *Incorrect*. Note that we use the natural language templates described in Section III to provide personalized feedback to the student. This approach also gives us the opportunity to enforce "bad patterns", i.e., those patterns that students should avoid using $\bar{t}(q,p) = 0$. One typical example is updating the value of the index (i++) more than once in a sentinel-controlled loop.

We iterate over the set of constraints related to each expected method q (line 17), and match each of them using *ConstraintMatching* (line 18), which takes all the current embeddings for each pattern and the extended program dependence graph as input. The matching is implemented following the definitions in Section III. We will output *NotExpected* for a constraint if it refers to any pattern that was marked as *NotExpected*. Otherwise, we check the constraint and output *Correct* or *Incorrect* depending on whether or not it is fulfilled.

Finally, we update the feedback to be provided B if the current feedback B' improves the existing one (lines 20 and 21). We use a cost function Λ as follows:

TABLE I. EXPERIMENTAL RESULTS OF OUR TECHNIQUE.

Assignment	S	L	T	P	C	M	D
Assignment 1	640,000	12.23	0.18s	6	4	0.03s	24
esc-LAB-3-P1-V1	442,368	15.17	0.20s	7	5	0.04s	8
esc-LAB-3-P2-V1	7,077,888	16.75	0.20s	8	13	0.03s	592
esc-LAB-3-P2-V2	144	7.67	0.17s	4	5	0.01s	0
esc-LAB-3-P3-V1	10,368	10.5	0.10s	7	6	0.01s	1
esc-LAB-3-P3-V2	589,824	15.42	0.19s	8	10	0.03s	4
esc-LAB-3-P4-V1	13,824	10.5	0.17s	7	6	0.01s	1
esc-LAB-3-P4-V2	9,437,184	17.42	0.26s	9	14	0.03s	248
mitx-derivatives	576	5.75	0.12s	3	4	0.03s	0
mitx-polynomials	768	6.67	0.12s	4	4	0.01s	0
rit-all-g-medals	559,872	24.67	0.32s	9	7	0.13s	1,872
rit-medals-by-ath	746,496	33.5	0.35s	9	7	0.25s	744

$$\Lambda(B) = \sum_{b \in B} \lambda(b) = \begin{cases} 1 & \text{if } b = Correct; \\ 0.5 & \text{if } b = Incorrect; \\ 0 & otherwise. \end{cases} \quad (3)$$

As a result, we use the personalized feedback provided to guide our best-effort submission matching with multiple methods. We assume that if a combination H of methods is providing better feedback than other combinations, H is the original intent of the student.

VI. EVALUATION

We implemented our technique using Java 8 and ANTLR 4.5.1 [26] to build extended program dependence graphs from Java submissions. We also used JGraphT 0.9.2[1] to store and manipulate graphs, and Guava 18[2] for auxiliary functionality. Our implementation is publicly available in a GitLab repository[3]. In the rest of this section, we present our results (Section VI-A), a discussion of those results (Section VI-B), and a comparison with state-of-the-art techniques in providing personalized feedback (Section VI-C).

A. Experimental results

Our experiments were run on a single-threaded Intel Xeon 5160 3.00GHz CPU and 8 GB RAM, running on Red Hat Enterprise Linux Server 6.8. We focus on twelve real-world assignments in total summarized in Table I as follows:

- Assignment 1 presented in Section III.
- esc-LAB-3-P1-V1: Print to console number n such that $n! \leq k < (n+1)!$ taking number k as input.
- esc-LAB-3-P2-V1: Same as esc-LAB-3-P1-V1 but using the Fibonacci sequence.
- esc-LAB-3-P2-V2: A number is special when the sum of cubes of its digits is equal to the number itself.
- esc-LAB-3-P3-V1: Find the difference of a positive number and its reverse.
- esc-LAB-3-P3-V2: Given numbers n and m, print to console the count of factorial numbers in $[n,m]$.

[1] http://jgrapht.org/
[2] http://github.com/google/guava
[3] https://gitlab.com/crivero/icde2017

- esc-LAB-3-P4-V1: Check if a given number k is a palindrome.
- esc-LAB-3-P4-V2: Same as esc-LAB-3-P3-V2 but using the Fibonacci sequence.
- mitx-derivatives: Computer the derivatives of an input polynomial represented by an array.
- mitx-polynomials: Compute the value of a polynomial at a given value.
- rit-all-g-medals: Count all the gold medals awarded in a given year in the Summer Olympic Games.
- rit-medals-by-ath: Count all the medals awarded to a given athlete in the Summer Olympic Games.

Assignments esc-LAB-X and rit-X are from the undergraduate introductory programming courses at IIT Kanpur and RIT, respectively. Assignments mitx-X are from the introductory programming course at *edX.org* offered by MIT.

To generate the submissions for these assignments, we followed the same hypothesis as Singh et al. [33] and Pu et al. [28] that errors in student submissions in introductory programming courses are predictable: all students are focusing on the same assignment after having attended the same lectures, so errors tend to follow predictable patterns. Singh et al. [33] use rules to represent mistakes of students of the form $i = 0 \rightarrow i = 1$, which indicates that an array index should start in 0 but some students initialize it to 1. Such rules define a search space to be explored. We took one or more reference solutions and, using similar rules for all the assignments, we explicitly generated the search space of student submissions, which are reported in Table I under column *S*. Column *L* presents the average number of lines of code of each submission.

We generated a set of functional tests to be performed over the previous submissions. We report the average running time in column *T* in Table I. Our next goal was checking our technique over the submissions. Columns *P* and *C* report the number of patterns and constraints that we used in each of the assignments, respectively. The number of unique patterns is twenty four, which gives an idea about their reusability. Column *M* stands for the average running time of our pattern matching technique. Note that all of them are in the range of milliseconds and, therefore, it shows that our approach is suitable to be used in an online context like MOOCs. Finally, *D* stands for the number of discrepancies between functional testing and our approach, i.e., when functional testing reported a submission was correct or incorrect, and our technique reported negative or positive feedback, respectively. We discuss these below.

B. Discussion

Our technique is comparable to executing functional tests over student submissions. The main difference is that feedback provided by such tests is usually not enough for novices, while our approach is mainly focused on such feedback. In this section, we discuss discrepancies that we found between our output and the results of functional tests, and benefits and limitations of our technique.

Assignment 1. There are twenty-four discrepancies in Assignment 1. Seventeen submissions initialize the index to access arrays as $i = 1$ when accessing odd positions, however, our technique suggests $i = 0$. Four submissions print to console in a different order than expected by the functional tests, however, our technique is independent of the order and provides correct feedback. Finally, three submissions are not using $if(i \,\%\, 2 == 0)$ to access even positions and they update twice the value of i, which is a different way of accessing even positions not currently allowed by our patterns. We could add more patterns to account for the different options, but since this could end up leading to many patterns, we intend to deal with pattern variability as future work (see Section VII).

esc-LAB-3-P1-V1. Eight submissions compute the interval $(n-1)! \leq k < (n+1)!$ instead of $n! \leq k < (n+1)!$, which is what the assignment states. The computed lower limit, $(n-1)!$, will never be greater than $n!$, so the condition that $n! \leq k$ still holds. We detect (and our technique provides feedback accordingly) all the pieces of the submission are correct except when computing the lower limit, where we say it is incorrectly done. How to handle cases in which functionality is preserved but do not strictly adhere to the assignment's statement should be up to the instructor. Our technique is capable of providing feedback in instances where the instructor decides that solutions for a particular assignment should not differ from what is stated. However, we would have to rely on any other technique, like functional testing, to check for these functionally equivalent cases and provide feedback for them.

esc-LAB-3-P2-V1. We found 592 discrepancies for this assignment. Similarly to esc-LAB-3-P1-V1, $fib(n-1) \leq k < fib(i+1)$ is computed instead of $fib(n) \leq k < fib(n+1)$.

esc-LAB-3-P3-V1. One single submission computes the number of digits of the input number k as $\lfloor \log_{10} k \rfloor$, which is not currently allowed by our patterns because of the pattern variability, since we expect $\lfloor \log_{10} k \rfloor + 1$.

esc-LAB-3-P3-V2. There are four discrepancies in this assignment. Functional testing considers them to be incorrect because the result we are expecting is the count of the factorial numbers in the interval without repetition. These assignments iterate from $i = 0$ to $i = m$ and compute the factorial of i every iteration. For example, for $n = 1$ and $m = 15$, the expected output in the functional testing is 3 (1, 2, 6); however, 1 is counted twice (as factorial of 0 and factorial of 1), thus returning 4 instead of 3.

esc-LAB-3-P4-V1. There is one discrepancy in which the same issue explained in esc-LAB-3-P3-V1 occurs because the submission tries to compute the number of digits of a given input number.

esc-LAB-3-P4-V2. We found 248 discrepancies because of the way the interval $[n, m]$ is computed. We considered the Fibonacci sequence as $1, 1, 2, 3, \ldots$, so the initial number in the sequence is 1. Therefore, we were expecting the computations to start in $i = 1$, but such submissions were initializing $i = 0$. We provided personalized feedback suggesting to modify the starting point.

rit-all-g-medals We obtained $1,872$ discrepancies in this assignment, which implies accessing a text file storing records of athletes in the history of Summer Olympic Games. The

```
void countGoldMedals (int year) {
 int i = 1, medals = 0, p = 0, y = 0;
 String fn = "", ln = "", e = "";
 Scanner s = new Scanner(
   new File("summer_olympics.txt"));
 while (s.hasNext()) {
  if (i % 5 == 4)
   e = s.next();
  if (i % 5 == 1)
   e = s.next();
  if (i % 5 == 1)
   e = s.next();
  if (i % 5 == 3)
   y = s.nextInt();
  if (i % 5 == 3)
   p = s.nextInt();
  if (i % 5 == 4 && y == year &&
    p == 1)
   medals += 1;
  i++;
 }
 s.close();
 System.out.println(medals);
}
```

Fig. 7. Functionally correct but semantically incorrect sample submission in rit-all-g-medals.

first and second positions of each record store the first and last names of an athlete, respectively. The third position the type of a medal (1 for gold, 2 for silver, and 3 for bronze). The fourth position stores the specific year of the event, and the fifth position is a record separator line. In our synthetic generation, we used all possible combinations of $i \% 2 == \{0, 1, 2, 3, 4\}$ and storing variables with that information, e.g., $p = s.nextInt()$ for the type of medal, or $y = s.nextInt()$ for the year. In these combinations, some of the submissions are functionally correct but they do not make sense from the semantics point of view, e.g., Figure 7 shows a sample submission that is functionally correct but semantically incorrect. We analyzed such submissions and we found that they used multiple times the same condition, for instance, by using $i \% 2 == 0$ twice, a submission advances the file index twice, so it retrieves correctly the medal type and year. We were able to successfully detect all of these submissions and provide personalized feedback in all cases.

rit-medals-by-ath. There are 744 discrepancies in this assignment due to the same reasons explained in rit-all-g-medals. They used the same condition several times to advance the file index.

As a conclusion, we were able to provide semantically correct personalized feedback in all cases using a limited number of patterns and constraints. One of our current limitations is that we are not able to deal with functionally equivalent variations of performing a given computation, e.g., to access even positions in an array, we can use either a loop controlled by $i \% 2 == 0$, or updating twice the index i+=2. We plan to address this issue by not considering patterns in isolation, i.e., we will create a hierarchy of patterns according to their semantics in which the same pattern can be performed in several ways. Our algorithms will take such hierarchy into account accordingly.

```
void assignment1(
    int[] a) {
 int i = 0;
 int o = 0;
 while (i < a.length ){
   if (i % 2 == 1)
    o += a[i];
   i++;
 }
 int e = 1;
 i = 0;
 while (i < a.length ){
   if (i % 2 == 0)
    e *= a[i];
   i++;
 }
 System.out.print(o);
 System.out.print(e);
}
```

(a) Reference solution.

```
void assignment1(
    int[] a) {
 int o = 0;
 int i = 0;
 while (i < a.length ){
   if (i % 2 == 1)
    o += a[i];
   i++;
 }
 i = 0;
 int e = 1;
 while (i < a.length ){
   if (i % 2 == 0)
    e *= a[i];
   i++;
 }
 System.out.print(e);
 System.out.print(o);
}
```

(b) Correct submission.

Fig. 8. Sample reference solution and a very similar correct submission for Assignment 1.

C. Comparison with AutoGrader and CLARA

We consider AutoGrader [33] and CLARA [15] as the state-of-the-art techniques to providing personalized feedback in MOOCs. Since AutoGrader is no longer available [15], we used Sketch [34] to perform our comparison, which is used under the hood by AutoGrader and is publicly available.

Reference solutions. AutoGrader suggests repairs to ensure functional equivalence between a submission and a single reference solution relying on Sketch. CLARA compares variable traces between reference solutions and submissions, so it needs "meaningful" inputs to compare such traces. Furthermore, traces are compared as a whole, so it needs a reference solution per any possible variation of the assignment, e.g., the reference solution in Figure 8a does not match the submission in Figure 8b for Assignment 1; note that both programs are functionally similar but the order of the variables are different. Our technique is not based on reference solutions but static analysis of the submissions, and it exploits patterns and constraints to provide personalized feedback.

Printing to console. Sketch compares the returning values of a reference solution and a submission, therefore, it is not able to deal with printing to console statements. A modification to solve this issue is to change submissions to concatenate the values of the variables to be returned. However, Sketch directly compares inputs, so all values should be returned in the same order as the reference solution. CLARA considers the standard output as another variable in the variable traces, so it is also dependent on the printing order. In our case, our patterns check that variables are printed to console independently from their order. If any specific order is required, it can be enforced using constraints correlating several patterns.

Loops. Sketch is not able to deal with infinite loops. Also, it requires loop unrolling and, therefore, it uses approximations to deal with loops similar to automated software verification tools [37]. Since it is implemented in C, Sketch requires having fixed array lengths. CLARA outputs a timeout when dealing with multiple loops. It does not rely on loop unrolling since

the user needs to provide inputs to compare variable traces of reference solutions and submissions. Our technique is based on static analysis and does not have any of these drawbacks.

Multiple methods. Sketch inlines methods in the entry point method. When dealing with method calls in loops, this inlining has the same problems as dealing with general loops: unrolling them requires approximations that introduce errors, e.g., in esc-LAB-3-P1-V1, Sketch only works when providing constant input values. CLARA is not able to match the same (copy and paste) submissions with several methods since it implies different variables traces. The repair algorithm outputs feedback of the form "Change $current = factorial(i)$ to $current = factorial(i)$", which may confuse novice students. Some of the assignments used to evaluate CLARA are appealing to contain a couple of methods, however, how to deal with this issue is not addressed in the original paper [15]. In our technique, the instructor needs to specify several methods that she expects to find along with patterns and constraints.

Structural requirements. Sketch is not able to enforce that students are following some specific strategy to solve an assignment, e.g., only one single loop in our Assignment 1. CLARA can enforce structural requirements by using only reference solutions using certain strategies. In our case, instructors can enforce these strategies by including the appropriate combination of patterns and constraints.

Scalability. Sketch can provide up to four repairs beyond which its performance degrades significantly [15]. Furthermore, in order for the search space to be manageable, the user needs to set bounds for the arguments of a method because Sketch exhausts the whole domain to compare a reference solution and a submission, e.g., in our Assignment 1, we need to bound both the number of integer bits and the maximum length of the array. We found that CLARA is able to deal with small but not large inputs, e.g., in esc-LAB-3-P1-V1, it outputs a timeout error when $k = 100,000$, when running such functional test takes milliseconds. Our technique took milliseconds on average in all of our experiments and is independent of the input values.

Matching and repair. Sketch matches the output of the reference solution over a student submission. The matching process in CLARA is disconnected from the repair algorithm, e.g., two submissions with a *for* and a *while* loops match under the same cluster but, when we select the one with the *for* loop as the reference solution and use the other submission to compute repairs, CLARA suggests to change the *while* into a *for* loop, which is an incorrect behavior. In our technique, we leverage provided feedback to improve the matching process by means of a cost function.

VII. Conclusions

The current "boom" in computing courses is making challenging the task of explaining what and why students did correctly and/or incorrectly in their introductory programming assignments. The challenge scales in MOOCs since there may be hundreds of thousands of submissions for the same assignment. One approach to address the challenge consists of using human resources (graders and/or peer discussions), which entails several problems like lack of uniformity in the feedback, or waiting days for feedback that may be incomplete or incorrect. Automated approaches have mainly focused on functional testing, in which provided feedback does not sufficiently guide novice students; automated software verification tools, which have not been specifically designed to address this challenging task; or reference solutions, which require multiple solutions as the comparison is performed as a whole, have scalability problems, or require pre-existing correct submissions.

We propose a semantic-aware technique that aims to find code patterns that an instructor is expecting to find in student submissions. We rely on extended program dependence graphs to model submissions, and patterns with feedback attached to them are encoded as subgraph patterns. Personalized feedback is provided by leveraging subgraph matching techniques from the graph database field. Constraints correlating multiple patterns are used to provide fine-grained assessments of submissions. We have evaluated our technique over several real-world assignments from different courses, including MOOCs. Our technique is scalable since it deals with every submission in milliseconds on average. Different from AutoGrader [33] and CLARA [15], current state-of-the-art techniques, our approach is based on understanding the semantics of the submissions and the original intentions of students when dealing with an assignment. Patterns are reusable among several assignments.

As future work, patterns will be clustered by variations to achieve the same semantics, e.g., a student can access even positions in an array using $if(i \% 2 == 0)$ or updating twice the value of *i*. Our patterns will support *else* expressions, e.g., a pattern to ensure accessing odd positions in a submission using $if(i \% 2 == 0) \{\dots\}$ *else* $\{\dots\}$ will only work by computing the functional equivalence, i.e., transforming *else* into $if(i \% 2 == 1)$. We are planning to use more sophisticated methods to match Java expressions rather than regular expressions like abstract syntax trees. We will use a program model similar to CLARA [15] to deal with additional imperative programming languages like C or Python. We will also deal with multiple, non-expected methods by the instructor by combining function inlining and approximate subgraph matching. Finally, we will predefine certain combinations of patterns and constraints to ensure specific algorithmic strategies to solve assignments.

References

[1] G. K. Baah, A. Podgurski, and M. J. Harrold. The probabilistic program dependence graph and its application to fault diagnosis. *IEEE Trans. Software Eng.*, 36(4):528–545, 2010.

[2] A. Bhattacharjee and H. M. Jamil. CodeBlast: A two-stage algorithm for improved program similarity matching in large software repositories. In *SAC*, pages 846–852, 2013.

[3] F. Bi, L. Chang, X. Lin, L. Qin, and W. Zhang. Efficient subgraph matching by postponing cartesian products. In *SIGMOD*, pages 1199–1214, 2016.

[4] T. Camp, S. H. Zweben, D. Buell, and J. Stout. Booming enrollments: Survey data. In *SIGCSE*, pages 398–399, 2016.

[5] B. Cheang, A. Kurnia, A. Lim, and W. Oon. On automated grading of programming assignments in an academic institution. *Computers & Education*, 41(2):121–131, 2003.

[6] A. Drummond, Y. Lu, S. Chaudhuri, C. M. Jermaine, J. Warren, and S. Rixner. Learning to grade student programs in a massive open online course. In *ICDM*, pages 785–790, 2014.

[7] S. H. Edwards. Using software testing to move students from trial-and-error to reflection-in-action. In *SIGCSE*, pages 26–30, 2004.

[8] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[9] S. Fitzgerald, B. Hanks, R. Lister, R. McCauley, and L. Murphy. What are we thinking when we grade programs? In *SIGCSE*, pages 471–476, 2013.

[10] D. D. Garcia, J. Campbell, J. DeNero, M. L. Dorf, and S. Reges. CS10K teachers by 2017?: Try CS1K+ students NOW! Coping with the largest CS1 courses in history. In *SIGCSE*, pages 396–397, 2016.

[11] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[12] 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, 2015.

[13] K. Gouda and M. Hassaan. CSI_GED: An efficient approach for graph edit similarity computation. In *ICDE*, pages 265–276, 2016.

[14] S. Gulwani, I. Radicek, and F. Zuleger. Feedback generation for performance problems in introductory programming assignments. In *SIGSOFT*, pages 41–51, 2014.

[15] S. Gulwani, I. Radicek, and F. Zuleger. Automated clustering and program repair for introductory programming assignments. *CoRR*, abs/1603.03165, 2016.

[16] M. Gupta, J. Gao, X. Yan, H. Cam, and J. Han. Top-k interesting subgraph discovery in information networks. In *ICDE*, pages 820–831, 2014.

[17] W. Han, J. Lee, and J. Lee. $\text{Turbo}_{\text{iso}}$: Towards ultrafast and robust subgraph isomorphism search in large graph databases. In *SIGMOD*, pages 337–348, 2013.

[18] D. Jackson. A software system for grading student computer programs. *Computers & Education*, 27(3-4):171–180, 1996.

[19] T. Kahsai, P. Rümmer, H. Sanchez, and M. Schäf. JayHorn: A framework for verifying Java programs. In *CAV*, pages 352–358, 2016.

[20] S. Kaleeswaran, A. Santhiar, A. Kanade, and S. Gulwani. Semi-supervised verified feedback generation. *CoRR*, abs/1603.04584, 2016.

[21] H. Keuning, J. Jeuring, and B. Heeren. Towards a systematic review of automated feedback generation for programming exercises. In *ITiCSE*, pages 41–46, 2016.

[22] P. A. Kirschner, J. Sweller, and R. E. Clark. Why minimal guidance during instruction does not work: An analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational Psychologist*, 41(2):75–86, 2006.

[23] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[24] J. Lee, W. Han, R. Kasperovics, and J. Lee. An in-depth comparison of subgraph isomorphism algorithms in graph databases. *PVLDB*, 6(2):133–144, 2012.

[25] C. Liu, C. Chen, J. Han, and P. S. Yu. GPLAG: Detection of software plagiarism by program dependence graph analysis. In *SIGKDD*, pages 872–881, 2006.

[26] T. Parr. *The Definitive ANTLR 4 Reference*. Pragmatic Bookshelf, 2nd edition, 2013.

[27] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. J. Guibas. Learning program embeddings to propagate feedback on student code. In *ICML*, pages 1093–1102, 2015.

[28] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay. sk_p: A neural program corrector for MOOCs. *CoRR*, abs/1607.02902, 2016.

[29] C. R. Rivero and H. M. Jamil. On isomorphic matching of large disk-resident graphs using an XQuery engine. In *ICDE Workshops*, pages 20–27, 2014.

[30] C. R. Rivero and H. M. Jamil. Efficient and scalable labeled subgraph matching using SGMatch. *Knowl. Inf. Syst.*, (available online first) 2016.

[31] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, 2009.

[32] N. B. Shah, J. Bradley, S. Balakrishnan, A. Parekh, K. Ramchandran, and M. J. Wainwright. Some scaling laws for MOOC assessments. In *SIGKDD Workshops*, 2014.

[33] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, pages 15–26, 2013.

[34] A. Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

[35] Y. Tian and J. M. Patel. TALE: A tool for approximate large graph matching. In *ICDE*, pages 963–972, 2008.

[36] N. Truong, P. Roe, and P. Bancroft. Static analysis of students' Java programs. In *ACE*, pages 317–325, 2004.

[37] M. Vujosevic-Janicic, M. Nikolic, D. Tosic, and V. Kuncak. Software verification and graph similarity for automated evaluation of students' assignments. *Information & Software Technology*, 55(6):1004–1016, 2013.

[38] R. P. Wiegand, A. Bucci, A. N. Kumar, J. L. Albert, and A. Gaspar. A data-driven analysis of informatively hard concepts in introductory programming. In *SIGCSE*, pages 370–375, 2016.

[39] J. M. Wing. Computational thinking. *Commun. ACM*, 49(3):33–35, 2006.

[40] S. Xu and Y. S. Chee. Transformation-based diagnosis of student programs for programming tutoring systems. *IEEE Trans. Software Eng.*, 29(4):360–384, 2003.

[41] S. Zweben and B. Bizot. 2015 Taulbee Survey. Technical report, Computing Research Association, 2016.