Automated Clustering and Program Repair for Introductory Programming Assignments*

Sumit Gulwani
Microsoft Corporation
USA
sumitg@microsoft.com

Ivan Radiček
TU Wien
Austria
radicek@forsyte.at

Florian Zuleger
TU Wien
Austria
zuleger@forsyte.at

Abstract

Providing feedback on programming assignments is a tedious task for the instructor, and even impossible in large Massive Open Online Courses with thousands of students. Previous research has suggested that program repair techniques can be used to generate feedback in programming education. In this paper, we present a novel fully automated program repair algorithm for introductory programming assignments. The key idea of the technique, which enables automation and scalability, is to use the existing correct student solutions to repair the incorrect attempts. We evaluate the approach in two experiments: (I) We evaluate the number, size and quality of the generated repairs on 4,293 incorrect student attempts from an existing MOOC. We find that our approach can repair 97% of student attempts, while 81% of those are small repairs of good quality. (II) We conduct a preliminary user study on performance and repair usefulness in an interactive teaching setting. We obtain promising initial results (the average usefulness grade 3.4 on a scale from 1 to 5), and conclude that our approach can be used in an interactive setting.

CCS Concepts • **Applied computing → Computer-assisted instruction**; • **Software and its engineering → Software testing and debugging**;

Keywords programming education, MOOC, dynamic analysis, program repair, clustering

ACM Reference Format:
Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated Clustering and Program Repair for Introductory Programming Assignments. In *Proceedings of 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'18).* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3192366.3192387

*Supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF).

PLDI'18, June 18–22, 2018, Philadelphia, PA, USA

ACM ISBN 978-1-4503-5698-5/18/06.
https://doi.org/10.1145/3192366.3192387

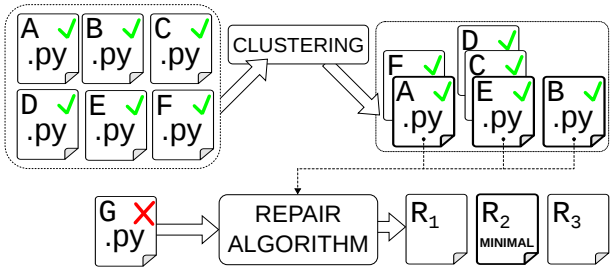


Figure 1. High-level overview of our approach.

1 Introduction

Providing feedback on programming assignments is an integral part of a class on introductory programming and requires substantial effort by the teaching personnel. This problem has become even more pressing with the increasing demand for programming education, which universities are unable to meet (it is predicted that in the US by 2020 there will be one million more programming jobs than students [1]). This has given rise to several Massive Open Online Courses (MOOCs) that teach introductory programming [28]; the biggest challenge in such a setting is scaling personalized feedback to a large number of students.

The most common approach to feedback generation is to present the student with a failing test case; either generated automatically using test input generation tools [40] or selected from a comprehensive collection of representative test inputs provided by the instructor. This is useful feedback, especially since it mimics the setting of how programmers debug their code. However, this is not sufficient, especially for students in an introductory programming class, who are looking for more guided feedback to make progress towards a correct solution.

A more guided feedback can be generated from modifications that make a student's program correct, using a program repair technique as pioneered by the AutoGrader tool [33]. Generating feedback from program repair is an active area of research: One line of work focuses on improving the *technical capabilites of program repair* in introductory education [9, 31, 32], while another line of research focuses

on *pedagogical questions* such as how to best provide repair-based feedback to the students [20, 37]. In this paper we propose a new completely automated approach for repairing introductory programming assignments, while sidelining the pedagogical questions for future work.

Our approach The key idea of our approach is to use the *wisdom of the crowd*: we use the *existing correct* student solutions to repair the *new incorrect* student attempts[1]. We exploit the fact that MOOC courses already have tens of thousands of existing student attempts; this was already noticed by Drummond et al. [13].

Fig. 1 gives a high-level overview of our approach: (A) For a given programming assignment, we automatically *cluster the correct student solutions* (**A**-**F** in the figure), based on a notion of *dynamic equivalence* (see §2.1 for an overview and §4 for details). (B) Given an *incorrect student attempt* (**G** in the figure) we run the *repair algorithm* against all clusters, and then select a *minimal repair* ($\mathbf{R_2}$ in the figure) from the generated repair candidates ($\mathbf{R_1}$-$\mathbf{R_3}$ in the figure). The repair algorithm uses expressions from *multiple correct solutions* to generate a repair (see §2.2 for an overview and §5 for details).

Intuitively, our clustering algorithm groups together similar correct solutions. Our repair algorithm can be seen as a generalization of the clustering approach of correct solutions to incorrect attempts. The key motivation behind this approach is as follows: to help the student, with an incorrect attempt, our approach finds the set of most similar correct solutions, written by other students, and generates the smallest modifications that get the student to a correct solution.

We have implemented the proposed approach in a tool called Clara and evaluated it in two experiments:

(I) On 12,973 correct and 4,293 incorrect (total 17,266) student attempts from an *MITx* MOOC, written in Python, we evaluate the *number*, *size* and *quality* of the generated repairs. Clara is able to repair 97% of student attempts, in 3.2s on average; we study the quality of the generated repairs by manual inspection and find that *81%* of the generated repairs are *of good-quality* and the size of the generated repair *matches the size of the required changes* to the student's program. Additionally, we compare AutoGrader and Clara, on the same MOOC data.

(II) We performed a preliminary user study about the *performance* and *usefulness* of Clara's repairs in an interactive teaching setting. The study consisted of 52 participants who were asked to solve 6 programming assignments in C. The participants judged the usefulness of the generated repair-based feedback by 3.4 in average on a scale from 1 to 5.

Our experimental results demonstrate that Clara can, completely automatically, generate repairs of high quality, for a large number of incorrect student attempts in an interactive teaching setting for introductory programming problems.

This paper makes the following contributions:

- We propose an algorithm to automatically cluster correct student solutions based on a dynamic program analysis.
- We propose a completely automated algorithm for program repair of incorrect student attempts that leverages the *wisdom of the crowd* in a novel way.
- We evaluate our approach on a large MOOC dataset and show that Clara can repair almost all the programs while generating repairs of high quality.
- We find in a real-time user study that Clara is sufficiently fast to be used in an interactive teaching setting and obtain promising preliminary results from the participants of the user study on the usefulness of the generated feedback.

Differences to our earlier Technical Report. Our approach first appeared as a technical report[2], together with a publicly available implementation[3]. While our core ideas are the same as in the technical report, we since have made several improvements of which we state here the two most important: (1) The repair algorithm uses expressions from different correct solutions in a cluster, as opposed to using only a single correct solution from a cluster. (2) We conducted a user study and added an extensive manual investigation of the generated repairs to the experimental evaluation.

Structure of the paper. In §2 we present an overview of our approach, in §3 we describe a simple imperative language, used for formalizing the notions of matching and clustering in §4 and the repair procedure in §5. We discuss our implementation and the experimental evaluation in §6, overview the related work in §7, discuss limitations and directions for future work in §8, and conclude in §9.

2 Overview

We discuss the high-level ideas of our approach on the student attempts to the assignment `derivatives`: "*Compute and return the derivative of a polynomial function (represented as a list of floating point coefficients). If the derivative is* `0`*, return* `[0.0]`." Fig. 2 (a), (b), (e) and (f) show four student attempts to the above programming assignment: **C1** and **C2** are functionally correct, while **I1** and **I2** are incorrect.

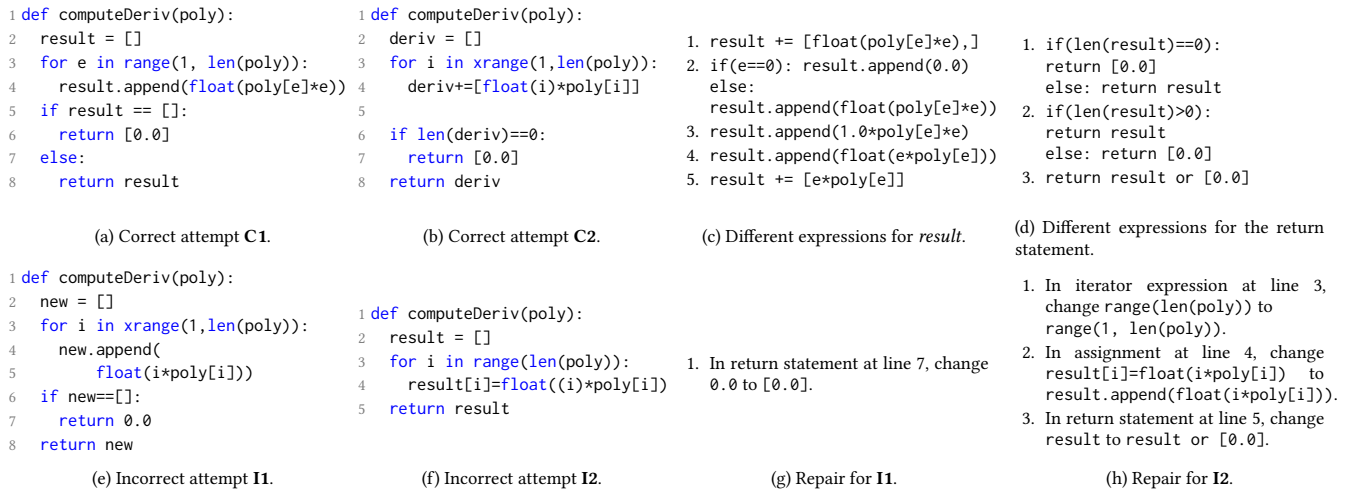
2.1 Clustering of correct student solutions

The goal of clustering is two-fold. (1) *Scalability*: elimination of *dynamically equivalent* correct solutions that the repair algorithm would otherwise consider separately. (2) *Diversity of repairs*: mining of *dynamically equivalent*, but *syntactically different* expressions from the same cluster, which are used

[1] We distinguish correct and incorrect attempts by running them on a set of inputs, and comparing their output to the expected output. This is the standard way of assessing correctness of student attempts in most introductory programming classes.

[2] https://arxiv.org/abs/1603.03165v1

[3] https://github.com/iradicek/clara



```
def computeDeriv(poly):
  result = []
  for e in range(1, len(poly)):
    result.append(float(poly[e]*e))
  if result == []:
    return [0.0]
  else:
    return result
```

(a) Correct attempt **C1**.

```
def computeDeriv(poly):
  deriv = []
  for i in xrange(1,len(poly)):
    deriv+=[float(i)*poly[i]]

  if len(deriv)==0:
    return [0.0]
  return deriv
```

(b) Correct attempt **C2**.

1. `result += [float(poly[e]*e),]`
2. `if(e==0): result.append(0.0)`
   `else:`
   `result.append(float(poly[e]*e))`
3. `result.append(1.0*poly[e]*e)`
4. `result.append(float(e*poly[e]))`
5. `result += [e*poly[e]]`

(c) Different expressions for *result*.

1. `if(len(result)==0):`
   `return [0.0]`
   `else: return result`
2. `if(len(result)>0):`
   `return result`
   `else: return [0.0]`
3. `return result or [0.0]`

(d) Different expressions for the return statement.

```
def computeDeriv(poly):
  new = []
  for i in xrange(1,len(poly)):
    new.append(
        float(i*poly[i]))
  if new==[]:
    return 0.0
  return new
```

(e) Incorrect attempt **I1**.

```
def computeDeriv(poly):
  result = []
  for i in range(len(poly)):
    result[i]=float((i)*poly[i])
  return result
```

(f) Incorrect attempt **I2**.

1. In return statement at line 7, change `0.0` to `[0.0]`.

(g) Repair for **I1**.

1. In iterator expression at line 3, change `range(len(poly))` to `range(1, len(poly))`.
2. In assignment at line 4, change `result[i]=float(i*poly[i])` to `result.append(float(i*poly[i]))`.
3. In return statement at line 5, change `result` to `result or [0.0]`.

(h) Repair for **I2**.

Figure 2. Motivation examples of real student attempts on the programming assignment `derivatives`.

to repair incorrect student attempts. We discuss the notion of *dynamic equivalence* next.

Matching The clusters in our approach are the *equivalence classes* of a *matching* relation. We say that two programs P and Q match, written $P \sim Q$, when: (1) they have the same control-flow (see the discussion below), and (2) there is a total bijective relation between the variables of P and Q, such that related variables take the same values, in the same order, during the program execution on the same inputs. This is inspired by the notion of a *simulation relation* [29], adapted for a dynamic program analysis: whereas a simulation relation establishes that a program P produces exactly the same values as program Q at corresponding program locations *for all inputs*, we are interested only in a *fixed finite set of inputs*. Therefore, we also call this notion *dynamic* equivalence, to stress that we use dynamic program analysis.

Control-flow Our algorithms consider two control-flows the same if they have the same *looping structure*. That is, any loop-free sequence of code is treated as a single block; in particular, blocks can include (nested) if-then-else statements without loops (similar to Beyer et al. [7]). We point out that we could have picked a different *granularity* of control-flow; e.g., to treat only straight line of code (without branching) as a block. We have picked this granularity because it enables matching of programs that have different branching-structure, and as a result our algorithm is able to generate repairs that involve missing if-then-else statements.

Example. Programs **C1** and **C2**, from Fig. 2 (a) and (b), *match*, because: (1) they have the same control-flow since there is only a single loop in both programs; (2) there is the bijective variable relation $\tau = \{poly \mapsto poly, deriv \mapsto result, ? \mapsto ?, i \mapsto e, return \mapsto return\}$, where variables ? and *return* are *special variables* denoting the loop condition and the return value, which we need to make the control-flow and the return values explicit. For example, on the input $poly = [6.3, 7.6, 12.14]$, the variable *result*, takes the value [] before the loop, the sequence of values $[7.6], [7.6, 24.28]$ inside the loop, and the value $[7.6, 24.28]$ after the loop. Exactly the same values are taken by the variable *deriv*; and similarly for all the other variables. Therefore, **C1** and **C2** belong to the same cluster, which we denote by C. For the further discussion, we need to fix one correct solution as a *cluster representative*; we pick **C1**, although it is irrelevant which program from the cluster we pick.

Although the expressions in the assignments to variables *result* and *deriv* generate the same values (i.e., they match or they are dynamically equivalent), the assignment expressions inside the loops are syntactically quite different; we state the expressions in terms of the variables of the cluster representative **C1**:

- `append(result, float(poly[e]*e))` in **C1**, and
- `result + [float(e)*poly[e]]` in **C2**, where

the second expression has been obtained by replacing the variables from **C2** with variables from **C1** using the variable relation τ (e.g., we have replaced *deriv* with *result*, because $\tau(deriv) = result$). In our benchmark we found 15 syntactically different, but dynamically equivalent, ways to write expressions for the assignment to *result*, and 6 different ways to write the return expression (observing only different ASTs, and ignoring formatting differences). Some of these examples are shown in Fig. 2 (c) and (d), respectively. As we discuss in the next section, these different expressions are used by our repair algorithm.

2.2 Repair of incorrect student attempts

Our repair algorithm takes an incorrect student attempt (which we call an *implementation*) and a *cluster* (of correct

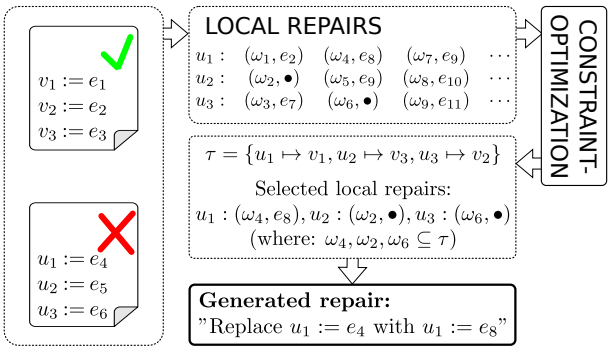


Figure 3. High-level overview of the repair algorithm.

programs), and returns a *repair*; a repair modifies the implementation such that the repaired implementation and the cluster representative *match*. The top-level repair procedure takes an implementation and runs the repair algorithm on each cluster separately. Each repair has a certain cost w.r.t. some cost metric. In this paper we use syntactic distance. Finally, the repair with the minimal cost is chosen.

Fig. 2 (e) and (f) shows two incorrect programs, **I1** and **I2**, and the generated repairs in (g) and (h), respectively. These repairs were generated using cluster C, with its representative **C1**. Our algorithm also considered other cluster besides C (which are not discussed here), but found the smallest repair using C. In the rest of this section we discuss the repair algorithm on a single cluster.

Our algorithm generates repairs in two steps, which we discuss in more detail next: (1) The algorithm generates a set of *local repairs* for each implementation variable (its assigned expression). (2) Using constraint-optimization techniques, the algorithm selects a *consistent* subset of the local repairs with the smallest cost. The high-level overview of the algorithm is given in Fig. 3.

Local Repairs A local repair ensures that an implementation expression e_{impl}, either modified or unmodified, *matches* a corresponding cluster representative expression e_C. To establish that the expressions match, we need a variable relation that translates implementation variables to cluster variables, so that the expressions range over the same variables. We point out that for *expression matching* it is sufficient to consider *partial variable relations* that relate variables of the expressions, as opposed to *total variable relations* that relate all program variables, which we considered above for *program matching*.

Let v be an implementation variable, with expression e_{impl} assigned to it at some program location. We say that $(\omega, e_{repaired})$ is a local repair for v, if the expressions e_C and $e_{repaired}$ match, where ω is a partial variable relation that establishes the matching. In this case the implementation expression e_{impl} is modified to the *repaired expression* $e_{repaired}$. We discuss below how the algorithm generates the repaired expression $e_{repaired}$. We say that $(\omega, \bullet)$ is a local repair for v, if the expressions e_C and e_{impl} match, where ω is a partial variable relation that establishes the matching. In this case the implementation expression e_{impl} remains unmodified.

We illustrate the notion of local repairs on **I1** with regard to the cluster representative **C1**:

1. $(\omega_1, \bullet)$ is a local repair for *new* before the loop, where $\omega_1 = \{new \mapsto result\}$, since the expressions of *new* and *result* match (i.e., they take the same values).
2. (ω_2, `if new==[]: return [0.0] else: return new`) is a local repair for *return* after the loop, where $\omega_2 = \{new \mapsto result, return \mapsto return\}$, since then the return expressions match.
3. (ω_3, `if poly==[]: return [0.0] else: return poly`) is a local repair for *return* after the loop, where $\omega_3 = \{poly \mapsto result, return \mapsto return\}$, since then the return expressions match.

The algorithm generates a set of such local repairs for each variable and each program location in the implementation.

Finding a repair A (whole program) repair is a *consistent subset* of the generated local repairs, such that there is *exactly one local repair for each variable and each program location in the implementation.* A set of local repairs is consistent when all partial variable relations in the local repairs are subsets of some total variable relation. For example, local repairs (1) and (3) from above are *inconsistent*, since we have $\omega_1(new) = result$ and $\omega_3(poly) = result$, and hence there is no total variable relation that is consistent with both ω_1 and ω_3. On the other hand, local repairs (1) and (2) are consistent, since there is a total variable relation consistent both with ω_1 and ω_2: $\{poly \mapsto poly, new \mapsto result, i \mapsto e, ? \mapsto ?, return \mapsto return\}$.

There are many choices for a whole program repair, that is, choices for a consistent subset of the generated local repairs; however, we are interested in one that has the smallest cost. Our algorithm finds such a repair using constraint-optimization techniques.

Generating the set of potential local repairs The repair algorithm takes expressions from the different correct solutions in order to generate the set of potential local repairs ($e_{repaired}$ discussed above). The algorithm translates these expressions to range over the implementation variables, using a partial variable relation. Since each cluster expression matches a corresponding cluster representative expression (recall the discussion in the previous sub-section), this partial variable relation ensures that $e_{repaired}$ matches a corresponding cluster representative expression (i.e., that it is a correct repair).

For example, the generated repair for **I2**, shown in Fig. 2 (h), combines the following expressions from the cluster:

- The first modification is generated using the expression `range(1, len(poly))` from **C1**, at line 3, using the partial variable relation $\{poly \mapsto poly\}$.
- The second modification is generated using the expression (4.) from Fig. 2 (c), using the partial variable relation $\{result \mapsto result, poly \mapsto poly, i \mapsto e\}$.
- The third modification is generated using the expression (3.) from Fig. 2 (d), using the partial variable relation $\{return \mapsto return, result \mapsto result\}$.

3 Program Model

In this section we define a program model that captures key aspects of imperative languages (e.g., C, Python). This model allows us to formalize our notions of program matching and program repair.

Definition 3.1 (Expressions)**.** Let V be a set of variables, K a set of constants, and O a set of operations. The set of expressions E is built from variables, constants and operations in the usual way. We fix a set of *special variables* $V^\sharp \subseteq V$. We assume that $V^\sharp$ includes at least the variable ?, which we will use to model conditions, and the variable *return*, which we will use to model return values.

Def. 3.1 can be instantiated by a concrete programming language. For example, for the C language, K can be chosen to be the set of all C constants (e.g., integer, float), and O can be chosen to be the set of unary and binary C operations as well as library functions. The special variables are assumed to not appear in the original program text, and are only used for modelling purposes.

Definition 3.2 (Program)**.** A *program* $P = (L_P, \ell_{init}, V_P, \mathcal{U}_P, \mathcal{S}_P)$ is a tuple, where L_P is a (finite) set of *locations*, $\ell_{init} \in L_P$ is the *initial location*, V_P is a (finite) set of *program variables*, $\mathcal{U}_P : (L_P \times V_P) \to E$ is the *variable update function* that assigns an expression to every location-variable pair, and $\mathcal{S}_P : (L_P \times \{\texttt{true}, \texttt{false}\}) \to (L_P \cup \{end\})$ is the *successor function*, which either returns a successor location in L_P or the special value *end* (we assume $end \notin L_P$). We drop the subscript P when it is clear from the context.

We point the reader to the discussion around the semantics below for an intuitive explanation of the program model.

Definition 3.3 (Computation domain, Memory)**.** We assume some (possibly infinite) set D of values, which we call the *computation domain*, containing at least the following values: (1) `true` (*bool true*); (2) `false` (*bool false*); and (3) $\perp$ (*undefined*).

Let V be a set of variables. A *memory* $\sigma : (V \cup V') \to D$ is a mapping from program variables to values, where the set $V' = \{v' \mid v \in V\}$ denotes the primed version of the variables in V; let Σ_V denote the set of all memories over variables $V \cup V'$.

Intuitively, the primed variables are used to denote the variable values after a statement has been executed (see the discussion around the semantics below).

Definition 3.4 (Evaluation)**.** A function $[\![\cdot]\!] : E \to \Sigma \to D$ is an *expression evaluation function*, where $[\![e]\!](\sigma) = d$ denotes that e, on a memory σ, evaluates to a value d.

The function $[\![\cdot]\!]$ is defined by a concrete programming language. The function returns the undefined value ($\perp$) when an error occurs during the execution of an actual program.

Definition 3.5 (Program Semantics)**.** Let $P = (L_P, \ell_{init}, V_P, \mathcal{U}_P, \mathcal{S}_P)$ be a program. A sequence of location-memory pairs $\gamma \in (L_P \times \Sigma_{V_P})^*$ is called a *trace*. Given some (*input*) memory ρ, we write $[\![P]\!](\rho) = (\ell_1, \sigma_1) \cdots (\ell_n, \sigma_n)$ if: (1) $\ell_1 = \ell_{init}$; (2) $\sigma_1(v) = \rho(v)$ for all $v \in V_P$; (3) (a) $\sigma_j(v') = [\![\mathcal{U}_P(\ell_j, v)]\!](\sigma_j)$, and (b) $\sigma_{j+1}(v) = \sigma_j(v')$, and (c) $\ell_{j+1} = \mathcal{S}_P(\ell_j, \sigma_j(?'))$, for all $v \in V_P$ and $0 \le j < n$; and (4) $\mathcal{S}_P(\ell_n, b) = end$, for any $b \in \{\texttt{true}, \texttt{false}\}$.

For some trace element $(\ell, \sigma) \in \gamma$ and a variable v, $\sigma(v)$ denotes the value of v before the location ℓ is evaluated (the *old value* of v at ℓ), and $\sigma(v')$ denotes the value of v after the location ℓ is evaluated (the *new value* of v at ℓ). The definition of $[\![P]\!](\rho)$ then says:

(1) The first location of the trace is the initial location ℓ_{init}.
(2) The *old values* of the variables at the initial location ℓ_{init} are defined by the input memory ρ.
(3a) The *new value* of variable v at location ℓ_j is determined by the semantic function $[\![\cdot]\!]$ evaluated on the expression $\mathcal{U}_P(\ell_j, v)$.
(3b) The old value of variable v at location ℓ_{j+1} is equal to the new value at location ℓ_j.
(3c) The next location ℓ_{j+1} in a trace is determined by the successor function $\mathcal{S}_P$ for the current location ℓ_j and the new value of ? at ℓ_j (i.e., $\sigma_j(?')$).
(4) The successor of the last location, ℓ_n, for any Boolean $b \in \{\texttt{true}, \texttt{false}\}$, is the end location *end*.

Modelling of if-then-else statements In our implementation we model if-then-else statements differently, according to when they contain loops and when they are loop-free (as mentioned in §2.1). In the former case, the branching is modelled, as usual, directly in the control-flow. In the latter case, (loop-free) statements are (recursively) converted to `ite` expressions that behave like a C ternary operator (as in the example that follows).

Example. We now show how a concrete program (**C1** from Fig. 2 (a)) is represented in our model. The set of locations is $L = \{\ell_{before}, \ell_{cond}, \ell_{loop}, \ell_{after}\}$, where $\ell_{init} = \ell_{before}$ is the location before the loop, and the initial location, ℓ_{cond} is the location of the loop condition, ℓ_{loop} is the loop body, and ℓ_{after} is the location after the loop. The successor function is given by $\mathcal{S}(\ell_{before}, \texttt{true}) = \mathcal{S}(\ell_{before}, \texttt{false}) = \ell_{cond}$, $\mathcal{S}(\ell_{cond}, \texttt{true}) = \ell_{loop}$, $\mathcal{S}(\ell_{cond}, \texttt{false}) = \ell_{after}$, $\mathcal{S}(\ell_{loop}, \texttt{true}) = \mathcal{S}(\ell_{loop}, \texttt{false})$

$= \ell_{cond}$, and $\mathcal{S}(\ell_{after}, \texttt{true}) = \mathcal{S}(\ell_{after}, \texttt{true}) = end$. Note that for non-branching locations the successor functions points to the same location for both `true` and `false`.

The set of variables is $V = \{poly, result, e, return, ?, it\}$, where *it* is used to model Python's *for-loop iterator*. An iterator is a sequence whose elements are assigned, one by one, to some variable (*e* in this example) in each loop iteration. The expression labeling function is given by:

- $\mathcal{U}(\ell_{before}, result)$ = `[]`,
- $\mathcal{U}(\ell_{before}, it)$ = `range(1,len(poly))`,
- $\mathcal{U}(\ell_{cond}, ?)$ = `len(it)>0`,
- $\mathcal{U}(\ell_{loop}, e)$ = `ListHead(it)`,
- $\mathcal{U}(\ell_{loop}, it)$ = `ListTail(it)`,
- $\mathcal{U}(\ell_{loop}, result)$ = `append(float(poly[e]*e))`,
- $\mathcal{U}(\ell_{after}, return)$ = `ite(result==[], [0.0], result)`.

For any variable v that is unassigned at some location ℓ we set $\mathcal{U}(\ell, v) = v$, i.e., the variable remains unchanged.

Finally, we state the trace when **C1** is executed on $\rho = \{poly \mapsto [6.3, 7.6, 12.14]\}$. We state only defined variables that change from one trace element to the next. Otherwise, we assume the values remain the same or are undefined ($\perp$) (if no previous value existed). $[\![\mathbf{C1}]\!](\rho) = (\ell_{before}, \{poly \mapsto [6.3, 7.6, 12.14], result' = [], i' = 0, it' = [1, 2]\}), (\ell_{cond}, \{?' = \texttt{true}\}), (\ell_{loop}, \{e' \mapsto 1, it' = [2], i' \mapsto 1, result' \mapsto [7.6]\}), (\ell_{cond}, \{?' = \texttt{true}\}), (\ell_{loop}, \{e' \mapsto 2, it' = [], i \mapsto 3, result' \mapsto [7.6, 24.28]\}), (\ell_{cond}, \{?' = \texttt{false}\}), (\ell_{after}, \{return \mapsto [7.6, 24.28]\})$.

4 Matching and Clustering

In this section we formally define our notion of *matching*.

Informally, two programs match, if (1) the programs have the *same control-flow* (i.e., the same looping structure), and (2) the corresponding variables in the programs take the same values in the same order. For the following discussion we fix two programs, $P = (L_P, \ell_{init_P}, V_P, \mathcal{U}_P, \mathcal{S}_P)$ and $Q = (L_Q, \ell_{init_Q}, V_Q, \mathcal{U}_Q, \mathcal{S}_Q)$.

Definition 4.1 (Program Structure)**.** Programs P and Q have the *same control-flow* if there exists a bijective function, called *structural matching*, $\pi : L_Q \to L_P$, s.t., for all $\ell \in L_Q$ and $b \in \{\texttt{true}, \texttt{false}\}$, $\mathcal{S}_P(\pi(\ell), b) = \pi(\mathcal{S}_Q(\ell, b))$.

We remind the reader, as discussed in §2 and §3, that we encode any loop-free program part as single control-flow location; as a result we compare only the *looping structure* of two programs.

We note that both our matching and repair algorithms require the existence of a *structural matching* π between programs. Therefore, in the rest of the paper we assume that such a π exists between any two programs that we discuss, and assume that $L = L_P = L_Q$ and $\mathcal{S} = \mathcal{S}_P = \mathcal{S}_Q$, since they can be always converted back and forth using π. Next, we state two definitions that will be useful later on.

```
1  fun Match(Programs P, Q, Inputs I):
2      π = structural matching or abort
3      γ_{P,ρ} = [[P]](ρ) for all ρ ∈ I
4      γ_{Q,ρ} = [[Q]](ρ) for all ρ ∈ I
5      M = V_Q × V_P
6      for v_2, v_1 ∈ V_Q × V_P:
7          for ρ ∈ I:
8              if γ_{Q,ρ}|_{v_2} ≠ γ_{P,ρ}|_{v_1}:
9                  M = M \ {(v_2, v_1)}
10                 break
11     return BijectiveMapping(M)
```

Figure 4. The Matching Algorithm.

Definition 4.2 (Variables of expression)**.** Let e be some expression, by $\mathcal{V}(e) = \{v \mid v \in e\}$ we denote the *set of variables used* in the expresison e. We also say that e *ranges over* $\mathcal{V}(e)$.

Definition 4.3 (Variable substitution)**.** Let $\tau : V_1 \to V_2$ be some function that maps variables V_1 to variables V_2. Given an expression e over variables V_1, i.e., $\mathcal{V}(e) \subseteq V_1$, by $\tau(e)$ we denote the expression, which we obtain from e, by substituting v with $\tau(v)$ for all $v \in V_1$. Note that $\mathcal{V}(\tau(e)) \subseteq V_2$.

Given a memory $\sigma \in \Sigma_{V_1}$, over variables V_1, we define $\tau(\sigma) = \{\tau(v) \mapsto \sigma(v), \tau(v)' \mapsto \sigma(v') \mid v \in V_1\}$, that is, the memory where v_1 is substituted with v_2, for all $v_2 = \tau(v_1)$. We lift substitution to a trace $\gamma = (\ell_1, \sigma_1) \cdots (\ell_n, \sigma_n) \in (L \times \Sigma_{V_1})^*$, by applying it to the each element: $\tau(\gamma) = (\ell_1, \tau(\sigma_1)) \cdots (\ell_n, \tau(\sigma_n)) \in (L \times \Sigma_{V_2})^*$.

In the rest of the paper we call a *bijective function* $\tau : V_1 \to V_2$, between two sets of variables V_1 and V_2, a **total variable relation** between V_1 and V_2. Next we give a formal definition of matching between two programs. Afterwards, we give a formal definition of matching between two expressions. The definitions involve execution of the programs on a set of inputs.

Definition 4.4 (Program Matching)**.** Let I be a set of inputs, and let $\gamma_{P,\rho} = [\![P]\!](\rho)$ and $\gamma_{Q,\rho} = [\![Q]\!](\rho)$ be sets of traces obtained by executing P and Q on $\rho \in I$, respectively.

We say that P and Q *match* over a set of inputs I, denoted by $P \sim_I Q$, if there exists a total variable relation $\tau : V_Q \to V_P$, such that $\gamma_{P,\rho} = \tau(\gamma_{Q,\rho})$, for all inputs $\rho \in I$. We call τ a *matching witness*.

Intuitively, a matching witness τ defines a way of translating Q to range over variables V_P, such that P and Q translated with τ produce the same traces.

Given a set of inputs I, $\sim_I$ is an *equivalence relation* on a set of programs $\mathcal{P}$: the *identity* relation on program variables gives a matching witness for *reflexivity*, the *inverse* τ^{-1} of some total variable relation τ gives a matching witness for *symmetry*, and the *composition* $\tau_1 \circ \tau_2$ of some total variables relations τ_1, τ_2 gives a matching witness for *transitivity*.

The algorithm for finding τ is given in Fig. 4: given two programs P and Q and a set of inputs I, it returns a matching

witness τ, if one exists. The algorithm first executes both programs on the inputs (lines 3 and 4), and then for each variable $v_2 \in V_Q$ finds a set of variables from V_P that take the same values during execution, thus defining a set of potential matches $M \subseteq V_Q \times V_P$ (lines 5-10); here $\gamma|_v$, denotes a projection of values of variable v from a trace γ, that is: $((\ell_1, \sigma_1) \cdots (\ell_n, \sigma_n))|_v = \sigma_1(v') \cdots \sigma_n(v')$. The matching witness is then a bijective mapping $\tau \subseteq M$, if one exists (line 11); τ can be found in M by constructing a *maximum bipartite matching* in the *bipartite graph* defined by M. We note that this problem can be solved in polynomial time, w.r.t. the number of the edges and vertices in M (e.g.,Uno [41]).

Definition 4.5 (Expression matching)**.** Let $\Gamma \subseteq (L \times \Sigma_{V_P})^*$ be a set of traces over variables V_P, and let e_1 and e_2 be two expressions over variables V_P, at some location $\ell \in L$.

We say that e_1 and e_2 *match* over a set of traces Γ, denoted $e_1 \simeq_{\Gamma,\ell} e_2$, if $[\![e_1]\!](\sigma) = [\![e_2]\!](\sigma)$, for all $(\sigma, \ell_i) \in \gamma$ where $\ell_i = \ell$, and all $\gamma \in \Gamma$.

Expression matching says that two expressions produce the same values, when considering the memories at location ℓ, in a set of traces Γ. In the following lemma we state that expression matching is equivalent to program matching; this lemma will be useful for our repair algorithm, which we will state in the next section.

Lemma 4.6 (Matching Equivalence)**.** *Let I be a set of inputs, and let $\Gamma_I = \{[\![P]\!](\rho) \mid \rho \in I\}$ be a set of traces obtained by executing P on I. We have the following equivalence: $P \sim_I Q$ witnessed by $\tau : V_Q \to V_P$, if and only if, $e_P \simeq_{\Gamma_I,\ell} \tau(e_Q)$, for all $(\ell, v_1) \in L \times V_P$, where $v_1 = \tau(v_2)$, $e_P = \mathcal{U}_P(\ell, v_1)$, and $e_Q = \mathcal{U}_Q(\ell, v_2)$.*

Proof. "⇒": Directly from the definitions. "⇐": By induction on the length of the trace $\gamma = [\![P]\!](\rho)$ for some $\rho \in I$. □

Clustering We define clusters as the equivalence classes of $\sim_I$. For the purpose of matching and repair we pick an arbitrary class representative from the class and collect expressions from all programs in the same cluster:

Definition 4.7 (Cluster)**.** Let $\mathcal{P}$ be a set of (correct) programs. A *cluster* $C \subseteq \mathcal{P}$ is an *equivalence class* of $\sim_I$. Given some cluster C, we fix some *arbitrary class representative* $P_C \in C$.

We define the set $\mathcal{E}_C(\ell, v_1)$ of *cluster expressions* for a pair $(\ell, v_1) \in L \times V_{P_C}$: $e_1 \in \mathcal{E}_C(\ell, v_1)$ iff there is some $Q \in C$ witnessed by $\tau : V_Q \to V_{P_C}$ such that $v_1 = \tau(v_2)$, $e_2 = \mathcal{U}_Q(\ell, v_2)$ and $e_1 = \tau(e_2)$.

Note that it is irrelevant which program from C is chosen as cluster representative P_C; we just need to fix some program in order to be able to define the expressions $\mathcal{E}_C$ over a common set of variables V_{P_C}. We note that by definition the sets of expressions $\mathcal{E}_C$ have the following property: for all $e_1, e_2 \in \mathcal{E}_C(\ell, v)$ we have $e_1 \simeq_{\Gamma_I,\ell} e_2$ that is, expressions e_1 and e_2 match.

Example. In §2.1 we discussed why the solutions **C1** and **C2** match; therefore these two solutions belong to the same cluster, which we denote here by C, and chose **C1** as its representative P_C. Also, in Fig. 2 (c) and (d) we gave examples of different equivalent expressions of assignment to the variable *result* inside the loop body and the return statement after the loop, respectively. To be more precise, these were examples of sets $\mathcal{E}_C(\ell_{loop}, result)$ and $\mathcal{E}_C(\ell_{after}, return)$, respectively.

5 Repair Algorithm

In the previous section we defined the notion of a matching between two programs. In this section we consider an implementation P_{impl} and a cluster C (with its representative P_C) between which there is no matching. We assume that P_{impl} and P_C have the same control-flow. The goal is to *repair* P_{impl}; that is, to modify P_{impl} minimally, w.r.t. some notion of cost, such that the repaired program matches the cluster. More precisely, the repair algorithm searches for a program $P_{repaired}$, such that $P_C \sim_I P_{repaired}$, and $P_{repaired}$ should be syntactically close to P_{impl}. Therefore, our repair algorithm can be seen as a *generalization of clustering to incorrect programs.*

We first define a version of the repair algorithm that does not change the set of variables, i.e., $V_{impl} = V_{repaired}$. Below we extend this algorithm to include changes of variables, i.e., we allow $V_{impl} \neq V_{repaired}$. In both cases the control-flow of P_{impl} remains the same.

For the following discussion we fix some set of inputs I. Let $\Gamma = \{[\![P_C]\!](\rho) \mid \rho \in I\}$ be the set of traces of cluster representative P_C for the inputs I.

As we discussed in the previous section, two programs match if all of their corresponding expressions match (see Lemma 4.6). Therefore the key idea of our repair algorithm is to consider a set of *local repairs* that modify individual implementation expressions. We first discuss local repairs; later on we will discuss how to combine local repairs into a full program repair.

Local repairs are defined with regard to *partial variable relations.* It is enough to consider partial variable relations (as opposed to the total variable relations needed for matchings) because these relations only need to be defined for the expressions that need to be repaired.

Definition 5.1 (Local Repair)**.** Let $(\ell, v_2) \in L \times V_{impl}$ be a location-variable pair from P_{impl}, and let $e_{impl} = \mathcal{U}_{impl}(\ell, v_2)$ be the corresponding expression. Further, let $\omega : V_{impl} \rightharpoonup V_{P_C}$ be a *partial variable relation* such that $v_2 \in dom(\omega)$, let $v_1 = \omega(v_2)$ be the related cluster representative variable, and let $e_C = \mathcal{U}_{P_C}(\ell, v_1)$ be the corresponding expression.

A pair $r = (\omega, e_{repaired})$, where $e_{repaired}$ is an expression over implementation variables V_{impl}, is a *local repair* for (ℓ, v_2) when $e_C \simeq_{\Gamma,\ell} \omega(e_{repaired})$ and $\mathcal{V}(e_{repaired}) \subseteq dom(\omega)$. A pair $r = (\omega, \bullet)$ is a *local repair* for (ℓ, v_2) when $e_C \simeq_{\Gamma,\ell} \omega(e_{impl})$ and $\mathcal{V}(e_{impl}) \subseteq dom(\omega)$.

We define the cost of a local repair $r = (\omega, e_{repaired})$ as $cost(r) = diff(e_{impl}, e_{repaired})$ and the cost of a local repair $r = (\omega, \bullet)$ as $cost(r) = 0$.

We comment on the definition of a local repair. Let $(\ell, v_2) \in L \times V_{impl}$ be a location-variable pair from P_{impl}, let $e_{impl} = \mathcal{U}_{impl}(\ell, v_2)$ be the corresponding expression, and let r be a local repair for some (ℓ, v_2). In case $r = (\omega, \bullet)$, the expression e_{impl} matches the corresponding expression of the cluster representative under the partial variable mapping ω; this repair has cost zero because the expression e_{impl} is not modified. In case $r = (\omega, e_{repaired})$, the expression $e_{repaired}$ constitutes a modification of e_{impl} that matches the corresponding expression of the cluster representative under the partial variable mapping ω; this repair has some cost $diff(e_{impl}, e_{repaired})$. In our implementation we define $diff(e_{impl}, e_{repaired})$ to be the tree edit distance [38, 43] between the abstract syntax trees (ASTs) of the expressions e_{impl} and $e_{repaired}$.

Example. We remind the reader that in §2.2 we discussed three examples of local repairs for **I1** (Fig. 2). Formally, example (1) is a local repair for (ℓ_1, new), while (2) and (3) are local repairs for $(\ell_4, return)$. Next we state how to combine local repairs into a full program repair.

Definition 5.2 (Repair)**.** Let R be a function that assigns to each pair $(\ell, v) \in L \times V_{impl}$ a *local repair* for (ℓ, v). We say that R is *consistent*, if there exists a total variable relation $\tau : V_{impl} \to V_{P_C}$, such that $\omega \subseteq \tau$, for all $R(\ell, v) = (\omega, -)$. A consistent R is called a *repair*. We define the *cost* of R as the sum of the costs of all local repairs: $cost(R) = \sum_{(\ell,v)\in L\times V_{impl}} cost(R(\ell, v))$.

A repair R defines a *repaired implementation* $P_{repaired} = (L, \ell_{init}, V_{impl}, \mathcal{U}_{repaired}, \mathcal{S})$, where $\mathcal{U}_{repaired}(\ell, v) = e_{repaired}$ if $M(\ell, v) = (-, e_{repaired})$, and $\mathcal{U}_{repaired}(\ell, v) = \mathcal{U}_{impl}(\ell, v)$ if $M(\ell, v) = (-, \bullet)$, for all $(\ell, v) \in L \times V_{impl}$.

Theorem 5.3 (Soundness of Repairs)**.** $P_C \sim_I P_{repaired}$.

Proof. (sketch) From the definition of $R(\ell, v_2)$, we have $e_C \simeq_{\Gamma,\ell} \tau(e_{repaired})$, for all $(\ell, v_2) \in L \times V_{impl}$, where $v_1 = \tau(v_2)$, $e_C = \mathcal{U}_C(\ell, v_1)$ and $e_{repaired} = \mathcal{U}_{repaired}(\ell, v_2)$. Then it follows from Lemma 4.6 that $P_C \sim_I P_{repaired}$. □

In the above definition we use notation $r = (\omega, -)$ when $e_{repaired}$ or $\bullet$ is not important in r; similarly we use $r = (-, e_{repaired})$ and $r = (-, \bullet)$ when ω is not important in r.

Example. The repair for example **I1** (Fig. 2) corresponds to the total variable relation $\tau = \{poly \mapsto poly, new \mapsto result, e \mapsto i, return \mapsto return, ? \mapsto ?\}$. The repair M includes local repairs (1) and (2) from the previous examples, where only (2) has cost > 0 (see the repair in Fig. 2 (g)).

Next we discuss the algorithm for finding a repair.

The repair algorithm The algorithm is given in Fig. 5: given a cluster C, an implementation P_{impl}, and a set of inputs I, it returns a repair R. The algorithm has two main parts: First, the algorithm constructs a set of *possible local repairs*; we define and discuss the possible local repairs below. Second, the algorithm searches for a consistent subset of the possible local repairs, which has minimal cost; this search corresponds to solving a constraint-optimization system.

```
1  fun Repair(Cluster C, Implementation P_impl, Inputs I):
2      π = structural matching or abort
3      Γ = {[[P_C]](ρ) | ρ ∈ I}
4      for (ℓ, v_2) ∈ L × V_impl:
5          LR(ℓ, v_2) = ∅
6          e_impl = U_impl(ℓ, v_2)
7          for v_1 ∈ V_{P_C}:
8              e_C = U_{P_C}(ℓ, v_1)
9              for ω : (V(e_impl) ∪ {v_2}) → V_{P_C} s.t. ω(v_2) = v_1:
10                 if e_C ≃_{Γ,ℓ} ω(e_impl):
11                     LR(ℓ, v_2) = LR(ℓ, v_2) ∪ {(ω, •)}
12             for e ∈ E_C(ℓ, v_1):
13                 for ω : (V(e) ∪ {v_1}) → V_impl s.t. ω(v_1) = v_2:
14                     LR(ℓ, v_2) = LR(ℓ, v_2) ∪ {(ω^{-1}, ω(e))}
15     return FindRepair(LR)
```

Figure 5. The Repair Algorithm.

Definition 5.4 (Set of possible local repairs)**.** For all $(\ell, v) \in L \times V_{impl}$, we define the set of *possible local repairs* $LR(\ell, v)$ as: (1) $(\omega, e) \in LR(\ell, v)$, if $\omega(e) \in \mathcal{E}_C(\ell, \omega(v))$; and (2) $(\omega, \bullet) \in LR(\ell, v)$, if $e_C \simeq_{\Gamma,\ell} \omega(e_{impl})$, where $e_C = \mathcal{U}_{P_C}(\ell, \omega(v))$ and $e_{impl} = \mathcal{U}_{impl}(\ell, v)$.

The set of possible local repairs $LR(\ell, v)$ includes all expressions from the cluster $\mathcal{E}_C(\ell, \omega(v))$, translated by some partial variable relation ω in order to range over implementation variables. It also includes $(\omega, \bullet)$ if e_{impl} matches e_C under partial variable mapping ω. Next we describe how the algorithm constructs the set $LR(\ell, v)$ (at lines 4-14).

For the following discussion we fix a pair $(\ell, v_2) \in L \times V_{impl}$ (corresponding to line 4), and some $v_1 \in V_{P_C}$ (corresponding to line 7); we set $e_{impl} = \mathcal{U}_{impl}(\ell, v_2)$ and $e_C = \mathcal{U}_{P_C}(\ell, v_1)$. Possible local repairs for (ℓ, v_2) are constructed in two steps: In the first step, the algorithm checks if there are partial variable relations $\omega : V_{impl} \rightharpoonup V_{P_C}$ s.t. $e_C \simeq_{\Gamma,\ell} \omega(e_{impl})$ (at line 9), and in that case adds a pair $(\omega, \bullet)$ to $LR(\ell, v_2)$ (at line 11). In the second step, the algorithm iterates over all cluster expressions $e = \mathcal{E}_{P_C}(\ell, v_1)$ (at line 12), and all partial variable relations $\omega : V_{P_C} \rightharpoonup V_{impl}$ (at line 13), and then adds a pair $(\omega^{-1}, \omega(e))$ to $LR(\ell, v_2)$ (at line 14). We note that $\omega^{-1}(\omega(e)) = e \in \mathcal{E}_C(\ell, v_1) = \mathcal{E}_C(\ell, \omega^{-1}(v_2))$, and thus $(\omega^{-1}, \omega(e))$ is a possible local repair as in Def. 5.4.

We remark that in both steps, the algorithm iterates over all possible variable relations ω. However, since ω relates only the variables of a single expression — usually a small subset of all program variables, this iteration is feasible.

Finding a repair with the smallest cost Finally, the algorithm uses sub-routine FindRepair (at line 15) that, given a set of possible local repairs LR, finds a repair with smallest cost. FindRepair encodes this problem as a *Zero-One Integer Linear Program (ILP)*, and then hands it to an off-the-shelf

ILP solver. Next, we define the ILP problem, describe how we encode the problem of finding a repair as an ILP problem, and how we decode the ILP solution to a repair.

Definition 5.5 ((Zero-One) ILP)**.** An ILP problem, over variables $\mathcal{I} = \{x_1, \ldots, x_n\}$, is defined by an *objective function* $O = \boxdot \sum_{1 \le i \le n} w_i \cdot x_i$, and a set of *linear (in)equalities* C, of the form $\sum_{1 \le i \le n} a_i \cdot x_i \trianglerighteq b$. Where $\boxdot \in \{\min, \max\}$ and $\trianglerighteq = \{\ge, =\}$. A solution to the ILP problem is a variable assignment $\mathcal{A} : \mathcal{I} \to \{0, 1\}$, such that all (in)equalities hold, and the value of the objective functions is minimal (resp. maximal) for $\mathcal{A}$.

We encode the problem of finding a consistent subset of possible local repairs as an ILP problem with variables $\mathcal{I} = \{x_{v_1 v_2} \mid v_1 \in V_{P_C} \text{ and } v_2 \in V_{impl}\} \cup \{x_r \mid r \in LR(\ell, v), (\ell, v) \in L \times V_{impl}\}$; that is, one variable for each pair of variables (v_1, v_2), and one variable for each possible local repair r. The set of constraints C is defined as follows:

$$\left(\sum_{v_2 \in V_{impl}} x_{v_1 v_2}\right) = 1 \text{ for each } v_1 \in V_{P_C} \quad (1)$$

$$\left(\sum_{v_1 \in V_{P_C}} x_{v_1 v_2}\right) = 1 \text{ for each } v_2 \in V_{impl} \quad (2)$$

$$\left(\sum_{r \in LR(\ell, v)} x_r\right) = 1 \text{ for each } (\ell, v) \in L \times V_{impl} \quad (3)$$

$$-x_r + x_{u_1 u_2} \ge 0 \text{ for each } r = (\omega, -) \in LR \text{ and each } \omega(u_2) = u_1 \quad (4)$$

Intuitively, the constraints encode:

1. Each $v_1 \in V_{P_C}$ is related to *exactly one* of $v_2 \in V_{impl}$.
2. Each $v_2 \in V_{impl}$ is related to *exactly one* of $v_1 \in V_{P_C}$. Together (1) and (2) encode that there is a total variable relation $\tau : V_{impl} \to V_{P_C}$.
3. For each $(\ell, v) \in L \times V_{impl}$ *exactly one* local repair is selected.
4. Each selected local repair $r = (\omega, -) \in LR$ is *consistent* with τ, i.e., $\omega \subseteq \tau$.

The objective function $O = \min\left(\sum_{r \in LR} cost(r) \cdot x_r\right)$ ensures that the sum of the costs of the selected local repairs is minimal.

Let $\mathcal{A} : \mathcal{I} \to \{0, 1\}$ be a solution of the ILP problem. We obtain the following total variable relation from $\mathcal{A}$: $\tau(v_2) = v_1$ iff $\mathcal{A}(x_{v_1 v_2}) = 1$. For $\mathcal{A}(x_r) = 1$, where $LR(\ell, v) = r$, we set $R(\ell, v) = r$.

Adding and Deleting Variables The repair algorithm described so far does not change the set of variables, i.e., $V_{repaired} = V_{impl}$. However, since the repair algorithm constructs a bijective variable relation, this only works when the implementation and cluster representative have the same number of variables, i.e., $|V_{impl}| = |V_{P_C}|$. Hence, we extend the algorithm to also allow the addition and deletion of variables.

We extend total variable relations $\tau : V_{impl} \to V_{P_C}$ to relations $\tau \subseteq (V_{impl} \cup \{\star\}) \times (V_{P_C} \cup \{-\})$. We relax the condition about τ being total and bijective: $\star$ and $-$ can be related to multiple variables or none. When some variable $v \in V_{P_C}$ is related to $\star$, that is $\tau(\star) = v$, it denotes that a *fresh variable is added* to P_{impl}, in order to match v. Conversely, when some variable $v \in V_{impl}$ is related to $-$, that is $\tau(v) = -$, variable v is *deleted* from P_{impl}, together with all its assignments.

An example of a repair where a fresh variable is added is given in the extended version [19].

Completeness of the algorithm We point out that with this extension the repair algorithm is *complete* (assuming P_{impl} and P_C have the same control-flow). This is because the repair algorithm can always generate a *trivial repair*: all variables $v_2 \in V_{impl}$ are deleted, that is $\tau(v_2) = -$ for all $v_2 \in V_{impl}$; and a fresh variable is introduced for every variable $v_1 \in V_{P_C}$, that is $\tau(\star) = v_1$ for all $v_1 \in V_{P_C}$. Clearly, this trivial repair has high cost, and in practice it is very rarely generated, as witnessed by our experimental evaluation in the next section.

6 Implementation and Experiments

We now describe our implementation (§6.1) and an experimental evaluation, which consists of two parts: (I) an evaluation on MOOC data (§6.2), and (II) a user study about the usefulness of the generated repairs (§6.3). The evaluation was performed on a server with an *AMD Opteron 6272 2.1GHz* processor and *224 GB* RAM.

6.1 Implementation

We implemented the proposed approach in the publicly available tool Clara[4] (for **CL**uster **A**nd **R**ep**A**ir). The tool currently supports programs in the programming languages C and Python, and consists of: (1) Parsers for C and Python that convert programs to our internal program representation; (2) Program and expression evaluation functions for C and Python, used in the matching and repair algorithms; (3) Matching algorithm; (4) Repair algorithm; (5) Simple feedback generation system. We use the *lpsolve* [2] *ILP* solver, and the *zhang-shasha* [4] tree-edit-distance algorithm.

Feedback generation We have implemented a simple feedback generation system that generates the location and a textual description of the required modifications (similar to AutoGrader). Other types of feedback can be generated from the repair as well, and we briefly discuss it in §9.

6.2 MOOC evaluation

Setup In the first experiment we evaluate Clara on data from the MITx introductory programming MOOC [3], which is similar to the data used in evaluation of AutoGrader [33].

This data is stripped from all information about student identity, i.e., there are not even anonymous identifiers. To avoid the threat that a student's attempt is repaired by her own future correct solution, we split the data into two sets. From the first (chronologically earlier) set we take only the

[4] https://github.com/iradicek/clara

Table 1. List of the problems with evaluation results for the MOOC data (with AutoGrader comparison).

Problem name	LOC median	AST size median	# correct	# clusters (% of # correct)	# incorrect	# repaired (% of # incorrect)		avg. (median) time in s	
						CLARA	AutoGrader	CLARA	AutoGrader
derivatives	14	33	1472	532 (36.14%)	481	472 (98.13%)	235 (48.86%)	4.9s (4.4s)	6.6s (5.2s)
oddTuples	10	25	9001	454 (5.04%)	3584	3514 (98.05%)	576 (16.07%)	3.0s (2.6s)	25.5s (13.3s)
polynomials	13	25	2500	234 (9.36%)	228	197 (86.40%)	17 (7.46%)	1.9s (1.6s)	4.3s (4.0s)
Total	11	25	12973	1220 (9.40%)	4293	4183 (97.44%)	828 (19.29%)	3.2s (2.7s)	19.7s (6.3s)

correct solutions: these solutions are then clustered, and the obtained clusters are used during the repair of the incorrect attempts. From the second (chronologically later) set we take only the incorrect attempts: on these attempts we perform repair. We have split the data in 80 : 20 ratio since then we have a large enough number (12973; see the discussion below) of correct solutions that our approach requires, while still having quite a large number (4293) of incorrect attempts for the repair evaluation. We point out that this is precisely the setting that we envision our approach to be used in: a large number of existing correct solutions (e.g., from a previous offering of a course) are used to repair new incorrect student submissions.

Results The evaluation summary is in Table 1; the descriptions of the problems are in the extended version of the paper [19]. CLARA **automatically** generates a repair for *97.44%* of attempts. As expected, CLARA can generate repairs in almost all the cases, since there is always the *trivial* repair of completely replacing the student implementation with some correct solution of the same control flow. Hence, it is mandatory to study the quality and size of the generated repairs. We evaluate the following questions in more detail: *(1) What are the reasons when CLARA fails? (2) Does CLARA generate non-trivial repairs? (3) What is the quality and size of the generated repairs?*

We discuss the results of this evaluation below, while further examples can be found in the extended version [19].

(1) CLARA fails CLARA fails to generate repair in 110 cases: in 69 cases there are unsupported PYTHON features (e.g., nested function definitions), in 35 cases there is no correct attempt with the same control-flow, and in 6 cases a numeric precision error occurs in the ILP solver. The only fundamental problem of our approach is the inability to generate repairs without matching control-flow; however, since this occurs very rarely, we leave the extension for a future work. *Hence, we conclude that CLARA can repair almost all programs.*

(2) Non-trivial repairs Since a correct repair is also a *trivial* one that *completely replaces* a student's attempt with a correct program, we measure *how much a repair changes the student's program.* To measure this we examine the *relative repair size*: the *tree-edit-distance* of the repair divided by *the size of the AST* of the program. Intuitively, the tree-edit-distance tells us how many changes were made in a program,

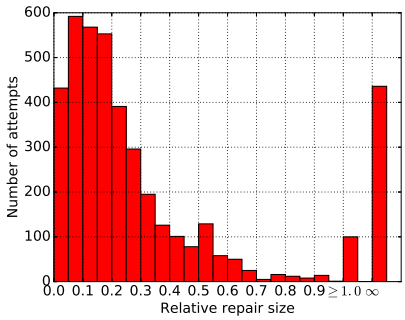
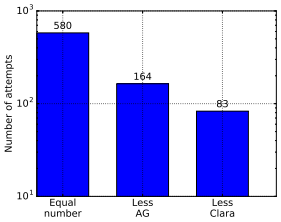
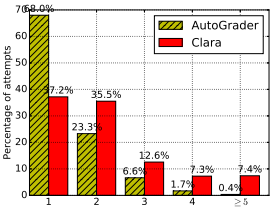


Figure 6. Histogram of relative repair sizes.

and normalization with the total number of AST nodes gives us the *ratio of how much of the whole program changed.* Note, however that this ratio can be > 1.0, or even ∞ if the program is empty. Fig. 6 shows a histogram of relative repair sizes. We note that 68% of all repairs have relative size < 0.3, 53% have < 0.2 and 25% have < 0.1; the last column (∞) is caused by 436 completely empty student attempts. As an example, the two repairs in Fig. 2 (g) and (h), have relative sizes of 0.03 and 0.24. *We conclude that CLARA in almost all cases generates a non-trivial repair that is not a replacement of the whole student's program.*



(a) The number of modified expressions per repair.



(b) Distribution of the number of modified expressions per repair.

Figure 7. Comparison of the generated repairs size between AutoGrader and CLARA.

(3) Repair quality and repair size We inspected 100 randomly selected generated repairs, with the goal of evaluating their quality and size. Our approach of judging repair quality and size mirrors a human teacher helping a student: the teacher has to guess the student's idea and use subjective judgment on what feedback to provide. We obtained the following results: (a) In 72 cases CLARA generates *the smallest,*

most natural repair; (b) In 9 cases the repair is almost the smallest, but involves an additional modification that is not required; (c) In 11 cases we determined the repair, although correct, to be different from the student's idea; (d) In 8 cases it is not possible to determine the idea of the student's attempt, although Clara generates some correct repair.

For the cases in (d), we found that program repair is not adequate and further research is needed to determine what kind of feedback is suitable when the student is far from any correct solution. For the cases in (c), the set of correct solutions does not contain any solution which is syntactically close to the student's idea; we conjecture that Clara's results in these cases can partially be improved by considering different cost functions which do not only take syntactic differences into account but also make use of semantic information (see the discussion in §9). However, in 81 cases (the sum of (a) and (b)), Clara *generates good quality repairs. We conclude that Clara mostly produces good quality repairs.*

Summary *Our large-scale experiment on the MOOC dataset shows that Clara can fully automatically repair almost all programs and the generated repairs are of high quality.*

Clusters Finally, we briefly discuss the correct solutions, since our approach depends on their existence. The quality of the generated repair should increase with the number of clusters, since then the algorithm can generate more diverse repairs. Thus, it is interesting to note that we experienced *no performance issues with a large number of clusters*; e.g., on `derivatives`, with *532* clusters, a repair is generated on average in *4.9s*. This is because the repair algorithm processes multiple clusters in parallel. Nonetheless, clustering is important for *repair quality*, since it enables repairs that combine expressions taken from different correct solutions from the same cluster, which would be impossible without clustering. We found that 2093 (around 50%) repairs were generated using at least two different correct solutions, and 110 (around 3%) were generated using at least three different correct solutions, in the same cluster.

6.2.1 Comparison with AutoGrader

While the setting of AutoGrader is different (a teacher has to provide an error model, while our approach is fully automated), the same high-level goal (finding a minimal repair for a student attempt to provide feedback) warrants an experimental comparison between the approaches.

Setup and Data We were not able to obtain the data used in AutoGrader's evaluation, which stems from an internal MIT course, because of privacy concerns regarding student data. Hence, we compare the tools on the same MITx introductory programming MOOC data, which we used in the paper for Clara evaluation. This dataset is similar to the dataset used in AutoGrader's evaluation. AutoGrader's authors provided us with an AutoGrader version that is optimized to scale to a MOOC, that is, it has a weaker error model than in the original AutoGrader's publication [33]. According to the authors, some error rewrite rules were intentionally omitted, since they are too slow for interactive online feedback generation.

Results The evaluation summary is in Table 1. AutoGrader is able to generate a repair for *19.29%* of attempts, using *manually* specified rewrite-rules, compared to *97.44% automatically* generated repairs by Clara. (We note that AutoGrader is able to repair fewer attempts than reported in the original publication [33] due to the differences discussed in the previous paragraph.) As Clara can generate repairs in almost all the cases, these numbers are not meaningful on their own; the numbers are, however, meaningful in conjunction with our evaluation of the following questions: *(1) How many repairs can one tool generate that other cannot, and what are the reasons when AutoGrader fails? (2) What are the sizes of repairs? (3) What is the quality of the generated repairs, in case both tools generate a repair?*

We summarize the results of this evaluation below, while a more detailed discussion can be found in the extended version [19].

(1) Repair numbers In all but one case, when Clara fails to generate a repair, AutoGrader also fails. Further, we manually inspected 100 randomly selected cases where AutoGrader fails, and determined that in 77 cases there is a fundamental problem with AutoGrader's approach: The modifications require fresh variables, new statements or larger modifications, which are beyond AutoGrader's capabilities. *This shows that Clara can generate more complicated repairs than AutoGrader.*

In the 100 cases we manually inspected we also determined that *in 74 cases Clara generates good quality repairs, when AutoGrader fails.*

(2) Repair sizes We do not report the relative repair size metric for AutoGrader, because we were not able to extract the repair size from its (textual) output. However, Fig. 7 (a) compares the relation of the number of modified expressions when both tools generate a repair. We note that the number of modified expressions is a weaker metric than the tree-edit-distance, however, we were only able to extract this metric for the repairs generated by AutoGrader. *We conclude that AutoGrader produces smaller repair in around 10% of the cases.*

Fig. 7 (b) also compares the overall (not just when both tools generate a repair) distribution of the number of changed expression per repair. *We notice that most of AutoGrader's repairs modify a single expression, and the percentage falls faster than in Clara's case.*

(3) Repair quality Finally, we manually inspected 100 randomly selected cases where both tools generate a repair. In 61 cases we found both tools to produce the same repair; in

Table 2. List of the problems with evaluation details for user study.

Problem	LOC median	# correct (exist.+study)	# clusters (exist.+study)	# incorr.	# feedback (% of # incorr.)	# repair feedback (% of # feedback)	time (in s) avg.	time (in s) median	# grades 1/2/3/4/5
Fibonacci sequence	12	512+84	70 + 17 (14.60%)	572	539 (94.23%)	440 (81.63%)	10.44	8.51	1 / 7 / 9 / 16 / 13
Special number	15	358+59	39 + 3 (10.07%)	121	109 (90.08%)	94 (86.24%)	3.77	2.38	2 / 3 / 8 / 9 / 13
Reverse Difference	17	342+46	48 + 8 (14.43%)	103	77 (74.76%)	68 (88.31%)	4.39	3.07	4 / 4 / 5 / 3 / 5
Factorial interval	14	391+44	56 + 8 (14.71%)	234	232 (99.15%)	185 (79.74%)	3.33	3.17	2 / 5 / 4 / 5 / 13
Trapezoid	14	281+41	36 + 15 (15.84%)	143	129 (90.21%)	121 (93.80%)	7.55	4.82	7 / 5 / 7 / 7 / 5
Rhombus	21	264+38	73 + 22 (31.46%)	525	417 (79.43%)	192 (46.04%)	9.16	5.35	6 / 9 / 6 / 5 / 3

19 cases different, although of the same quality; in 9 cases we consider AutoGrader to be better; in 5 cases we consider Clara to be better; and in 6 cases we found that AutoGrader generates an incorrect repair. *We conclude that there is no notable difference between the tools when both tools generate a repair.*

6.3 User study on usefulness

In the second experiment we performed a user study, evaluating Clara in real time. We were interested in the following questions: (1) *How often and fast is feedback generated (performance)?* (2) *How useful is the generated repair-based feedback?*

Setup To answer these questions we developed a web interface for Clara and conducted a user study, which we advertised on programming forums, mailing lists, and social networks. Each participant was asked to solve six introductory C programming problems, for which the participants received feedback generated by Clara. There was one additional problem, not discussed here, that was almost solved, and whose purpose was to familiarize the participants with the interface. After solving a problem each participant was presented with the question: *"How useful was the feedback provided on this problem?"*, and could select a grade on the scale from 1 ("*Not useful at all*") to 5 ("*Very useful*"). Additionally, each participant could enter an additional textual comment for each generated feedback individually and at the end of solving a problem.

We also asked the participants to assess their programming experience with the question: *"Your overall programming experience (your own, subjective, assessment)"*, with choices on the scale from 1 ("*Beginner*") to 5 ("*Expert*").

The initial correct attempts were taken from an introductory programming course at *IIT Kanpur, India.* The course is taken by around 400 students of whom several have never written a program before. We selected problems from two weeks where students start solving more complicated problems using loops. Of the 16 problems assigned in these two weeks, we picked those 6 that were sufficiently different.

Results Table 2 shows the summary of the results; detailed descriptions of all problems are available in the extended paper version [19]. The columns **# correct** and **# clusters** show the number of correct attempts and clusters obtained from: (a) the existing ESC 101 data (*exist.* in the table), and (b) during the case study from participants' correct attempts (*study* in the table). We plan to make the complete data, with all attempts, grades and textual comments publicly available.

Performance of Clara Feedback was generated for 1503 (**88.52%**) of incorrect attempts. In the following we discuss the 3 reasons why feedback could not be generated: (1) In 57 cases there was a bug in Clara, which we have fixed after the experiment finished. Then we confirmed that in all 57 cases the program is correctly repaired and feedback is generated (note that this bug was only present in this real-time experiment, i.e., it did not impact the experiment described in the previous section); (2) In 43 cases a timeout occurred (set to 60s); (3) In 95 cases a program contained an unsupported C construct, or there was a syntactic compilation error that was not handled by the web interface (Clara currently provides no feedback on programs that cannot be even parsed). Further, the average time to generate feedback was **8** seconds. *These results show that Clara provides feedback on a large percentage of attempts in real time.*

Feedback usefulness The results are based on *191* grades given by *52* participants. Note that problems have a different number of grades. This is because we asked for a grade only when feedback was successfully generated (as noted above, in 88.52% cases), and because some of the participants did not complete the study. The average grade over all problems is **3.4**. *This shows very promising preliminary results on the usefulness of Clara.* However, we believe that these results can be further improved (see §9).

The participants declared their experience as follows: 22 as 5, 19 as 4, 9 as 3, 0 as 2, and 2 as 1. While these are useful preliminary results, a study with beginner programmers is an important future work.

Note. In the case of a very large repair (*cost* > 100 in our study), we decided to show a generic feedback explaining a general strategy on how to solve the problem. This is because the feedback generated by such a large repair is usually not useful. We generated such a general strategy in 403 cases.

6.4 Threats to Validity

Program size We have evaluated our approach on small to medium size programs typically found in introductory programming problems. The extension of our approach to larger programming problems, as found in more advanced courses,

is left for future work. Focusing on small to medium size programs is in line with related work on automated feedback generation for introductory programming (e.g., D'Antoni et al. [9], Singh et al. [33], Head et al. [20]). We stress that the state-of-the-art in teaching is manual feedback (as well as failing test cases); thus, automation, even for small to medium size programs, promises huge benefits. We also mention that our dataset contains larger and challenging attempts by students which use *multiple functions*, *multiple* and *nested loops*, and our approach is able to handle them.

The focus of our work differs from the related work on program repair (see §7 for a more detailed discussion). Our approach is specifically designed to perform well on small to medium size programs typically found in introductory programming problems, rather than the larger programs targeted in the literature on automated program repair. In particular, our approach addresses the challenges of (1) a high number of errors (education programs are expected to have higher error density [33]), (2) complex repairs, and (3) a runtime fast enough for use in an interactive teaching environment. These goals are often out of reach for program repair techniques. For example, the general purpose program repair techniques discussed in Goues et al. [17] on the IntroClass benchmark, either repair a small number of defects (usually <50%) or take a long time (i.e., over one minute).

Unsoundness Our approach guarantees only that repairs are correct over a given set of test cases. This is in accordance with the state-of-the-art in teaching, where testing is routinely used by course instructors to grade programming assignments and provide feedback (e.g., for ESC101 at IIT Kanpur, India [10]). When we manually inspected the repairs for their correctness, we did not find any problems with soundness. We believe that this due to the fact that programming problems are small, human-designed problems that have comprehensive sets of test cases.

In contrast to our dynamic approach, one might think about a sound static approach based on symbolic execution and SMT solving. We decided for a dynamic analysis because symbolic execution can sometimes take a long time or even fail when constructs are not supported by an SMT solver. For example, reasoning about floating points and lists is difficult for SMT solvers. On the other hand, our method only executes given expressions on a set of inputs, so we can handle any expression, and our method is fast. Further, our evaluation showed our dynamic approach to be precise enough for the domain of introductory programming assignments. The investigation of a static verification of the results generated by our repair approach is an interesting direction for future work: one could take the generated repair expressions and verify that they indeed establish a simulation with the cluster against which the program was repaired.

7 Related Work

Automated Feedback Generation Ihantola et al. [21] present a survey of tools for the automatic assessment of programming exercises. Pex4Fun [40] and its successor CodeHunt [39] are browser-based, interactive platforms where students solve programming assignments with hidden specifications, and are presented with a list of automatically generated test cases. LAURA [5] heuristically applies program transformations to a student's program and compares it to a reference solution, while reporting mismatches as potential errors (they could also be correct variations). Apex [25] is a system that automatically generates error explanations for bugs in assignments, while our work automatically clusters solutions and generates repairs for incorrect attempts.

Trace analysis Striewe and Goedicke [35] have proposed presenting full program traces to the students, but the interpretation of the traces is left to the students. They have also suggested automatically comparing the student's trace to that of a sample solution [36]. However, the approach misses a discussion of the situation when the student's code enters an infinite loop, or has an error early in the program that influences the rest of the trace. The approach of Gulwani et al. [18] uses a dynamic analysis based approach to find a strategy used by the student, and to generate feedback for *performance aspects*. However, the approach requires specifications *manually provided* by the teacher, written in a specially designed specification language, and it only matches specifications to correct attempts, i.e., it *cannot provide feedback on incorrect attempts*.

Program Classification in Education CodeWebs [30] classifies different AST sub-trees in equivalence classes based on probabilistic reasoning and program execution on a set of inputs. The classification is used to build a search engine over ASTs to enable the instructor to search for similar attempts, and to provide feedback on some class of ASTs. OverCode [15] is a visualization system that uses a lightweight static and dynamic analysis, together with manually provided rewrite rules, to group student attempts. Drummond et al. [13] propose a statistical approach to classify interactive programs in two categories (*good* and *bad*). Head et al. [20] cluster incorrect programs by the type of the required modifications. CoderAssist [23] provides feedback on student implementations of dynamic programming algorithms: the approach first clusters both correct and incorrect programs based on their syntactic features; feedback for incorrect program is generated from a counterexample obtained from an equivalence check (using SMT) against a correct solution in the same cluster.

Program Repair The research on program repair is vast; we mention some work, with emphasis on introductory education. The non-education program repair approaches are based on SAT [16], symbolic execution [26], games [22, 34],

program mutation [11], and genetic programming [6, 14]. In contrast, our approach uses dynamic analysis for scalability. These approaches aim at repairing large programs, and therefore are not able to generate complex repairs. Our approach repairs small programs in education and uses multiple correct solutions to find the best repair suggestions, and therefore is able to suggest more complex repairs.

Prophet [27] mines a database of successful patches and uses these patches to repair defects in large, real-world applications. However, it is unclear how this approach would be applicable to our educational setting. SearchRepair [24] mines a body of code for short snippets that it uses for repair. However, SearchRepair has different goals than our work and has not been used or evaluated in introductory education. Angelic Debugging [8] is an approach that identifies *at most one* faulty expression in the program and tries to replace it with a correct *value* (instead of *replacement expression*).

Yi et al. [42] explore different automated program repair (APR) systems in the context of generating feedback in intelligent tutoring systems. They show that using APR out-of-the-box seems infeasible due to the low repair rate, but discuss how these systems can be used to generate *partial repairs*. In contrast, our approach is designed to provide complete repairs. They also conclude that further research is required to understand how to generate the most effective feedback for students from these repairs.

AutoGrader [33] takes as input an incorrect student program, along with a reference solution and a set of potential corrections in the form of expression rewrite rules (both provided by the course instructor), and searches for a set of minimal corrections using program synthesis. In contrast, our approach is completely automatic and can generate more complicated repairs. Refazer [32] learns programs transformations from example code edits made by the students, and then uses these transformations to repair incorrect student submissions. In comparison to our approach, Refazer does not have a cost model, and hence the generated repair is the first one found (instead of the smallest one). Rivers and Koedinger [31] transform programs to a canonical form using semantic-preserving syntax transformations, and then report syntax difference between an incorrect program and the closest correct solution; the paper reports evaluation on loop-less programs. In contrast, our approach uses *dynamic equivalence*, instead of (canonical) syntax equivalence, for better robustness under syntactic variations of semantically equivalent code. Qlose [9] automatically repairs programs in education based on different program distances. The idea to consider different semantic distances is very interesting, however the paper reports only a very small initial evaluation (on 11 programs), and Qlose is only able to generate small, template-based repairs.

8 Future Work

In this section we briefly discuss the limitations of our approach, and possible directions for future work.

Cost function The cost function in our approach compares only the syntactic difference between the original and the replacement expressions (specifically, we use the tree-edit-distance in our implementation). We believe that the cost function could take into account more information; e.g., variable roles [12] or semantic distance [9].

Control-flow The clustering and repair algorithms are restricted to the analysis of programs with the same control-flow. As the case of "no matching control-flow to generate a repair" rarely occurs in our experiments (only 35 cases in the MOOC experiment), we have left the extension of our algorithm to programs with different control-flow for future work. We conjecture that our algorithm could be extended to programs with similar control-flow (e.g., different looping-structure).

Feedback Our tool currently outputs a textual description of the generated repair, very similar to the feedback generated by AutoGrader. We believe that the generated repairs could be used to derive other types of feedback as well. For example, a more abstract feedback with the help of a course instructor: A course instructor could annotate variables in the correct solutions with their descriptions, and when a repair for some variable is required, a matching feedback is shown to a student.

While this paper is focused on the technical problem of finding possible repairs, an interesting orthogonal direction for future work is to consider pedagogical research questions, for example: (1) How much information should be revealed to the student (the line number, an incorrect expression, the whole repair)? (2) Should the use of automated help be penalized? (3) How much do students learn from the automated help?

9 Conclusion

We present novel algorithms for clustering and program repair in introductory programming education. The key idea behind our approach is to use the *existing correct student solutions*, which are available in tens of thousands in large MOOC courses, to *repair incorrect student attempts*. Our evaluation shows that Clara can generate a large number of repairs without any manual intervention, can perform complicated repairs, can be used in an interactive teaching setting, and generates good quality repairs in a large percentage of cases.

References

[1] [n. d.]. Analysis: The exploding demand for computer science education, and why America needs to keep up. http://www.geekwire.com/2014/analysis-examining-computer-science-education-explosion/.
[2] [n. d.]. lpsolve - ILP solver. http://sourceforge.net/projects/lpsolve/.

[3] [n. d.]. MITx MOOC. https://www.edx.org/school/mitx.

[4] [n. d.]. zhang-shasha - tree-edit-distance algorithm implemented in Python. https://github.com/timtadh/zhang-shasha.

[5] Anne Adam and Jean-Pierre Laurent. 1980. LAURA, a system to debug student programs. *Artificial Intelligence* 15, 1âĂŞ2 (1980), 75 – 122. https://doi.org/10.1016/0004-3702(80)90023-5

[6] Andrea Arcuri. 2008. On the Automation of Fixing Software Bugs. In *Companion of the 30th International Conference on Software Engineering (ICSE Companion '08)*. ACM, New York, NY, USA, 1003–1006. https://doi.org/10.1145/1370175.1370223

[7] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, S. F. University, and R. Sebastiani. 2009. Software model checking via large-block encoding. In *2009 Formal Methods in Computer-Aided Design*. 25–32. https://doi.org/10.1109/FMCAD.2009.5351147

[8] Satish Chandra, Emina Torlak, Shaon Barman, and Rastislav Bodik. 2011. Angelic Debugging. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE '11)*. ACM, New York, NY, USA, 121–130. https://doi.org/10.1145/1985793.1985811

[9] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program Repair with Quantitative Objectives. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. 383–401. http://dx.doi.org/10.1007/978-3-319-41540-6_21

[10] Rajdeep Das, Umair Z. Ahmed, Amey Karkare, and Sumit Gulwani. 2016. Prutor: A System for Tutoring CS1 and Collecting Student Programs for Analysis. *CoRR* abs/1608.03828 (2016). http://arxiv.org/abs/1608.03828

[11] V. Debroy and W.E. Wong. 2010. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*. 65–74. https://doi.org/10.1109/ICST.2010.66

[12] Yulia Demyanova, Helmut Veith, and Florian Zuleger. 2013. On the concept of variable roles and its use in software analysis. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. 226–230. http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=6679414

[13] A. Drummond, Y. Lu, S. Chaudhuri, C. Jermaine, J. Warren, and S. Rixner. 2014. Learning to Grade Student Programs in a Massive Open Online Course. In *Data Mining (ICDM), 2014 IEEE International Conference on*. 785–790. https://doi.org/10.1109/ICDM.2014.142

[14] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. A Genetic Programming Approach to Automated Software Repair. In *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation (GECCO '09)*. ACM, New York, NY, USA, 947–954. https://doi.org/10.1145/1569901.1570031

[15] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip Guo, and Robert Miller. 2014. OverCode: Visualizing Variation in Student Solutions to Programming Problems at Scale. In *Proceedings of the Adjunct Publication of the 27th Annual ACM Symposium on User Interface Software and Technology (UIST'14 Adjunct)*. ACM, New York, NY, USA, 129–130. https://doi.org/10.1145/2658779.2658809

[16] Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. 2011. Specification-based Program Repair Using SAT. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems: Part of the Joint European Conferences on Theory and Practice of Software (TACAS'11/ETAPS'11)*. Springer-Verlag, Berlin, Heidelberg, 173–188. http://dl.acm.org/citation.cfm?id=1987389.1987408

[17] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. 2015. The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs. *IEEE Transactions on Software Engineering* 41, 12 (Dec 2015), 1236–1256. https://doi.org/10.1109/TSE.2015.2454513

[18] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2014. Feedback Generation for Performance Problems in Introductory Programming Assignments. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 41–51. https://doi.org/10.1145/2635868.2635912

[19] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. 2018. Automated Clustering and Program Repair for Introductory Programming Assignments. *CoRR* abs/1603.03165 (2018). arXiv:1603.03165 http://arxiv.org/abs/1603.03165

[20] Andrew Head, Elena Glassman, Gustavo Soares, Ryo Suzuki, Lucas Figueredo, Loris D'Antoni, and Björn Hartmann. 2017. Writing Reusable Code Feedback at Scale with Mixed-Initiative Program Synthesis. In *Proceedings of the Fourth (2017) ACM Conference on Learning @ Scale (L@S '17)*. ACM, New York, NY, USA, 89–98. https://doi.org/10.1145/3051457.3051467

[21] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research (Koli Calling '10)*. ACM, New York, NY, USA, 86–93. https://doi.org/10.1145/1930464.1930480

[22] Barbara Jobstmann, Andreas Griesmayer, and Roderick Bloem. 2005. Program Repair As a Game. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*. Springer-Verlag, Berlin, Heidelberg, 226–238. https://doi.org/10.1007/11513988_23

[23] Shalini Kaleeswaran, Anirudh Santhiar, Aditya Kanade, and Sumit Gulwani. 2016. Semi-supervised Verified Feedback Generation. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*. ACM, New York, NY, USA, 739–750. https://doi.org/10.1145/2950290.2950363

[24] Yalin Ke, Kathryn T. Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing Programs with Semantic Code Search (T). In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE) (ASE '15)*. IEEE Computer Society, Washington, DC, USA, 295–306. http://dx.doi.org/10.1109/ASE.2015.60

[25] Dohyeong Kim, Yonghwi Kwon, Peng Liu, I. Luk Kim, David Mitchel Perry, Xiangyu Zhang, and Gustavo Rodriguez-Rivera. 2016. Apex: Automatic Programming Assignment Error Explanation. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 311–327. https://doi.org/10.1145/2983990.2984031

[26] Robert Könighofer and Roderick Bloem. 2011. Automated Error Localization and Correction for Imperative Programs. In *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD '11)*. FMCAD Inc, Austin, TX, 91–100. http://dl.acm.org/citation.cfm?id=2157654.2157671

[27] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. *SIGPLAN Not.* 51, 1 (Jan. 2016), 298–312. http://doi.acm.org/10.1145/2914770.2837617

[28] Ken Masters. 2011. A Brief Guide To Understanding MOOCs. *The Internet Journal of Medical Education* 1, 2 (2011). https://doi.org/10.5580/1f21

[29] Robin Milner. 1971. *An Algebraic Definition of Simulation Between Programs*. Technical Report. Stanford, CA, USA.

[30] Andy Nguyen, Christopher Piech, Jonathan Huang, and Leonidas Guibas. 2014. Codewebs: Scalable Homework Search for Massive Open Online Programming Courses. In *Proceedings of the 23rd International Conference on World Wide Web (WWW '14)*. ACM, New York, NY, USA, 491–502. https://doi.org/10.1145/2566486.2568023

[31] Kelly Rivers and Kenneth R. Koedinger. 2017. Data-Driven Hint Generation in Vast Solution Spaces: a Self-Improving Python Programming Tutor. *International Journal of Artificial Intelligence in Education* 27, 1 (01 Mar 2017), 37–64. https://doi.org/10.1007/s40593-015-0070-z

[32] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Piscataway, NJ, USA, 404–415. https://doi.org/10.1109/ICSE.2017.44

[33] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2491956.2462195

[34] Stefan Staber, Barbara Jobstmann, and Roderick Bloem. 2005. Finding and Fixing Faults. In *Correct Hardware Design and Verification Methods*, Dominique Borrione and Wolfgang Paul (Eds.). Lecture Notes in Computer Science, Vol. 3725. Springer Berlin Heidelberg, 35–49. https://doi.org/10.1007/11560548_6

[35] Michael Striewe and Michael Goedicke. 2011. Using run time traces in automated programming tutoring. In *ITiCSE*. 303–307.

[36] Michael Striewe and Michael Goedicke. 2013. Trace Alignment for Automated Tutoring. In *CAA*.

[37] Ryo Suzuki, Gustavo Soares, Elena Glassman, Andrew Head, Loris D'Antoni, and Björn Hartmann. 2017. Exploring the Design Space of Automatically Synthesized Hints for Introductory Programming Assignments. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems (CHI EA '17)*. ACM, New York, NY, USA, 2951–2958. https://doi.org/10.1145/3027063.3053187

[38] Kuo-Chung Tai. 1979. The Tree-to-Tree Correction Problem. *J. ACM* 26, 3 (July 1979), 422–433. https://doi.org/10.1145/322139.322143

[39] Nikolai Tillmann, Judith Bishop, R. Nigel Horspool, Daniel Perelman, and Tao Xie. 2014. Code Hunt: Searching for Secret Code for Fun. *Proceedings of the International Conference on Software Engineering (Workshops)* (June 2014). http://research.microsoft.com/apps/pubs/default.aspx?id=210651

[40] Nikolai Tillmann, Jonathan De Halleux, Tao Xie, Sumit Gulwani, and Judith Bishop. 2013. Teaching and Learning Programming and Software Engineering via Interactive Gaming. In *Proc. 35th International Conference on Software Engineering (ICSE 2013), Software Engineering Education (SEE)*. http://www.cs.illinois.edu/homes/taoxie/publications/icse13see-pex4fun.pdf

[41] Takeaki Uno. 1997. Algorithms for Enumerating All Perfect, Maximum and Maximal Matchings in Bipartite Graphs. In *ISAAC*. 92–101.

[42] Jooyong Yi, Umair Z. Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 740–751. https://doi.org/10.1145/3106237.3106262

[43] K. Zhang and D. Shasha. 1989. Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems. *SIAM J. Comput.* 18, 6 (Dec. 1989), 1245–1262. https://doi.org/10.1137/0218082