QLOSE: Program Repair with Quantitative Objectives*

Loris D'Antoni[1], Roopsha Samanta[2], and Rishabh Singh[3]

[1] University of Wisconsin-Madison
loris@cs.wisc.edu

[2] IST Austria
roopsha.samanta@ist.ac.at

[3] Microsoft Research
risin@microsoft.com

Abstract. The goal of automatic program repair is to identify a set of syntactic changes that can turn a program that is incorrect with respect to a given specification into a correct one. Existing program repair techniques typically aim to find *any* program that meets the given specification. Such "best-effort" strategies can end up generating a program that is quite different from the original one. Novel techniques have been proposed to compute syntactically minimal program fixes, but the smallest syntactic fix to a program can still significantly alter the original program's behaviour. We propose a new approach to program repair based on *program distances*, which can quantify changes not only to the program syntax but also to the program semantics. We call this the *quantitative program repair problem* where the "optimal" repair is derived using multiple distances. We implement a solution to the quantitative repair problem in a prototype tool called QLOSE (Quantitatively close), using the program synthesizer SKETCH. We evaluate the effectiveness of different distances in obtaining desirable repairs by evaluating QLOSE on programs taken from educational tools such as CodeHunt and edX.

1 Introduction

Recent years have seen the emergence of computer-aided personalized education as a new, important research field. Sophisticated techniques relying on formal methods, programming languages, and program synthesis have been designed to assist teachers in grading and providing feedback for introductory programming assignments [34], automata constructions [1], and geometric constructions [18, 21]. In this paper, we propose a novel program repair framework that enhances the state of the art in automated feedback generation for students in introductory programming courses.

* This research was supported in part by the European Research Council (ERC) under grant 267989 (QUAREM) and by the Austrian Science Fund (FWF) under grants S11402-N23 (RiSE) and Z211-N23 (Wittgenstein Award).

The goal of automatic program repair is to identify a set of syntactic changes that can turn a program that is incorrect with respect to a given specification into a correct one. In the context of automated feedback generation, repairing a program corresponds to finding a "fix" to the student's incorrect solution. The specification can be as simple as a set of test cases.

Existing program repair techniques are typically "best-effort" and aim to find *any* repair that meets the given specification. Such techniques can end up generating a program that is quite different from the original one. Although this may be acceptable in some settings, in the context of education, the goal should be to slowly guide the students towards a correct solution. In particular, if a student solution is close to a correct one, a teacher wouldn't point the student to a completely different program, but would rather show how the student solution can be corrected with small changes. Advanced program repair techniques address this problem by computing syntactically minimal program repairs [34, 33, 28].

In this paper, we argue that even the smallest syntactic fix to a program can significantly alter its behaviour. We propose a new approach to program repair based on *program distances*, which quantify changes not only to the program syntax but also to the program semantics. While syntactic distances capture the number of edits to the program, semantic distances quantify the number of changes to the "behaviour" of a program with respect to a given set of tests. We formalize the *quantitative program repair problem* in which the "optimal" repair is defined as a correct program that minimizes an objective function over multiple program distances. Although our framework is general, we present two types of syntactic distances, three types of semantic distances, and propose a solution to the quantitative program repair problem with respect to these distances.

We have implemented our techniques in a prototype tool called QLOSE (Quantitatively close), which is built on top of the SKETCH synthesis system [36]. In QLOSE, we encode the functional correctness property of the student solution with respect to a set of tests as a *hard* constraint and the syntactic and semantic distances with respect to the original solution as *soft* constraints. The repair generated by SKETCH maximizes the number of soft constraints that can be satisfied, while satisfying the hard constraints. We evaluate QLOSE on 11 representative benchmark programs taken from student submissions to the Microsoft CodeHunt platform [39] and the Introduction to Programming course on the edX platform. Our preliminary results show that encoding quantitative program repair using syntactic and semantic distances is practically feasible for small student solutions and leads to more desirable repairs.

Contributions. This paper makes the following key contributions.

- We define new notions of syntactic and semantic distances between programs with respect to a given set of tests and use these notions to formalize the quantitative program repair problem (§ 3, § 4).
- We encode the quantitative repair problem in a prototype tool called QLOSE, which is built on top of the SKETCH synthesis system (§ 5).
- We evaluate QLOSE and the strengths of the different distances on 11 representative student submissions taken from education platforms (§ 6).

Test set

Input			Output	
s	c	k	expected	actual
ab?	?	2	true	false
aba?gc	?	5	true	true
?aba	?	3	true	true

```
FindCBuggy(str s, ch c, int k){
  for(int j=0;j<k;j++)
    if(s[j]==c)
        return true;
  return false;
}
```

(a)

```
FindCBadFix(str s, ch c, int k){
  for(int j=0;j<k;j++)
    if(c=='?') //new condition
        return true;
  return false;
}
```

(b)

```
FindCGoodFix(str s, ch c, int k){
  for(int j=0;j<=k;j++) // new guard
    if(s[j]==c)
        return true;
  return false;
}
```

(c)

Fig. 1: A buggy program (a) with two possible syntactic repairs (b) and (c).

2 Motivating Example

We use the example in Fig. 1 to show that semantic distances can sometimes yield more intuitive program repairs than syntactic distances.[4] Fig. 1 contains a set of tests that is representative of the intended semantics of a desired program. Given this test set, the student has come up with the program FindCBuggy in Fig. 1(a) which fails the first test but passes the other ones. The desired program is one that given a string s, a character c, and an integer k outputs whether the character c appears in s at some position $j \leq k$. Besides a small imprecision, the student solution captures the intended algorithm in the sense that successful executions of the program are not far from those in the correct algorithm.

Limitation of syntactic distances. To give feedback to the student, one can try fixing the student solution using existing program repair techniques that minimize the number of *syntactic changes* to an incorrect program. Techniques like the ones presented in [28, 34] would return one of the two programs at the bottom of Fig. 1. Both of these programs differ from the one in Fig. 1(a) by exactly one expression. However, one of the repaired programs is, in some sense, more “disruptive” than the other. In particular, although the program in Fig. 1(b) simply changes the guard of the if-statement, its executions on previously correct tests are now very different: on all tests the loop is now executed only once! On the other hand, for the program in Fig. 1(c), the executions on correct tests are the same as for the original program. Syntactic program distances cannot distinguish between these two candidate repairs and are inadequate for this example.

Semantic distances. One can capture the intuition that the program in Fig. 1(c) is a better repair for the student solution than the program in Fig. 1(b) by examining the execution of these programs on successful tests. For example, if

[4] This example is a slight variation of the one appearing in Fig. 3 of [28].

we were to track the locations (lines of code) traversed by the three programs on the second input test with $s =$ `aba?gc` we would get the following sequences of locations: (a) $2, 3, 2, 3, 2, 3, 2, 3, 4$; (b) $2, 3, 4$; and (c) $2, 3, 2, 3, 2, 3, 2, 3, 4$. These sequences highlight that the program in Fig. 1(c) is *semantically closer* to the student solution.

A similar argument can be made for repairing using only a semantic distance as the repaired program may be syntactically very far from the original one. In summary, in order to repair programs in a meaningful way, it is often necessary to take into account multiple quantitative objectives such as the number of syntactic edits and the *distance* between program behaviours.

3 Program Repair

In this section, we formalize programs, correctness specifications, and permissible program edits. We then use these notions to define the program repair problem.

3.1 Programs

We fix a simple imperative programming language, in which a program P consists of a function definition $f(i_1, \ldots, i_q) : o$, a set of program variables V, and a sequence of labeled statements $\sigma = s_1 \ldots s_n$. A statement is one of the following: `skip`, `return`, assignment, conditional or loop statement.[5] Each statement in σ is labeled with a unique location identifier from the set $L = \{\ell_0, \ell_1, \ldots, \ell_p, exit\}$. The function f has a designated set of input variables $I = \{i_1, \ldots, i_q\}$ and a designated output variable o. The program statements are allowed to use an auxiliary set of variables $V = \{v_1, \ldots, v_r\}$. We assume a universe $\mathcal{U}$ of values. We also assume that all variables are associated with a given type and and are only assigned values from $\mathcal{U}$ with the proper types.

We now define the semantics of our programs. The semantics of program statements is standard. Without loss of generality, we assume execution of a `return` statement assigns a value to the output variable and transfers control to a designated location *exit*.

A program configuration η is a pair (ℓ, ν) where $\ell \in L$ is a location and $\nu : I \cup \{o\} \cup V \mapsto \mathcal{U} \cup \{nd\}$ is a valuation function that assigns values to all variables. The element nd indicates that a variable has not been assigned a value yet or is out of scope. We write $(\ell, \nu) \rightarrow (\ell', \nu')$ if execution of the statement at location ℓ under variable valuation ν transfers control to location ℓ' with variable valuation ν'.

The execution $\pi(\nu)$ of program P on a valuation ν is a sequence of configurations $\eta_0, \eta_1, \ldots$, where $\eta_0 = (\ell_0, \nu)$ and for each h, $\eta_h \rightarrow \eta_{h+1}$. An execution terminates once the location *exit* is reached.

Here we are only interested in executions for which the initial valuation ν is such that for every input variable $x \in I$, $\nu(x) \neq nd$, and for every non-input

[5] Our implementation supports a richer subset of the Python language including lists, strings, and function calls.

variable $y \in V \cup \{o\}$, $\nu(y) = nd$. Given a partial valuation $\nu_I : I \mapsto \mathcal{U}$ assigning values to the input variables, let ν_I^+ be the valuation such that for every input variable $x \in I$, $\nu_I^+(x) = \nu_I(x)$, and for every other variable $y \notin I$, $\nu_I^+(y) = nd$. We denote by $[\![P]\!] : (I \mapsto \mathcal{U}) \mapsto \mathcal{U}$ the partial function computed by a program P, and define it as $[\![P]\!](\nu_I) = res$ iff $\pi(\nu_I^+)$ terminates with output valuation ν' and $\nu'(o) = res$.

Example 1. Consider the program FINDCBUGGY in Fig. 1(a). The input variables I are $\{s, c, k\}$ and the designated output variable is o. The set of program variables is the singleton $\{j\}$. The execution of FINDCBUGGY on ν such that $\nu_I(s) = \mathsf{ab?}, \nu_I(c) = \mathsf{?}, \nu_I(k) = 2$ is illustrated in the following table:

	η_0	η_1	η_2	η_3	η_4	η_5	η_6
loc	2	3	2	3	2	5	*exit*
s	ab?	ab?	ab?	ab?	ab?	ab?	ab?
c	?	?	?	?	?	?	?
k	2	2	2	2	2	2	2
j	*nd*	0	0	1	1	2	*nd*
o	*nd*	*nd*	*nd*	*nd*	*nd*	*nd*	false

We thus have $[\![$FINDCBUGGY$]\!]$(ab?, ?, 2) = false.

3.2 Test Sets as Specifications

A test t is a pair (ν_I, res), where $\nu_I : I \mapsto \mathcal{U}$ is a valuation over the input variables, and $res \in \mathcal{U}$ is the expected output value. A program P satisfies a test t if $[\![P]\!](\nu_I) = res$. A program P satisfies a test set T if it satisfies all the tests $t \in T$.

We use $\dot{\pi}(t)$, read as "execution of a program on a test t", to refer to $\pi(\nu_I^+)$.

Example 2. Consider the test set and the program FINDCBUGGY in Fig. 1. Clearly, FINDCBUGGY does not satisfy the test set. In particular, on the first test from Ex. 1 we have $[\![$FINDCBUGGY$]\!]$(ab?, ?, 2) $\neq$ true.

3.3 The Program Repair Problem

In our repair model we permit program expressions to be changed, but not program statements. For example, we permit replacement of loop guards and right-hand sides of assignments and disallow replacement of an assignment with a `return` statement. Formally, a permissible program edit applied to a labeled statement ℓ : `stmt` in program P is any modification of `stmt` that replaces an expression in `stmt` with another expression over the same domain, and leaves the label ℓ unchanged.

Given program P and a subset of locations LOC $\subseteq L$ of P, let $\mathcal{R}_{\text{LOC}}(P)$ be the set of all programs that can be obtained by applying permissible program edits to labeled statements with labels in LOC. The following proposition holds trivially.

Proposition 1. *Given programs P, P', with locations L, L', the following statements are equivalent:*

(i) *there exists unique* $\textsc{loc} \subseteq L$ *such that* $P' \in \mathcal{R}_{\textsc{loc}}(P)$

(ii) *there exists unique* $\textsc{loc}' \subseteq L'$ *such that* $P \in \mathcal{R}_{\textsc{loc}'}(P')$

(iii) $L = L'$ *and there exists unique* $\textsc{loc} \subseteq L$ *such that* $P' \in \mathcal{R}_{\textsc{loc}}(P)$ *and* $P \in \mathcal{R}_{\textsc{loc}}(P')$.

Example 3. In Fig. 1, FindCBadFix $\in \mathcal{R}_{\{3\}}$(FindCBuggy) as FindCBadFix replaces the guard $s[j] == c$ in location 3 of FindCBuggy with the guard $c ==?$. Similarly, FindCGoodFix $\in \mathcal{R}_{\{2\}}$(FindCBuggy) as FindCGoodFix replaces the loop guard $j < k$ in location 2 of FindCBuggy with the guard $j \leq k$[6].

Given a program P and a test set T such that P does not satisfy T, the goal of program repair is to compute P' such that: (1) P' satisfies T, and (2) there exists $\textsc{loc} \subseteq L$ such that $P' \in \mathcal{R}_{\textsc{loc}}(P)$.

Example 4. Consider the programs in Fig. 1. The programs FindCBadFix and FindCGoodFix are possible repairs of the program FindCBuggy with respect to the test set shown in the figure. They are both correct on the test set and, from Ex. 3, FindCBadFix $\in \mathcal{R}_{\{3\}}$(FindCBuggy) and FindCGoodFix $\in \mathcal{R}_{\{2\}}$(FindCBuggy).

4 Quantitative Program Repair

In this section, we define program distances and the quantitative program repair problem. Given two programs P, P' and a test set T, a program distance[7] is a function over P, P' and T that quantifies how *close* are P and P' w.r.t. T. We classify program distances as *syntactic* and *semantic* distances. A syntactic program distance simply tracks the syntactic change between P and P', independent of the test set T. Hence, a syntactic program distance is a function over P and P'. A semantic program distance tracks the semantic differences between P and P' with respect to executions on the test set T. In particular, a semantic program distance tracks the differences in the executions of P and P' on all tests t such that both P and P' satisfy t. In what follows, we define several syntactic and semantic distances. One could easily define more sophisticated distances and we invite the reader to do so. The following distances sufficed for our "proof of concept" experiments with quantitative program repair.

[6] In our implementation, each location is associated with a single expression. To keep our presentation simple, we associate all 3 expressions in the `for` loop in FindCBuggy with location 2, instead of mapping each expression to a different location.

[7] Our program distances are not necessarily *distance metrics*. In particular, some of them are not symmetric in P and P'.

4.1 Syntactic Distances

A syntactic distance between programs P and P' is defined modulo an *expression distance* ε. An expression distance tracks the syntactic difference between two expressions. In this work, we use two simple expression distances, defined below.

Boolean expression distance, $\varepsilon_{\text{BOOL}}$, is a Boolean-valued distance that simply tracks if two expressions are equal or not:

$$\varepsilon_{\text{BOOL}}(expr, expr') = \begin{cases} 0 & \text{if } expr = expr' \\ 1 & \text{otherwise.} \end{cases}$$

Expression-size distance, $\varepsilon_{\text{SIZE}}$, tracks the size of the repaired expression:

$$\varepsilon_{\text{SIZE}}(expr, expr') = \begin{cases} 0 & \text{if } expr = expr' \\ size(expr') & \text{otherwise,} \end{cases}$$

where $size(expr')$ can be defined in different ways. For example, $size(expr')$ could be the total number of symbols and operators in $expr'$. In Section 5, we present the definition of $size(expr')$ used in our implementation. Note that $\varepsilon_{\text{SIZE}}(expr, expr')$ is not a symmetric function.

A syntactic program distance between programs P and P' is finite only if P and P' can be obtained from each other by applying a set of permissible program edits. Given an expression distance ε, a syntactic program distance accumulates the expression distance across all expression changes between P and P'. Formally:

$$d^{\varepsilon}_{syn}(P, P') = \begin{cases} \infty & \text{if } \forall \text{LOC} : P' \notin \mathcal{R}_{\text{LOC}}(P) \\ \sum\limits_{\ell \in \text{LOC} : P' \in \mathcal{R}_{\text{LOC}}(P)} \varepsilon(expr_\ell, expr'_\ell) & \text{otherwise.} \end{cases}$$

Note that Prop. 1 ensures the uniqueness of LOC in the second case. Here, $expr_\ell, expr'_\ell$ denote expressions in ℓ-labeled statements of P, P', respectively. Thus, if $\varepsilon = \varepsilon^{\text{BOOL}}$, $d^{\varepsilon}_{syn}(P, P')$ equals the number of permissible program edits required to transform P to P'. Similarly, if $\varepsilon = \varepsilon_{\text{SIZE}}$, $d^{\varepsilon}_{syn}(P, P')$ equals the total size of all new expressions in P'.

Example 5. Consider the programs in Figure 1. For $\varepsilon = \varepsilon_{\text{BOOL}}$, one can see that $d^{\varepsilon}_{syn}(\text{FindCBuggy}, \text{FindCBadFix})$ and $d^{\varepsilon}_{syn}(\text{FindCBuggy}, \text{FindCGoodFix})$ both equal 1, as there is exactly one permissible program edit in each case. For $\varepsilon = \varepsilon_{\text{SIZE}}$, if expression size is given by the total number of symbols and operators, $d^{\varepsilon}_{syn}(\text{FindCBuggy}, \text{FindCBadFix})$ and $d^{\varepsilon}_{syn}(\text{FindCBuggy}, \text{FindCGoodFix})$ both equal 3. Neither syntactic distance can distinguish between FindCBadFix and FindCGoodFix.

4.2 Semantic Distances

The semantic distance between programs P and P' with respect to a test set T is defined modulo an *execution distance* ζ. An execution distance tracks the differences between two executions. In this paper, we consider three types of execution distances, defined on terminating executions.

Let $T_{sat} \subseteq T$ consist of all tests t such that P and P' both satisfy t. Given a test t in T_{sat}, let $\dot{\pi}(t)$, $\dot{\pi}'(t)$ denote executions of P, P', respectively on t. In what follows, we fix $\dot{\pi}(t) = \eta_0, \eta_1, \ldots, \eta_M$ and $\dot{\pi}'(t) = \eta'_0$, η'_1, ..., η'_K. Recall that a configuration η_h is a tuple of the form (ℓ_h, ν_h).

Our execution distances essentially compute the Hamming distance between two executions, using different abstractions of configurations. For executions of equal lengths, this distance equals the minimum number of configuration substitutions required to transform one execution into another. For executions of differing lengths, this distance additionally includes the difference in the execution lengths. All three execution distances can be defined as follows:

$$\zeta(\dot{\pi}(t), \dot{\pi}'(t)) = \begin{cases} |M - K| + \sum_{h=0}^{min(M,K)} \mathtt{diff}(\eta_h, \eta'_h) & \text{if } M, K < \infty \\ \infty & \text{otherwise,} \end{cases}$$

where the definition of $\mathtt{diff}(\eta_h, \eta'_h)$ varies for each execution distance.

Concrete execution distance, ζ_{CONC}*,* compares both locations and variable values in two executions: $\mathtt{diff}_{conc}(\eta_h, \eta'_h) = 0$ if $\eta_h = \eta'_h$ and 1 otherwise.

Value execution distance, ζ_{VAL}*,* only compares the variable values in two executions: $\mathtt{diff}_{val}(\eta_h, \eta'_h) = 0$ if $\nu_h = \nu'_h$ and 1 otherwise.

Location execution distance, ζ_{LOCS}*,* only compares the locations in two executions: $\mathtt{diff}_{loc}(\eta_h, \eta'_h) = 0$ if $\ell_h = \ell'_h$ and 1 otherwise.

A semantic program distance between programs P, P' w.r.t. test set T is finite only if P and P' can be obtained from each other by applying a set of permissible program edits and T_{sat} is not empty. Given an execution distance ζ and the set T_{sat}, a semantic program distance accumulates the execution distance between executions of P and P' on tests in T_{sat}. Formally:

$$d^{\zeta}_{sem}(P, P', T) = \begin{cases} \infty & \text{if } \forall \text{LOC} : P' \notin \mathcal{R}_{\text{LOC}}(P) \text{ or } T_{sat} \text{ is empty} \\ \sum\limits_{t \in T_{sat}} \zeta(\dot{\pi}(t), \dot{\pi}'(t)) & \text{otherwise.} \end{cases}$$

Example 6. The executions of FindCBuggy and FindCBadFix from Figure 1 on ν such that $\nu_I(s) = \mathtt{aba?gc}, \nu_I(c) = \mathtt{?}, \nu_I(k) = 5$ are shown in Fig. 2. The last 3 rows of the table show $\mathtt{diff}(\eta_h, \eta'_h)$ for $h = 0, 1, 2, 3$. Note that $\mathtt{diff}_{val}$ doesn't distinguish between η_2 and η'_2 as these configurations share the same variable

	step	0	1	2	3	4	5	6	7	8	9
FindCBuggy	loc	2	3	2	3	2	3	2	3	4	exit
	j	nd	0	0	1	1	2	2	3	3	exit
	o	nd	nd	nd	nd	nd	nd	nd	nd	nd	true
FindCBadFix	loc	2	3	4	exit						
	j	nd	0	0	exit						
	o	nd	nd	nd	true						
`diff`	$\texttt{diff}_{conc}$	0	0	1	1						
	$\texttt{diff}_{val}$	0	0	0	1						
	$\texttt{diff}_{loc}$	0	0	1	1						

Fig. 2: Semantic distances between executions. We do not show the input variables s, c, and k as their values are never modified.

values. The difference in lengths of the given two executions is 6. Thus, for these executions, $\zeta_{\text{CONC}} = \zeta_{\text{LOCS}} = 6 + 2 = 8$, and $\zeta_{\text{VAL}} = 6 + 1 = 7$.

The execution of FindCGoodFix on the same ν with $\nu_I(s) = \texttt{aba?gc}, \nu_I(c) = ?, \nu_I(k) = 5$ is exactly the same as the execution of FindCBuggy shown in Fig. 2. Hence, $\zeta_{\text{CONC}} = \zeta_{\text{LOCS}} = \zeta_{\text{VAL}} = 0$ for the executions of FindCBuggy and FindCGoodFix. Our semantic program distances can distinguish between FindCBadFix and FindCGoodFix.

4.3 The Quantitative Program Repair Problem

Given a program P and a test set T such that P does not satisfy T, syntactic distance functions $d^1_{syn}, \dots, d^x_{syn}$, semantic distance functions $d^1_{sem}, \dots, d^y_{sem}$, and *objective functions* $f_1, \dots, f_z$ over $d^1_{syn}, \dots, d^x_{syn}, d^1_{sem}, \dots, d^y_{sem}$, the goal of quantitative program repair is to compute P' such that:

(1) P' satisfies T,
(2) there exists $\text{LOC} \subseteq L$ such that $P' \in \mathcal{R}_{\text{LOC}}(P)$, and
(3) $$P' = \operatorname*{arg\,min}_{\exists \widehat{\text{LOC}} \subseteq L : \widehat{P} \in \mathcal{R}_{\widehat{\text{LOC}}}(P)} \underset{1 \le i \le z}{\text{AGGREGATE}} \{ f_i(d^1_{syn}(P, \widehat{P}), \dots, d^y_{sem}(P, \widehat{P}, T)) \}.$$

Here AGGREGATE allows multiple objective functions to be combined. For example, AGGREGATE could enforce Pareto optimality.

Example 7. Consider the programs in Fig. 1. In Ex. 4, we showed that both FindCBadFix and FindCGoodFix satisfy conditions (1) and (2) of the quantitative program repair problem for program FindCBuggy and the test set shown in Fig. 1.

In Ex. 5, we showed that both $d^{\varepsilon}_{syn}(\text{FindCBuggy, FindCBadFix})$ and $d^{\varepsilon}_{syn}(\text{FindCBuggy}, \text{FindCGoodFix})$ equal 1 for $\varepsilon = \varepsilon_{\text{BOOL}}$. For $\varepsilon = \varepsilon_{\text{SIZE}}$, $d^{\varepsilon}_{syn}(\text{FindCBuggy}, \text{FindCBadFix})$ and $d^{\varepsilon}_{syn}(\text{FindCBuggy}, \text{FindCGoodFix})$ both equal 3.

The set T_{sat} consists of the last two tests in the test set T in Fig. 1. Let the test with $s =$ aba?gc be denoted t_1 and the test with $s =$?aba be denoted t_2. Let $\dot{\pi}(t_1)$ and $\dot{\pi}(t_2)$ denote the executions of FindCBuggy on t_1 and t_2, respectively. Let $\dot{\pi}'(t_1)$, $\dot{\pi}'(t_2)$ and $\dot{\pi}''(t_1)$, $\dot{\pi}''(t_2)$ denote the executions of FindCBadFix and FindCGoodFix on t_1, t_2, respectively.

We have seen in Ex. 6 that $\zeta_{\textsc{conc}}(\dot{\pi}(t_1), \dot{\pi}'(t_1)) = 8$ and $\zeta_{\textsc{conc}}(\dot{\pi}(t_1), \dot{\pi}''(t_1)) = 0$. It's not hard to see that $\zeta_{\textsc{conc}}(\dot{\pi}(t_2), \dot{\pi}'(t_2)) = 0$ and $\zeta_{\textsc{conc}}(\dot{\pi}(t_2), \dot{\pi}''(t_2)) = 0$. Thus, we can compute $d_{sem}^{\zeta_{\textsc{conc}}}(\textsc{FindCBuggy}, \textsc{FindCBadFix}, T) = 8$ and $d_{sem}^{\zeta_{\textsc{conc}}}(\textsc{FindCBuggy}, \textsc{FindCGoodFix}, T) = 0$.

If we choose $d_{syn}^{\varepsilon_{\textsc{bool}}}$, $d_{syn}^{\varepsilon_{\textsc{size}}}$ as our syntactic distances, $d_{sem}^{\zeta_{\textsc{conc}}}$ as our semantic distance and our objective function f to simply be the sum of $d_{syn}^{\varepsilon_{\textsc{bool}}}$, $d_{syn}^{\varepsilon_{\textsc{size}}}$ and $d_{sem}^{\zeta_{\textsc{conc}}}$, the value of f is 4 for FindCGoodFix and is 12 for FindCBadFix. Hence, this instance of the quantitative program repair problem will prefer the program FindCGoodFix as a repair candidate.

5 Quantitative Program Repair using Sketch

In this section, we describe the formulation of the quantitative program repair problem as an instance of the MAX-SMT problem. We encode the program semantics using a symbolic Boolean encoding and specify the functional correctness of the program w.r.t the given test set T as a *hard* constraint. The syntactic and semantic distances are encoded using *soft* constraints. The repair generated by the MAX-SMT solver maximizes the number of soft constraints that can be satisfied while ensuring the satisfaction of the hard constraints. We perform a syntax-directed translation from the source imperative language to Sketch [36], and use the minimization algorithm in Sketch to solve the MAX-SMT constraints. Instead of using a general MAX-SMT solver, we use the Sketch solver because of the ease in translation of the buggy programs into constraints. The Sketch solver allows for optimization constraints similar to MAX-SMT, but uses several algorithmic optimizations before encoding the problem into low-level SMT constraints. We now describe the key ideas in the formulation and translation of the quantitative program repair problem using the Sketch system.

5.1 Background on Sketch

Sketch is a synthesis system for writing partial programs (with holes) together with some high-level specifications of the programs. The synthesis algorithm fills the holes automatically using a constraint-based, counterexample-guided inductive synthesis (CEGIS) algorithm such that the completed program satisfies the given specifications. For example, consider the Sketch program shown in Fig. 3(a). One possible completion synthesized by the Sketch system is shown in Fig. 3(b). The hole expressions ?? can take any constant integer value, and they can further be composed to construct more complex unknown expressions.

```
harness int triple(int x){
 int y = ?? * x;
 if(x==10) assert y == 30;
 return y;
}
```

(a)

```
harness int triple(int x){
 int y = 3 * x;
 if(x==10) assert y == 30;
 return y;
}
```

(b)

Fig. 3: (a) A simple SKETCH program and (b) a possible completion.

```
bit FindCBuggySketch(str s, ch c, int k){
  for(int j= f1(s,c,k); f2(j,s,c,k);
       j=f3(j,s,c,k))
     if(f4(j,s,c,k))
        return f5(j,s,c,k);
  return f6(j,s,c,k);
}
```

(a)

```
int f1(str s, ch c, int k){
   if (b_f1 == 0)
      return 0;
   else
      return ??1,1*s[??] +
           ??1,2 * c+??1,3*k + ??1,4;
}
```

(b)

Fig. 4: The SKETCH translation for the FINDCBUGGY program from Fig. 1(a).

5.2 Space of Expression Edits

For our quantitative program repair encoding, we restrict the class of expressions that can potentially be modified by the solver to (i) the set of conditional expressions and (ii) the right hand side expressions of assignment statements. Furthermore, to restrict the space of possible repairs, we use an expression template corresponding to a linear combination of constants and all program variables in scope at the program location. In SKETCH, the modifiable expressions are replaced by functions that either allow for returning the original unmodified expression in the program or some instantiation of the expression template.

For example, the SKETCH translation for the buggy program in Fig. 1(a) is shown in Fig. 4(a). The conditional expressions and the right hand side expressions of the assignment statements are translated to *change functions* f_i. An example change function f_1 is shown in Fig. 4(b). Each change function f_i is associated with a Boolean variable b_{f_i} that indicates if the original expression is selected ($b_{f_i} = 0$) or some new expression is selected for the completion of the function f_i ($b_{f_i} = 1$). Each set of possible new expressions is represented as a linear combination of program variables of appropriate types where the coefficients of the variables are denoted using unknown values $??_{i,j}$. For expressions involving strings, the change function restricts the edit expression to consist of only 1 character from that string. The characters are then interpreted as integers in SKETCH.

5.3 Encoding Distances

We now describe how the syntactic and semantic distances are encoded as constraints in the SKETCH system.

```
$m_2$ = 0; $m_3$ = 0; $m_4$ = 0;
for(int j= f1(s,c,k); f2(j,s,c,k); j=f3(j,s,c,k)){
   $S_2^t$[$m_2$++] = [j,s,c,k];
   if(f4(j,s,c,k)){
      $S_3^t$[$m_3$++] = [j,s,c,k]; return f5(j,s,c,k);}
}
$S_4^t$[$m_4$++] = [j,s,c,k];
return f6(j,s,c,k);
```

(a)

```
int semDistance($S, S^{\mathsf{orig}}, T_{sat}$){
   $d_\ell$ = 0;
   foreach(test $t \in T_{sat}$)
     foreach(loc $\ell \in |S|$)
        $d_\ell += \phi(S^t_\ell, S^{\mathsf{orig,t}}_\ell)$;
   return $d_\ell$;
   }
```

(b)

Fig. 5: Encoding semantic distance in Sketch.

Syntactic distances. We encode our syntactic distances modulo our two expression distances in Sketch as follows.

- *Boolean expression distance:* The syntactic distance d^{ε}_{syn} for $\varepsilon = \varepsilon_{\textsc{bool}}$ computes the number of expression changes that are performed by the solver and is computed as $\Sigma_i b_{f_i}$. The Boolean variable b_{f_i} is set to 0 if the expression corresponding to function f_i remains unchanged in the final solution and is set to 1 otherwise.
- *Expression-size distance:* The syntactic distance d^{ε}_{syn} for $\varepsilon = \varepsilon_{\textsc{size}}$ computes the total size of modified expressions, where the size of a modified linear arithmetic expression corresponding to f_i is computed as the sum of all of its coefficients $|??_{i,j}|$. Thus, $d^{\varepsilon_{\textsc{size}}}_{syn}$ is defined as $\Sigma_i \Sigma_j \, |??_{i,j}|$.

Semantic distances. We encode our semantic distance modulo the *concrete execution distance*. The Sketch translation is instrumented to capture program states at different program locations as shown in Fig. 5(a), where $S^t_\ell[j]$ denotes the program state for j^{th} loop iteration at program location ℓ for a test case t. The concrete execution distance $\zeta_{\textsc{conc}}$ between the original program and the modified program on a test case t in T_{sat} is computed as $\Sigma_{\ell,j}\phi(S^t_\ell[j], S^{\mathsf{orig,t}}_\ell[j])$, where the function ϕ counts the number of variables that do not have equal values across two states $S^t_\ell[j]$ and $S^{\mathsf{orig,t}}_\ell[j]$, as shown in Fig. 5(b). Our encoding enforces a bound on the length of program executions by unrolling loops a fixed number of times.

Quantitative Objective. The final quantitative objective in the Sketch translation is encoded as the following constraint:

$$\mathtt{assert}\ d^{\varepsilon_{\textsc{bool}}}_{syn} < N \ \wedge\ \mathtt{minimize}\ (d^{\varepsilon_{\textsc{size}}}_{syn} + d^{\zeta_{\textsc{conc}}}_{sem})$$

We use a linear search to first find the minimum number of expression changes N that are needed to repair the buggy program using a linear iterative search. After computing the value n, we then add the minimization constraint to find a repair with minimum semantic distance $d^{\zeta_{\textsc{conc}}}_{sem}$ and simpler expression modifications $d^{\varepsilon_{\textsc{size}}}_{syn}$. The Sketch solver uses an incremental search methodology to compute the repair that corresponds to the minimum objective function value [34].

The hard constraints specifying functional correctness w.r.t a test set T is encoded in a standard way using `assert` statements in SKETCH. If we refer back to the definition of quantitative program repair in Section 4.3, the resulting repaired program is

$$P' = \underset{\exists \widehat{\text{LOC}} \subseteq L : \widehat{P} \in \mathcal{R}_{\widehat{\text{LOC}}}(P)}{\arg\min} \langle d_{syn}^{\varepsilon_{\text{BOOL}}}(P, \widehat{P}), d_{syn}^{\varepsilon_{\text{SIZE}}}(P, \widehat{P}) + d_{sem}^{\zeta_{\text{CONC}}}(P, \widehat{P}) \rangle .$$

In this case, the aggregation operator is the one that first minimizes the left element of the pair and then the right one.

6 Evaluation

We implemented a prototype tool QLOSE that given a (simplified) C# program, a set of test cases, and the desired types of distances, constructs a SKETCH program with the corresponding constraints to encode the quantitative program repair problem. We evaluated QLOSE on 11 representative benchmark programs using the distances presented in Sec. 5. Our preliminary results suggest that QLOSE is practically feasible for small student solutions and generates more desirable repairs while using a combination of syntactic and semantic distances.[8]

6.1 Benchmarks

Our benchmark set consists of 11 representative buggy programs taken from student submissions to introductory programming courses and recent program repair literature. The `LargestGap` problem is taken from the Microsoft Code-Hunt platform [39] and asks students to write a program to compute the largest difference amongst any two values in a given input array of integers. The `FindC` program is the same as FINDCBUGGY in Fig. 1. The `tcas-semfix` benchmark is taken from the SEMFIX [29] system and corresponds to a code excerpt from the `Tcas` benchmark[9]. The `max3` problem asks students to compute the maximum of 3 integers. The `iterPower`, `epoly`, and `multIA` problems are taken from the Introduction to Programming course taught on the edX platform. The `iterPower` problem asks students to write an iterative program that, given two integers m and n, computes the value m^n. The `epoly` problem evaluates a polynomial (defined using an array of integer coefficients) on an integer value, and the `multIA` problem requires students to write a program to compute multiplication of two integers using successive additions.

The number of lines of code (LOC), the number of variables (|Vars|), and the number of test input-output pairs ($|T_{sat}|$) for each benchmark problem[10]

[8] The experiments were performed on a 40-core 2.4GHz Intel Xeon CPU with 100GB RAM, with a timeout of 20 minutes. Although this is powerful hardware, we point out that SKETCH only uses a single core and in our experiments the maximum memory usage was less than 500MB RAM.

[9] http://www.irit.fr/wiki/doku.php?id=wtc:benchmarks:tcas

[10] The benchmark problems and the translated SKETCH files are available at: bit.ly/cav16-qlose

Problem	LOC	|Vars|	$\|T_{sat}\|$	Syntactic		Semantic		Syntactic + Semantic	
FindC	4	4	4	1.5s	✗	2.5s	✓	2.2s	✓
LargestGap-1	10	4	4	9.8s	✓	184.9s	✗	13.4s	✓
LargestGap-2	7	4	4	6.9s	✗	TO	-	18.2s	✓
LargestGap-3	8	4	4	7.3s	✓	15.7s	✓	14.4s	✓
tcas-semfix	10	4	5	12.6s	✓	27.8s	✓	18.4s	✓
max3	5	3	4	1.1s	✓	1.7s	✗	1.9s	✓
iterPower-1	7	3	4	2.3s	✗	10.3s	✓	3.4s	✓
iterPower-2	7	3	4	2.1s	✓	15.2s	✗	2.7s	✓
ePoly-1	8	4	3	1.8s	✗	3.6s	✓	2.5s	✓
ePoly-2	10	4	3	2.4s	✓	4.6s	✓	2.8s	✓
multIA	5	4	4	1.8s	✗	21.5s	✗	2.4s	✓

Fig. 6: Solving times and the desiredness of the generated repairs for different distances. TO denotes that the solver timed out ($>$ 20 minutes), The symbol ✓ (resp. ✗) denotes that the generated repair was (resp. wasn't) the desired one.

is shown in Fig. 6. The number of lines in the benchmarks varied from 4 to 10 lines, whereas the number of variables and the number of test cases varied from 3 to 5. For the CodeHunt benchmarks, we reused the test input-output pairs automatically generated by the CodeHunt engine. For the `tcas-semfix` benchmark, we use the tests from the SemFix paper [29]. For the benchmarks obtained from the edX class, we manually selected the relevant test cases that exposed different corner case behaviors.

6.2 Desired Repairs

The experimental results obtained by running Qlose on different benchmarks using different distances are shown in Fig. 6. We manually inspected the repairs generated using different distance metrics and classified them into desired (✓) or not (✗). For performing this classification, we did not inspect the reference code for the problem, but instead inspected the original buggy program and manually inferred the algorithm the student (or programmer) likely intended to implement. We then checked whether the repaired program matched the intended algorithm.

We can observe that using only syntactic or semantic distance sometimes leads to undesired repairs whereas combining the two distances always leads to the desired fixes in our benchmark set. For example, for the `LargestGap-2` program shown in Fig. 7(a), the syntactic distance encoding causes the solver to come up with a fix that sets the loop initialization variable i to 0 instead of 1. Although, this repair is correct on the test cases, it is less desirable than the repair that assigns $a[0]$ to the low variable l, which corresponds to the solution that student had in mind. Qlose generates this repair when it uses both syntactic and semantic distances. A similar example of a desirable repair generated by

Qlose using both syntactic and semantic distances is illustrated in Fig. 7 for the `ePoly-1` benchmark.

```
int LargestGap(int[] a){
  int h = a[0], l=0;
  int N = a.Length;
  for(int i=1; i<N;++i){
    h = max(h,a[i]);
    l = min(l,a[i]);
  }
  return h - l;
}
```

```
int ePoly-1(int[]p,int
     x){
  int n = p[0];
  int i = p.Length-1;
  while(i >= 0){
    n += p[i]*pow(x,i);
    i--;
  }
  return n;
}
```

(a) Original

```
int LargestGap(int[] a){
  int h = a[0], l=0;
  int N = a.Length;
  for(int i=0; i<N;++i){
    h = max(h,a[i]);
    l = min(l,a[i]);
  }
  return h - l;
}
```

```
int ePoly-1(int[]p,int
     x){
  int n = 0;
  int i = p.Length-1;
  while(i >= 0){
    n += p[i]*pow(x,i);
    i--;
  }
  return n;
}
```

(b) Syntactic

```
int LargestGap(int[] a){
  int h = a[0], l=a[0];
  int N = a.Length;
  for(int i=1; i<N;++i){
    h = max(h,a[i]);
    l = min(l,a[i]);
  }
  return h - l;
}
```

```
int ePoly-1(int[]p,int
     x){
  int n = p[0];
  int i = p.Length-1;
  while(i > 0){
    n += p[i]*pow(x,i);
    i--;
  }
  return n;
}
```

(c) Syntactic+Semantic

Fig. 7: (a) The original `LargestGap-2` and `ePoly-1` programs, (b) the repair generated by the syntactic distance, and (c) the repair generated by the combination of syntactic and semantic distances that corresponds to the desired repair.

6.3 Solving Time

The solving times for different combinations of syntactic and semantic distances are shown in Fig. 6. As expected, the syntactic distances take the smallest amount of time to resolve the sketches. For some problems, the semantic distances also resolve within a few seconds, but there are some cases where the solver takes much longer (including a case where the solver times out at 20 minutes). Our hypothesis for this phenomenon is that the semantic constraints by themselves under-constrain the space of repairs, which causes the solver to search a larger space for finding the optimal solution for the minimization objective. On the other hand, by combining syntactic and semantic distances, Qlose can solve the sketches with minimization constraints within 20s for each benchmark.

6.4 Repairs with Different Test Sets

In this experiment, we evaluate the effect of using different sets of tests on the repairs generated by Qlose. We empirically observe that the combination of syntactic and semantic distances is more robust with respect to changes in the test set as compared to individual distances. For example, if we look at Fig. 8,

Tests over variables s, c, and k	**Syntactic Fix**
(adb?,?,3),(bgc?cg,?,5),(?aba,?,3),(abcdd?,g,4)	for(i=0;i<k;i++) if(**c==’?’**)
(adb?,?,3),(bgc?cg,?,5),(gaba,?,3),(abcdd?,g,4)	for(i=0;**s[i]!=’e’**;i++) . . .
(ab?,?,2),(aba?cg,?,5), (?aba,?,3),(abcdd?,?,4)	for(i=0;**s[i]!=’d’**;i++) . . .

Fig. 8: Repairs obtained for `FindC` with syntactic distance for different test sets.

we can see that when we vary the test set for the `FindC` benchmark, using only syntactic distances yields different and undesired repairs. On the contrary, we obtain the same desired repair using the combined distance for these test sets.

7 Related Work

We review relevant work focussing on sequential, imperative software programs.

The authors in [42] were the first to emphasize the need to look for repaired programs that are semantically close to the original program. But they did not develop a quantitative formulation of the problem and relied on choosing sets of traces of the original program to be preserved exactly. There are several program repair approaches that aim to find repairs that are syntactically close to the original program [25, 33, 28, 34]. As we have discussed in the paper, focussing just on syntactic changes can lead to non-intuitive repairs. The AutoProf system [34] uses the Sketch solver to compute the minimum number of syntactic changes to incorrect student solutions based on a manual error model. Qlose, on the other hand, uses additional syntactic and semantic distances, and generalizes the set of expression modifications using linear combinations of constants with program variables.

There is also a growing and interesting body of work on quantitative notions for verification and synthesis [7, 8, 19], which formalize distances between specifications and systems or between systems themselves. However, these distances mostly apply to reactive systems and temporal logic specifications. There have also been many proposals for scaling program repair and synthesis to large programs. These are based on techniques ranging from constraint-solving [37, 29, 38], winning strategies in games [22], abstractions [27, 15, 32], mutations [11], genetic algorithms [2, 13], using contracts [43], and focusing on data structure manipulations [35, 44]. As we develop Qlose further, we hope to leverage some of these techniques and improve the scope of our approach.

Many fault localization algorithms are based on analyzing error traces [45, 3, 24, 9]. Some of these techniques can be used as a preprocessing step to improve the efficiency of our algorithm. A recent paper [26] finds the root cause of an equivalence failure in binaries using a notion of semantic similarity between programs. The problem setting is quite different from ours and the notions of similarity mostly refer to the program abstract semantics rather than to concrete executions. We wish to explore whether the distances proposed in [26] can be instantiated in our framework.

A more general question is whether the notions of program distances appearing in quantitative program analysis and program repair can be modeled in Qlose. While simple limits on the number of syntactic edits clearly fall in our framework [28], some complex distances could take into account features that we currently do not model. For example [33] uses location-specific costs that cannot be captured using our current definitions. Extending Qlose to more complex distances is an interesting research direction.

In this paper we use manual code inspection to decide which repair is most *natural*. Recently, many data-driven driven techniques have been proposed to reason about code *naturalness*[20, 30]. These techniques learn language models of source code from a large code corpus and then use these models for several applications such as learning natural coding conventions, code suggestions and auto-completion, improving code style, suggesting variable and method names etc. Using such automatic techniques to classify repairs is an interesting direction.

8 Limitations and Conclusion

We introduce the *quantitative program repair* problem informally described as follows: *given a set D of syntactic and semantic distances, a program P, and a set of test cases T, find the closest program P' (with respect to some function over the distances in D) such that P' is correct on all the tests in T.* We differentiated ourselves from previous approaches by showing that, to find "natural" program repairs, both semantic and syntactic distances are necessary. Our techniques have been implemented in a prototype tool Qlose, but some limitations need to be addressed. The most important ones are that the distances are tailored to specifications given as test sets and that Qlose only handles programs with tens of lines of code. Addressing these limitations is part of our research agenda.

Acknowledgements

The authors would like to thank the anonymous reviewers for their insightful feedback. Roopsha Samanta would like to thank Krishnendu Chatterjee and Tom Henzinger for inspiring discussions on quantitative repair for reactive systems.

References

1. R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. Automated grading of dfa constructions. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI'13, pages 1976–1982. AAAI Press, 2013.
2. A. Arcuri. On the Automation of Fixing Software Bugs. In *International Conference on Software Engineering (ICSE)*, pages 1003–1006. ACM, 2008.

3. T. Ball, M. Naik, and S. K. Rajamani. From Symptom to Cause: Localizing Errors in Counterexample Traces. In *Principles of Programming Languages (POPL)*, pages 97–105. ACM, 2003.
4. T. Ball and S. K. Rajamani. Boolean Programs: A Model and Process for Software Analysis. Technical Report 2000-14, MSR, 2000.
5. R. Bloem, K. Chatterjee, T. A. Henzinger, and B. Jobstmann. Better Quality in Synthesis through Quantitative Objectives. In *Computer Aided Verification (CAV)*, pages 140–156. Springer, 2009.
6. C. Blog. A triple milestone: 107 partners, 532 courses, 5.2 million students and counting!, 2013.
7. P. Cerný and T. A. Henzinger. From Boolean to Quantitative Synthesis. In *Proceedings of the 11th International Conference on Embedded Software (EMSOFT)*, pages 149–154, 2011.
8. P. Cerný, T. A. Henzinger, and A. Radhakrishna. Simulation Distances. *Theor. Comput. Sci.*, 413(1):21–35, 2012.
9. S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic Debugging. In *International Conference on Software Engineering (ICSE)*, pages 121–130. ACM, 2011.
10. L. de Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of TACAS'08*, LNCS. Springer, 2008.
11. V. Debroy and W. E. Wong. Using Mutation to Automatically Suggest Fixes for Faulty Programs. In *Software Testing, Verification and Validation (ICST)*, pages 65–74, 2010.
12. I. Dillig, T. Dillig, and A. Aiken. Automated Error Diagnosis using Abductive Inference. In *Programming Language Design and Implementation*, pages 181–192. ACM, 2012.
13. C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *International Conference on Software Engineering (ICSE)*, pages 3–13. IEEE Press, 2012.
14. C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.
15. A. Griesmayer, R. Bloem, and B. Cook. Repair of Boolean Programs with an Application to C. In *Computer Aided Verification (CAV)*, pages 358–371, 2006.
16. A. Griesmayer, S. Staber, and R. Bloem. Automated Fault Localization for C Programs. In *Verification and Debugging (V&D)*, 2006.
17. A. Groce, S. Chaki, D. Kroening, and O. Strichman. Error Explanation with Distance Metrics. *Int. J. Softw. Tools Technol. Transf.*, 8(3):229–247, 2006.
18. S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. *SIGPLAN Not.*, 46(6):50–61, June 2011.
19. T. A. Henzinger and J. Otop. From Model Checking to Model Measuring. In *Proceedings of Concurrency Theory (CONCUR)*, pages 273–287, 2013.
20. A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 837–847, Piscataway, NJ, USA, 2012. IEEE Press.
21. S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. Solving geometry problems using a combination of symbolic and numerical reasoning. In K. McMillan, A. Middeldorp, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 8312 of *Lecture Notes in Computer Science*, pages 457–472. Springer Berlin Heidelberg, 2013.

22. B. Jobstmann, A. Griesmayer, and R. Bloem. Program Repair as a Game. In *Computer Aided Verification (CAV)*, pages 226–238. Springer-Verlag, 2005.
23. K. Jordan. Mooc completion rates: The data, 2014.
24. M. Jose and R. Majumdar. Cause Clue Clauses: Error Localization using Maximum Satisfiability. In *Programming Language Design and Implementation (PLDI)*, pages 437–446. ACM, 2011.
25. R. Könighofer and R. Bloem. Automated Error Localization and Correction for Imperative Programs. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 91–100, 2011.
26. S. K. Lahiri, R. Sinha, and C. Hawblitzel. Automatic Rootcausing for Program Equivalence Failures in Binaries. In *Computer Aided Verification (CAV)*, pages 362–379, 2015.
27. F. Logozzo and T. Ball. Modular and Verified Automatic Program Repair. In *Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 133–146. ACM, 2012.
28. S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 448–458, Piscataway, NJ, USA, 2015. IEEE Press.
29. H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 772–781, Piscataway, NJ, USA, 2013. IEEE Press.
30. N. Partush and E. Yahav. Abstract semantic differencing via speculative correlation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 811–828, 2014.
31. D. Patterson. Why are english majors studying computer science?, 2013.
32. R. Samanta, J. V. Deshmukh, and E. A. Emerson. Automatic Generation of Local Repairs for Boolean Programs. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 1–10, 2008.
33. R. Samanta, O. Olivo, and E. A. Emerson. Cost-Aware Automatic Program Repair. In *Proceedings of Static Analysis Symposium (SAS)*, pages 268–284, 2014.
34. R. Singh, S. Gulwani, and A. Solar-Lezama. Automatic Feedback Generation for Introductory Programming Assignments. In *Proceedings of Programming Language Design and Implementation (PLDI)*, pages 15–26, 2013.
35. R. Singh and A. Solar-Lezma. Synthesizing Data-Structure Manipulations from Storyboards. In *Foundations of Software Engineering (FSE)*, pages 289–299, 2011.
36. A. Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.
37. A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial Sketching for Finite Programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415. ACM, 2006.
38. S. Srivastava, S. Gulwani, and J. S. Foster. From Program Verification to Program Synthesis. In *Principles of Programming Languages (POPL)*, pages 313–326. ACM, 2010.
39. N. Tillmann, J. de Halleux, T. Xie, and J. Bishop. Code hunt: Gamifying teaching and learning of computer science at scale. In *Proceedings of the First ACM Conference on Learning @ Scale Conference*, L@S '14, pages 221–222, New York, NY, USA, 2014. ACM.
40. N. Y. Times. Instruction for masses knocks down campus walls, 2012.

41. P. Černý, T. A. Henzinger, and A. Radhakrishna. Simulation Distances. In P. Gastin and F. Laroussinie, editors, *Proceedings of Concurrency Theory (CONCUR)*, volume 6269 of *LNCS*, pages 253–268. Springer, 2010.
42. C. von Essen and B. Jobstmann. Program Repair without Regret. In N. Sharygina and H. Veith, editors, *Proceedings of Computer Aided Verification (CAV)*, volume 8044 of *LNCS*, pages 896–911. Springer, 2013.
43. Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated Fixing of Programs with Contracts. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 61–72. ACM, 2010.
44. R. N. Zaeem, D. Gopinath, S. Khurshid, and K. S. McKinley. History-Aware Data Structure Repair using SAT. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 2–17. Springer-Verlag, 2012.
45. A. Zeller and R. Hilebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, 2002.