Highlights

MENTOR: Fixing Introductory Programming Assignments With Formula-Based Fault Localization and LLM-Driven Program Repair

Pedro Orvalho[1], Mikoláš Janota, Vasco Manquinho

- MENTOR addresses the Automated Program Repair (APR) problem, for the C programming language, through an LLM-driven Counterexample-Guided Inductive Synthesis (CEGIS) approach.
- MENTOR employs MaxSAT-based Fault Localization to guide and minimize LLMs' patches to incorrect programs by feeding them bug-free program sketches.
- MENTOR combines state-of-the-art modules for program clustering, variable alignment, and MaxSAT-based fault localization; within an LLM-driven program fixer that orchestrates the repair process.
- Experimental results show that our approach enables all six evaluated LLMs to fix more programs and produce smaller patches compared to alternative configurations and symbolic tools.
- All code and experiments are publicly available on Zenodo [1].

[1]Part of this work was conducted at INESC-ID, Instituto Superior Técnico, U. Lisboa.

MENTOR: Fixing Introductory Programming Assignments With Formula-Based Fault Localization and LLM-Driven Program Repair

Pedro Orvalho[1a], Mikoláš Janota[b], Vasco Manquinho[c]

[a]*Department of Computer Science, University of Oxford, Oxford, United Kingdom*
[b]*CIIRC, Czech Technical University in Prague, Prague, Czech Republic*
[c]*INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal*

Abstract

The increasing demand for programming education has led to online evaluations like MOOCs, which rely on introductory programming assignments (IPAs). A major challenge in these courses is providing personalized feedback at scale. This paper introduces MENTOR, a semantic automated program repair (APR) framework designed to fix faulty student programs. MENTOR validates repairs through execution on a test suite, and returns the repaired program or highlights faulty statements.

Unlike symbolic repair tools like Clara and Verifix, which require correct implementations with identical control flow graphs (CFGs), MENTOR's LLM-based approach enables flexible repairs without strict structural alignment. MENTOR clusters successful submissions regardless of CFGs, and employs a Graph Neural Network (GNN)-based variable alignment module for enhanced accuracy. Next, MENTOR's fault localization module leverages MaxSAT techniques to pinpoint buggy code segments precisely. Finally, MENTOR's program fixer integrates Formal Methods (FM) and Large Language Models (LLMs) through a Counterexample Guided Inductive Synthesis (CEGIS) loop, iteratively refining repairs. Experimental results show that MENTOR significantly improves repair success rates, achieving 64.4%, far surpassing Verifix (6.3%) and Clara (34.6%). By merging formula-based fault localization, and LLM-driven repair, MENTOR provides an innovative, scalable framework for programming education.

[1]Part of this work was conducted at INESC-ID, Instituto Superior Técnico, U. Lisboa.

Keywords:
Automated Program Repair, Program Analysis, Formula-based Fault Localization, LLM-Driven Program Repair, Computer-Aided Education.

1. Introduction

Every year, thousands of students enroll in programming-oriented courses. With the rapid growth of Computer Science courses, providing personalized and timely feedback on *introductory programming assignments* (IPAs) and software projects has become a significant challenge, requiring substantial time and effort from faculty [2].

Automated Program Repair (APR) has emerged as a promising solution to this challenge, aiming to deliver automated, comprehensive, and personalized feedback to students about their programming errors [2, 3, 4, 5, 6]. Traditional semantic APR techniques based on *Formal Methods* (FM), while providing high-quality fixes, are often slow and may struggle when the correct implementation diverges significantly from the erroneous one [7, 8]. These APR approaches do not guarantee minimal repairs, as they align an incorrect submission with a correct implementation for the same IPA. If the alignment is not possible, these tools return a structural mismatch error, leaving the program unrepaired. In the past decade, there has been a surge in Machine Learning (ML) techniques for APR [9, 10, 11, 12, 13, 14, 15]. ML-based approaches require multiple correct implementations to generate high-quality repairs, and need considerable time and resources to train on correct programs. While these approaches generate repairs more quickly, they often produce imprecise and non-minimal fixes [4].

More recently, *Large Language Models* (LLMs) trained on code (LLMCs) have shown great potential in generating program fixes [16, 17, 18, 19, 20, 21, 22, 23]. LLM-based APR can be performed using zero-shot learning [24], few-shot learning [22] or fine-tuned models [18]. Fine-tuned models are the most commonly used, where the model is trained for a specific task. In contrast, zero-shot learning refers to the ability of a model to correctly perform a task without having seen any examples of that task during training. Few-shot learning refers to the LLMs's ability to perform tasks correctly with only a small number of examples provided. Furthermore, the ability to generalize using zero or few-shot learning enables LLMs to handle a wide range of tasks without the need for costly retraining or fine-tuning. Nonetheless, few-shot

Table 1: Test-suite with three tests (t_0, t_1, and t_2). Given as input three numbers (num1, num2, and num3), each test should return the value in its respective output column.

	Input			**Output**
	num1	**num2**	**num3**	
t_0	1	2	3	3
t_1	6	2	1	6
t_2	-1	-2	-3	-1

learning can lead to larger fixes than necessary, as it is based on a limited number of examples. Moreover, LLMs do not guarantee minimal repairs and typically rewrite most of the student's implementation to fix it, rather than making minimal adjustments, making their fixes harder for students to learn from.

In this work, we present MENTOR, a clustering-based semantic program repair tool for the C programming language capable of providing auto**m**ated f**e**edback for i**n**troduc**t**ory pr**o**gramming exe**r**cises. MENTOR leverages past student submissions to assist in repairing incorrect code. It combines *Maximum Satisfiability* (MaxSAT)-based fault localization with large language models (LLMs), guiding them using bug-free program sketches. MENTOR's main objective is to repair buggy programs using correct implementations, even when their control-flow graphs (CFGs) differ, a capability not supported by state-of-the-art semantic repair tools.

1.1. Problem Description

Consider the program presented in Listing 1, which aims to determine the maximum among three given numbers. Based on the test suite shown in Table 1, the program is buggy as its output differs from the expected results. The set of minimal faulty lines in this program includes lines 4 and 8, as these two `if` conditions are incorrect. A good way to provide personalized feedback to students on their IPAs is to highlight these two buggy lines. However, it is essential to check these faults by fixing the program and evaluating the repaired program against the test suite.

Using traditional Automated Program Repair (APR) tools for introductory programming assignments (IPAs) based on Formal Methods, such as Clara [2] or Verifix [3], the program in Listing 1 cannot be fixed within 90 seconds. Clara takes too long to compute a 'minimal' repair by considering several correct implementations for the same IPA, while Verifix returns a compilation error. Conversely, using state-of-the-art LLMs trained for coding tasks (LLMCs), Granite [25] or CodeGemma [26], would in-

Listing 1: Semantically incorrect program. Faulty lines: {4,8}.

```c
1  int main(){// finds max of 3 numbers
2     int f,s,t;
3     scanf("%d%d%d",&f,&s,&t);
4     if (f < s && f >= t)//fix: f >= s
5        printf("%d",f);
6     else if (s > f && s >= t)
7        printf("%d",s);
8     else if (t < f && t < s)
9     // fix: t > f and t > s
10       printf("%d",t);
11
12    return 0;
13 }
```

Listing 2: Reference implementation.

```c
1  int main() {
2     int m1,m2,m3,m;
3     scanf("%d%d%d",&m1,&m2,&m3);
4     m = m1 > m2 ? m1 : m2;
5     m = m3 > m ? m3 : m;
6     printf("%d\n", m);
7
8     return 0;
9  }
```

Listing 3: Program sketch with holes.

```c
1  int main(){
2     int f,s,t;
3     scanf("%d%d%d",&f,&s,&t);
4     @ HOLE 1 @
5        printf("%d",f);
6     else if (s > f && s >= t)
7        printf("%d",s);
8     @ HOLE 2 @
9        printf("%d",t);
10
11    return 0;
12 }
```

Listing 4: GRANITE's fix using the program sketch.

```c
1  int main(){
2     int f,s,t;
3     scanf("%d%d%d",&f,&s,&t);
4     if (f >= s && f >= t)
5        printf("%d",f);
6     else if (s > f && s >= t)
7        printf("%d",s);
8     else
9        printf("%d",t);
10
11    return 0;
12 }
```

volve providing the description of the programming assignment and some examples of input-output tests. Even with these features, neither LLM could fix the buggy program in Listing 1 within 90 seconds when repeatedly testing and refining their fixes. If the lecturer's reference implementation shown in Listing 2 is suggested as a reference in the prompt, both LLMs simply copy the correct program, ignoring instructions not to do so.

Hence, symbolic approaches demand an excessive amount of time to produce an answer, and LLMs, while fast, often produce incorrect fixes. Thus, MENTOR aims to address the limitations of current state-of-the-art APR frameworks for IPAs by combining the strengths of both approaches. State-of-the-art Maximum Satisfiability (MaxSAT)-based Fault Localization [27] can rigorously identify buggy statements, which can then be highlighted in the LLM prompt to focus on the specific parts of the program that need fixing. Listing 3 shows an example of a program sketch, which is a partially incomplete program where each buggy statement from the original incorrect program in Listing 1 is replaced with a `@ HOLE @`. Instructing the LLMs to complete this program allows both Granite and CodeGemma to fix the buggy program in a single interaction, returning the program in Listing 4.

Thus, MENTOR aims to repair a student's incorrect submission without relying on a correct solution with the same program structure, while also minimizing changes to the student's code, similar to symbolic approaches [2, 3].

1.2. Paper Overview and Contributions

Section 2 presents the basic definitions and notation used throughout this paper. Next, Section 3 describes the architecture of MENTOR, which consists of four main modules: (1) program clustering, (2) variable alignment, (3) fault localization, and (4) program fixer. MENTOR employs state-of-the-art tools at each stage: InvAASTCluster [8] for clustering, a GNN-based approach [15] for variable mapping, and CFaults [27] for formula-based fault localization. Furthermore, Section 4 describes our experimental results on C-Pack-IPAs [28], a benchmark of IPAs. These experiments demonstrate that MENTOR's hybrid repair method, integrating Formal Methods (FM)-based fault localization with LLMs, significantly enhances repair success rates while yielding smaller, more precise fixes. This innovative approach outperforms other existing repair strategies and state-of-the-art symbolic tools. For instance, Verifix repairs only 6.3% of C-Pack-IPAs, and Clara achieves a repair rate of 34.6%. In contrast, depending on the prompt configuration and the specific LLM utilized, MENTOR achieves impressive repair rates, ranging from 37.3% to 64.4%. Finally, Section 5 reviews related work, and the paper concludes in Section 6.

The main novelty of MENTOR, compared to our earlier work [29], lies in its program fixer module, which integrates the outputs of three key components: (1) cluster representatives, (2) variable mappings, and (3) localized faulty statements. These elements are incorporated into structured prompts that drive an LLM-based, counterexample-guided program repair process. Furthermore, in addition to the research questions explored in our earlier work, this paper introduces several new dimensions of analysis that further advance the understanding of LLM-based program repair. Specifically, in Section 4, we investigate: (i) the usefulness of providing LLMs with a variable mapping between a buggy program and a correct implementation of the same IPA (RQ6); (ii) the effectiveness of MENTOR in repairing programs that Clara fails to address due to control-flow mismatches (RQ7); (iii) how LLM repair performance is influenced by varying levels of program complexity (RQ8); and (iv) the number of CEGIS iterations typically required by LLMs to repair programs, and what this reveals about their repair efficiency (RQ9). These additional research questions broaden the scope of our study and provide new insights into the capabilities and limitations of LLM-based automated program repair.

In summary, this work presents MENTOR, a semantic automated program repair framework, for the C programming language, designed to provide automated feedback in IPAs. This paper makes the following contributions:

- MENTOR addresses the Automated Program Repair (APR) problem through an LLM-driven Counterexample-Guided Inductive Synthesis (CEGIS) approach;
- MENTOR employs MaxSAT-based Fault Localization to guide and minimize LLMs' patches to incorrect programs by feeding them bug-free program sketches.
- MENTOR combines state-of-the-art modules for program clustering [8], variable alignment [15], and MaxSAT-based fault localization [27]; within an LLM-driven program fixer that orchestrates the repair process.
- Experimental results show that our approach enables all six evaluated LLMs to fix more programs and produce smaller patches compared to alternative configurations and symbolic tools.
- All code and experiments will be made publicly available on GitHub: `https://github.com/pmorvalho/MENTOR`, and on Zenodo [1].

2. Preliminaries

This section provides definitions used throughout this work.

Maximum Satisfiability (MaxSAT). The *Boolean Satisfiability* (SAT) problem is the decision problem for propositional logic [30]. A propositional formula in Conjunctive Normal Form (CNF) is a conjunction of clauses where each clause is a disjunction of literals. The *Maximum Satisfiability* (MaxSAT) problem is an optimization version of SAT, i.e., the goal is to find an assignment that maximizes the number of satisfied clauses in a formula [31].

Basic Block (BB). A *basic block* is a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed). It may, of course, have many predecessors and many successors and may even be its own successor. Program entry blocks might not have predecessors that are in the program; program terminating blocks never have successors in the program [32].

Control dependence. A *control dependence* occurs between a statement and a control predicate whose value dictates if the statement is executed or not [33].

Control flow path. A *control flow path* is defined by the order in which program statements, instructions, and function calls are executed.

Control flow Graph (CFG). A *control flow graph (CFG)* is a directed graph in which the nodes represent basic blocks, and the edges represent control flow paths [32].

Program. A *program* is considered sequential, comprising standard statements such as assignments, conditionals, loops, and function calls, each adhering to their conventional semantics in C. A program P is deemed to contain a bug when an assertion violation occurs during its execution with input I. Conversely, if no assertion violation occurs, the program is considered correct for input I. In cases where a bug is detected for input I, it is possible to define an error trace, representing the sequence of statements executed by program P on input I [27].

Abstract Syntax Tree (**AST*****).*** An AST is a syntax tree in which each node represents an operation, and the node's children represent the arguments of the operation for a given programming language described by a Context-Free Grammar. An AST depicts a program's grammatical structure [34].

Synthesis Problem. For a given program's specification S (e.g., input-output examples), G a context-free grammar, and O be the semantics for a particular Domain-specific language, the goal of *program synthesis* is to infer a program P such that (1) the program is produced by G, (2) the program is consistent with O and (3) P is consistent with S [35, 36, 37].

Please refer to Aho et al. [34] for more context about context-free grammars (CFGs), and Domain-specific language (DSL).

Semantic Program Repair. Given (T, G, O, P), let T be a set of input-output examples (test suite), G be a grammar, O be the semantics for a particular Domain-specific language, and P be a syntactically well-formed program (i.e., sets of statements, instructions, expressions) consistent with G and O but semantically erroneous for at least one of the input-output tests (i.e., $\exists(t^i_{in}, t^i_{out}) \in T \;:\; P(t^i_{in}) \neq t^i_{out}$).

The goal of *Semantic Program Repair* is to find a program P_f by semantically change a subset S_1 of P's statements ($S_1 \subseteq P$) for another set of statements S_2 consistent with G and O, such that,

$$P_f = ((P \setminus S_1) \cup S_2)$$

and

$$\forall(t^i_{in}, t^i_{out}) \in T \;:\; P_f(t^i_{in}) = t^i_{out}.$$

Moreover, automated program repair tools typically perform program synthesis to synthesize the set of program statements S_2 needed to fix the erroneous program P [35, 38].

3. MENTOR

This section presents MENTOR, a semantic automated program repair framework designed to provide automated feedback in introductory programming assignments (IPAs).

The overall architecture of MENTOR is shown in Figure 1. MENTOR receives as input an incorrect submission for a given introductory programming assignment (IPA), a test suite, and a set of N correct submissions for the same IPA. MENTOR is divided into five main modules: (1) program clustering, (2) variable aligner, (3) fault localization, (4) program fixer, and (5) decider. MENTOR starts by clustering, using InvAASTCluster [8], all correct submissions using these program's sets of invariants and abstract syntax trees (ASTs) as described in Section 3.1. Next, MENTOR employs Graph Neural Networks (GNNs) to map the set of variables between each cluster's representative and the incorrect submission [15, 39] based on the ASTs of both programs (see Section 3.2). Additionally, MENTOR uses CFaults [27], a state-of-the-art formula-based fault localization (FBFL) technique for C programs capable of addressing multiple faults. CFaults, described in Section 3.3, leverages Model-Based Diagnosis [40] with multiple

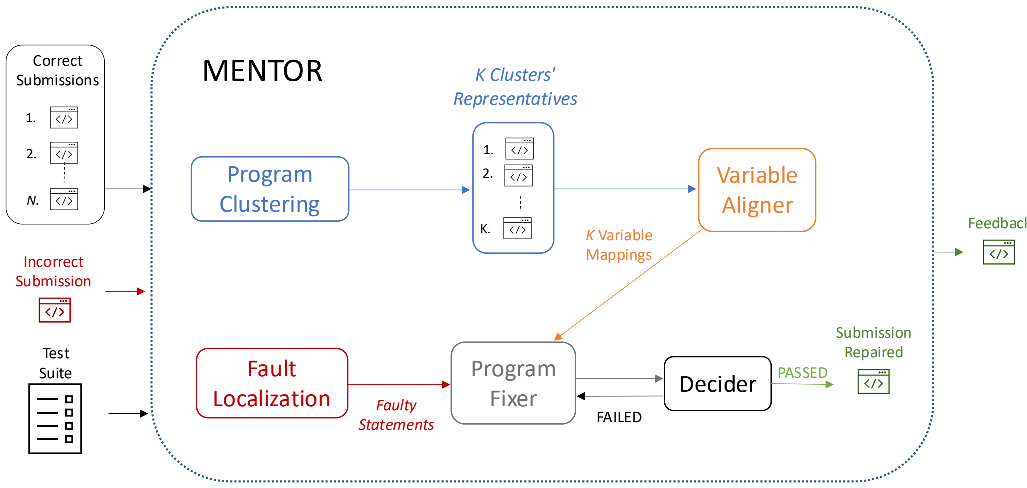


Figure 1: Overview of MENTOR.

observations, integrating all failing test cases into a single MaxSAT formula to ensure consistency in the fault localization process.

The program fixer module integrates all the features computed so far: (1) cluster representatives, (2) variable mappings, and (3) localized faulty statements. These features are incorporated into prompts to guide our LLM-driven counterexample guided program repair process [29], described in Section 3.4. Finally, in the decider module, MENTOR checks if the candidate program proposed by the program fixer is correct by evaluating it against the provided test suite. If the candidate program remains incorrect, the program fixer generates a new candidate. Otherwise, MENTOR returns personalized feedback to students, either by providing the fixed program or by highlighting the buggy statements in their code.

3.1. Program Clustering

To support the repair process, MENTOR aims to find a correct implementation for the same IPA that is structurally and semantically close to a given buggy submission. Instead of exhaustively comparing all correct submissions, MENTOR leverages InvAASTCluster [8], a state-of-the-art clustering tool for IPAs. InvAASTCluster groups correct programs based on their sets of program invariants and their *anonymized abstract syntax trees (AASTs)*, which are abstract syntax trees (ASTs) with variable and function identifiers removed, preserving only type information.

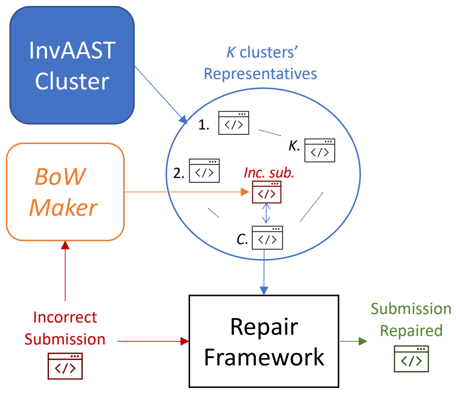


Figure 2: Finding the closest correct program, i.e., the closest correct program representative to the incorrect submission vector representation.

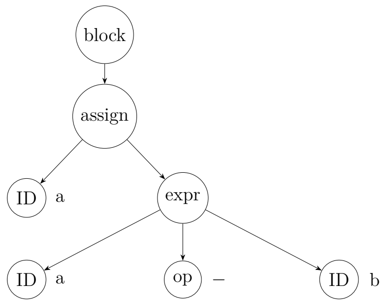
The primary goal of using InvAASTCluster is to reduce the potentially large set of correct programs into a manageable number of diverse clusters. For each cluster, a single representative is selected, which is the program closest to the cluster centroid in Euclidean space. InvAASTCluster achieves this by extracting structural features from each program's AAST and semantic features from its invariants. These features are combined into Bag-of-Words vector representations, enabling clustering into a small number of clusters. When a buggy program is submitted, InvAASTCluster computes its distance to each cluster representative and selects the closest one as the most suitable candidate to guide the repair process. As illustrated in Figure 2, if the incorrect program is closest to representative program C, InvAASTCluster returns C to MENTOR for further analysis and repair guidance.

3.2. Variable Alignment

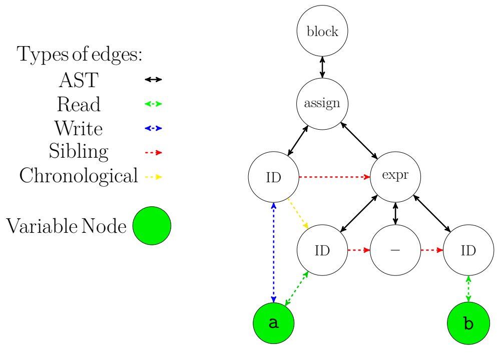
Since the problem of program equivalence, that is, determining whether two programs exhibit the same behavior, is undecidable [41], repairing an incorrect program based on a correct implementation is highly challenging. To compare the faulty and correct programs, repair tools must first establish a correspondence between their respective sets of variables. To address this, MENTOR uses a state-of-the-art graph-based program representation to map variables between incorrect and correct implementations. This representation [15] captures structural information from the abstract syntax trees (ASTs) of imperative programs and leverages graph neural networks (GNNs) to learn how to align variables across program pairs.

Listing 5: Small example of a C code block with an expression.

```c
{ // a and b are ints
  a = a - b;
}
```



(a) Part of the AST representation of Listing 5.



(b) Our program representation for the snippet presented in Listing 5.

Figure 3: AST and our graph representation for the small code snippet presented in Listing 5.

Programs are represented as directed graphs to allow bidirectional information flow in the GNN. These graphs are derived from the program's AST and are enriched with semantic information [15]. Figure 3a shows the AST for the code snippet in Listing 5, while Figure 3b shows its corresponding graph representation. Each variable in the program is assigned a unique node, and all occurrences of that variable are connected to it. For example, in Figure 3b, the green nodes `a` and `b` represent unique variable nodes.

The graph includes five types of edges:

- *Child edges* (black): connect parent and child nodes in the AST and are bidirectional.
- *Sibling edges* (red dashed): connect each child to its next sibling, capturing argument order.
- *Write edges* (blue dashed): connect an `ID` node to its variable node when the variable is written.
- *Read edges* (green dashed): connect an `ID` node to its variable node when the variable is read.

- *Chronological edges* (yellow dashed): connect `ID` nodes of the same variable to preserve usage order.

All edges are bidirectional, except sibling and chronological edges. This graph structure enables the GNN to propagate information through the program in a semantically meaningful way. After several message-passing rounds through these edges, the GNN generates a numerical vector for each variable node in both programs. Then, scalar products are computed between all pairs of variable vectors from the incorrect and correct programs, followed by a softmax operation to yield the final variable mapping probabilities.

3.3. Fault Localization

Effectively identifying the root causes of program failures is essential for generating meaningful repairs [27, 42, 43]. To this end, MENTOR employs CFaults, a state-of-the-art MaxSAT-based fault localization technique. CFaults is designed to identify minimal sets of faulty statements in C programs, even in the presence of multiple bugs. Grounded in Model-Based Diagnosis (MBD) [40], CFaults encodes multiple failing test cases into a single MaxSAT formula. Each failing test case is treated as an observation, and all are simultaneously represented in a relaxed and unrolled version of the input program. This unified formulation ensures that the identified faults are consistent across all failing test cases.

The approach begins with static analysis to detect simple issues (e.g., uninitialized variables or division by zero). If no issues are found, the program is unrolled and instrumented with relaxation variables that selectively deactivate code statements. The resulting program is then transformed into a CNF formula, and a MaxSAT solver is used to find the minimal set of statements whose relaxation would make all failing test cases pass. By encoding multiple test executions into a single MaxSAT instance, CFaults improves precision in scenarios with multiple faults.

3.4. Counterexample Guided Automated Repair

Large Language Models (LLMs) excel at completing strings, while Formal Methods (FM)-based fault localization excel at identifying buggy parts of a program. Therefore, our approach combines the strengths of both FM and LLMs to enhance automated program repair (APR). Firstly, we employ MaxSAT-based fault localization techniques (see Section 3.3) to rigorously identify the minimal set of buggy parts of a program [44, 27]. Afterwards,

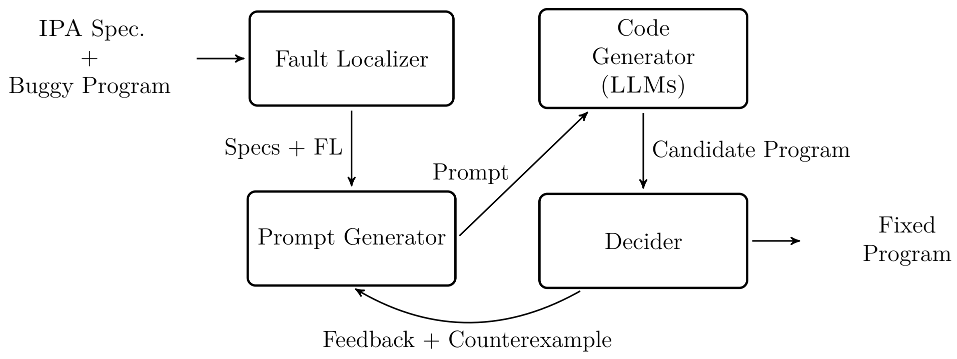


Figure 4: Counterexample Guided Automated Repair.

we leverage LLMs to quickly synthesize the missing parts in the program sketch. Finally, we use a counterexample (i.e., a failing test case from the test suite) to guide LLMs in generating patches that make the synthesized program compliant with the entire test suite, thus completing the repair. The rationale of our approach follows a Counterexample Guided Inductive Synthesis (CEGIS) [45] loop to iteratively refine the program. Figure 4 provides an overview of our APR approach. The input is a buggy program and the specifications for an introductory programming assignment (IPA), including a test suite, a description of the IPA, and the lecturer's reference implementation.

We start by using MaxSAT-based fault localization techniques using all failing test cases to identify the program's minimal set of faulty statements (see Section 3.3). Next, the prompt generator builds a prompt based on the specifications of the IPA and a bug-free program sketch reflecting the localized faults, then feeds this information to the LLM. The LLM generates a program based on the provided prompt. After each iteration, the Decider module evaluates the synthesised program against a test suite, i.e., it checks whether the program passes all the tests in the test suite. If the program is incorrect, a counterexample chosen from the test suite is sent back to the prompt generator, which then provides this counterexample to the LLM to prompt a revised synthesis.

Prompts. The prompts fed to LLMs can contain various types of information related to the IPA. The typical information available in every programming course includes the description of the IPA, the test suite to check the students' submissions corresponding to the IPA's specifications, and the lecturer's reference implementation.

The syntax used in our prompts is similar to that in other works on LLM-driven program repair [16]. We have evaluated several types of prompts. Basic prompts are the simplest prompts that can be fed to an LLM without additional computation, including all the programming assignment's basic information. An example of such a prompt is the following:

~~~
Fix all semantic bugs in the buggy program below.
Modify the code as little as possible.
Do not provide any explanation.

### Problem Description ###
Write a program that determines and prints the largest of three
integers  given by the user.

### Test Suite
#input:
6 2 1
#output:
6
// The other input-output tests

# Reference Implementation (Do not copy this program) <c> #
```c
int main(){
  // Reference Implementation
}
```

### Buggy Program <c> ###
```c
int main(){
  // Buggy program from Listing 1
}
```

### Fixed Program <c> ###
```c
~~~

In order to incorporate information about the faults localized in the program using MaxSAT-based fault localization, we utilized two different types of prompts: (1) `FIXME` annotations and (2) program sketches. `FIXME` annotated prompts are prompts where each buggy line identified by the fault localization tool is marked with a `/* FIXME */` comment. These prompts are quite similar to the basic prompt described previously, with the primary differences being the annotations in the buggy program and the first command given to the LLMs, which is modified as follows:

```
Fix all buggy lines with '/* FIXME */' comments in the buggy
program below.
```

In the second type of prompt, to address program repair as a string completion problem, we evaluated the use of prompts where the buggy program is replaced by an incomplete program (program sketch), with each line identified as buggy by our fault localization module replaced by a *hole*. The command given to the LLMs is now to complete the incomplete program. Consequently, the sections 'Buggy Program' and 'Fixed Program' are replaced by 'Incomplete Program' and 'Complete Program', respectively, as follows:

~~~
Complete all the '@ HOLES N @' in the incomplete program below.
// ...
### Incomplete Program <c> ###
// ...
### Complete Program <c> ###
```c
~~~

Feedback. If the candidate program generated by the LLM is not compliant with the test suite, this feedback is provided to the LLM in a new message through iterative querying. This new prompt indicates that the LLM's previous suggestion to fix the buggy program was incorrect and provides a counterexample (i.e., an input-output test) where the suggested fixed program produces an incorrect output. Hence, we provide the LLM with a feedback prompt similar to the following:

```
### Feedback ###
Your previous suggestion was incorrect!
```

~~~
Try again.
Code only.
Provide no explanation.

### Counterexample  ###
#input:
6 2 1
#output:
6

### Fixed Program <c> ###
```c
~~~

4. Experimental Results

The goal of our evaluation is to answer the following research questions:

- **RQ1.** How effective is MENTOR using LLMs in repairing introductory programming assignments (IPAs) compared to different state-of-the-art symbolic repair approaches?
- **RQ2.** How do different prompt configurations impact the performance of LLMs?
- **RQ3.** How does FM-based fault localization impact LLM-driven APR?
- **RQ4.** What is the performance impact of using a Counterexample Guided approach in LLM-driven APR?
- **RQ5.** How helpful is it to provide a reference implementation for the same IPA to the LLMs?
- **RQ6.** How helpful is it to provide LLMs with a variable mapping between the buggy and a correct implementation of the same IPA?
- **RQ7.** How effective is MENTOR at fixing programs that Clara fails to repair due to control-flow issues?
- **RQ8.** How does LLM performance vary with different levels of program complexity?

- **RQ9.** How many CEGIS iterations do LLMs typically require to repair programs, and what does this indicate about their repair efficiency?

Experimental Setup. All LLMs were run using NVIDIA RTX A4000 graphics cards with 16GB of memory on an Intel(R) Xeon(R) Silver 4130 CPU @ 2.10GHz with 48 CPUs and 128GB RAM. All experiments related to repair tasks were conducted on an Intel(R) Xeon(R) Silver computer with 4210R CPUs @ 2.40GHz, using a CPU memory limit of 10GB and a timeout of 90s. For Clara and Verifix, we set a CPU memory limit of 32GB, as these tools do not utilize GPUs, and a timeout of 90s.

Evaluation Benchmark. To evaluate MENTOR, we used C-Pack–IPAs [28], which is a set of student programs developed during an introductory programming course in the C programming language. These programs were collected over three distinct lab classes for 25 different programming assignments throughout three academic years. Each lab class focuses on a different topic of the C programming language. The first class deals with integers and input-output operations. Secondly, the second class focuses on loops and chars. Lastly, in the third lab class, students learn to use vectors and strings. Only submissions that compile without any errors were selected. The set of submissions was split into two sets: correct submissions and incorrect submissions. The students' submissions that satisfied a set of input-output test cases for each IPA were considered correct. The submissions that failed at least one input-output test were considered incorrect. C-Pack-IPAs contains 1431 semantically incorrect programs submitted for 25 different IPAs.

4.1. Large Language Models (LLMs)

In our experiments, we used only open-access LLMs available on Hugging Face [46] with approximately 7 billion parameters for three primary reasons. Firstly, closed-access models like Chat-GPT are cost-prohibitive and raise concerns over student data privacy. Secondly, models with a very large number of parameters (e.g., 70B) need significant computational resources, such as GPUs with higher RAM capacities, and take longer to generate responses, which is unsuitable for a classroom setting. Thirdly, we used these off-the-shelf LLMs to evaluate the publicly available versions without fine-tuning them. This approach ensures that the LLMs used in this section are available to anyone without investing time and resources into fine-tuning these models. Thus, we evaluated six different LLMs for this study through iterative querying. Three of these models are LLMCs, i.e., LLMs fine-tuned for coding tasks: IBM's Granite [25], Google's CodeGemma [26] and Meta's CodeLlama [47]. The other three models are general-purpose LLMs not specifically tailored for coding tasks: Google's Gemma [48], Meta's Llama3 (latest version of the Llama family [49]) and Microsoft's Phi3 [50].

We selected specific variants of each model to optimize their performance for our program repair tasks. For Meta's Llama3, we utilized the 8B-parameter instruction-tuned variant. This model is designed to follow instructions more accurately, making it suitable for a range of tasks, including program repair. For CodeLlama, we used the 7B-parameter instruct-tuned version, which is specifically designed for general code synthesis and understanding, making it highly effective for coding tasks. We employed the Granite model with 8B-parameters, fine-tuned to respond to coding-related instructions. For Phi3, we opted for the mini version, which has 3.8B-parameters and a context length of 128K. This smaller model is efficient yet capable of handling extensive context, making it practical for educational settings. For Gemma, we used the 7B-parameter instruction-tuned version, optimized to follow detailed instructions. Lastly, for CodeGemma, we selected the 7B-parameter instruction-tuned variant, designed specifically for code chat and instruction, enhancing its capability to handle programming-related queries and tasks. To fit all LLMs into 16GB GPUs, we used model quantization of 4bit. Moreover, all LLMs were run using Hugging Face's Pipeline architecture. By using these different LLMs, we aimed to balance computational efficiency with the ability to effectively generate and refine code, facilitating a practical APR approach in an educational environment.

4.2. Evaluation

To assess the effectiveness of the program fixes generated by the LLMs under different prompt configurations, we used two key metrics: the number of programs successfully repaired and the quality of the repairs. For assessing the patch quality, we use the *Tree Edit Distance* (TED) [51, 52] to compute the distance between the student's buggy program and the fixed program returned by the LLMs. TED computes the structural differences between two Abstract Syntax Trees (ASTs) by calculating the minimum number of edit operations (i.e., insertions, deletions, and substitutions) needed to transform one AST into another.

Based on this metric for measuring program distances, we computed the *distance score*, defined by Equation 1. This score aims to identify and penalize

	Prompt Configurations				
LLMs	**CE**	**De**	**De-TS**	**De-TS-CE**	**FIXME**
CodeGemma	451 (31.5%)	564 (39.4%)	597 (41.7%)	606 (42.3%)	413 (28.9%)
CodeLlama	447 (31.2%)	472 (33.0%)	492 (34.4%)	500 (34.9%)	403 (28.2%)
Gemma	379 (26.5%)	462 (32.3%)	496 (34.7%)	492 (34.4%)	328 (22.9%)
Granite	529 (37.0%)	584 (40.8%)	626 (43.7%)	624 (43.6%)	459 (32.1%)
Llama3	496 (34.7%)	533 (37.2%)	564 (39.4%)	590 (41.2%)	400 (28.0%)
Phi3	367 (25.6%)	462 (32.3%)	494 (34.5%)	489 (34.2%)	313 (21.9%)
Portfolio (All LLMs)	732 (51.2%)	821 (57.4%)	842 (58.8%)	846 (59.1%)	655 (45.8%)
LLMs	**FIXME_De-CE**	**FIXME_De-TS**	**FIXME_De-TS-CE**	**Sk**	**Sk_De-CE**
CodeGemma	563 (39.3%)	592 (41.4%)	601 (42.0%)	484 (33.8%)	615 (43.0%)
CodeLlama	485 (33.9%)	481 (33.6%)	463 (32.4%)	467 (32.6%)	547 (38.2%)
Gemma	443 (31.0%)	446 (31.2%)	444 (31.0%)	367 (25.6%)	524 (36.6%)
Granite	574 (40.1%)	566 (39.6%)	583 (40.7%)	550 (38.4%)	653 (45.6%)
Llama3	531 (37.1%)	535 (37.4%)	557 (38.9%)	434 (30.3%)	565 (39.5%)
Phi3	448 (31.3%)	460 (32.1%)	474 (33.1%)	367 (25.6%)	506 (35.4%)
Portfolio (All LLMs)	806 (56.3%)	796 (55.6%)	820 (57.3%)	717 (50.1%)	890 (62.2%)
LLMs	**Sk_De-TS**	**Sk_De-TS-CE**	**TS**	**Portfolio (All Configurations)**	
CodeGemma	682 (47.7%)	**688 (48.1%)**	511 (35.7%)	850 (59.4%)	
CodeLlama	**573 (40.0%)**	561 (39.2%)	466 (32.6%)	748 (52.3%)	
Gemma	532 (37.2%)	**534 (37.3%)**	404 (28.2%)	780 (54.5%)	
Granite	**691 (48.3%)**	681 (47.6%)	577 (40.3%)	887 (62.0%)	
Llama3	578 (40.4%)	**591 (41.3%)**	505 (35.3%)	929 (64.9%)	
Phi3	**547 (38.2%)**	535 (37.4%)	400 (28.0%)	759 (53.0%)	
Portfolio (All LLMs)	900 (62.9%)	**907 (63.4%)**	767 (53.6%)	1050 (73.4%)	
Verifix [3]	90 (6.3%)				
Clara [2]	495 (34.6%)				

Table 2: The number of programs fixed by each LLM under various configurations. Row *Portfolio (All LLMs)*, shows the best results across all LLMs for each configuration. Column *Portfolio (All Configurations)* shows the best results for each LLM across all configurations. Mapping abbreviations to configuration names: **CE** - *Counterexample*, **De** - *IPA Description*, **FIXME** - *FIXME Annotations*, **SK** - *Sketches*, **TS** - *Test Suite*.

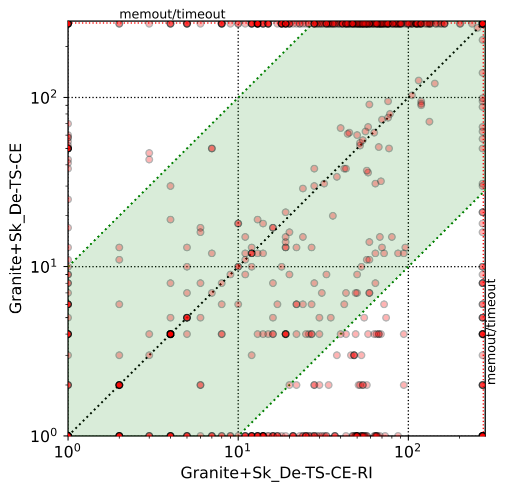
LLMs that replace the buggy program with the reference implementation rather than fixing it. The distance score is zero when the TED of the original buggy program (T_o) to the program suggested by the LLM (T_f) is the same as the TED of the reference implementation (T_r) to T_o. Otherwise, it penalizes larger fixes than necessary to align the program with the correct implementation.

$$ds(T_f, T_o, T_r) = max\Big(0, 1 - \frac{\text{TED}(T_f, T_o)}{\text{TED}(T_r, T_o)}\Big) \quad (1)$$

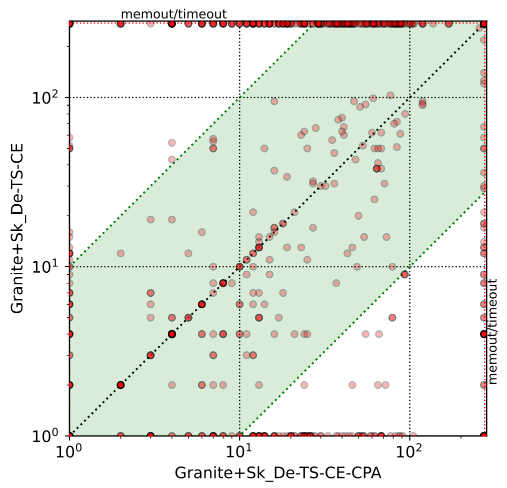
Baseline. We used two state-of-the-art traditional semantic program repair tools for IPAs as baselines: Verifix [3] and Clara [2]. Verifix employs MaxSMT to align a buggy program with a reference implementation provided by the lecturer, while Clara clusters multiple correct implementations and selects the one that produces the smallest fix when aligned with the buggy program. Both tools require an exact match between the control flow graphs (e.g., branches, loops) and a bijective relationship between the variables; otherwise, they return a structural mismatch error. Verifix was provided with each buggy program, the reference implementation, and a test suite. Clara was given all correct programs from different academic years to generate clusters for each IPA. Within a 90-second time limit, Clara repairs 495 programs (34.6%), times out without producing a repair on 154 programs (10.8%), and fails to repair 738 programs (54.7%). Furthermore, Clara's clustering step was performed offline and is not included in the 90-second time limit. In comparison, Verifix repairs 91 programs (6.3%), reaches the time limit on 0.6%, and fails to repair 1338 programs (93.5%). The main reason for these failures is that both tools rely on structure mismatch errors.

Tables 2 and 3 present the number of programs successfully repaired by each LLM under various configurations. The row labeled `Portfolio` represents the best possible outcomes by selecting the optimal configuration for each program across all LLMs. Meanwhile, `Portfolio` column highlights the best results achieved by a particular LLM across all tested configurations. The configurations yielding the highest success rates for the six evaluated LLMs involve incorporating a reference implementation of the IPA into the prompt (see Table 3). However, rather than genuinely fixing the buggy program, the LLMs often replace it with the reference implementation. To address this, we separately analyzed configurations that include (see Section 4.2.1) and exclude access to a reference implementation.

When no reference implementation is provided (see Table 2), Granite still leads among the LLMs, fixing up to 62.0% of the programs across all configurations and 48.3% when using sketches (SK), the IPA description, and a test suite (SK_De-TS). CodeGemma also performs well, achieving up to 59.4% success in a portfolio approach and showing particular strength in configurations involving sketches (SK). For instance, CodeGemma can repair 48.1% of the evaluation benchmark using bug-free sketches, IPA description, test suite, and counterexample (SK_De-TS-CE). Configurations incorporating sketches (SK) and FIXME annotations generally yield better results. Including counterexamples (CE), IPA descriptions, and test suites (De-TS) further boosts the success rate across different LLMs. The portfolio approach, which combines the strengths of all LLMs and configurations without using reference implementation, achieves the highest overall success



(a) Reference Implementation.



(b) Closest Program using AASTs.

Figure 5: Comparison of tree edit distances (TED) for GRANITE's repairs when using (x-axis) versus not using (y-axis) correct implementations with configuration Sk_De-TS-CE.

rate, fixing 73.4% of the programs. This demonstrates that leveraging multiple LLMs together can significantly enhance repair success.

We are now prepared to answer the first four research questions:

A1. All six LLMs used by MENTOR, across different prompt configurations, repair more programs than traditional tools such as CLARA and VERIFIX.

A2. Prompt configurations with FL-based Sketches, IPA description and test suite yield the most successful repair outcomes.

A3. Incorporating FL-based Sketches (or FIXME annotations) allows LLMs to repair more programs than only providing the buggy program.

A4. Employing a Counterexample guided approach significantly improves the accuracy of LLM-driven APR across various configurations.

	Prompt configurations with access to Reference Implementations				
LLMs	**CPA**	**CPIA**	**De-TS-CE-CPA**	**De-TS-CE-RI**	**FIXME_De-TS-CE-CPA**
CodeGemma	505 (35.3%)	447 (31.2%)	578 (40.4%)	576 (40.3%)	637 (44.5%)
CodeLlama	532 (37.2%)	517 (36.1%)	528 (36.9%)	525 (36.7%)	565 (39.5%)
Gemma	633 (44.2%)	364 (25.4%)	595 (41.6%)	607 (42.4%)	563 (39.3%)
Granite	758 (53.0%)	516 (36.1%)	773 (54.0%)	828 (57.9%)	794 (55.5%)
Llama3	595 (41.6%)	464 (32.4%)	685 (47.9%)	691 (48.3%)	657 (45.9%)
Phi3	465 (32.5%)	367 (25.6%)	552 (38.6%)	444 (31.0%)	545 (38.1%)
Portfolio (All LLMs)	1021 (71.3%)	726 (50.7%)	1033 (72.2%)	1046 (73.1%)	1011 (70.6%)

LLMs	**FIXME_De-TS-CE-RI**	**FIXME_De-TS-CPA**	**FIXME_De-TS-RI**	**RI**	**Sk_De-TS-CE-CPA**
CodeGemma	638 (44.6%)	647 (45.2%)	640 (44.7%)	445 (31.1%)	725 (50.7%)
CodeLlama	609 (42.6%)	604 (42.2%)	611 (42.7%)	455 (31.8%)	633 (44.2%)
Gemma	616 (43.0%)	580 (40.5%)	655 (45.8%)	582 (40.7%)	664 (46.4%)
Granite	857 (59.9%)	838 (58.6%)	882 (61.6%)	775 (54.2%)	838 (58.6%)
Llama3	681 (47.6%)	662 (46.3%)	661 (46.2%)	555 (38.8%)	725 (50.7%)
Phi3	492 (34.4%)	572 (40.0%)	508 (35.5%)	358 (25.0%)	639 (44.7%)
Portfolio (All LLMs)	1056 (73.8%)	1036 (72.4%)	1082 (75.6%)	1039 (72.6%)	1050 (73.4%)

LLMs	**Sk_De-TS-CE-RI**	**Sk_De-TS-CPA**	**Sk_De-TS-RI**	**Portfolio (All Configurations)**
CodeGemma	739 (51.6%)	**744 (52.0%)**	729 (50.9%)	950 (66.4%)
CodeLlama	675 (47.2%)	673 (47.0%)	**677 (47.3%)**	959 (67.0%)
Gemma	**732 (51.2%)**	681 (47.6%)	720 (50.3%)	1025 (71.6%)
Granite	876 (61.2%)	881 (61.6%)	**921 (64.4%)**	1169 (81.7%)
Llama3	730 (51.0%)	**783 (54.7%)**	706 (49.3%)	1073 (75.0%)
Phi3	647 (45.2%)	**661 (46.2%)**	653 (45.6%)	959 (67.0%)
Portfolio (All LLMs)	1077 (75.3%)	1080 (75.5%)	**1089 (76.1%)**	1218 (85.1%)

Table 3: The number of programs fixed by each LLM under various configurations *with access to a reference implementation* of each IPA. Row *Portfolio (All LLMs)*, shows the best results across all LLMs for each configuration. Column *Portfolio (All Configurations)* shows the best results for each LLM across all configurations. Mapping abbreviations to configuration names: **CE** - *Counterexample*, **CPA** - *Closest Program using AASTs*, **CPIA** - *Closest Program using Invariants + AASTs*, **De** - *IPA Description*, **FIXME** - *FIXME Annotations*, **RI** - *Reference Implementation*, **SK** - *Sketches*, **TS** - *Test Suite*. Entries highlighted in bold correspond to the highest success rates for each LLM.

	Metric: **sum(Distance Score)**				
	Prompt Configurations				
LLMs	**FIXME_De-TS**	**FIXME_De-TS-CE**	**FIXME_De-TS-CE-CPA**	**FIXME_De-TS-CE-RI**	**FIXME_De-TS-CPA**
CodeGemma	469.6	479.3	249.2	426.3	250.6
CodeLlama	413.5	403.9	240.4	409.2	258.3
Gemma	284.3	282.2	142.2	264.0	144.1
Granite	463.1	470.6	171.2	298.8	160.4
Llama3	353.4	353.6	175.9	392.7	176.0
Phi3	276.0	291.9	96.8	223.8	96.8
LLMs	**FIXME_De-TS-RI**	**RI**	**Sk**	**Sk_De-CE**	**Sk_De-TS**
CodeGemma	435.7	373.5	421.8	473.8	524.4
CodeLlama	417.3	332.3	421.6	464.8	**477.9**
Gemma	272.5	227.2	280.1	328.6	338.8
Granite	282.7	93.8	479.4	506.2	**539.8**
Llama3	390.6	409.8	354.6	383.1	379.8
Phi3	232.6	203.2	265.1	299.5	**326.5**
LLMs	**Sk_De-TS-CE**	**Sk_De-TS-CE-CPA**	**Sk_De-TS-CE-RI**	**Sk_De-TS-CPA**	**Sk_De-TS-RI**
CodeGemma	**529.5**	249.8	497.3	268.8	484.3
CodeLlama	464.5	251.3	459.0	270.8	452.1
Gemma	**340.3**	156.4	316.2	153.9	307.2
Granite	533.6	172.3	334.5	175.4	349.4
Llama3	384.5	172.7	**423.0**	196.4	407.3
Phi3	321.4	98.2	253.4	97.2	256.6

Table 4: The cumulative distance scores for each program repaired by each LLM across various configurations. Entries highlighted in bold correspond to the highest score for each LLM.

4.2.1. Correct Implementations

Furthermore, in Table 3, we provide the results of LLMs with a reference implementation. The correct implementation can either be the lecturer's reference implementation (RI) for the same IPA or the most similar correct program from a previously submitted student. The closest correct program is determined using Tree Edit Distance (TED) values, computed over the AASTs (CPA) of the programs, or over the AASTs and their invariants (CPIA). The intent was to allow the model to reuse correct code snippets to generate repairs. Results show that including a reference implementation allows for better repair results. However, as mentioned earlier, the LLMs often simply copy the provided reference implementation.

Table 4 presents the sum of the distance scores (see Equation 1) for the top-performing LLMs from Tables 2 and 3 across different configurations. This summation aims to penalize LLMs that either copy the provided reference implementation or generate unnecessarily large repairs. For example, Granite using configuration Sk_De-TS-CE-RI can repair 876 programs but yields a total distance score of 334.5, whereas using the same configuration without a correct implementation repairs 681 programs resulting in a higher distance score of 533.6.

Figure 5a shows a scatter plot that compares the tree edit distance (TED) of the buggy program to the program fixed by Granite with and without a reference implementation, using configuration Sk_De-TS-CE. Each point represents a faulty program, where the x-value (resp. y-value) represents the TED cost of Granite' with access to a reference implementation (resp. without it). Points below the diagonal indicate that fixing a program with access to a correct implementation incurs a higher TED cost than fixing it without access. This suggests that while access to a reference implementation enables Granite and other LLMs to repair more programs, it often results in larger changes to the student's program than when no correct implementation is given. Similarly, Figure 5b shows a scatter plot that compares the TED cost between the buggy program and the program fixed by Granite using the closest correct implementation determined by *anonymized abstract syntax trees* (see Section 3.1), AASTs (CPA), and without using this implementation. This plot shows that while having a correct implementation helps Granite repair more programs, it generally results in fewer modifications to the student's code when the CPA is used instead of the lecturer's solution. Moreover, we did not run the same prompt configuration using the closest correct program considering both AASTs and invariants (CPIA) because, as shown in Table 3, this approach did not contribute positively to the repair process of the LLMs.

To answer our fifth research question (RQ5):

A5. Including a reference implementation allows for more repaired programs but with potentially less efficient fixes

4.2.2. Variable Mappings

As explained in Section 3.2, mapping variables between two programs is essential for various applications, such as program repair [15, 39, 53]. In this context, we have evaluated the impact of incorporating variable mappings between the buggy program and a given correct implementation into the prompts used by LLMs. These variable mappings are computed using Graph Neural Networks (GNNs) [15], which map the set of variables between each correct program and the incorrect submission based on both programs' ASTs.

GNNs. The specific GNN architecture used in this work is the relational graph convolutional neural network (RGCN), which can handle multiple edges or relation types within one graph [54]. The numerical representation of nodes in the graph is updated in the message passing step according to the following equation:

$$\mathbf{x}_i' = \mathbf{\Theta}_{\text{root}} \cdot \mathbf{x}_i + \sum_{r \in \mathcal{R}} \sum_{j \in \mathcal{N}_r(i)} \frac{1}{|\mathcal{N}_r(i)|} \mathbf{\Theta}_r \cdot \mathbf{x}_j,$$

where $\mathbf{\Theta}$ are the trainable parameters, $\mathcal{R}$ stands for the different edge types that occur in the graph, and $\mathcal{N}_r$ the neighbouring nodes of the current node i that are connected with the edge type r [55]. After each step, we apply Layer Normalization [56] followed by a Rectified Linear Unit (ReLU) non-linear function. These GNNs [15] were trained on the first lab class of C-Pack–IPAs [28], with MultIPAs [57] augmenting C-Pack-IPAs by generating pairs of buggy/correct programs. At training time, since the incorrect program is generated, the mapping between the variables of both programs is known. The network is trained by minimizing the cross entropy loss between the labels (which are categorical integer values indicating the correct mapping) and the values in each corresponding row of the matrix $\mathcal{P}$. As an optimizer, we used

	Prompt configurations with access to Reference Implementations and Variable Mappings				
LLMs	**CPA-VM**	**CPIA-VM**	**De-TS-CE-CPA-VM**	**De-TS-CE-RI-VM**	**FIXME_De-TS-CE-CPA-VM**
CodeGemma	597 (41.7%)	579 (40.5%)	651 (45.5%)	624 (43.6%)	700 (48.9%)
CodeLlama	660 (46.1%)	**699 (48.8%)**	589 (41.2%)	568 (39.7%)	614 (42.9%)
Gemma	738 (51.6%)	700 (48.9%)	675 (47.2%)	680 (47.5%)	656 (45.8%)
Granite	827 (57.8%)	721 (50.4%)	821 (57.4%)	869 (60.7%)	846 (59.1%)
Llama3	591 (41.3%)	565 (39.5%)	729 (50.9%)	711 (49.7%)	689 (48.1%)
Phi3	552 (38.6%)	516 (36.1%)	598 (41.8%)	555 (38.8%)	601 (42.0%)
Portfolio (All LLMs)	1036 (72.4%)	991 (69.3%)	1042 (72.8%)	1070 (74.8%)	1033 (72.2%)
LLMs	**FIXME_De-TS-CE-RI-VM**	**RI-VM**	**Sk_De-TS-CE-CPA-VM**	**Sk_De-TS-CE-RI-VM**	**Portfolio (All Configurations)**
CodeGemma	705 (49.3%)	526 (36.8%)	**782 (54.6%)**	780 (54.5%)	959 (67.0%)
CodeLlama	618 (43.2%)	543 (37.9%)	681 (47.6%)	677 (47.3%)	984 (68.8%)
Gemma	650 (45.4%)	639 (44.7%)	756 (52.8%)	**766 (53.5%)**	1082 (75.6%)
Granite	882 (61.6%)	832 (58.1%)	901 (63.0%)	**921 (64.4%)**	1167 (81.6%)
Llama3	669 (46.8%)	559 (39.1%)	**792 (55.3%)**	720 (50.3%)	1060 (74.1%)
Phi3	531 (37.1%)	519 (36.3%)	**691 (48.3%)**	**691 (48.3%)**	988 (69.0%)
Portfolio (All LLMs)	1075 (75.1%)	1082 (75.6%)	1078 (75.3%)	**1093 (76.4%)**	1210 (84.6%)

Table 5: The number of programs fixed by each LLM under various configurations with access to variable mappings. Row *Portfolio (All LLMs)*, shows the best results across all LLMs for each configuration. Column *Portfolio (All Configurations)* shows the best results for each LLM across all configurations. Mapping abbreviations to configuration names: **CE** - *Counterexample*, **CPA** - *Closest Program using AASTs*, **CPIA** - *Closest Program using Invariants + AASTs*, **De** - *IPA Description*, **FIXME** - *FIXME Annotations*, **RI** - *Reference Implementation*, **SK** - *Sketches*, **TS** - *Test Suite*, **VM** - *Variable Mapping.*

	Metric: **sum(Distance Score)**				
	Prompt Configurations				
LLMs	**CPA-VM**	**CPIA-VM**	**De-TS-CE-CPA-VM**	**De-TS-CE-RI-VM**	**FIXME_De-TS-CE-CPA-VM**
CodeGemma	268.6	382.2	257.6	470.5	275.5
CodeLlama	217.3	351.6	253.9	426.8	231.0
Gemma	137.2	185.3	156.5	243.4	166.1
Granite	78.1	109.4	148.1	283.5	141.8
Llama3	249.9	316.3	199.2	426.0	189.1
Phi3	134.2	182.1	76.4	214.2	75.3
LLMs	**FIXME_De-TS-CE-RI-VM**	**RI-VM**	**Sk_De-TS-CE-CPA-VM**	**Sk_De-TS-CE-RI-VM**	
CodeGemma	475.3	410.1	286.9	**509.5**	
CodeLlama	424.4	372.7	250.0	**455.2**	
Gemma	263.3	195.7	171.5	**306.4**	
Granite	281.3	114.8	165.7	**330.9**	
Llama3	407.0	429.8	196.1	**432.1**	
Phi3	213.0	**252.4**	78.2	247.5	

Table 6: The cumulative distance scores for each program successfully repaired by each LLM across various configurations considering variable mappings. Entries highlighted in bold correspond to the highest score for each LLM.

the Adam algorithm with its default settings in PyTorch [58]. The batch size was 1. As there are many different programs generated by the mutation procedures, we took one sample from each mutation for each student. Each network was trained for 20 full passes (epochs) over this dataset while shuffling the order of the training data before each pass. For validation purposes, data corresponding to 20% of the students from the first year of the dataset was kept separate and not trained on. Please refer to [15] for more details about the training and evaluation of the GNNs used in MENTOR.

Table 5 shows the results for LLMs when provided with a correct implementation for the same IPA and a variable mapping between the buggy

LLMs	CodeGemma +Sk_De-TS-CE	CodeLlama +Sk_De-TS	Gemma +Sk_De-TS-CE	Granite +Sk_De-TS	Llama3 +Sk_De-TS-CE	Phi3 +Sk_De-TS
#Programs Fixed	270 (34.5%)	210 (26.8%)	210 (26.8%)	**290 (37.0%)**	199 (25.4%)	199 (25.4%)

Table 7: The number of programs repaired by each LLM using their best-performing prompt configuration, specifically on the subset of programs where Clara fails to repair due to control-flow issues (54.7% of C-Pack-IPAs).

Quartiles for Average Cyclomatic Complexity	CodeGemma +Sk_De-TS-CE	CodeLlama +Sk_De-TS	Gemma +Sk_De-TS-CE	Granite +Sk_De-TS	Llama3 +Sk_De-TS-CE	Phi3 +Sk_De-TS	Clara
Q1: 1.0-2.5	**231 (76.5%)**	178 (58.9%)	183 (60.6%)	201 (66.6%)	217 (71.9%)	198 (65.6%)	164 (54.3%)
Q2: 2.5-3.5	**231 (65.3%)**	212 (59.9%)	169 (47.7%)	**231 (65.3%)**	192 (54.2%)	180 (50.9%)	163 (46.1%)
Q3: 3.5-7.0	168 (45.3%)	140 (37.7%)	129 (34.8%)	**190 (51.2%)**	139 (37.5%)	125 (33.7%)	124 (33.4%)
Q4: 7.0-26	58 (14.4%)	43 (10.6%)	53 (13.1%)	**69 (17.1%)**	43 (10.6%)	44 (10.9%)	44 (10.9%)

Table 8: The number of programs repaired by each LLM using their best-performing prompt configuration, considering the average cyclomatic complexity of programs [59].

and correct implementation. The correct implementation can either be the lecturer's reference implementation (RI) for the same IPA or the most similar correct program from a previously submitted student. The closest correct program is determined using Tree Edit Distance (TED) values, computed over the AASTs (CPA) of the programs, or over the AASTs and their invariants (CPIA). As indicated in Table 5, all LLMs, except for Granite, are able to repair more programs when they have access to variable mappings between the buggy and correct programs. For instance, CodeGemma, using the prompt configuration Sk_De-TS-CE-CPA without variable mappings, fixes 725 programs, while the same configuration plus variable mappings fixes 782 programs, representing an improvement of nearly 8%.

Table 6 presents the sum of the distance scores (see Equation 1) for the LLMs from Table 5 across different prompt configurations. Notably, only Llama3's score improves compared to Table 4, increasing from a distance score of 423 to 432.1 in Table 6. This suggests that while access to variable mappings aids LLMs in repairing more programs, it does not significantly enhance the LLMs' distance score in 80% of the evaluated models.

To answer our sixth research question (RQ6):

A6. Incorporating a variable mapping alongside the reference implementation enables even more programs to be repaired, though it may similarly lead to less efficient fixes.

4.3. Discussion

In response to RQ7, we evaluated the effectiveness of LLMs in repairing programs that Clara fails to handle due to control-flow issues, which account for 54.7% of C-Pack-IPAs (738 programs). Table 7 presents these results. Among the best-performing configurations, Granite with Sk_De-TS achieved the highest repair rate, successfully fixing 290 programs (37.0%) in this subset. This highlights Granite's strong capability to handle complex program structures where traditional constraint-based tools fail. CodeGemma with Sk_De-TS-CE also performed well, repairing 270 programs (34.5%), demonstrating the advantage of incorporating counterexamples (CE) alongside the Sketches (Sk) configuration. In contrast, models such as Llama3 and Phi3 achieved lower success rates, each repairing only 199 programs (25.4%), suggesting limitations in their ability to generalize and address intricate control-flow issues.

A7. We evaluated 738 programs that Clara fails to repair due to control-flow issues, representing 54.7% of C-Pack-IPAs. MENTOR repaired 37.0% of these programs using Granite with Sk_De-TS, followed by CodeGemma with Sk_De-TS-CE at 34.5%, while Llama3 and Phi3 performed worse, each repairing only 25.4%.

Furthermore, to answer RQ8, and to gain deeper insights into MENTOR's performance across varying levels of program complexity, we evaluated the average cyclomatic complexity of each program in C-Pack-IPAs using `lizard` [59]. Table 8 summarizes these findings, divided into quartiles based on cyclomatic complexity. For simpler programs (Q1: 1.0–2.5), CodeGemma +Sk_De-TS-CE excelled, achieving a 76.5% repair rate. However, as program complexity increased (Q3: 3.5–7.0), Granite +Sk_De-TS outperformed the other models with a 51.2% repair rate, underscoring its robustness in tackling moderately complex programs. In the most challenging cases (Q4: 7.0–26), Granite retained its lead, repairing 17.1% of programs. These results suggest that while CodeGemma is highly effective for simpler errors, Granite exhibits superior adaptability and resilience when addressing programs of greater complexity.

A8. All evaluated models, including Clara, face significant challenges when repairing programs with an average cyclomatic complexity above seven. Nevertheless, our LLM-based approach successfully repairs more complex programs than Clara.

Finally, considering only CEGIS loops where LLMs were able to repair the program within the time limit, the minimum number of iterations is one, the maximum number of iterations to fix a program is seven, and the average number of iterations is 1.14. In 89% of the cases, the program is repaired on the first attempt. Thus, to answer our final research question (RQ9):

A9. LLMs typically require very few CEGIS iterations to repair programs. On average, only 1.14 iterations are needed, with 89% of repairs completed on the first attempt and no case requiring more than seven iterations.

5. Related Work

Automated Program Repair (APR). Several constraint-based program repair techniques have been proposed to check if a program is semantically correct: clustering-based [2], implementation-driven [4, 3, 5, 60], semantic code search [61, 62, 63, 64], generate-and-validate [65, 66], and semantic-based techniques [67, 68, 69]. *Clustering-based repair* frameworks [2, 70, 4, 8] address incorrect student submissions by leveraging a test suite and a pool of N correct submissions for the same IPA. To improve scalability, they first eliminate dynamically equivalent solutions using the provided test suite and clustering techniques, ensuring that only semantically distinct programs are retained. The remaining correct solutions are then grouped into K semantically different clusters ($K << N$), each represented by a cluster exemplar. These representatives serve as the basis for repairing the incorrect submission, reducing redundancy while maintaining coverage of diverse solution strategies. *Implementation-driven repair* tools use one reference implementation to repair a given incorrect submission [3, 5, 60]. AutoGrader [60] finds potential path differences between the executions of a student's submission and a reference implementation using symbolic execution. Verifix [3] aligns an incorrect program with the reference solution into an automaton. Then, using that alignment relation and MaxSMT solving, Verifix proposes fixes to the incorrect program. Moreover, constraints-based repair tools are developed for specific programming languages and typically support only a subset of that language, based on the focus of the programming course. Extending the supported subset is often hard and time-consuming. Furthermore, *code search methods* [61, 62, 63, 64] form another family of semantic program repair techniques. These approaches rely on a specification (e.g., input–output tests) to query large code repositories and identify fragments that satisfy the specification. Unlike clustering-based or implementation-driven approaches, semantic code search does not rely on the closest correct implementation of the same IPA or on a lecturer-provided reference solution. Instead, it searches across other correct programs to extract code fragments that can be used to repair the incorrect submission. *Generate-and-validate techniques* [65, 66] apply predefined mutation operators to the input program to produce multiple candidate solutions, which are then validated against the test suite for correctness. Finally, *semantic-based* approaches rely on symbolic techniques that encode programs into logical formulae for repair. SemFix [67] leverages fault localization to identify suspicious statements and replaces them with symbolic expressions, which are then evaluated against the test suite. DirectFix [68] extends this idea by encoding the program as a MaxSMT formula, enabling the synthesis of multi-line repairs. While more powerful than SemFix, this approach suffers from scalability issues due to the complexity of solving large formulas. To balance scalability and expressiveness, Angelix [69] introduces the concept of angelic values, states, paths, and forests, which capture the intended program behavior in symbolic form. An angelic value is the expected value an expression should return to pass a given test. An angelic state is the set of variables that are visible in the scope of the expression being analyzed. An angelic path is encoded as a triple containing the faulty expression and its respective angelic value and state, these paths can be achieved by symbolic execution of programs. Finally, an angelic forest is the set of angelic paths that encode a repair problem. Angelix uses these forests to synthesize multi-line fixes.

Large Language Models (LLMs). LLMs trained on code (LLMCs) have demonstrated significant effectiveness in generating program fixes [16, 17, 18, 19, 20, 21, 71, 72, 73]. For instance, RING [16] is a multilingual repair engine powered by an LLMC that uses fault localization information from error messages and leverages the few-shot capabilities of LLMCs for code transformation. In the context of APR for programming education, several works have explored the use of LLMs for coding tasks [22, 23, 74]. PyDex [22], for example, employs iterative querying with CODEX, an LLMC version of ChatGPT, using test-based few-shot selection and structure-based program chunking to repair syntax and semantic errors in Python assignments. Similarly, CODEHELP [74] utilizes OpenAI's LLMs to provide textual feedback to students on their programming assignments. However, to the best of our knowledge, no existing work has explored the use of LLMs guided by formula-based fault localization.

Fault localization (FL). Fault localization techniques typically fall into two main families: *spectrum-based (SBFL)* and *formula-based (FBFL)*. SBFL methods [75, 76, 77, 78, 79, 80] estimate the likelihood of a statement being faulty based on test coverage information from both passing and failing test executions. While SBFL techniques are generally fast, they may lack precision, as not all identified statements are likely to be the cause of failures [81, 82]. In contrast, FBFL approaches [83, 42, 43, 84, 85, 86, 87, 88, 89] are considered exact. FBFL methods encode the fault localization problem into several optimization problems aimed at identifying the minimum number of faulty statements within a program. Typically, these methods perform a MaxSAT call for each failing test, allowing them to individually identify a minimal set of faults for each failing test case rather than simultaneously addressing all failing test cases.

6. Conclusion

In conclusion, MENTOR presents an innovative solution for automated program repair (APR) in the context of introductory programming assignments (IPAs). By addressing the challenges faced by state-of-the-art tools like CLARA and VERIFIX, MENTOR introduces a novel approach that combines advanced techniques for greater flexibility in repairing student submissions. Unlike CLARA, which requires strict structural alignment between faulty and correct programs, MENTOR's LLM-based approach enables flexible repairs without strict structural alignment. Furthermore, MENTOR employs a clustering-based strategy that takes advantage of previously correct submissions to repair programs with different control flow graphs. In addition, MENTOR uses Graph Neural Networks (GNNs) to map variables between programs more effectively in its variable alignment module, and its fault localization module, CFAULTS is based on Maximum Satisfiability (MaxSAT) techniques, which allows it to pinpoint buggy code with precision.

One of MENTOR's key innovations is the combination this MaxSAT-based fault localization with a program fixer module that integrates Formal Methods (FM) and Large Language Models (LLMs), MENTOR iteratively refines student programs, leading to smaller, more precise fixes. This hybrid approach leverages the strengths of both FM, which excels at fault localization, and LLMs, which are adept at generating code patches, exceeding the capabilities of symbolic repair tools alone.

Experimental results on C-Pack-IPAs demonstrate MENTOR's superior performance, with repair success rates ranging from 37.3% to 64.4%, depending on the prompt configuration and LLM used. This marks a significant improvement over the state-of-the-art.

To conclude, MENTOR aims to support both students and lecturers by offering automated personalized feedback in IPAs. For lecturers, MENTOR simplifies the evaluation process and allows programming courses to scale to larger enrollments, as the need for direct personalized feedback from the lecturer is significantly reduced. For students, MENTOR can improve their self-learning experience by providing targeted feedback on syntactic and semantic errors in a shorter time frame.

Currently, MENTOR provides personalized feedback to programmers by offering the repaired program generated by the program fixer or by highlighting bugs in their code. For future work, we propose using the repaired program to provide more detailed feedback, rather than just highlighting errors. This could include providing a sketch of the repaired program as a hint to guide students through fixing their buggy code. Furthermore, future work includes extending MENTOR to programming languages other than C. Moreover, we plan to conduct a user study of MENTOR to gather qualitative insights on its practical utility in educational settings.

Acknowledgments

PO acknowledges partial support from the ERC under the European Union's Horizon 2020 research and innovation programme (FUN2MODEL, grant agreement No. 834115). This work was also supported by Portuguese national funds through FCT, under projects UID/50021/2025, UID/PRR/50021/-2025, and 2023.14280.PEX (DOI: 10.54499/2023.14280.PEX) and grant SFRH/BD/07724/2020 (DOI: 10.54499/2020.07724.BD). This work was also supported by the MEYS within the program ERC CZ under the project

POSTMAN no. LL1902 and co-funded by the EU under the project *ROBOPROX* (reg. no. CZ.02.01.01/00/22_008/0004590).

References

[1] P. Orvalho, M. Janota, V. Manquinho, MENTOR: Fixing Introductory Programming Assignments With Formula-Based Fault Localization and LLM-Driven Program Repair, 2025. URL: `https://zenodo.org/records/15678692`. doi:`10.5281/zenodo.15678691`.

[2] S. Gulwani, I. Radicek, F. Zuleger, Automated clustering and program repair for introductory programming assignments, in: PLDI 2018, ACM, 2018, pp. 465–480.

[3] U. Z. Ahmed, Z. Fan, J. Yi, O. I. Al-Bataineh, A. Roychoudhury, Verifix: Verified repair of programming assignments, ACM Trans. Softw. Eng. Methodol. (2022). URL: `https://doi.org/10.1145/3510418`. doi:`10.1145/3510418`.

[4] K. Wang, R. Singh, Z. Su, Search, align, and repair: data-driven feedback generation for introductory programming exercises, in: PLDI 2018, ACM, 2018, pp. 481–495.

[5] Y. Hu, U. Z. Ahmed, S. Mechtaev, B. Leong, A. Roychoudhury, Refactoring based program repair applied to programming assignments, in: 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019, IEEE, 2019, pp. 388–398.

[6] P. Orvalho, M. Janota, V. Manquinho, GitSEED: A Git-backed Automated Assessment Tool for Software Engineering and Programming Education, in: Proceedings of the 2024 ACM Virtual Global Computing Education Conference, SIGCSE Virtual 2024, volume 1, 2024. URL: `https://doi.org/10.1145/3649165.3690106`. doi:`10.1145/3649165.3690106`.

[7] M. R. Contractor, C. R. Rivero, Improving program matching to automatically repair introductory programs, in: S. Crossley, E. Popescu (Eds.), Intelligent Tutoring Systems, Springer International Publishing, Cham, 2022, pp. 323–335.

[8] P. Orvalho, M. Janota, V. Manquinho, InvAASTCluster: On Applying Invariant-Based Program Clustering to Introductory Programming Assignments, J. Syst. Softw. 230 (2025) 112481. doi:10.1016/J.JSS.2025.112481.

[9] R. Gupta, S. Pal, A. Kanade, S. K. Shevade, Deepfix: Fixing common C language errors by deep learning, in: S. P. Singh, S. Markovitch (Eds.), AAAI 2017, AAAI Press, 2017, pp. 1345–1351.

[10] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, E. Aftandilian, Deepdelta: learning to repair compilation errors, in: ESEC/SIGSOFT FSE 2019, ACM, 2019, pp. 925–936.

[11] R. Gupta, A. Kanade, S. K. Shevade, Deep reinforcement learning for syntactic error repair in student programs, in: The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, AAAI Press, 2019, pp. 930–937.

[12] M. Yasunaga, P. Liang, Graph-based, self-supervised program repair from diagnostic feedback, in: ICML 2020, volume 119 of *Proceedings of Machine Learning Research*, PMLR, 2020, pp. 10799–10808.

[13] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, B. Hartmann, Learning syntactic program transformations from examples, in: S. Uchitel, A. Orso, M. P. Robillard (Eds.), ICSE 2017, IEEE / ACM, 2017, pp. 404–415.

[14] S. Bhatia, P. Kohli, R. Singh, Neuro-symbolic program corrector for introductory programming assignments, in: ICSE 2018, ACM, 2018, pp. 60–70.

[15] P. Orvalho, J. Piepenbrock, M. Janota, V. M. Manquinho, Graph Neural Networks for Mapping Variables Between Programs, in: ECAI 2023 - 26th European Conference on Artificial Intelligence, volume 372 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, Poland, 2023, pp. 1811–1818. URL: https://doi.org/10.3233/FAIA230468. doi:10.3233/FAIA230468.

[16] H. Joshi, J. P. C. Sánchez, S. Gulwani, V. Le, G. Verbruggen, I. Radicek, Repair is nearly generation: Multilingual program repair with llms, in:

B. Williams, Y. Chen, J. Neville (Eds.), Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023, AAAI Press, 2023, pp. 5131–5140. URL: https://doi.org/10.1609/aaai.v37i4.25642. doi:10.1609/AAAI.V37I4.25642.

[17] C. S. Xia, Y. Ding, L. Zhang, The plastic surgery hypothesis in the era of large language models, in: 38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023, IEEE, 2023, pp. 522–534. URL: https://doi.org/10.1109/ASE56229.2023.00047. doi:10.1109/ASE56229.2023.00047.

[18] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, A. Svyatkovskiy, Inferfix: End-to-end program repair with llms, in: S. Chandra, K. Blincoe, P. Tonella (Eds.), Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023, ACM, 2023, pp. 1646–1656. URL: https://doi.org/10.1145/3611643.3613892. doi:10.1145/3611643.3613892.

[19] Y. Wei, C. S. Xia, L. Zhang, Copiloting the copilots: Fusing large language models with completion engines for automated program repair, in: S. Chandra, K. Blincoe, P. Tonella (Eds.), Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023, ACM, 2023, pp. 172–184. URL: https://doi.org/10.1145/3611643.3616271. doi:10.1145/3611643.3616271.

[20] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, S. H. Tan, Automated repair of programs from large language models, in: 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023, IEEE, 2023, pp. 1469–1481. URL: https://doi.org/10.1109/ICSE48619.2023.00128. doi:10.1109/ICSE48619.2023.00128.

[21] C. S. Xia, Y. Wei, L. Zhang, Automated program repair in the era of large pre-trained language models, in: 45th IEEE/ACM International Conference-

ence on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023, IEEE, 2023, pp. 1482–1494. URL: https://doi.org/10.1109/ICSE48619.2023.00129. doi:10.1109/ICSE48619.2023.00129.

[22] J. Zhang, J. P. Cambronero, S. Gulwani, V. Le, R. Piskac, G. Soares, G. Verbruggen, Pydex: Repairing bugs in introductory python assignments using llms, Proc. ACM Program. Lang. 8 (2024) 1100–1124. URL: https://doi.org/10.1145/3649850. doi:10.1145/3649850.

[23] T. Phung, J. Cambronero, S. Gulwani, T. Kohn, R. Majumdar, A. Singla, G. Soares, Generating high-precision feedback for programming syntax errors using large language models, in: M. Feng, T. Käser, P. P. Talukdar, R. Agrawal, Y. Narahari, M. Pechenizkiy (Eds.), Proceedings of the 16th International Conference on Educational Data Mining, EDM 2023, Bengaluru, India, July 11-14, 2023, International Educational Data Mining Society, 2023. URL: https://educationaldatamining.org/2023.EDM-short-papers.37/index.html.

[24] C. S. Xia, L. Zhang, Less training, more repairing please: revisiting automated program repair via zero-shot learning, in: A. Roychoudhury, C. Cadar, M. Kim (Eds.), Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022, ACM, 2022, pp. 959–971. URL: https://doi.org/10.1145/3540250.3549101. doi:10.1145/3540250.3549101.

[25] M. Mishra, M. Stallone, G. Zhang, Y. Shen, A. Prasad, A. M. Soria, M. Merler, P. Selvam, S. Surendran, S. Singh, M. Sethi, X. Dang, P. Li, K. Wu, S. Zawad, A. Coleman, M. White, M. Lewis, R. Pavuluri, Y. Koyfman, B. Lublinsky, M. de Bayser, I. Abdelaziz, K. Basu, M. Agarwal, Y. Zhou, C. Johnson, A. Goyal, H. Patel, S. Y. Shah, P. Zerfos, H. Ludwig, A. Munawar, M. Crouse, P. Kapanipathi, S. Salaria, B. Calio, S. Wen, S. Seelam, B. Belgodere, C. A. Fonseca, A. Singhee, N. Desai, D. D. Cox, R. Puri, R. Panda, Granite code models: A family of open foundation models for code intelligence, CoRR abs/2405.04324 (2024). URL: https://doi.org/10.48550/arXiv.2405.04324. doi:10.48550/ARXIV.2405.04324. arXiv:2405.04324.

[26] H. Zhao, J. Hui, J. Howland, N. Nguyen, S. Zuo, A. Hu, C. A. Choquette-Choo, J. Shen, J. Kelley, K. Bansal, L. Vilnis, M. Wirth, P. Michel,

P. Choy, P. Joshi, R. Kumar, S. Hashmi, S. Agrawal, Z. Gong, J. Fine, T. Warkentin, A. J. Hartman, B. Ni, K. Korevec, K. Schaefer, S. Huffman, Codegemma: Open code models based on gemma, CoRR abs/2406.11409 (2024). URL: `https://doi.org/10.48550/arXiv.2406.11409`. doi:`10.48550/ARXIV.2406.11409`. `arXiv:2406.11409`.

[27] P. Orvalho, M. Janota, V. Manquinho, CFaults: Model-Based Diagnosis for Fault Localization in C Programs with Multiple Test Cases, in: Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, 2024, Proceedings, volume 14933 of *Lecture Notes in Computer Science*, Springer Nature Switzerland, Cham, 2024, pp. 463–481. doi:`10.1007/978-3-031-71162-6\_24`.

[28] P. Orvalho, M. Janota, V. Manquinho, C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments, in: 2024 IEEE/ACM International Workshop on Automated Program Repair (APR), ACM, ., 2024, pp. 14–21. URL: `https://doi.org/10.1145/3643788.3648010`. doi:`10.1145/3643788.3648010`.

[29] P. Orvalho, M. Janota, V. Manquinho, Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault Localization, in: In the 39th Annual AAAI Conference on Artificial Intelligence, AAAI 2025, 2025.

[30] A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, volume 185 of *Frontiers in Artificial Intelligence and Applications*, IOS Press, 2009.

[31] F. Bacchus, M. Järvisalo, R. Martins, Maximum Satisfiability, in: A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, IOS Press, 2021, pp. 929 – 991.

[32] F. E. Allen, Control flow analysis, in: R. S. Northcote (Ed.), Proceedings of a Symposium on Compiler Optimization, Urbana-Champaign, Illinois, USA, July 27-28, 1970, ACM, 1970, pp. 1–19.

[33] J. Ferrante, K. J. Ottenstein, J. D. Warren, The program dependence graph and its use in optimization, ACM Trans. Program. Lang. Syst. 9 (1987) 319–349. URL: `https://doi.org/10.1145/24039.24041`. doi:`10.1145/24039.24041`.

[34] A. V. Aho, R. Sethi, J. D. Ullman, Compilers: Principles, Techniques, and Tools, Addison-Wesley series in computer science / World student series edition, Addison-Wesley, 1986.

[35] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, V. M. Manquinho, Encodings for enumeration-based program synthesis, in: Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings, 2019, pp. 583–599.

[36] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, V. M. Manquinho, SQUARES : A SQL synthesizer using query reverse engineering, Proc. VLDB Endow. 13 (2020) 2853–2856. URL: `http://www.vldb.org/pvldb/vol13/p2853-orvalho.pdf`.

[37] P. da Silva, SQUARES : A SQL Synthesizer Using Query Reverse Engineering, Master's thesis, Instituto Superior Técnico, Lisboa, Portugal, 2019.

[38] P. M. O. M. da Silva, MENTOR: Automated Feedback for Introductory Programming Exercises, Ph.D. thesis, INSTITUTO SUPERIOR TÉCNICO, 2025.

[39] P. Orvalho, J. Piepenbrock, M. Janota, V. Manquinho, Project Proposal: Learning Variable Mappings to Repair Programs, in: 7th Conference on Artificial Intelligence and Theorem Proving, AITP, 2022.

[40] R. Reiter, A theory of diagnosis from first principles, Artif. Intell. 32 (1987) 57–95. URL: `https://doi.org/10.1016/0004-3702(87)90062-2`. doi:`10.1016/0004-3702(87)90062-2`.

[41] H. G. Rice, Classes of recursively enumerable sets and their decision problems, Transactions of the American Mathematical Society 74 (1953) 358–366.

[42] M. Jose, R. Majumdar, Cause clue clauses: error localization using maximum satisfiability, in: Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, ACM, 2011, pp. 437–446.

[43] S. Lamraoui, S. Nakajima, A formula-based approach for automatic fault localization of multi-fault programs, J. Inf. Process. 24 (2016) 88–98. URL: `https://doi.org/10.2197/ipsjjip.24.88`. doi:`10.2197/IPSJJIP.24.88`.

[44] A. Ignatiev, A. Morgado, G. Weissenbacher, J. Marques-Silva, Model-based diagnosis with multiple observations, in: S. Kraus (Ed.), Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019, ijcai.org, 2019, pp. 1108–1115. URL: `https://doi.org/10.24963/ijcai.2019/155`. doi:`10.24963/IJCAI.2019/155`.

[45] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, V. A. Saraswat, Combinatorial sketching for finite programs, in: J. P. Shen, M. Martonosi (Eds.), International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2006, pp. 404–415.

[46] HuggingFace, , `https://huggingface.co`, 2024. [Online; accessed 1-July-2024].

[47] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. Canton-Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, G. Synnaeve, Code llama: Open foundation models for code, CoRR abs/2308.12950 (2023). URL: `https://doi.org/10.48550/arXiv.2308.12950`. doi:`10.48550/ARXIV.2308.12950`. `arXiv:2308.12950`.

[48] T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M. S. Kale, J. Love, P. Tafti, L. Hussenot, A. Chowdhery, A. Roberts, A. Barua, A. Botev, A. Castro-Ros, A. Slone, A. Héliou, A. Tacchetti, A. Bulanova, A. Paterson, B. Tsai, B. Shahriari, C. L. Lan, C. A. Choquette-Choo, C. Crepy, D. Cer, D. Ippolito, D. Reid, E. Buchatskaya, E. Ni, E. Noland, G. Yan, G. Tucker, G. Muraru, G. Rozhdestvenskiy, H. Michalewski, I. Tenney, I. Grishchenko, J. Austin, J. Keeling, J. Labanowski, J. Lespiau, J. Stanway, J. Brennan, J. Chen, J. Ferret, J. Chiu, et al., Gemma: Open models based on gemini research and technology, CoRR abs/2403.08295 (2024).

URL: https://doi.org/10.48550/arXiv.2403.08295. doi:10.48550/ARXIV.2403.08295. arXiv:2403.08295.

[49] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, G. Lample, Llama: Open and efficient foundation language models, CoRR abs/2302.13971 (2023). URL: https://doi.org/10.48550/arXiv.2302.13971. doi:10.48550/ARXIV.2302.13971. arXiv:2302.13971.

[50] M. I. Abdin, S. A. Jacobs, A. A. Awan, J. Aneja, A. Awadallah, H. Awadalla, N. Bach, A. Bahree, A. Bakhtiari, H. S. Behl, A. Benhaim, M. Bilenko, J. Bjorck, S. Bubeck, M. Cai, C. C. T. Mendes, W. Chen, V. Chaudhary, P. Chopra, A. D. Giorno, G. de Rosa, M. Dixon, R. Eldan, D. Iter, A. Garg, A. Goswami, S. Gunasekar, E. Haider, J. Hao, R. J. Hewett, J. Huynh, M. Javaheripi, X. Jin, P. Kauffmann, N. Karampatziakis, D. Kim, M. Khademi, L. Kurilenko, J. R. Lee, Y. T. Lee, Y. Li, C. Liang, W. Liu, E. Lin, Z. Lin, P. Madan, A. Mitra, H. Modi, A. Nguyen, B. Norick, B. Patra, D. Perez-Becker, T. Portet, R. Pryzant, H. Qin, M. Radmilac, C. Rosset, S. Roy, O. Ruwase, O. Saarikivi, A. Saied, A. Salim, M. Santacroce, S. Shah, N. Shang, H. Sharma, X. Song, M. Tanaka, X. Wang, R. Ward, G. Wang, P. Witte, M. Wyatt, C. Xu, J. Xu, S. Yadav, F. Yang, Z. Yang, D. Yu, C. Zhang, C. Zhang, J. Zhang, L. L. Zhang, Y. Zhang, Y. Zhang, Y. Zhang, X. Zhou, Phi-3 technical report: A highly capable language model locally on your phone, CoRR abs/2404.14219 (2024). URL: https://doi.org/10.48550/arXiv.2404.14219. doi:10.48550/ARXIV.2404.14219. arXiv:2404.14219.

[51] K. Tai, The tree-to-tree correction problem, J. ACM 26 (1979) 422–433. URL: https://doi.org/10.1145/322139.322143. doi:10.1145/322139.322143.

[52] K. Zhang, D. E. Shasha, Simple fast algorithms for the editing distance between trees and related problems, SIAM J. Comput. 18 (1989) 1245–1262. URL: https://doi.org/10.1137/0218082. doi:10.1137/0218082.

[53] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, X. Chen, Shaping program repair space with existing patches and similar code, in: F. Tip, E. Bodden

(Eds.), Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018, ACM, 2018, pp. 298–309. URL: `https://doi.org/10.1145/3213846.3213871`. doi:`10.1145/3213846.3213871`.

[54] M. S. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, M. Welling, Modeling relational data with graph convolutional networks, in: The Semantic Web - 15th International Conference, ESWC 2018, volume 10843 of *Lecture Notes in Computer Science*, Springer, 2018, pp. 593–607. URL: `https://doi.org/10.1007/978-3-319-93417-4_38`. doi:`10.1007/978-3-319-93417-4\_38`.

[55] PyTorchGeometric, Documentation, `https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#torch_geometric.nn.conv.RGCNConv`, 2022. Accessed 2022-08-12.

[56] L. J. Ba, J. R. Kiros, G. E. Hinton, Layer normalization, CoRR http://arxiv.org/abs/1607.06450 (2016). URL: `http://arxiv.org/abs/1607.06450`. `arXiv:1607.06450`.

[57] P. Orvalho, M. Janota, V. M. Manquinho, MultIPAs: Applying Program Transformations To Introductory Programming Assignments For Data Augmentation, in: A. Roychoudhury, C. Cadar, M. Kim (Eds.), Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022, ACM, 2022, pp. 1657–1661. URL: `https://doi.org/10.1145/3540250.3558931`. doi:`10.1145/3540250.3558931`.

[58] D. P. Kingma, J. Ba, Adam: A method for stochastic optimization, in: 3rd International Conference on Learning Representations, ICLR 2015, 2015. URL: `http://arxiv.org/abs/1412.6980`.

[59] Lizard, Lizard: A simple code complexity analyser, `https://github.com/terryyin/lizard`, 2024. Accessed: 2024-05-01.

[60] X. Liu, S. Wang, P. Wang, D. Wu, Automatic grading of programming assignments: an approach based on formal semantics, in: S. Beecham, D. E. Damian (Eds.), Proceedings of the 41st International Conference

on Software Engineering: Software Engineering Education and Training, ICSE (SEET) 2019, IEEE / ACM, 2019, pp. 126–137.

[61] Y. Ke, K. T. Stolee, C. L. Goues, Y. Brun, Repairing programs with semantic code search (T), in: M. B. Cohen, L. Grunske, M. Whalen (Eds.), 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, IEEE Computer Society, 2015, pp. 295–306.

[62] A. Afzal, M. Motwani, K. T. Stolee, Y. Brun, C. L. Goues, Sosrepair: Expressive semantic search for real-world program repair, IEEE Trans. Software Eng. 47 (2019) 2162–2181. doi:10.1109/TSE.2019.2944914.

[63] K. T. Stolee, S. G. Elbaum, D. Dobos, Solving the search for source code, ACM Trans. Softw. Eng. Methodol. 23 (2014) 26:1–26:45. URL: https://doi.org/10.1145/2581377. doi:10.1145/2581377.

[64] S. P. Reiss, Semantics-based code search, in: 31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings, IEEE, 2009, pp. 243–253. doi:10.1109/ICSE.2009.5070525.

[65] C. L. Goues, T. Nguyen, S. Forrest, W. Weimer, Genprog: A generic method for automatic software repair, IEEE Trans. Software Eng. 38 (2012) 54–72.

[66] F. Long, M. Rinard, Automatic patch generation by learning correct code, in: R. Bodík, R. Majumdar (Eds.), Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016, ACM, 2016, pp. 298–312. doi:10.1145/2837614.2837617.

[67] H. D. T. Nguyen, D. Qi, A. Roychoudhury, S. Chandra, Semfix: program repair via semantic analysis, in: D. Notkin, B. H. C. Cheng, K. Pohl (Eds.), 35th International Conference on Software Engineering, ICSE '13, IEEE Computer Society, 2013, pp. 772–781.

[68] S. Mechtaev, J. Yi, A. Roychoudhury, Directfix: Looking for simple program repairs, in: A. Bertolino, G. Canfora, S. G. Elbaum (Eds.), 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, IEEE Computer Society, 2015, pp. 448–458.

[69] S. Mechtaev, J. Yi, A. Roychoudhury, Angelix: scalable multiline program patch synthesis via symbolic analysis, in: L. K. Dillon, W. Visser, L. A. Williams (Eds.), ICSE 2016, ACM, 2016, pp. 691–701.

[70] D. M. Perry, D. Kim, R. Samanta, X. Zhang, Semcluster: clustering of imperative programming assignments based on quantitative semantic features, in: K. S. McKinley, K. Fisher (Eds.), Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019, ACM, 2019, pp. 860–873.

[71] I. Bouzenia, P. T. Devanbu, M. Pradel, Repairagent: An autonomous, llm-based agent for program repair, in: 47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025, IEEE, 2025, pp. 2188–2200. URL: `https://doi.org/10.1109/ICSE55347.2025.00157`. doi:`10.1109/ICSE55347.2025.00157`.

[72] C. Spiess, D. Gros, K. S. Pai, M. Pradel, M. R. I. Rabin, A. Alipour, S. Jha, P. Devanbu, T. Ahmed, Calibration and correctness of language models for code, in: 47th IEEE/ACM International Conference on Software Engineering, ICSE 2025, Ottawa, ON, Canada, April 26 - May 6, 2025, IEEE, 2025, pp. 540–552. URL: `https://doi.org/10.1109/ICSE55347.2025.00040`. doi:`10.1109/ICSE55347.2025.00040`.

[73] H. Ye, M. Monperrus, ITER: iterative neural repair for multi-location patches, in: Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024, ACM, 2024, pp. 10:1–10:13. URL: `https://doi.org/10.1145/3597503.3623337`. doi:`10.1145/3597503.3623337`.

[74] M. H. Liffiton, B. E. Sheese, J. Savelka, P. Denny, Codehelp: Using large language models with guardrails for scalable support in programming classes, in: A. Mühling, I. Jormanainen (Eds.), Proceedings of the 23rd Koli Calling International Conference on Computing Education Research, Koli Calling 2023, Koli, Finland, November 13-18, 2023, ACM, 2023, pp. 8:1–8:11. URL: `https://doi.org/10.1145/3631802.3631830`. doi:`10.1145/3631802.3631830`.

[75] R. Abreu, P. Zoeteweij, A. J. C. van Gemund, Spectrum-based multiple fault localization, in: ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009, IEEE Computer Society, 2009, pp. 88–99. URL: `https://doi.org/10.1109/ASE.2009.25`. doi:`10.1109/ASE.2009.25`.

[76] W. E. Wong, V. Debroy, B. Choi, A family of code coverage-based heuristics for effective fault localization, J. Syst. Softw. 83 (2010) 188–208. URL: `https://doi.org/10.1016/j.jss.2009.09.037`. doi:`10.1016/J.JSS.2009.09.037`.

[77] L. Naish, H. J. Lee, K. Ramamohanarao, A model for spectra-based software diagnosis, ACM Trans. Softw. Eng. Methodol. 20 (2011) 11:1–11:32. URL: `https://doi.org/10.1145/2000791.2000795`. doi:`10.1145/2000791.2000795`.

[78] W. E. Wong, V. Debroy, R. Gao, Y. Li, The dstar method for effective software fault localization, IEEE Trans. Reliab. 63 (2014) 290–308. URL: `https://doi.org/10.1109/TR.2013.2285319`. doi:`10.1109/TR.2013.2285319`.

[79] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A survey on software fault localization, IEEE Trans. Software Eng. 42 (2016) 707–740. URL: `https://doi.org/10.1109/TSE.2016.2521368`. doi:`10.1109/TSE.2016.2521368`.

[80] R. Abreu, P. Zoeteweij, R. Golsteijn, A. J. C. van Gemund, A practical evaluation of spectrum-based fault localization, J. Syst. Softw. 82 (2009) 1780–1792. URL: `https://doi.org/10.1016/j.jss.2009.06.035`. doi:`10.1016/J.JSS.2009.06.035`.

[81] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, Y. Le Traon, You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems, in: 2019 12th IEEE conference on software testing, validation and verification (ICST), IEEE, 2019, pp. 102–113.

[82] B. Rothenberg, O. Grumberg, Must fault localization for program repair, in: S. K. Lahiri, C. Wang (Eds.), Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24,

2020, Proceedings, Part II, volume 12225 of *Lecture Notes in Computer Science*, Springer, 2020, pp. 658–680. URL: `https://doi.org/10.1007/978-3-030-53291-8_33`. doi:`10.1007/978-3-030-53291-8\_33`.

[83] M. Jose, R. Majumdar, Bug-assist: Assisting fault localization in ANSI-C programs, in: G. Gopalakrishnan, S. Qadeer (Eds.), Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings, volume 6806 of *Lecture Notes in Computer Science*, Springer, 2011, pp. 504–509. URL: `https://doi.org/10.1007/978-3-642-22110-1_40`. doi:`10.1007/978-3-642-22110-1\_40`.

[84] J. K. Feser, S. Chaudhuri, I. Dillig, Synthesizing data structure transformations from input-output examples, in: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015, 2015, pp. 229–239.

[85] S. Lamraoui, S. Nakajima, A formula-based approach for automatic fault localization of imperative programs, in: S. Merz, J. Pang (Eds.), Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings, volume 8829 of *Lecture Notes in Computer Science*, Springer, 2014, pp. 251–266. URL: `https://doi.org/10.1007/978-3-319-11737-9_17`. doi:`10.1007/978-3-319-11737-9\_17`.

[86] A. Griesmayer, S. Staber, R. Bloem, Automated fault localization for C programs, in: R. Bloem, M. Roveri, F. Somenzi (Eds.), Proceedings of the Workshop on Verification and Debugging, V&D@FLoC 2006, Seattle, WA, USA, August 21, 2006, volume 174 of *Electronic Notes in Theoretical Computer Science*, Elsevier, 2006, pp. 95–111. URL: `https://doi.org/10.1016/j.entcs.2006.12.032`. doi:`10.1016/J.ENTCS.2006.12.032`.

[87] F. Wotawa, M. Nica, I. Moraru, Automated debugging based on a constraint model of the program and a test case, J. Log. Algebraic Methods Program. 81 (2012) 390–407. URL: `https://doi.org/10.1016/j.jlap.2012.03.002`. doi:`10.1016/J.JLAP.2012.03.002`.

[88] Y. Xie, A. Aiken, Scalable error detection using boolean satisfiability, in: J. Palsberg, M. Abadi (Eds.), Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005, ACM, 2005, pp.

351–363. URL: https://doi.org/10.1145/1040305.1040334. doi:10.1145/1040305.1040334.

[89] R. Könighofer, R. Bloem, Automated error localization and correction for imperative programs, in: P. Bjesse, A. Slobodová (Eds.), International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011, FMCAD Inc., 2011, pp. 91–100. URL: http://dl.acm.org/citation.cfm?id=2157671.