Automated Feedback Generation for Competition-Level Code

Jialu Zhang
Yale University
New Haven, Connecticut, USA

De Li
The MathWorks, Inc.
Natick, Massachusetts, USA

John C. Kolesar
Yale University
New Haven, Connecticut, USA

Hanyuan Shi
Independent Researcher
Hangzhou, Zhejiang, China

Ruzica Piskac
Yale University
New Haven, Connecticut, USA

ABSTRACT

Competitive programming has become a popular way for programmers to test their skills. Competition-level programming problems are challenging in nature, and participants often fail to solve the problem on their first attempt. Some online platforms for competitive programming allow programmers to practice on competition-level problems, and the standard feedback for an incorrect practice submission is the first test case that the submission fails. Often, the failed test case does not provide programmers with enough information to resolve the errors in their code, and they abandon the problem after making several more unsuccessful attempts.

We present Clef, the first data-driven tool that can generate feedback on competition-level code automatically by repairing programmers' incorrect submissions. The key development is that Clef can learn how to generate repairs for incorrect submissions by examining the repairs that other programmers made to their own submissions over time. Since the differences between an incorrect program and a correct program for the same task may be significant, we introduce a new data structure, *merge trees*, to capture the changes between submissions. Merge trees are versatile: they can encode both large algorithm-level redesigns and small statement-level alterations. We evaluated Clef on six real-world problems from Codeforces, the world's largest platform for competitive programming. Clef achieves 41.8% accuracy in repairing programmers' incorrect submissions. When given incorrect submissions from programmers who never found the solution to a problem on their own, Clef repairs the users' programs 34.1% of the time.

CCS CONCEPTS

• **Applied computing** → **Computer-assisted instruction**.

KEYWORDS

Automated feedback generation, competitive programming, programming education, program repair

ACM Reference Format:
Jialu Zhang, De Li, John C. Kolesar, Hanyuan Shi, and Ruzica Piskac. 2022. Automated Feedback Generation for Competition-Level Code. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3560425

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10.
https://doi.org/10.1145/3551349.3560425

1 INTRODUCTION

Competitive programming enjoys widespread popularity. The International Collegiate Programming Contest (ICPC), one of the most prestigious programming contests for college students, has been held annually for more than 50 years. Each year, more than 50,000 students from over 3,000 universities in over 100 countries compete for medals in the contest [7]. Moreover, competitive programming has had a significant impact in industry as well. Platforms such as Codeforces[1] and Topcoder[2] host large-scale online programming contests that attract millions of experienced programmers. Software companies view the finalists in the competitions as strong candidates for hiring since the finalists demonstrate solid algorithmic problem-solving skills and an outstanding ability to handle on-the-spot stress. Some software companies, such as Google [5], Meta [6], Microsoft [3], Yandex [8], and HP [2], even hold their own online programming contests for recruiting purposes [5].

In a programming competition, participants receive a description of a problem and a short list of sample tests illustrating how the program should behave. The participants develop solutions for the problem and submit them. The solutions are evaluated automatically on a number of different tests that are hidden from the participants. If a solution passes every test in the suite, it is accepted as correct. Competition-level problems are non-trivial: correct implementations sometimes require hundreds of lines of code, and the entire program needs to be efficient and bug-free. The competition platform's automatic tests can involve carefully-designed corner cases, along with hard limits on execution time and memory usage.

State-of-the-art feedback generation for programmers' incorrect submissions. Some competitive programming platforms allow programmers to practice on problems from past competitions. Currently, the standard response to a programmer's incorrect practice submission is simply to expose the first test case that the submission fails. Even with test case exposure for failures, many programmers still fail to solve the problem in the end, and they abandon the problem after several attempts. For the problems that we surveyed, 52.2% of the incorrect submissions had no correct follow-up submission. Forums can serve as a helpful source of feedback for programmers while they are practicing, but users who post questions on a forum have no guarantee of a timely response. Furthermore, the feedback from other forum users can be incomplete

[1] https://codeforces.com
[2] https://www.topcoder.com

or incorrect, and the users who post the questions might not ask for the right information in the first place because they misunderstand their own problems. A tool that repairs programs or provides useful feedback automatically would be a helpful alternative.

In recent years, researchers in the automated program repair community have worked on generating feedback automatically for intro-level programming assignments [18, 20, 40, 42–44]. State-of-the-art feedback generators use data-driven approaches. They all take advantage of databases of previously-written submissions to learn how to repair new incorrect submissions. Unfortunately, these tools target only problems from intro-level programming courses, and their feedback generation techniques do not suffice for competition-level problems. Two major differences exist between intro-level and competition-level programming:

- **The difficulty level of the problems.** Intro-level programming problems focus primarily on training programmers to use the features of a language correctly [19]. On the other hand, competition-level programming problems require programmers to understand complex natural-language descriptions, to master a wide range of algorithms and data structures, and to implement solutions that may involve hundreds of lines of code.
- **The evaluation metrics.** Intro-level programming problems usually do not have rigorous evaluation metrics. Restrictions on execution time and memory consumption are rare: generally, the test suites for intro-level problems only cover functional correctness. On the other hand, evaluation suites for competition-level problems perform rigorous checks on execution time and memory consumption. Any timeout or excessive memory usage causes the suite to mark a submission as incorrect even if the program behaves perfectly in terms of functional correctness.

Furthermore, state-of-the-art intro-level feedback generators suffer from a variety of weaknesses, including the inability to generate complex repairs [20, 40, 44], the tendency to enlarge programs excessively with repairs [18], and dependence on manual guidance from users [42]. Finding an effective automatic feedback generation method for competition-level code remains an open problem.

Problem 1475A from Codeforces illustrates some of the major differences between intro-level and competition-level problems. The input is an integer n ($2 < n < 10^{14}$), and the goal is to determine whether n has any odd divisor greater than one.[3] The execution time limit for Problem 1475A is two seconds per test, and the memory limit is 256 MB per test. One solution for the problem on Stack Overflow [1] performs an exhaustive search by iterating over every odd number in $(1, n)$ and checking whether n is divisible by it:

```
for (unsigned long long i=3; i<n; i+=2){
   if (n%i == 0) return true; //It has an odd divisor
}
return false; // n%i == 0 was never true so it doesn't
     have an odd divisor
```

This submission is syntactically and semantically correct. However, it fails to pass the evaluation suite: the test suite marks it as incorrect with "Time limit exceeded on test 2" as the feedback. Solving the problem correctly within the time limit requires a more efficient algorithm, and finding that efficient algorithm requires an important insight: the odd divisor problem reduces to checking whether n is a power of two. An efficient program for Problem 1475A right-shifts n repeatedly to remove trailing zeroes and then checks at the end that the remaining one is the only one in n:

```
while (!(n&1)) n >>= 1;
if (n==1) return false; else return true;
```

Problem 1475A presents a challenge for automated feedback generation tools. A submission that approaches the problem incorrectly may require a complete algorithm-level redesign, not just a small local repair. The automatic feedback provided by Codeforces did not help the programmer who wrote the exhaustive-search implementation to see that a completely different approach was necessary. Additionally, state-of-the-art tools [18, 20, 40, 42, 44] cannot make the repairs that the program needs because they view correctness only in terms of input-output correspondence, not efficiency.

Our approach: Clef. We introduce CLEF (Competition-Level Effective Feedback), a new tool that generates feedback automatically for competition-level problems by repairing errors in programmers' incorrect submissions. Clef learns the repair patterns that it uses by analyzing the repairs that other programmers made for the same problem across their submission histories. Clef applies the patterns that it learns to target programs outside the database to generate candidate repaired programs.

Main technical challenges in designing Clef. The main technical challenge for Clef is having an effective method for learning how programmers repair errors in their own submissions. The repair patterns that Clef needs to learn range from small statement-level fixes to algorithm-level control flow redesigns. Other data-driven feedback generators cannot alter the control flow of a program [18, 44], so large-scale algorithm-level changes, precisely the kind of changes that incorrect submissions for competition-level problems often require, are impossible for them to make. Clef employs two techniques that no other feedback generator has used previously:

- We introduce *merge trees*, a new data structure for encoding both statement-level and algorithm-level changes between incorrect programs and corrected versions of the same programs.
- We propose a new matching and repairing algorithm that takes advantage of similarities between the target program and programs in the database. With the new algorithm, Clef can repair incorrect submissions even if the errors in the submission have no exact matches in the database.

Evaluation. To evaluate our tool, we have run Clef on thousands of submissions for six real-world competitive programming problems obtained from Codeforces. Clef provides feedback successfully for 41.8% of the incorrect solutions overall. Whenever the database contains both incorrect submissions and a correct submission for an individual user, we have Clef attempt to fix the incorrect submissions without seeing the correct version, and then we compare Clef's repaired version to the real user's correct version in terms of program editing distance. In 40.6% of the cases, Clef generates a repaired program that is syntactically closer to the original incorrect submission than the user's own corrected version is. For the cases where a user made incorrect submissions but never made a

[3] https://codeforces.com/contest/1475/problem/A

correct submission, Clef repairs the user's incorrect submissions successfully 34.1% of the time.

In summary, we make the following contributions:

- We conduct a survey to assess the characteristics and challenges of competitive programming.
- We present a data-driven tool, Clef, that generates feedback for users' incorrect submissions automatically using its knowledge of how other users repair their own programs.
- We propose a new data structure for capturing both small and large changes in repaired submissions.
- We evaluate Clef on real-world competitive programming problems. For the incorrect submissions that were later repaired by the same user, Clef provides correct repairs 49.4% of the time. In 40.6% of these cases, Clef generates repaired programs that are closer to the original incorrect submission than the user's own correct submission. For the incorrect submissions that were never repaired by their authors, Clef provides correct repairs 34.1% of the time.

2 UNDERSTANDING COMPETITIVE PROGRAMMING

In this section, we present our survey of real-world competitive programming. We illustrate the challenges involved with solving competition-level problems through some concrete examples, and we discuss the implications that drive the design of Clef.

In a programming competition, a contestant writes a program to perform a specific task and submits the code to an online evaluation platform. The platform compiles the program, runs it on some predesigned test cases, and reports one of the following outcomes:

- **Accepted.** The submission produces the correct output for every test and never violates the time and memory limits.
- **Compile-Time Error.** The submission has a compilation error. Most programs with syntax errors fall into this group.
- **Runtime Error.** The submission encounters an error at runtime for a test. Common errors include buffer overflow and invalid array indices.
- **Time Limit Exceeded.** The program surpasses the execution time limit on a test.
- **Memory Limit Exceeded.** The program surpasses the memory usage limit on a test.
- **Wrong Answer.** The program returns an incorrect output for a test.
- **Other.** A non-deterministic error, such as a network outage.

The major sources of difficulty for competition-level problems are categorically different from the sources of difficulty for intro-level problems. The aim of a competition-level problem's design is not to teach contestants how to write programs but to push contestants to the limits of their knowledge. We now highlight some of the patterns in competition-level problems' designs.

Pattern 1: Challenging Problem Descriptions. The first step in solving a competition-level problem is converting the natural-language problem description into an idea for an algorithm. Intro-level programming problems generally have short, straightforward descriptions, but competition-level problems can have lengthy descriptions designed to mislead contestants. The length of a challenging problem description comes not from insignificant clutter but from complicated explanations of problem details meant to test how well programmers can bridge the gap between an end goal and an algorithm to accomplish it. For instance, consider Problem 405A from Codeforces:[4]

A box contains n columns of toy cubes arranged in a line. Initially, gravity pulls all of the cubes downward, but, after the cubes are settled in place, gravity switches to pulling them to the right side of the box instead. The input is the initial configuration of the cubes in the box, and the goal is to print the configuration of the box after gravity changes. The sample case example provided by Codeforces is shown in Subfigure 1a in Figure 1.

The prompt of Problem 405A is designed to test programmers' ability to reduce a complex problem to a well-known simple algorithm, namely sorting. Attempting to write a brute-force implementation that treats the cubes as distinct entities is a tedious and error-prone process. A key insight for solving the problem is the fact that, when gravity changes, the highest columns always appear at the right end of the box and are of the same height as the highest columns at the start. Subfigure 1b in Figure 1 illustrates this. The possibility of reducing the problem to sorting a one-dimensional array becomes clear after a programmer notices how the columns behave.

Pattern 2: Challenging Implementation Details. Not every competition-level programming problem is a simple task hidden behind a complex description. Often, implementing an effective algorithm for the problem is a genuinely difficult task involving minor details that are easy to mishandle. Problem 579A from Codeforces[5] is one such problem:

Start with an empty box. Each morning, you can put any number of bacteria into the box. Each night, every bacterium in the box will split into two bacteria. To get exactly x ($1 \le x \le 10^9$) bacteria in the box at some moment, what is the minimum number of bacteria you need to put into the box across some number of days?

An important detail to notice is the fact that every bacterium placed in the box will become 2^n bacteria after n days. What the problem really requires is an algorithm that can divide one integer into a sum of powers of two. A natural implementation for this algorithm is to count the number of ones that appear in the binary representation of x. Using the provided integer representation of x makes this easy, but a program that converts x into a binary string instead to count the number of ones can fall victim to certain errors

[4] https://codeforces.com/contest/405/problem/A
[5] https://codeforces.com/problemset/problem/579/A

(a) Original Sample Illustration (b) Required New Understanding

Figure 1: Subfigure 1a is the original sample illustration from Codeforces. The initial configuration of the cubes in the box appears on the left, and the final configuration appears on the right. The cubes whose positions change are highlighted in orange. The top cube of the first column falls to the top of the last column, the top cube of the second column falls to the top of the third column, and the middle cube of the first column falls to the top of the second column. Subfigure 1b shows the same example input but highlights a different detail. The tallest column at the end is of the same height as the tallest column at the start, but it appears at the right end of the box. The number of columns of a given height is preserved, so the two-dimensional gravity flip problem reduces to one-dimensional array sorting.

if implemented carelessly. String operations may misinterpret the base-2 string as a base-10 string, and this can lead to incorrect answers or even overflow errors.[6] To avoid overflow errors, a better program for Problem 579A never converts x into an alternative format. Instead, it operates directly on the binary representation of the integer. It right-shifts x repeatedly one bit at a time and counts the number of iterations where the right-shifted version of x is odd:

```
while (x > 0){
    if (x & 1)  r += 1;
    x >>= 1;
}
```

Pattern 3: Challenging Efficiency Requirements. For other problems, meeting the evaluation suite's efficiency requirements is the main source of difficulty. Problem 1475A, shown in Section 1, is an example of this.

3 MOTIVATING EXAMPLES

Effective feedback generation for competition-level code requires the ability to apply complex changes to incorrect submissions. This includes modifying programs' control flow and making major statement-level alterations. Along with the ability to perform complex modifications, high repair quality is another priority for Clef: it returns the smallest repairs that it can find. To illustrate the repair process that Clef follows, we use a number of real submissions for Problem 579A from Codeforces as examples. The prompt for the problem appears in the discussion of Pattern 2 in Section 2.

Incorrect program needs control flow changes. An example of a control flow modification that Clef applies appears in Figure 2. The original incorrect submission made by a user for Problem 579A appears in Subfigure 2a. The high-level design of the implementation is correct, but the control flow needs correction. Computing the number of ones in the binary representation of the integer `x` requires a loop rather than a conditional. Other users in the database repaired their programs by making a similar control flow change (converting `if` to `while`), so Clef applies the same repair in Subfigure 2b. In this situation, Clef generates a high-quality repair that not only passes all of the test cases but also makes minimal changes to the structure of the original incorrect program. The same user's own fix for the problem appears in Subfigure 2c. If we use the

[6] https://stackoverflow.com/questions/52548304/converting-decimal-to-binary-with-long-integer-values-c

```
if ((x/2)!=0)
{
  if ((x%2)==1)
    c++;
  x = x/2;
}
printf("%lld",c+1);
```

(a) Incorrect Program

```
while ((x/2)!=0)
{
  if ((x%2)==1)
    c++;
  x = x/2;
}
printf("%lld",c+1);
```

(b) Clef's Repair

```
while(x)
{
  if ((x%2)==1)
    c++;
  x = x/2;
}
printf("%lld",c);
```

(c) User's Repair

Figure 2: An example repair involving control flow modification. The differences between the programs are highlighted in red. The variable `x` is the input for the program, representing the desired number of bacteria to have in the end. The variable `c` is the output, the number of bacteria that need to be inserted.

```
while (x>0)
{
  if (x%2==0)
    u++;
  x = x/2;
}
printf("%d", u);
```

(a) Incorrect Program

```
while (x>0)
{
  if (x%2)
    u++;
  x = x/2;
}
printf("%d", u);
```

(b) Clef's Repair

```
while (x!=0)
{
  u += x%2;
  x /= 2;
}
printf("%d",u);
```

(c) User's Repair

Figure 3: A example repair involving a statement-level change. The differences between the programs are highlighted in red. The variable `x` is the input to this program, and `u` is the output.

Zhang-Shasha algorithm [51] to measure tree edit distances, the repair generated by Clef has a distance of 1 from the original flawed program, whereas the user's own repair has a distance of 6 from the original program.

Incorrect program needs statement-level changes. In addition to making algorithm-level control flow changes, Clef is able to generate repairs that require small statement-level changes. Figure 3 shows an example of a statement-level repair. The control flow in the original submission is correct, but the guard in the `if` statement contains a numerical error. The repair that Clef produces for the submission appears in Subfigure 3b. The Zhang-Shasha algorithm gives the new program generated by Clef a tree edit distance of

only 2 from the original program. Although this repair is not the smallest possible repair, which would be changing `x%2==0` to `x%2==1`, Clef still generates a repair that is closer to the user's original incorrect submission than the user's own repaired program is. The user's own correction of the program involves three major changes: changing the guard in the `while` statement, removing the `if` statement inside the while loop, and computing the output variable `u` differently by adding the remainder of `x` mod 2 to it in each loop iteration.

4 SYSTEM DESCRIPTION

We design and build Clef, a tool that can generate repairs for competition-level code automatically by learning the ways that users repair their own programs. Figure 4 gives an overview of Clef's architecture. It consists of three main modules: (1) The preprocessor, described in Section 4.1, takes the database programs as input, parses them, and generates abstract syntax trees for them. (2) The pattern learner, described in Section 4.2, uses a new data structure, *merge trees*, to represent the algorithm-level and statement-level changes that users in the database apply to their own programs over time. (3) The repair generator, described in Section 4.3, applies program transformation patterns to the incorrect target program to generate repair candidates, and it also validates the candidates with the provided test suite for the problem.

4.1 Preprocessor

The preprocessor parses all of the programs in the database into ASTs offline for later use in repair pattern learning. The preprocessor groups the program ASTs into pairs of the form (i, c), where i and c are ASTs for an incorrect program and a correct program, respectively, written by the same user. If a user made multiple incorrect or correct submissions, the preprocessor makes program pairs for all of the possible combinations. It also discards programs that have syntax errors in this stage.

4.2 Pattern Learner

After obtaining the program pairs from the preprocessor, Clef produces a collection of program transformation patterns based on the changes between the incorrect and correct programs. The program transformation patterns fall into two categories: additions and deletions. Clef uses a merge tree to represent the AST changes that occur between the incorrect and correct versions of a program.

Merge trees. A merge tree encodes the differences between two abstract syntax trees. An example merge tree appears in Figure 5. The main structure of a merge tree resembles the unchanged parts of the two ASTs being compared (node `0`, node `1` and node `2`) , but the merge tree also includes special nodes that represent additions (node `5`) and deletions of sub-trees (node `3` and node `4`).

An important characteristic of merge trees is their generality: they can match a variety of patterns in ordinary ASTs rather than only a single pattern. For example, if the new incorrect version of a program contains a node `3` where the correct version contains a different node `5`, the merge tree for the transformation can apply to ASTs that contain `3`, `5`, `(3;5)`, `(5;3)`, or neither statement, as long as the surrounding parts of the AST bear a sufficiently close resemblance to the merge tree. In contrast, a simpler encoding of the transformation [40] would only apply to ASTs that contain `3`. The merge tree's ability to be applied to any program that contains a combination of the two statements allows it to cover a much larger range of programs than a simpler encoding does.

Computing program differences. The standard approach for identifying differences between two programs is to apply the Zhang-Shasha algorithm directly [51] to compute the edits needed to convert one AST into the other. Multiple state-of-the-art intro-level feedback generators follow this approach [40, 44]. However, the Zhang-Shasha algorithm on its own is not the best method for computing program differences for competition-level code. First, the Zhang-Shasha algorithm runs in $O(m^2n^2)$ time, where m and n are the numbers of nodes in the two input trees. A faster alternative that involves flattening is possible, as we will explain later.

The second and more important reason for not using the Zhang-Shasha algorithm on full program ASTs is that the algorithm treats every node in a tree as having equal weight. The Zhang-Shasha algorithm is a general-purpose algorithm for trees of any kind, not just program ASTs, so it pays no attention to the semantic significance of the edits it uses for measuring distance. In the case of program ASTs, not all nodes deserve equal weight: some node modifications are more significant than others. For example, changing an `if` node to a `while` node generally qualifies as a major change because it alters the control flow of a program. A method for measuring the edit distance between two programs should count such a control flow change as having a higher impact than changing an `x=1` statement to `x=0`. When we compute tree edit distances, we assign a higher cost to control flow edits than to other changes.

Algorithm 1 Learning Program Transformations

Input: P_i : User's incorrect program submission (AST).
Input: P_c : User's correct program submission (AST).
Output: *patternPool* : A set of program transformation patterns that reflect the changes that users made in repairing their own programs.

```
1:  procedure learnTransformation(P_i, P_c)
2:      patternPool = []
3:      alignedCF, unmatchedCF := ControlFlowAlign(P_i, P_c)
4:      for (T_i, T_c) in alignedCF do
5:          flatAST_i, flatAST_c := Flatten(T_i, T_c)
6:          edits := Zhang-Shasha(flatAST_i, flatAST_c)
7:          mergeTree += merge(edits, flatAST_i, flatAST_c)
8:      for (deletedCF_i) in unmatchedCF do
9:          mergeTree := augment(deletedCF_i, mergeTree)
10:     patternPool += mergeTree
        return patternPool
```

For Clef, we designed a custom algorithm, shown as Algorithm 1, that computes the program transformation patterns between the incorrect and correct versions of a program. To detect algorithm-level changes between the two versions, Clef uses top-down control flow alignment. The nodes that count as control flow nodes for our purposes are `if`, `while`, and `for` statements as well as function calls. Two control flow nodes align if they have the same type (i.e. they are both `if`, both `while`, both `for`, or both function calls)

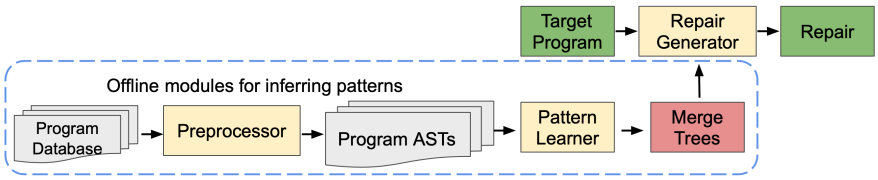

Figure 4: Clef overview. The green blocks are the input that Clef receives from users and the output that it provides for them. The yellow blocks are the key modules of Clef.

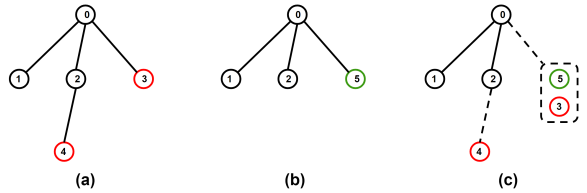

Figure 5: An example of a merge tree. Subfigure (a) is an incorrect program, and subfigure (b) is its correct version. Subfigure (c) is the merge tree for the two. Each small circle represents one AST node. The red nodes are deleted, the green node is inserted, and the black nodes remain the same after the program transformation.

and they satisfy some extra type-specific conditions for alignment. Two `if` statements need to have matching guards or matching true and false branches. Two `while` or `for` statements need to have matching guards or matching bodies. Two function calls need to have all of their arguments match. For two sub-expressions to match in any case, they need to be mostly the same. Variable names and function names are not required to be the same, but values of variables and numbers and types of parameters must be. Clef identifies all of the pairs of control flow nodes that align with each other and merges their sub-trees.

Control flow nodes in the incorrect program with no matches in the correct program are regarded as deletions. Correspondingly, control flow nodes in the correct program with no matches in the incorrect program are regarded as insertions. Clef uses special nodes as markers in merge trees to represent these deletions and insertions. At the end of pattern learning, Clef returns a set of all the merge trees it generated to use as program transformation patterns.

A minor detail that makes merge tree generation more efficient is the fact that the merge tree for a control flow node does not cover changes in the sub-trees of the two versions' control flow nodes. We leave the sub-trees for other merge trees to cover. Clef handles program changes within the current control flow node by flattening all of the control flow nodes inside its sub-trees and treating the interior as an empty node. To simplify the process of computing edits, we run the Zhang-Shasha algorithm only on pairs of these flattened sub-trees rather than on the full original ASTs of the incorrect and correct programs. If m and n are the numbers of nodes of all types in the two input trees and p and q are the numbers of control flow nodes in the two input trees, then this flattening reduces the time complexity of merge tree generation from $O(m^2n^2)$ to $O(\frac{m^2n^2}{p^2q^2})$.

After implementing Clef, we conducted a study of the performance gain that comes from our optimization of the Zhang-Shasha algorithm for merge tree generation. We ran Clef on our full evaluation suite twice over, once with our optimization of the Zhang-Shasha algorithm in place and once without it. We found that, with our optimization in place, the overall running time of Clef was about 5% faster on average than it was without the optimization.

4.3 Repair Generator

The repair generator takes the incorrect target program and the set of merge trees as input, and it returns a repaired version of the target program. The repair generator's algorithm consists of three main steps. First, it converts the incorrect target program into an AST just as the preprocessor described in Section 4.1 does for the database programs. Second, the repair generator identifies merge trees that match the target program and produces candidate repaired programs by applying transformations based on the merge trees that match the target program. During this step, variable usage analysis helps with the removal of spurious candidate programs. Third, the repair generator validates the candidate programs with the pre-defined test suite for the problem.

Matching the target program with merge trees. Merge trees represent the changes that programmers made to their own programs in the database. The goal of the matching process is to apply similar program edits to repair the target program. Intuitively, the repair generator takes advantage of the similarities between the incorrect target program and the merge trees to generate repairs.

To start, the repair generator analyzes the target program's AST and identifies the sub-trees that have a control flow node as their root. Such a sub-tree matches a merge tree if all of the nodes and edges of the sub-tree are contained within the merge tree. We allow the names of variables and functions to be different for different programs, but the initial values of variables (if applicable) and the types and parameters of functions are required to be the same for a match. If the repair generator finds a match between the target program and a merge tree, it can modify the target program by performing a repair based on the merge tree. The fact that matching only requires the merge tree to contain the sub-tree and not the other way around helps the repair generator to find ways to fix incorrect submissions in situations where the errors in the submission have no exact matches in the database. Clef generates a modified version of the target program by replacing the matched sub-tree with a new sub-tree based on the merge tree's transformation.

Replacing a sub-tree using the merge tree's transformation might introduce usages of undefined variables. To account for this, the repair generator tries conservatively fitting different combinations

of defined variable names onto the undefined variables that are inserted. Then it performs variable usage analysis on modified versions of the target program to remove candidates that are invalid simply because of their variable usage. The repair generator discards candidate programs that still contain undefined variables after variable alignment or define variables without using them. This filtering reduces the number of candidates to be validated with the test suite, improving the performance of the repair generator.

Validation. The repair generator validates candidate programs simply by running the provided test suite on them. As soon as the repair generator finds a candidate program that passes all of the tests, it returns that candidate program as output.

Because small repairs are more beneficial for users, we prioritize candidates with small transformations over candidates with large ones for validation. The repair generator starts by applying only one merge tree to the target program at a time to generate candidates. If we fail to find a valid candidate program after trying every option among the individual merge trees, the repair generator begins creating candidate programs from combinations of multiple transformations. The repair generator continues trying progressively larger repairs until it hits a timeout or the number of test suite runs reaches a preset limit. For our evaluation, we do not impose a time limit, but we set a limit of 1,000 on the number of candidate programs to validate with the test suite.

5 EVALUATION

We answer the following questions with our evaluation of Clef:

- How effectively can Clef repair incorrect submissions for competition-level problems?
- How high is the quality of Clef's feedback? More specifically, how closely do the repaired programs that Clef generates resemble users' original programs?
- How does Clef's repair rate on competition-level problems compare to the repair rates of similar tools?

5.1 Implementation and Experimental Setup

Our implementation of Clef uses a mix of Python and open-source software libraries. As it is now, Clef operates on C programs. We rely on PYCPARSER [4], a complete C99 parser, to convert C programs into abstract syntax trees. Also, we use the Zhang-Shasha algorithm [50] to compute tree edit distances.

Benchmark Setup. For our evaluation suite, we use six problems from Codeforces, the world's largest online platform for competitive programming. Codeforces assigns difficulty scores to its problems, and we group the problems into three categories based on their difficulty scores. We categorize problems with a difficulty score of 800 or less as *easy*, problems with a score between 800 and 1000 as *medium*, and problems with a score of at least 1000 as *hard*. Each of the six problems that we selected received more than 700 submissions written in C. (Multiple submissions made by the same user count as distinct.) The number of evaluated problems and the numbers of submissions for each are on par with existing work[18, 20, 43]. For each submission, we collected not only the text of the program but also its execution result, running time, and memory usage.[7] Additionally, we have access to the test suite used by Codeforces for each of the problems. Table 1 names the six problems and the specific challenges that each problem presents.

Table 2 provides a breakdown of the six selected competition-level programming problems. We can see from analyzing the execution results for the database programs that 25.6% (2093/8187) of the submissions were rejected because of errors rather than incorrect outputs. Among those, 4.4% (359/8187) of the programs were classified as incorrect because of runtime errors, time limit violations, or memory limit violations. Furthermore, of the 5597 incorrect submissions, 2921 (52.2%) come from programmers who never made a correct submission.

Clef aims to provide effective feedback for programmers practicing on competition-level problems. To assess whether Clef meets this goal, we split the users into two groups for each problem:

- **Group One.** Some users made incorrect submissions but never managed to produce a correct submission. Generating repairs for these users' programs is generally a challenging task: since the users never managed to repair their own programs, their submissions may contain major errors.
- **Group Two.** Other users made incorrect submissions initially but then managed to produce a correct submission afterward on their own. These users' incorrect submissions are easier to handle in general: the fact that the users found a solution eventually means that they were likely close to a right answer with their earlier incorrect submissions.

For each problem, we perform some cleaning of the database before we use it as a training set. To clean the database, we discard all programs with syntax errors and all submissions from users who solved the problem on their first attempt. Next, we label each user with a distinct anonymous identifier. After that, we allocate 80% of the users for the training set and 20% for the evaluation set. Some existing feedback generators use the chronologically earliest 80% of submissions as the training set and the remaining 20% as the evaluation set [37], but we divide the users at random instead to avoid skewing the results. The setup of Codeforces makes problems easier for programmers who submit their programs later: on Codeforces, users practicing on a specific problem can view every other user's submission history for the same problem. Consequently, there is a risk that programmers who submitted later fixed the mistakes in their code by copying someone else's correct submission rather than by finding a solution on their own. Grouping the users randomly rather than chronologically allows us to distribute the users who copied other users' submissions more fairly between the training set and the evaluation set, if there are any such users.

5.2 Results

Group One. Table 3 shows Clef's results for incorrect programs abandoned by their authors. For this group, Clef has an overall fix rate of 34.1% across the six problems. Since the programs' authors never addressed their mistakes fully on their own, they would benefit from receiving repaired versions of their programs as feedback. To measure the quality of our repairs for Group One, we introduce a new metric: the *dissimilarity* between each target program and

[7] The owners of Codeforces gave us permission to collect data from the six problems.

Table 1: Six Representative Competition-Level Programming Problems.

Problem ID	Difficulty Level	Codeforces Tag	Challenges	Problem Description
1312A	Easy	Number Theory	Description	Given two integers n and m, determine whether a convex regular polygon with m sides can be inscribed in a convex regular polygon with n sides such that their centers and vertices coincide
1519B		Math	Description	Given an n-by-m grid, with different costs for moving in different directions, check whether it is possible to reach cell (n, m) with exactly cost k
1238A	Medium	Number Theory	Algo Design & Implementation	Given two integers x and y, determine whether there is a prime integer p such that subtracting p from x any number of times makes x equal to y
1295A		Greedy	Description & Algo Design	Find the largest integer that can be shown on a seven-segment (digital clock) display that requires no more than n segments to be turned on in total
579A	Hard	Bit Mask	Algo Design & Implementation	Find the minimum number of bacteria that need to be placed into a box over some number of days in order to have x bacteria in the box at some moment
1199B		Geometry	Algo Design & Implementation	Find the depth of a body of water given the distance that a vertical line segment extending from the bottom can tilt before being submerged

Table 2: Statistics for the six selected competition-level programming problems. The categorizations for submissions here come from Codeforces. AC: Accepted, WA: Wrong Answer, CE: Compile-Time Error, RE: Runtime Error, TLE: Time Limit Exceeded, MLE: Memory Limit Exceeded, OT: Other.

Problem ID	# Submissions	# AC	# WA	# CE	# RE	# TLE	# MLE	# OT	Average LOC
1312A	1160	494	327	301	15	19	3	1	21.1
1519B	724	349	211	137	12	14	1	0	25.1
1238A	1345	303	520	358	39	121	0	4	27.3
1295A	1024	251	349	368	13	40	3	0	33.3
579A	1889	780	758	288	23	36	1	3	19.8
1199B	2045	413	1339	269	18	1	0	5	11.9
Total	8187	2590	3504	1721	120	231	8	13	21.4

the correct programs in the database. To measure dissimilarity, we compute the minimum tree edit distance between a target program and any of the correct programs in the database using the Zhang-Shasha algorithm, and then we divide this distance by the size of the target program. This dissimilarity metric quantifies the difficulty of repairing each program: if a program is not syntactically close to any correct program in the database, generating a repair for it is difficult, and any repairs found are likely to be large.

Clef generates high-quality repairs for Group One according to our standard. We define the *relative repair size* for a problem as the tree edit distance between a repaired program and its target program divided by the size of the target program. For three of the six problems, Clef has an average relative repair size smaller than the average dissimilarity between the target programs and the correct programs. The high dissimilarity values for Group One make the higher average relative repair sizes for the other three problems understandable. Another important finding is that the average dissimilarity across all six problems is 0.38, so a typical target program needs a large portion of its code to be changed to become identical to any of the correct database programs.

Group Two. Table 4 shows Clef's results for incorrect programs later fixed by their authors. Clef has an overall fix rate of 49.4% across the six problems, which is better than the result for Group One. Since we have the authors' own repairs for the programs in Group Two, we use the authors' repairs as the ground truth for assessing the quality of Clef's feedback. A repair generated by Clef counts as a *high-quality repair* if the tree edit distance between it and the target program is smaller than the tree edit distance between the user's own repair and the target program. For four out of the six problems, more than 50% of the repairs Clef generates are closer to the target program than the ground truth is, so Clef does in fact generate high-quality repairs for programs in Group Two.

5.3 Comparison with State-of-the-Art: Clara

To the best of our knowledge, there is no feedback generator for competition-level problems, so we compare Clef with a feedback generator for intro-level programming assignments instead. Among the state-of-the-art data-driven feedback generators for intro-level problems[18, 43, 44], we selected Clara as our baseline because Clara is the only state-of-the-art feedback generator that is publicly available[8] and operates on C programs [18].

Clara's input format is relatively restricted, so we needed to translate every target program from Codeforces manually into a format that Clara can accept. We could not perform these translations automatically because there is too much syntactic variation among the original programs. An example translation appears in Figure 6. If a program contains an outer loop, we remove the loop since many common syntactic patterns for loops cause Clara to run into errors. Because these loop removals are necessary, we have the programs take a single line of input rather than a multi-line input of variable length. For the same reason, we replace all uses of `scanf` for reading inputs with function arguments. Apart from the changes relating to loops and inputs, we kept the reformulated programs' semantics as close to the semantics of the originals as

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

Table 3: Evaluation of Clef on incorrect programs abandoned by their authors (Group One). The second column shows the number of incorrect-correct program pairs in the training set for each problem, not the number of individual programs. The penultimate column shows the average dissimilarity of the target programs that had a successful repair by Clef. The last column shows the average dissimilarity of the target programs that Clef failed to generate a successful repair.

Problem ID	# Pairs in Training Set	# Programs in Test Set	# Programs Repaired	Accuracy (Repair Rate)	Avg. Relative Repair Size	Average Dissimilarity	Avg. Dissimilarity for Successes	Avg. Dissimilarity for Failures
1312A	475	90	51	56.7%	0.24	0.26	0.16	0.39
1519B	203	31	12	38.7%	0.38	0.43	0.27	0.53
1238A	1304	316	119	37.7%	0.26	0.40	0.27	0.48
1295A	277	127	38	29.9%	0.86	0.56	0.55	0.57
579A	1654	362	98	27.1%	0.73	0.44	0.41	0.45
1199B	4027	558	82	14.7%	0.15	0.19	0.06	0.21

Table 4: Evaluation of Clef on incorrect programs later repaired by their authors (Group Two). The training set for each problem is the same as it is in Table 3.

Problem ID	# Programs in Test Set	# Programs Repaired	Accuracy (Repair Rate)	High-Quality Repairs	Avg. Relative Repair Size	Average Dissimilarity	Avg. Dissimilarity for Successes	Avg. Dissimilarity for Failures
1312A	62	44	71.0%	52.3%	0.19	0.17	0.14	0.21
1519B	71	55	77.5%	7.3%	0.33	0.19	0.12	0.40
1238A	55	34	61.8%	58.8%	0.29	0.34	0.24	0.50
1295A	53	22	41.5%	13.6%	0.54	0.38	0.37	0.39
579A	107	25	23.4%	52.0%	0.53	0.40	0.38	0.41
1199B	176	37	21.0%	59.5%	0.21	0.15	0.08	0.17

```
int main() {
int i, t, m, n;
scanf("%d",&t);
for (i = 0; i < t; i++) {
    scanf("%d%d",&n,&m);
    if (n%m == 0)
        printf("YES");
    else
        printf("NO");
}
return 0;
}
```

(a) Clef's Input

```
int main(int n, int m) {
if (n%m == 0)
    printf("YES");
else
    printf("NO");
return 0;
}
```

(b) Clara's Input

Figure 6: An example of necessary reformulation.

possible. To compare Clara and Clef, we ran Clara on the reformulated programs, applied Clara's suggested edits manually, and then passed the programs back to Codeforces to evaluate their correctness. The programs passed to Codeforces have Clara's suggested edits applied but also have our reformulations undone. More specifically, the programs passed to Codeforces have all of the original loops and take their arguments with `scanf`, but they have Clara's suggested edits applied to the main loop body. We collected and reformulated the correct programs in our training set to build the training set for Clara.

Table 5 shows the results of the basic comparison between Clef and Clara. For the comparison, we compare only the repair rates of the two tools. There is no meaningful way to compare the running times of the tools because Clef applies its repairs automatically, whereas Clara simply suggests repairs without applying them itself. Also, we cannot give a fair comparison of the two tools' repair sizes either. Since Clara operates on a database of reformulated

Table 5: Basic Comparison: Clef against Clara

	Problem ID	# Programs	Clara Accuracy (Repair Rate)	Clef Accuracy (Repair Rate)	Improvement
Group One	1312A	90	43.3%	56.7%	30.9%
	1519B	31	29.0%	38.7%	33.4%
Group Two	1312A	62	48.4%	71.0%	46.7%
	1519B	71	49.3%	77.5%	57.2%
	1238A	55	7.3%	61.8%	7x
	1295A	53	7.5%	41.5%	5x

programs that are smaller than the ones that Clef uses directly, Clara is necessarily limited to a smaller range of possible edits than Clef is.

We use two easy problems and two medium problems for the comparison. For the two easy problems, we tested Clef and Clara on submissions from both Group One and Group Two, but for the medium problems, we used submissions from Group Two only. For the two easy problems, 1312A and 1519B, Clef outperforms Clara by at least 30% in repair accuracy in both Group One and Group Two. For the easy problems, we found that the main reason why Clara fails to generate a repair is the tool's limited language feature support. Language features that cause Clara to fail include structs, `Typedef`, simultaneous array declaration and initialization (such as a[100]={0}), and some C functions such as `strrev`.

For the two medium problems, 1238A and 1295A, Clef outperforms Clara by a wide margin (500% or more). Because of the significant manual effort required for reformulation and validation, along with the fact that Clef enjoys such a large performance gain over Clara on the easy and medium problems, we chose not to evaluate Clef and Clara on the full suite that we used for Clef alone.

We investigated why Clara fails to generate repairs for medium problems. Again, Clara suffers from limited language feature support. For example, for Problem 1238A,[9] Clara does not distinguish between the types `int` and `long long int`. However, the distinction is crucial because the inputs may be as large as 10^{18} in this problem. There are incorrect submissions that fail solely because they use `int` to take inputs. Clara cannot generate the feedback that these programs need, namely a change to a variable's type.

Multiple factors prevent us from performing a full direct comparison. First, we cannot evaluate Clara directly on the same suite of programs from Codeforces that we used for our evaluation because Clara does not support the full range of C's syntax. For example, Clara cannot translate loop guards of the form `while(t--)` into its internal intermediate representation language. We found that, if Clara encounters such a loop guard, it may enter an infinite loop and time out after approximately five seconds.[10] The programs that we collected from Codeforces for problems such as Problem 1312A and Problem 1519B use loop guards of this form extensively, usually to iterate through input lines. We believe that this bug is a small implementation mistake rather than a fundamental limitation of Clara. Nevertheless, it prevents us from having a fair comparison of Clara and Clef on the same programs.

Another source of difficulty is the fact that Clara does not apply program edits on its own automatically. Instead, it provides feedback in the form of a list of edits for the programmer to make, expressed in Clara's low-level intermediate representation. To evaluate the correctness of Clara's feedback, we needed to apply the suggested edits to each target program manually.

Overall, our comparison between Clara and Clef highlights the major differences between intro-level and competitive programming. Programming contests test programmers' knowledge with language-specific issues, such as numerical types' ranges in Problem 1238A, while intro-level programming avoids such material.

5.4 Efficiency

Although efficiency is not our primary goal, we conduct an efficiency analysis to assess the running time of Clef. For the efficiency tests, we run Clef on a MacBook Pro with an Intel i7 CPU and 16GB of memory. Table 6 shows that Clef generates repairs for incorrect programs efficiently. For each of the problems in our main evaluation, we report an average running time based on a sample of 15 runs covering both successful and failed runs. On average, across the six problems, Clef takes around three minutes to run.

Three minutes may seem like a relatively long running time at first glance, but, given the setup of practice sessions on platforms like Codeforces, a three-minute delay for feedback generation is not a significant impediment to the utility of Clef. Codeforces practice sessions generally have a two-hour time limit and consist of four to six problems each,[11] so users have twenty minutes on average to solve an individual problem. Programmers can receive feedback for a problem multiple times over within that duration. This can be especially helpful for programmers who would abandon the problem otherwise after making a few failed attempts.

[9] https://codeforces.com/problemset/problem/1238/A
[10] https://github.com/iradicek/clara/issues/31
[11] https://codeforces.com/contests

Table 6: Clef Average Running Times

Problem ID	Average Running Time (s)
1312A	58
1519B	60
1238A	150
1295A	242
579A	53
1199B	26

6 DISCUSSION

State-of-the-art tools. Recent data-driven approaches for feedback generation utilize the wisdom of the crowd by selecting *donor programs* from their databases [18, 20, 40, 44]. A donor program is a program that bears a close resemblance to the program to be repaired. Tools that use donors repair their target programs by analyzing the differences between a target program and its donors.

State-of-the-art feedback generators choose their donor programs from a database of programs that are either all correct or all incorrect. Both options have limitations. Tools that draw their donors from databases of correct programs [18, 20, 44] operate under the faulty assumption that the target program differs from the correct database programs only because of the presence of errors in the program. Tools that draw their donors from databases of incorrect programs [40] suffer from low success rates because the mistakes in the donor programs are unlikely to coincide with the mistakes in the target program.

Clef takes a different approach. Its database includes both correct programs and incorrect programs, and it draws information from both sides of the database to produce merge trees that function like donor programs do for other tools. Merge trees offer a unique advantage over donor programs: they allow Clef to generate high-quality repairs in a way that mimics the debugging procedure that human programmers follow. The structure of a merge tree represents the changes that a user makes between the different versions of a program. Clef can learn to imitate the user's behavior by observing the differences between an early incorrect version of the program and the final correct version of the program.

Running Clef on Clara's evaluation suite. Just as Clara cannot run directly on Clef's evaluation suite, Clef cannot run on Clara's evaluation suite either. The database of programs used for Clara's original evaluation is not publicly available. Even if we did have access to Clara's target programs and training data, compatibility issues would still prevent us from evaluating Clef on Clara's suite. Clara's training set contains only finished correct programs, and the incorrect target programs are stored only as isolated individual submissions. Without any version histories for training, Clef cannot generate fixes because it cannot observe the changes that occur between the incorrect and correct versions of a program.

Threats to Validity. Clef validates candidate programs by running the test suite provided by Codeforces on them. Passing every test within the resource limits is not a guarantee of the correctness of a program but only a highly likely indicator of its correctness. A guarantee would require formal verification, which we do not perform. To our knowledge, this limitation is common to all existing data-driven feedback generation techniques [18, 20, 30, 40, 44].

Currently, Clef only supports feedback generation for C programs. However, the principles behind the design of Clef are applicable to programs written in any language. Our method for handling control flow nodes does assume C-like syntax, but nothing else about the underlying algorithm is tailored specifically for C.

7 RELATED WORK

Competitive programming. Researchers have devoted an increasing amount of attention to competitive programming recently because of its growing impact on programming training and education [12, 23, 38]. The first tool to generate solutions for programming problems with program synthesis comes from Zavershynskyi *et al.* [48]. Unfortunately, the tool's utility is limited significantly by the fact that it generates solutions only in a custom-made intermediate programming language. Hendrycks *et al.* [19] are the first to use large language models to generate solutions for competitive programming problems directly in Python. However, their approach produces solutions successfully less than 20% of the time. AlphaCode [27] is a significant improvement over the state of the art [14, 19, 48]. AlphaCode produces programs based on natural-language descriptions it receives as input. In contests with over 5,000 participants, AlphaCode places among the top 54.3% of participants on average [27]. In spite of the advances made in the field of competitive programming, no existing tool generates feedback or repairs for incorrect competition-level programs.

Automated feedback generation. Automatic feedback generation for programming assignments has been a popular topic in programming education over the last decade [15, 18, 20, 21, 26, 30, 35–37, 40, 42–45, 47, 52]. The first tools developed for the task [21, 42] rely on manual guidance from users, in the form of either reference solutions [21, 42] or an error model [42] that explicitly defines all repairs that the tool can make. Because of their heavy reliance on input from users, the early tools do not qualify as fully automatic.

More recent feedback generators are automatic, and they rely on data-driven approaches for the task. They learn how to generate repairs for programs by analyzing programs written by other users. Tools such as Clara [18], SARFGEN [44], Refactory [20], FAPR [30], and Cafe [43] use databases of existing correct solutions for a problem to learn how to repair incorrect programs written for the same problem. Some of the data-driven tools are limited by their heavy dependence on syntactic similarities between the target program and reference solutions from the database. Two of the tools for imperative languages cannot repair a flawed program unless their database contains a correct program with exactly the same control flow as the flawed program [18, 44]. Similarly, one of the tools for functional programs requires alignment for function call sites [43]. Multiple studies have shown that the assumption that a flawed program will have an exact control-flow match in the database of correct programs is too strong to be reliable [20, 25]. Other feedback generators suffer from different problems, such as the tendency to enlarge programs excessively with repairs [18, 20], the inability to fix errors that require changes to multiple parts of a program [40], and algorithms that ignore programs' semantics [30, 46].

Furthermore, state-of-the-art feedback generators [18, 43, 44] cannot generate the complex repairs that flawed competition-level programs need because the tools' creators designed them with intro-level programming assignments in mind. No existing tool can repair programs that require an algorithm-level redesign, but merge trees allow Clef to handle the task. The inspiration behind our usage of merge trees comes from algorithms for semi-structured merging [10, 13]. More importantly, no existing feedback generator attempts to make programs more efficient.

Automated program repair. Researchers have studied automated program repair techniques extensively for the past sixty years [11, 16, 17, 49]. Automated program repair techniques fall into three main categories: heuristic-based [22, 24, 39], semantics-based [31, 32], and learning-based [28, 29]. Heuristic-based approaches use some heuristic, such as genetic programming [24], randomization [39], or a predefined fitness function [22], to guide a search procedure to candidate patches for a program. Semantics-based techniques [9, 31, 32] combine symbolic execution with SMT solvers to synthesize repairs. Semantics-based techniques struggle to repair competition-level code reliably because of the limitations of their internal design. Programming competitions make heavy use of floating-point numbers for geometry problems and lists for string operation problems, both of which are difficult for SMT solvers to handle effectively. Lastly, learning-based techniques [28, 29, 41] learn code repair patterns from prior patches.

State-of-the-art automated program repair techniques work best when used to handle a small number of errors among millions of lines of code. However, for competition-level code, errors appear much more frequently relative to the size of users' programs.

Automatic repair for non-functional program properties (i.e. time and memory usage) has received a small amount of attention from researchers previously. However, unlike Clef, prior work in the area has targeted only specific program patterns [16, 17], such as unnecessary loop iterations [34] or repeated computations of the same value [33]. No prior research on the subject aims to improve the efficiency of competition-level code automatically.

8 CONCLUSION

We present Clef, a tool that generates feedback automatically for competition-level code. By observing how other users repair their own programs over time, Clef learns how to create repairs for its target programs. The improvement in quality that Clef provides over the standard feedback that programmers receive when practicing on competition-level problems will make online programming platforms that utilize Clef more user-friendly.

ACKNOWLEDGMENTS

We thank Andong Fan for valuable early feedback and Matt Elacqua for proofreading. Jialu Zhang is supported in part by NSF grants CCF-1715387 and CCF-2106845. John C. Kolesar is supported in part by NSF grant CNS-1565208. Ruzica Piskac is supported in part by NSF grants CCF-2131476 and CNS-1565208.

REFERENCES

[1] 2022. An example of repairing a faulty submission that need an algorithm-level redesign. https://stackoverflow.com/questions/65896295/finding-odd-divisors-with-bit-shifting
[2] 2022. HPE CODEWARS. https://hpecodewars.org/.
[3] 2022. Microsoft Imagine Cup. https://imaginecup.com/.
[4] 2022. pycparser: Complete C99 parser in pure Python. https://github.com/eliben/pycparser.

[5] 2022. The 10 Most Prestigious Programming Contests and Coding Challenges. https://www.mycplus.com/featured-articles/programming-contests-and-challenges/.

[6] 2022. The Facebook Hacker Cup. https://www.facebook.com/codingcompetitions/hacker-cup/.

[7] 2022. The International Collegiate Programming Contest. https://icpc.global/.

[8] 2022. The Yandex Algorithm Cup. https://yandex.com/cup/algorithm/.

[9] Umair Z. Ahmed, Zhiyu Fan, Jooyong Yi, Omar I. Al-Bataineh, and Abhik Roychoudhury. 2022. Verifix: Verified Repair of Programming Assignments. *ACM Trans. Softw. Eng. Methodol.* 31, 4, Article 74 (jul 2022), 31 pages. https://doi.org/10.1145/3510418

[10] Sven Apel, Jörg Liebig, Benjamin Brandl, Christian Lengauer, and Christian Kästner. 2011. Semistructured Merge: Rethinking Merge in Revision Control Systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 190–200. https://doi.org/10.1145/2025113.2025141

[11] Rohan Bavishi, Harshit Joshi, José Pablo Cambronero Sánchez, Anna Fariha, Sumit Gulwani, Vu Le, Ivan Radicek, and Ashish Tiwari. 2022. Neurosymbolic Repair for Low-Code Formula Languages. https://doi.org/10.48550/ARXIV.2207.11765

[12] Aaron Bloomfield and Borja Sotomayor. 2016. A programming contest strategy guide. In *Proceedings of the 47th ACM technical symposium on computing science education*. 609–614.

[13] Guilherme Cavalcanti, Paulo Borba, and Paola Accioly. 2017. Evaluating and Improving Semistructured Merge. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 59 (oct 2017), 27 pages. https://doi.org/10.1145/3133883

[14] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. https://doi.org/10.48550/ARXIV.2107.03374

[15] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program Repair with Quantitative Objectives. In *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 9780)*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer, 383–401. https://doi.org/10.1007/978-3-319-41540-6_21

[16] Claire Goues, Stephanie Forrest, and Westley Weimer. 2013. Current Challenges in Automatic Software Repair. *Software Quality Journal* 21, 3 (sep 2013), 421–443. https://doi.org/10.1007/s11219-013-9208-0

[17] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62, 12 (nov 2019), 56–65. https://doi.org/10.1145/3318162

[18] Sumit Gulwani, Ivan Radicek, and Florian Zuleger. 2018. Automated Clustering and Program Repair for Introductory Programming Assignments. *SIGPLAN Not.* 53, 4 (June 2018), 465–480. https://doi.org/10.1145/3296979.3192387

[19] Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring Coding Challenge Competence With APPS. *CoRR* abs/2105.09938 (2021). arXiv:2105.09938 https://arxiv.org/abs/2105.09938

[20] Y. Hu, U. Z. Ahmed, S. Mechtaev, B. Leong, and A. Roychoudhury. 2019. Re-Factoring Based Program Repair Applied to Programming Assignments. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 388–398.

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

[22] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-Written Patches. In *Proceedings of the 2013 International Conference on Software Engineering* (San Francisco, CA, USA) *(ICSE '13)*. IEEE Press, 802–811.

[23] Antti Laaksonen. 2020. *Guide to Competitive Programming - Learning and Improving Algorithms Through Contests, Second Edition*. Springer. https://doi.org/10.1007/978-3-030-39357-1

[24] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[25] Junho Lee, Dowon Song, Sunbeom So, and Hakjoo Oh. 2018. Automatic Diagnosis and Correction of Logical Errors for Functional Programming Assignments. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 158 (oct 2018), 30 pages. https://doi.org/10.1145/3276528

[26] Leping Li, Hui Liu, Kejun Li, Yanjie Jiang, and Rui Sun. 2022. Generating Concise Patches for Newly Released Programming Assignments. *IEEE Transactions on Software Engineering* (2022), 1–1. https://doi.org/10.1109/TSE.2022.3153522

[27] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode.

[28] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic Inference of Code Transforms for Patch Generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 727–739. https://doi.org/10.1145/3106237.3106253

[29] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[30] Yunlong Lu, Na Meng, and Wenxin Li. 2021. FAPR: Fast and Accurate Program Repair for Introductory Programming Courses. *CoRR* abs/2107.06550 (2021). arXiv:2107.06550 https://arxiv.org/abs/2107.06550

[31] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2018. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 129–139. https://doi.org/10.1145/3180155.3180247

[32] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 691–701. https://doi.org/10.1145/2884781.2884807

[33] Khanh Nguyen and Guoqing Xu. 2013. Cachetor: Detecting Cacheable Data to Remove Bloat. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering* (Saint Petersburg, Russia) *(ESEC/FSE 2013)*. Association for Computing Machinery, New York, NY, USA, 268–278. https://doi.org/10.1145/2491411.2491416

[34] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and Fixing Performance Problems That Have Non-Intrusive Fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, 902–912.

[35] Benjamin Paassen, Barbara Hammer, Thomas W. Price, Tiffany Barnes, Sebastian Gross, and Niels Pinkwart. 2018. The Continuous Hint Factory - Providing Hints in Vast and Sparsely Populated Edit Distance Spaces. *Journal of Educational Data Mining* 10, 1 (Jun. 2018), 1–35. https://doi.org/10.5281/zenodo.3554697

[36] David M. Perry, Dohyeong Kim, Roopsha Samanta, and Xiangyu Zhang. 2019. SemCluster: Clustering of Imperative Programming Assignments Based on Quantitative Semantic Features. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 860–873. https://doi.org/10.1145/3314221.3314629

[37] Yewen Pu, Karthik Narasimhan, Armando Solar-Lezama, and Regina Barzilay. 2016. Sk_p: A Neural Program Corrector for MOOCs. In *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity* (Amsterdam, Netherlands) *(SPLASH Companion 2016)*. Association for Computing Machinery, New York, NY, USA, 39–40. https://doi.org/10.1145/2984043.2989222

[38] Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, and Ulrich Finkler. 2021. Project CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. *CoRR* abs/2105.12655 (2021). arXiv:2105.12655 https://arxiv.org/abs/2105.12655

[39] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/2568225.2568254

[40] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. 2017. Learning Syntactic Program Transformations from Examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 404–415.

[41] Mark Santolucito, Jialu Zhang, Ennan Zhai, Jürgen Cito, and Ruzica Piskac. 2022. Learning CI Configuration Correctness for Early Build Feedback. In *2022*

IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER). 1006–1017. https://doi.org/10.1109/SANER53432.2022.00118

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

[43] Dowon Song, Woosuk Lee, and Hakjoo Oh. 2021. *Context-Aware and Data-Driven Feedback Generation for Programming Assignments*. Association for Computing Machinery, New York, NY, USA, 328–340. https://doi.org/10.1145/3468264.3468598

[44] Ke Wang, Rishabh Singh, and Zhendong Su. 2018. Search, Align, and Repair: Data-Driven Feedback Generation for Introductory Programming Exercises *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 481–495. https://doi.org/10.1145/3192366.3192384

[45] Ke Wang, Zhendong Su, and Rishabh Singh. 2018. Dynamic Neural Program Embeddings for Program Repair. In *International Conference on Learning Representations*. https://openreview.net/forum?id=BJuWrGW0Z

[46] Michihiro Yasunaga and Percy Liang. 2021. Break-It-Fix-It: Unsupervised Learning for Program Repair. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 11941–11952. http://proceedings.mlr.press/v139/yasunaga21a.html

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

[48] Maksym Zavershynskyi, Alexander Skidanov, and Illia Polosukhin. 2018. NAPS: Natural Program Synthesis Dataset. *CoRR* abs/1807.03168 (2018). arXiv:1807.03168 http://arxiv.org/abs/1807.03168

[49] Jialu Zhang, Todd Mytkowicz, Mike Kaufman, Ruzica Piskac, and Shuvendu K. Lahiri. 2022. Using Pre-Trained Language Models to Resolve Textual and Semantic Merge Conflicts (Experience Paper). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 77–88. https://doi.org/10.1145/3533767.3534396

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

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

[52] Kurtis Zimmerman and Chandan R. Rupakheti. 2015. An Automated Framework for Recommending Program Elements to Novices. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 283–288. https://doi.org/10.1109/ASE.2015.54