Improving the Flexibility of CLARA’s Automated Matching and Repair Processes

Maheen Riaz Contractor
mc1927@rit.edu

Follow this and additional works at: https://scholarworks.rit.edu/theses

Recommended Citation

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.
Improving the Flexibility of CLARA’s Automated Matching and Repair Processes

Maheen Riaz Contractor

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Master of Science in Computer Science

Department of Computer Science
B. Thomas Golisano College of Computing and Information Sciences

Rochester Institute of Technology
Rochester, NY
April 28, 2022
Committee Approval:

Dr. Carlos R. Rivero
Chair

Date

Dr. Matthew Fluet
Reader

Date

Dr. James Heliotis
Observer

Date
Abstract

More computer science researchers focus on automated program repair, with the world steadfastly moving towards automation. CLARA is an example of an automated program repair tool that provides feedback to novice programmers solving introductory programming assignments in Java, C++, and Python. CLARA involves test-based repair, requiring as input a correct program, an incorrect program, and its corresponding test case. Our work only focuses on Python.

CLARA has two main limitations. The first involves lacking support for commonly used language constructs such as standard input, standard output, and import statements. We address this issue by extending CLARA’s abstract syntax tree processor and interpreter to include these constructs. The second limitation is that CLARA requires both the correct and the incorrect program to have the same control flow. In a real-world setting, it is not easy to find such programs, reducing the true impact CLARA can have on the learning of novice programmers. Therefore, we implement a graph matching technique between the correct and incorrect programs that considers both the semantic and the topological information to help overcome this limitation. Using this matching, we modify the incorrect program to match its control flow with the correct program.

To verify that our technique overcomes the control flow limitation, we conduct experiments to run CLARA and compare the number of programs repaired with and without the graph matching technique. We also analyze the percentage of the program modified by CLARA and the number of correct programs needed to repair all valid incorrect programs. Our experiments show that CLARA can parse, process, and repair many more programs after our extensions. Additionally, our experiments indicate that we never enable CLARA to replace all the source code of an incorrect program with all the source code of a correct program.
Acknowledgements

I want to express my deepest and sincerest gratitude to my committee chair and advisor, Dr. Carlos Rivero. His faith in me and my abilities and constant support helped me successfully complete this project. I would also like to thank my committee members, Dr. Matthew Fluet and Dr. James Heliotis. You both have always provided me with great advice, and it has been a pleasure working with you throughout my journey at RIT.

I also want to say a special thanks to my parents, Riaz and Maleka. They never let me believe that I could not do whatever I wanted and taught me the importance of integrity and hard work. Without their constant support, I would never have been able to achieve my dreams. I hope I have made them proud. I would also like to thank my sister, Raina, grandmother, Munira, and my aunts, uncles, and cousins. Each of you has been an immense help during RIT and has provided me with emotional, mental, and financial support. Additionally, I would like to thank all my friends and lab mates. We have all shared great times in Rochester that I will remember for years.

Lastly, I would like to thank my undergraduate supervisor Dr. Junaid Haroon Siddiqui, for guiding me and pushing me to get my graduate degree.
## Contents

1 Introduction 1

2 Research Questions 3

3 CLARA 4

3.1 Models, Processing and Interpreting 5

3.1.1 Creating Functions and Locations 6

3.1.2 Expressions 7

3.1.3 Example of a Model 8

3.2 Single Function Matching and Repair 10

3.3 Control Flow and its Issues 18

4 Additions to CLARA 19

4.1 AST Processing and Interpreting 20

4.1.1 Print Statements 20

4.1.2 Boolean Values 21

4.1.3 Import Statements 22

4.1.4 Variable Assigning 23

4.1.5 Built-in Python functions 23

4.1.6 Variable Additions and Deletions 24

4.1.7 Input Statements 26

4.1.8 Main Function 28

4.2 Matching and Repair 28

5 Application of Repairs 29

6 Graph Matching 32
1 Introduction

Since the start of the 21st Century, tremendous advancements have been made in terms of technology, moving the world steadfastly towards automation [15]. This thesis focuses on automation in the education sector, specifically the Computer Science education sector. There has been a drastic increase in computer science undergraduate enrollment over the previous years [12]. This increase has impacted all members of the community; namely, the faculty, staff, and students [12]. Hence, researchers are constantly trying to improve automated grading, program repair, testing, and instruction [2, 6, 14, 16, 17]. In this thesis, we will be concentrating on automated program repair.

Automated Program Repair involves automatically finding a way to repair errors that exist in software without the involvement of any human interaction[11]. The survey published by Liu et al. [9] divides automated program repair into two main categories: state-based repair and behavior-based repair. Figure 1 displays the categorization of automated program repair techniques. State-based repair consists of a dynamic analysis of the incorrect program. It involves running the program and analyzing the values allocated to the registers, heaps, and stacks. These values are then manipulated by adding or removing assignments to ensure the program always produces the correct answer[4]. Behavior-based repair involves carrying out static analysis of the incorrect program and, therefore, rather than modifying the program memory, entails modification of the program’s source code.

Liu et al. [9]’s survey further categorizes behavior-based approaches into specification-based repair and test-based repair. Specification-based repair includes modifying a program based on functional logic specifications. These specifications are provided by the programmer and assert how the program is supposed to behave [5]. An example of a functional logic specification is as follows: insertion in a linked list should always happen at the front of the list. Test-based repair involves using given test cases to detect the errors in the incorrect program and, hence, aids in deciphering what and how the program should be repaired based
on the answer it is supposed to produce. Since the test-based process relies so heavily on the test cases themselves, we have to be very careful to provide test cases that cover the edge cases of the program we are repairing. The general program repair process involves three steps. The first is concerned with locating the errors within the source code that are causing the program to behave abnormally. The second involves deciphering how to repair the errors found by determining the necessary changes. The third is carrying out the actual repair.

In this thesis, we will be focusing on how to improve the flexibility of CLARA, an automated program repair tool. CLARA uses test-based automatic program repair to repair an incorrect program. It takes a correct program, an incorrect program, and the connected test case to evaluate as input and suggests the corresponding repairs. Rather than providing the repaired program, many tools such as CLARA, Sarfgen, and Refazer use the third step of the repair process mentioned above to provide feedback to students. The suggested feedback can be helpful for grading, hints, or the program repair itself.

To increase the flexibility of CLARA, we had to increase the number of assignments CLARA could parse and repair. Therefore, we analyzed introductory programming assignments in Python to see which language constructs are commonly used and then ran the assignments to see which CLARA did not recognize language constructs. We updated CLARA’s source code to include these language constructs, allowing CLARA to parse and process many more
programs.

We also noticed that CLARA could not repair programs that differed in control flow during our initial analysis. Control flow information encompasses the order in which statements get evaluated in a program. Since it is not common to find programs written by different people to contain the same control flow, we created a graph matching approach to help overcome this issue. The graph matching approach creates control flow graphs of each program and creates a matching between them, considering both semantic and topological information. We use the matching to recreate the intermediate representation of the incorrect program, such that the control flows of both the correct and the incorrect program match. We conduct experiments to test our graph matching approach and successfully verify that the approach helps overcome the control flow issue and can successfully repair many more programs.

The thesis is organized as follows: Section 2 presents the research questions addressed in the thesis; Section 3 provides background information about CLARA, expanding on its processes and how they work; Section 4, 5, 6, and 7 contain our contributions and addresses research questions 1, 2, and 3 by elaborating on the various additions made to CLARA to help improve the flexibility of the matching and repair processes; Section 8 contains our experimental results and addresses the research questions 4, and 5; and Section 9, and 10 recap our conclusions and future work.

2 Research Questions

Our goal is to increase the flexibility of CLARA such that it can run on a broader range of programs than it currently runs and repair real-world introductory programming assignments involving Python programs. In order to do that, we need to analyze submissions made by users to these assignments to detect common Python constructs and syntax and study how to add them to CLARA while maintaining its original functionality. Therefore, we present RQ1, RQ2, and RQ3:
**RQ1:** What Python statements/logic not currently parsed by CLARA are most commonly used in introductory programming assignments?

**RQ2:** What changes are necessary to the AST (Abstract Syntax Tree) processor to incorporate the new statements and create a new extended model?

**RQ3:** Are the current matching and repair processes able to function with the new extended model? If not, what changes are needed to resume functionality?

Once programs are successfully parsed and processed, we evaluate if the incorrect program is fully repaired, resulting in RQ4:

**RQ4:** Do the repairs suggested by CLARA actually repair the incorrect program, and, if so, is the program repaired only for the particular test case it was evaluated against or for all test cases?

One of CLARA’s limitations is that it requires the correct and the incorrect program to have the same control flow. To overcome this limitation, we present a graph-based matching approach between the correct and the incorrect program, leading to the proposition of RQ5.

**RQ5:** Can a graph-based matching process improve the current line by line in-sequence matching, reducing the structure mismatches?

### 3 CLARA

The automated CLustering And program RepAir tool, CLARA was first introduced in 2016. CLARA aids in providing feedback to students in introductory programming courses. An introductory programming course is a course that is usually taken by a student during
his or her first year of undergraduate study and typically has a course code ranging from 100 to 150. These courses are commonly taught in Python, Java, or C++ [13]. CLARA is a tool that works on all three of these programming languages. In this thesis, we focus on Python.

Figure 2 shows the workflow of CLARA. The AST processor receives a program as input, parses it, and creates a model. Models for the correct program (reference solution), the incorrect program (student submission), and the corresponding test case are provided to the repair process. The interpreter takes a model as input during the repair process and outputs its executed trace using the test case at hand. After finding the repairs, CLARA uses them to generate feedback to be delivered to students.

3.1 Models, Processing and Interpreting

Before performing any program repair or matching, CLARA creates models for every input program. The modeling process starts by providing the source code to the ast module of Python. The ast module is a standard library that can be imported. It takes a Python program as input, parses it, and returns an abstract syntax tree of the program. An abstract syntax tree is a graphical representation of source code containing nodes, where every node represents a language construct or operation like If, Return, Import [7]. CLARA traverses the returned abstract syntax tree, node by node, and creates a model. For every part of the abstract syntax tree, like FunctionDef (function definition), Expr (expression), or Call (function call), there is a different processing function and it has a different representation in
3.1.1 Creating Functions and Locations

The entire model consists of one program containing multiple functions. Each function is partitioned based on its control flow information. Control Flow information captures the order in which statements get evaluated. An example of a control flow statement is an if condition, as it adds a possible path for the program to take. Locations, a model construct of CLARA, represent control flow information. Therefore, each function consists of multiple locations. Locations contain expressions and are created based on branching control flow statements like if, while, or for. Other control flow statements like function calls or sequencing of statements are not involved in location creation. So if a function contains no branching control flow statement, it will only contain one location. Locations also contain information about which location to go to next, called transitions. There are two types of transitions, True and False. The True transition contains the following location to go to if the conditional expression inside the location evaluates to True. The False transition contains the location to visit next if the expression inside the location evaluates to False.

However, there is a possibility of having expressions inside a location that do not evaluate to a Boolean value, in which case, they will always go to a specific location. For example, transitioning back to the program body after executing the then or else branch inside an if condition. In this case, these transitions are always True. To maintain consistency, these locations also have true and false transitions, but the False transition always points to None, and the True transition always points to the next location. Hence, if there is only one possible transition, it will always be accessed using the True transition, leaving the False transition as None by default.
3.1.2 Expressions

CLARA contains three types of expressions, where the names in between the parenthesis refer to CLARA’s naming convention: variables (\texttt{Var}), operators (\texttt{Op}), and constants (\texttt{Const}). A \texttt{Const} can be a string, byte, number, or a name constant. A \texttt{Var} represents variables and, therefore, only comprises strings. An \texttt{Op} is the most complex type of expression as it encompasses all computations involving any time of operation. Such as creating a list, set, or tuple, computation involving comparisons, if conditions, or binary operations. Every \texttt{Op} comprises two components, the name of the operation and the arguments it has to operate on. Depending on the type of operation, \texttt{Op} can contain a different number of arguments. For example, the operator, \texttt{GetElement}, entails getting an element from a list, dictionary, or set, and contains two arguments: the object it needs to get the element from and then the element index. \texttt{SetInit}, which creates a set, has multiple arguments as each argument is an element inside the set.

All types of control flow statements are also \texttt{Ops}. However, while processing those statements, CLARA makes changes to the model. As mentioned earlier, processing control flow statements results in the addition of new locations to the model. The number of locations is different for each control flow statement. If the program contains an if condition, three or four locations will be added if it is not a nested condition; one for the condition of the if statement, another for the expressions inside the then branch, one for the expressions after the if statement, and finally, for the expressions inside the else branch. The last location for the else branch is optional, as we do not always have an else branch accompanying the then branch. On the other hand, loops always result in the addition of three locations, corresponding to the conditional, expressions, and the statements after the loop. Hence, models of programs in introductory programming courses typically have ten to fifteen locations, depending on the program’s functionality.

CLARA restricts its models by requiring a variable to appear only once on the left side of an expression per location. This restriction entails inspecting all the declarations of a
particular variable and nesting the declarations in the last use of the variable. Hence, during the repair and match processes, where variable values are matched between files, there is only one value per location, reducing the amount of work required by the repair process.

3.1.3 Example of a Model

Figures 3, and 4 show an example of Python source code and its corresponding model created by CLARA. This model is the pretty printed version created by CLARA to help improve readability. Since the source code contains a for loop, the model contains four locations corresponding to the code before the loop, the conditional, inside the loop, and after the loop. As shown in Figure 4, the sequence of locations do not necessarily correspond to the actual control flow; hence, the True and False transitions at the bottom of each location indicate how to traverse the model. If $\text{cond}$ in location 2 evaluates to True, we transition to location 4. If it evaluates to False, we transition to location 3. $\text{ind#0}$ correspond to the index of the for loop, i.e, $\text{i}$ and $\text{iter#0}$ corresponds to the value we are iterating over, i.e, $\text{a}$. CLARA internally creates both variables for every loop. As shown in the source code, $\text{b}$ appears on the left side of an expression three times (lines 2, 3, 4), but in the model, it only appears once in location 1. All three uses are nested inside one expression. $\text{GetElement(a', 0)}$, represents the expression on line 2. $\text{AssAdd(...)1}$ represents the expression on line 3, and $\text{AssAdd(..., c')}\) represents the expression line 4.

Whenever CLARA uses a variable defined earlier in an expression, it converts it to a
Loc 1 (around the beginning of function main)

\[
\begin{align*}
    a & := \text{ListInit}(5, 6) \\
    c & := \text{GetElement}(a', 1) \\
    b & := \text{AssAdd}(\text{AssAdd}(\text{GetElement}(a', 0), 1), c') \\
    \text{iter#0} & := a' \\
    \text{ind#0} & := 0 \\
\end{align*}
\]

True $\rightarrow$ 2 False $\rightarrow$ None

Loc 2 (the condition of the ‘for’ loop at line 6)

\[
\begin{align*}
    \text{cond} & := \text{Lt}(\text{ind#0}, \text{len}(\text{iter#0})) \\
\end{align*}
\]

True $\rightarrow$ 4 False $\rightarrow$ 3

Loc 3 (*after* the ‘for’ loop starting at line 6)

\[
\begin{align*}
\end{align*}
\]

True $\rightarrow$ None False $\rightarrow$ None

Loc 4 (inside the body of the ‘for’ loop beginning at line 7)

\[
\begin{align*}
    i & := \text{GetElement}(\text{iter#0}, \text{ind#0}) \\
    \text{ind#0} & := \text{Add}(\text{ind#0}, 1) \\
    c & := \text{AssAdd}(c, i') \\
\end{align*}
\]

True $\rightarrow$ 2 False $\rightarrow$ None

Figure 4: Model generated by CLARA
different variable rather than using the original variable. For example, a becomes a’. This is shown in Figure 4. CLARA uses this notation to determine whether a variable is being defined or used.

Since CLARA has a Python processor that creates its model, it also has an interpreter that helps execute this model in Python. The interpreter visits each expression in the model and recursively executes it since an expression can consist of multiple nested expressions. For example, the expression, \( b := \text{Eq}(\text{AssAdd}(\text{AssAdd}(\text{GetElement}(a',0),1),c'),0) \) needs to evaluate from the innermost expression, \( \text{GetElement}(a',0) \), outwards, until it reaches \( \text{Eq}(..,0) \). The interpreter contains functions for every type of operator, in this case, \( \text{Eq} \), \( \text{AssAdd} \), \( \text{GetElement} \), annotated with the word `execute_` as a prefix of each function. Therefore, while evaluating this expression, CLARA first executes \( \text{GetElement}(a',0) \), which translates to getting the element of a at index 0, involving the function `execute_GetElement()`. Then it goes on to evaluate \( \text{AssAdd}(...,1) \) and \( \text{AssAdd}(...,c') \), which involves incrementing the result by 1 and c respectively and then assigning the answer to b, both of which are done by the function `execute_AssAdd()`. Finally, it evaluates \( \text{Eq}(..,0) \), which first checks if the result is 0, given by the function `execute_Eq`. Figure 5 contains model expressions on the left and its translated Python code on the right.

\[ 
\begin{align*}
b &:= \text{GetElement}(a', 0) & b &= a[0] \\
b &:= \text{AssAdd}(b, 1) & b &= b + 1 \\
b &:= \text{AssAdd}(b, c') & b &= b + c \\
b &:= \text{Eq}(b, 0) & b &= b == 0 
\end{align*}
\]

Figure 5: Expressions and their translated Python code

3.2 Single Function Matching and Repair

CLARA comprises several workflows of tasks such as program matching, single program repair, and clustered program repair. Program matching is done between two programs,
which can be either correct or incorrect. CLARA starts by analyzing its structures and creates models from each program. If the control flow of both program models is the same, for every variable in one model, it finds a matching with a corresponding variable in the other model. Driven by variable tracing, the process involves comparing values of variables at each location of the program trace. If the matching variables hold the same values at every point in the trace of the program, the two programs are determined as a match.

Single program repair is a program repair done between two programs, where one is the correct program and the second is the incorrect program. The incorrect program is repaired against a cluster of correct programs in clustered program repair. All programs in that cluster must have the same control flow and must match with each other. In practice, it is not easy to find programs that contain the exact same control flow since a specific task like finding a minimum or sorting an array can be coded up in many different ways, making it challenging. A program in an introductory programming course typically contains several of these tasks intertwined, e.g., read the test case from console, compute the minimum and maximum values of the input, sort the input and provide its median value. The challenge comes from the combination of these tasks and the different approaches to solve them in source code. In this thesis, we focus on single program repair.

Like program matching, the repair process starts by creating models from each program and checking for a match in the control flow. Additionally, CLARA requires the programs to have at least one function and the same number of functions overall in the programs being compared. These functions also must have the same names and cannot be nested. These features determine the structure of the program. If there is a difference in the structure of the programs, CLARA throws an error called StructureMismatch and does not run at all.

Since the models contain multiple locations, where a number represents each location, the locations are matched based on their number of transitions. Algorithm 1 shows the pseudocode CLARA uses to map locations and determine if there is a structure mismatch. It takes as input the initial location from the correct program model, i.e., $u$, and the initial
Algorithm 1: CLARA’s Location Matching

**Input:** $G_C = (U, E)$ control flow graph (correct program)

$G_I = (V, F)$ control flow graph (incorrect program)

**input/output:** $\phi : U \rightarrow V$ program matching

$u \leftarrow$ initial location in $G_C$

$v \leftarrow$ initial location in $G_I$

**Function** $\text{CompareCFG}(u, v)$:

// If $u$ is already in $\phi$, it must be mapped to $v$

if $u \in \text{dom}(\phi)$ then
    return $\phi(u) = v$

end

// If $v$ is already in $\phi$, it should only be mapped to $u$

if $v \in \text{ran}(\phi)$ then
    return $\phi(u) = v$

end

// If both $u$ and $v$ are None

if $u = \text{None} \land v = \text{None}$ then
    return $\text{True}$

end

// Map $u$ to $v$

$\phi(u) \leftarrow v$

// $u'$ and $u''$ are the neighbors of $u$

Let $u \rightarrow^\text{True} u', u \rightarrow^\text{False} u''$

// $v'$ and $v''$ are the neighbors of $v$

Let $v \rightarrow^\text{True} v', v \rightarrow^\text{False} v''$

// Compare the neighbors of $u$ with the neighbors of $v$

return $\text{CompareCFG}(u', v') \land \text{CompareCFG}(u'', v'')$
location from the incorrect program model, i.e., $v$, and returns a Boolean declaring whether the two program models match. $\phi$ is a dictionary containing the mapping of locations. It starts by checking if $u$ has already been mapped, as it is possible to have cyclic transitions, like in the case of for loops. If $u$ is mapped, then it should only be mapped to $v$. If both $u$ and $v$ are None, CLARA return True and stop, otherwise CLARA run the the algorithm on its respective transitions.

Figure 6 shows an example of CLARA’s location mapping. For both the examples, the left graph represents the correct program, and the right represents the incorrect program. The solid arrows represent the True transitions, and the dotted arrows represent the False transitions. We are attempting to create a match between a location in the correct program model to a location in the incorrect program model. Figure 6a shows two models where the location mapped was successful, represented by the blue arrows. Hence, location 1 maps to location 1, location 2 maps to location 3, and location 3 maps to location 4 of the correct and incorrect programs, respectively. However, figure 6b shows an example of a structure...
mismatch, as there is no match for location 4 of the correct model to a location in the incorrect model.

After mapping the locations, CLARA traverses through these locations during the repair process. For every variable in one location of the correct program, CLARA searches for a match for the variable in the corresponding location of the incorrect program. Due to CLARA’s modeling, a variable can only have one expression per location, ensuring it can only hold one value. This makes the comparison with other variables possible. During the repair process, a mapping of a variable only deals with its specific expression and the expression it is mapped to. While mapping a variable from the correct program, every variable in the incorrect program is compared, whether or not it has already been mapped. Two variables are declared a match if their corresponding expressions evaluate to the same values using the same inputs, where the inputs denote the variables both the expressions depend on \[ \text{[6]} \]. This match is evaluated based on cost. Since both the correct expression and the incorrect expression depend on different variables, the cost of a match denotes the number of changes/steps it takes to transform the incorrect program’s expression into the correct program’s expression, where a variable from the incorrect program replaces each variable in the correct program’s expression.

In the repair process, CLARA assumes that the number of variables in the correct program is the minimal number of variables needed to solve the program, so the number of variables in the incorrect program needs to match the number of variables in the correct program precisely. Therefore, if the incorrect program contains extra variables, CLARA will suggest deleting the variables it cannot match. If the incorrect program contains fewer variables than the correct program, CLARA will suggest creating new variables needed during the matching process.

During cost calculation, all the variables in the correct program’s expression are substituted by variables from the incorrect program. If the expression of a variable is dependent on other variables, like the statement \( s = a + 2 \), CLARA would replace \( a \) by all other variables
Correct Model:  
\[
\begin{align*}
a & := 1 \\
b & := 2 \\
c & := \text{Add}(a', 1)
\end{align*}
\]

Incorrect Model:  
\[
\begin{align*}
x & := 1 \\
y & := 2 \\
z & := \text{Add}(y', 1)
\end{align*}
\]

Figure 7: Correct and incorrect program models for cost Tables 1, 2, and 3

<table>
<thead>
<tr>
<th>#</th>
<th>Variable (C)</th>
<th>Variable (IC)</th>
<th>Dependency Matching</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>*</td>
<td>None</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>a</td>
<td>x</td>
<td>None</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>a</td>
<td>y</td>
<td>None</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>z</td>
<td>None</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 1: Repair cost table for variable a from Figure 7

in the incorrect program and evaluate the corresponding cost. So, if the incorrect program contains three variables, \(x, y, z\), variable \(s\) in the expression, \(s = a + 2\) will be calculated with \(a\) substituted by the values of \(x, y\) and \(z\). Since there is a possibility of extra variables in the correct file, CLARA also calculates the cost of the variable as if it were a new variable, with no existence in the incorrect program. This value is initialized to the value of \(a\) with a cost of 1. CLARA finally saves all the costs for every location. It provides the cost array to a linear programming solver, which minimizes the overall cost and suggests matches for all the variables. Based on the matches and the cost, the final set of repairs are suggested, which can be of three types: variable additions, variable deletions, or variable changes.

Consider the models of two programs in Figure 7. The model on the left is the correct program model, and the one on the right is the incorrect program model. Figure 3 contains an example of a variable matching with its associated costs. The first column represents the comparison number. The second column contains the variable of the correct program it is trying to match. The third column represents the possible match for the variable in the first column from the incorrect program. The fourth column contains the possible substitution
<table>
<thead>
<tr>
<th>#</th>
<th>Variable (C)</th>
<th>Variable (IC)</th>
<th>Dependency Matching</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
<td>*</td>
<td>None</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>x</td>
<td>None</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>b</td>
<td>y</td>
<td>None</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>b</td>
<td>z</td>
<td>(a, y)</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
<td>z</td>
<td>None</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 2: Repair cost table for variable b from Figure 7

<table>
<thead>
<tr>
<th>#</th>
<th>Variable (C)</th>
<th>Variable (IC)</th>
<th>Dependency Matching</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>c</td>
<td>*</td>
<td>(a, *)</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>c</td>
<td>*</td>
<td>(a, x)</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>*</td>
<td>(a, y)</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>*</td>
<td>(a, z)</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>c</td>
<td>x</td>
<td>(a, *)</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>c</td>
<td>x</td>
<td>(a, y)</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>c</td>
<td>x</td>
<td>(a, z)</td>
<td>2</td>
</tr>
<tr>
<td>8</td>
<td>c</td>
<td>y</td>
<td>(a, *)</td>
<td>3</td>
</tr>
<tr>
<td>9</td>
<td>c</td>
<td>y</td>
<td>(a, x)</td>
<td>3</td>
</tr>
<tr>
<td>10</td>
<td>c</td>
<td>y</td>
<td>None</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>c</td>
<td>y</td>
<td>(a, z)</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>c</td>
<td>z</td>
<td>(a, *)</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>c</td>
<td>z</td>
<td>(a, x)</td>
<td>1</td>
</tr>
<tr>
<td>14</td>
<td>c</td>
<td>z</td>
<td>(a, y)</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 3: Repair cost table for variable c from Figure 7
for the dependent variable written in a tuple (i, j), where i is the dependent variable from
the correct program, and j is its possible substitution from the incorrect program. The fifth
column represents the cost for this match.

The cost calculation of \(a\) and \(b\) is given in Tables 1 and 2. They are calculated similar to
\(c\), by calculating the cost of each variable if it was matched with a new variable, represented
by \(\ast\), or the variables, \(x, y\) and, \(z\). Since \(c\) is dependent on \(a\), given by the statement
\(c := \text{Add}(a', 1)\), while attempting to find a match for \(c\), \(a\) is also replaced. The optimal
solution is a matching that minimizes the overall cost for all variables.

For example, consider the 9th, 10th and 14th rows of Table 3. In the 9th row, where \(c\) is matched with \(y\) and \(a\) is replaced by \(x\), its cost is 3. In this match, we are converting
\(\text{Add}(x, 1)\) into 2, as \(a\) has been substituted by \(x\). The cost for this match is 3 as we replace
the 2 with a 1, add the variable \(x\), and the addition operation, which are three operations. In
the 10th row the cost is 0 as both \(c\) and \(y\) hold the same trace values throughout the program
and \(a\) is not taken into consideration. Since we get the same value by assigning \(c\) to 2 or
adding 1 to \(a\), \(a\) is considered an extra computation and ignored. While evaluating the cost
for the 14th row, we equate \(\text{Add}(y, 1)\) with \(\text{Add}(y, 1)\), as \(a\) has been replaced by \(y\), returning
the cost 0, since the two expressions are equal. All of these possible matches, along with the
ones for \(a\) and \(b\), are the input of the linear programming solver. Let’s consider an example of
how the linear programming solver computes its final match. If we take the 10th row of Table
3 we get two matches, \(\phi_1 = \{c = y, a = x, b = z\}\), and \(\phi_2 = \{c = y, a = z, b = x\}\). The total
cost for each match is 3. After considering every combination, the linear programming solver
returns, \(a\) matched with \(x\), \(b\) matched with \(y\) and \(c\) matched with \(z\), having an overall cost
of 1. Since, \(a\) is matched with \(x\), the 13th row in Table 3 is chosen as a match for \(c\), giving us
the final repair Change \(z := \text{Add}(y', 1)\) to \(z := \text{Add}(x', 1)\) at line 3 (cost = 1.0).
3.3 Control Flow and its Issues

It is a common practice for program repair techniques to rely on control flow information to accomplish their task [3, 6, 1, 4]. Control flow statements like if, while, or for loops affect the order of evaluation of a program, as they execute a set of statements a certain number of times if a given condition is met, or only once if the condition is met. Hence, we know what path a program can take using control flow information. For tools like CLARA that use variable tracing during the program matching and repair processes, an ambiguous order of execution can cause many difficulties. Since variables can be reassigned and hold different values through different stages of computation during the execution of a program, it would prove tricky to match it to a particular variable. Suppose we match variables based on their final value in the program. In that case, the answers can be misleading as a specific value can be calculated in multiple ways, thereby, proving challenging to determine the location of the error in the program. If we know the control flow, we can follow the execution of a variable and, hence, pinpoint its value at any point in the program. Therefore, CLARA only works on programs that contain the same control flow. In practice, however, it is not very common for programs to contain the same order of while and for loops, and if/else conditions, limiting the number of programs CLARA can repair.

Figure 8 presents an example of a control flow graph. The left side of the figure displays a Python program that finds the first even number in a given list and prints it to the console. The right side of the figure contains its corresponding control flow graph. The solid edges in the graph represent the transition if the conditional inside the node evaluates to True. Similarly, the dotted edges represent the transition if the conditional evaluates to False. If there are no conditionals inside that node, it will automatically evaluate to True. There are two nodes containing the conditionals for the for loop and the if condition, and then three more nodes for the statements, before the loop, inside the conditional body, and after the loop. The figure shows that there is no separate node for the break statement. Instead, it is
a = 0
b = [13, 27, 32, 45, 58]

for i in b:
    if i % 2 == 0:
        a = i
        break

print(a)

(a) Source code

Figure 8: Python code and its corresponding control flow graph for finding the first even number in a list represented by a transition from the variable assignment of a to the print statement. If \( i \) is even, following the True transition, we save \( i \) in a, and then follow the True transition to exit the for loop.

4 Additions to CLARA

To increase CLARA’s flexibility, we analyzed introductory programming assignments to identify language constructs commonly used like print statements, input functions, and import statements. We then ran assignments containing these language constructs to decipher which constructs were not processed by CLARA. In most cases, if CLARA encountered these statements, it would throw an exception and stop as CLARA’s AST processor did not recognize these statements. Since the AST processor models these constructs and the Interpreter executes them, both the processes were updated to include these constructs. These changes are described in Section 4.1. After including these changes, we ran the repair
and matching processes to verify that repairs could be suggested for these constructs. The changes made to both the matching and repair processes are described in Section 4.2.

4.1 AST Processing and Interpreting

4.1.1 Print Statements

Most introductory programming courses worldwide and online programming practicing websites use printing to the console to verify whether a program is correct or incorrect. The verification of correctness determines whether a program should be repaired or not. Similar to other programming languages, computation in Python can be performed using variables or inside the parentheses of a print statement, removing the need for variables altogether. Hence, while evaluating the similarity between two programs or performing a repair, it is crucial to match the content inside these print statements. The original authors of CLARA did incorporate the parsing of print statements. However, when Python developers introduced Python 3.x, the print operation switched from a statement to a function. Furthermore, when the authors updated the system to work with Python 3.x, not all of the code was updated, leading to the loss of functionality of print operation. We updated the section of the Python AST processor that checks for function calls by checking if the print function was called and accordingly updated the model. Once the print function was correctly processed, the comparison of the print operation during the matching and repair processes was handled automatically.

Consider the two small programs in Figure 9. The program aims to calculate whether the input is an even number. Fig 9a is correct and 9b is incorrect. Hence, CLARA should say the programs do not match and output a repair asking the user to change the modulo value to 2. However, since CLARA does not match the print statements, no repairs are suggested, and the programs are declared a match. We can circumvent this issue by assigning the computation inside the print functions to a variable and then printing that variable. This
def isEven(val):
    print(val % 2 == 0)

(a) Correct program

def isEven(val):
    print(val % 1 == 0)

(b) Incorrect program

Figure 9: Correct and incorrect programs for isEven function

An option would enable CLARA to output the correct repair successfully. However, we found that many programs performed computation inside the print function during our testing, making this change necessary.

4.1.2 Boolean Values

The most basic and common types in a program in an introductory programming course are Int, Bool and String. Accordingly, to maximize the range of input, CLARA must be able to parse values of all these types. However, CLARA’s parser did not check if values had the type Bool, so the Boolean values, True and False, threw an error while parsing. Since values of this type were not parsed, any program that used flags, commonly used writing while loops, could not run in CLARA. An example of this is in Problem 1401C on CodeForces\textsuperscript{1}. This problem requires the student to check if it is possible to manipulate an array of numbers such that it is increasing in order. A few students used an infinite while loop to traverse the array and perform the operations provided. Without the addition of Boolean values to the AST processor, these programs could not run in CLARA. Since CLARA categorizes expressions into Constants, Variables and Operations, we added another Constant to the AST processor of type Boolean that can process the values True and False.

\textsuperscript{1}codeforces.com/problemset/problem/1401/C
import math
from math import sqrt
import math as m
from math import sqrt as sq

(a) Import Statements

math.sqrt(3)
sqrt(3)
sqrt(3)
m.sqrt(3)
sq(3)

(b) Function Calls

math : math
sqrt: [sqrt, math]
*: [*, math]
m : math
sq: [sqrt, math]

(c) Dictionary Representation

Figure 10: Import statements with its corresponding function calls and representation in CLARA

4.1.3 Import Statements

While problem solving, one can require the use libraries, like math, re, string. These libraries provide access to functions like sqrt, ceil, floor, ASCII, and format that can save students time and mental energy from not defining the functionality themselves. Hence, for CLARA to execute these functions while performing a repair or match, it is essential to have the functionality to parse and record the data inside these import statements. The original version of CLARA ignored all import statements, causing the program to occasionally crash during the repair or match process as their corresponding functions could not be found while executing the function trace.

Figure 10a shows the five different ways of writing import statements in Python, each resulting in a different version of an ast node created during parsing. These functions can appear by themselves or with a library/alias attached, making the process of recognizing them challenging. Using each import statement, we are calling the function sqrt. Figure 10b contains Python code displaying how to call sqrt using each type of import statement. sqrt is a function available in the math library. Each import statement carries out similar functionality so all 5 statements can not be added together in the same program. Since CLARA has its version of an AST processor and interpreter for Python, after parsing and processing the import statements, we saved the statements in a manner such that a function
would be successfully recognized during execution by the interpreter. As a result, we created a section in the AST processor for import processing and saved the imports and their data in a nested global dictionary, which was then passed to the interpreter so it could be accessed during execution. Examples of dictionaries for each import statement is given in Figure 10c, where the key is the name we use in the code and the value is the actual function/library name in Python.

4.1.4 Variable Assigning

When Python developers released Python3, they introduced a new technique of variable assignment, similar to the one in JavaScript ES6. This technique is based on list deconstruction or unpacking. Below is an example of this technique in Python.

\[ a, b, c = [1, 2, 3] \]

It is a convenient technique for assigning the value 1 to a, 2 to b, and 3 to c. Analyzing user submissions on CodeForces, we found that this technique was commonly used. Hence, we updated the AST processor to recognize and process multiple assignments from a single statement. We separated the assignments with their corresponding expressions and added each assignment as an individual expression to the model. Before this addition, CLARA would throw an error stating that multiple assignments within a single line are impossible and halting the repair and matching processes.

4.1.5 Built-in Python functions

CLARA has a interpreter that helps recognize functions in the model and execute them in Python. Therefore, common built-in functions like \texttt{max}, \texttt{sum}, \texttt{len} are individually defined in the interpreter using functions like \texttt{execute_max}, \texttt{execute_sum}, \texttt{execute_len} which implement their functionality. Since Python has a substantial collection of built-in functions, manual addition of every function was not included in the interpreter, causing CLARA to crash
def cap():
    a = [1, 2, 3]
    s = 0
    for x in a:
        s += a
    f = 2
    return a

def cap():
    a = [1, 2, 3]
    s = 0
    for x in a:
        s += a
    return a

(a) Correct program
(b) Incorrect program

1 Add assignment 'new_f := new_f' around the beginning of function 'cap' (cost=1.0)
2 Add assignment 'new_f := 2' after the 'for' loop starting at line 9 (cost=1.0)

(c) Feedback

Figure 11: Correct and incorrect programs and their corresponding feedback for repairs during the matching and repair processes. We found an approach to circumvent the need for manual addition of every function. Every time a function is called, we verify if it is a built-in Python function using Python’s internal dictionary named builtins and execute it.

4.1.6 Variable Additions and Deletions

CLARA expects all variable declarations to have a definition in the first location of the program. In other words, the start of the file contains assignments for every variable, and the rest of the file makes use of those variables. Hence, a new variable cannot be declared later in the file, leading to the output of unnecessary repairs, and the final mapping of variables being occasionally inaccurate. Figure 11 contains a correct program, an incorrect and the corresponding repairs given by CLARA. As one can see, line 1 of the feedback asks the user to create a variable and assign it to itself, which is unnecessary. We modified the AST processor by removing the restriction of requiring variables to be declared in the first location. Additionally, we modified the list of repairs generated by CLARA such that a repair similar to line 1 of the feedback is no longer suggested.
During the repair process, CLARA creates a mapping of variables from the correct program to an incorrect program. This mapping relies on the variable tracing performed throughout the repair process. A dictionary/map is used to store the mapping. It uses variables from the correct program as keys and incorrect program variables as values. While updating the source code to include our modifications, we noticed that, if more than one extra variable was declared, CLARA no longer suggests deleting more than one variable, resulting in an incorrect final variable mapping. An example of the feedback, along with the source code is shown in Figure 12. The final mapping after this repair is \{a : a, s : 0, x : x, − : g\}. The variable \(f\) has been left out of the mapping.

Variable addition in CLARA is represented by ∗ and variable deletion is represented by −. Since a dictionary is used for variable mapping, it overwrites one of the deletions, as the key is the same. Hence, if the variables \(f\) and \(g\) were in the correct program, CLARA would suggest to add new variables. This never results in an incorrect mapping problem, because, in the dictionary, they are represented as \{f : ∗, g : ∗\}. The deletions cannot be \{− : g, − : f\}, because the same key cannot be present multiple times, so we only get the suggestion to
remove one variable. Therefore, we adjusted the dictionary to save an array of values in the case of deletions. Nevertheless, this did not solve the final matching problem. After further investigation, we found that, before suggesting the final repairs, CLARA removes suggested repairs that had a cost of 0, causing certain variables to not be considered during the mapping. After updating CLARA to consider every variable in the mapping, and adding an array for deletions, we solved both these problems.

4.1.7 Input Statements

One of the primary practices of introductory programming courses is that they evaluate the correctness of a program based on test cases. The program must admit console input and output the correct answer. CLARA, however, did not support standard input. All inputs have to be provided via the command line as function arguments. This is not always possible since input arguments can be multiple lines long and do not have the same length. Therefore, we updated CLARA to read all of the inputs using an argument file and store them in a list accessible by the interpreter. We updated the interpreter to handle function calls to the `input()` function separately. Every time the function is called, we extract the first element from the list, and pass it to the variable calling the function, rather than actively it. Consequently, we can repair programs that had inputs of variable length.

However, while adding this feature, we encountered another problem. Since CLARA nests the expressions of variables while creating its model, it creates copies of the `input` function when there should only be one call. For example, the Python statement:

\[
\text{a, b, c = input().split()}
\]

becomes the following statements in the model.

\[
\text{a = GetElement(split(input()), 0)}
\]
\[
\text{b = GetElement(split(input()), 1)}
\]
\[ c = \text{GetElement}(\text{split}(\text{input}()), 2) \]

This change results in \texttt{input} being called three times, where it should have been called once. Since it is possible for this problem to occur in other cases, too, we updated the AST processor to create a new variable to store the result of calling the \texttt{input} function. Additionally, the expression referencing the function, now references the new variable instead. Therefore, using the above example, the statements in the model are as follows:

\begin{verbatim}
input_val = input()
\end{verbatim}
\begin{verbatim}
a = \text{GetElement}(\text{split}(\text{input}_\text{val}), 0)
b = \text{GetElement}(\text{split}(\text{input}_\text{val}), 1)
c = \text{GetElement}(\text{split}(\text{input}_\text{val}), 2)
\end{verbatim}

This change helped solve our problem by calling \texttt{input} only once.

Repetition of expressions is expected if we have multiple variable declarations in a single line during model creation. If these expressions include side-effecting functions, we can have a similar problem as we had with the \texttt{input} function. It is programmatically feasible to detect whether a function is side-effecting, and creating new variables for every single function call in a program can be challenging to handle due to the addition of multiple variables. Additionally, it can cause a mismatch in the number of variables between the correct program and the incorrect program, resulting in unexpected feedback. However, this is not a problem for the \texttt{input} function, as we expect both the programs to have the same number of calls to the \texttt{input} function. We adjusted the source code to address this issue to receive a list of the side-effecting functions via command line. New variables will be created for each of these functions similar to \texttt{input}, aiding us in solving the problem and limiting the addition of extra variables.
4.1.8 Main Function

One important feature of Python is that it allows us to write a program that does not contain any functions. In this case, everything is included as statements inside the main function and executed. Since CLARA requires programs to have named functions, it does not run on those programs. We updated CLARA to add all statements of a program manually, not already in functions, into a function named main and to automatically repair the main function if a function name is not provided. This required changes to both the AST processor and the model to recognize and group programs correctly.

4.2 Matching and Repair

Before making most of the additions to the AST processor, if we encountered a statement that the processor did not handle, the model would not be created, and an error would be thrown. Since both the matching and repair processes depend on creating a model, if the model was not created, both the processes would halt too. For most of the additions, once the statements were processed and recognized by CLARA, the matching and repair processes ran smoothly, requiring no further changes.

One addition that did involve changes to these processes was the addition of nested functions. Initially, CLARA was not be able to parse nested functions as functions could not store other functions in the model. A function was only allowed to store expressions. We updated the model to allow nested functions by creating a link between the two or more functions. After adding it to the model, we updated the matching and repair processes to match and repair these functions successfully.

Both the processes involve creating a one-to-one mapping between variables. Therefore, we need to ensure that the variables inside the nested functions are not involved in the mapping as those exist in a different environment. We treated the nested function as a variable, which was evaluated during trace execution, and its return value was substituted by the variable
calling the function. If the function was called on its own and did not have a return value, we checked if it was printing to standard output; if that was the case, the expression being printed was added to the standard output of the outer function. Since CLARA performs variable tracing and tracks the value a variable holds throughout the program, CLARA places more importance on the variable’s value than its expression. This helps eliminate the need for function inlining. Our aim while updating the matching and repair process was to avoid the suggestion of creating/deleting the nested function if the other program was performing the exact computation without the use of a nested function.

5 Application of Repairs

In their paper, the authors of CLARA suggest that, once all of the repairs have been applied, the program should be repaired entirely for the particular provided test case. We aimed to determine whether the program is repaired for only that particular test case or all test cases. We had to tackle the problems of how to convert the feedback into an actual repair, and, then, how to apply that repair to the model. Since the repair process works with models, we decided to repair the model rather than repair the source code itself.

Every repair statement that is converted to feedback by CLARA contains the variables involved in the repair for both the correct and incorrect program, the associated location of the correct program, and the associated expression from the correct program. Hence, we started by extracting the rest of the information, i.e., the location of the incorrect program and the incorrect program’s expression. Every function in a program groups and stores its variables and their associated expressions by location. There are three types of repairs: variable deletion, variable addition, and changing the variable definition. In the case of variable deletion, we access its corresponding location expressions and remove the variable from the list. Similarly, we add the variable and the new expression to its corresponding location expressions for variable additions. Likewise, we substitute the variable’s expression
in its corresponding location expressions for variable changes.

However, the feedback suggested by CLARA is never in any particular order, so for variable additions and changes, we must to ensure the expressions are added in such an order that the variables being used in its expressions exist. For example, if we had the following repairs:

* Change 'a = x + 5' to 'a = m + 5’
* Add ’m = 3’

We must to ensure that m is defined before a; otherwise, an error is thrown during program execution. As mentioned earlier, every time CLARA uses a variable that has been defined earlier in an expression, rather than using the original variable, it converts it to a different variable with ’ appended at the end. This is shown in Figure [13] on line 4 where a becomes a’. However, even though line 18 is using the variable, a, it is not using a’. This is because Location 1 contains an assignment for this variable while Location 3 does not. Therefore a further problem involving variable definition is caused. If we are adding line 4 into the model, we must ensure a already exists and is defined earlier in the location. If we are adding a definition of a into Location 3, we must change a to a’ on line 18. Therefore, every time a new variable is added or deleted, we execute a trace of the function to help us determine which variables we have access to before adding a repair. If all of the variables used in that expression are defined, we add the repair. Otherwise we continue adding the rest of the repairs and come back to the one we skipped earlier.

Once all the repairs have been added, we rerun the repair process to determine if further repairs are suggested for the same test case. We halt and conclude that the program is not successfully repaired if further repairs are proposed. Otherwise, we conclude that the incorrect program is repaired for that particular test case, run the repair process for all other available test cases, and record if any repairs are suggested. If no other repairs are suggested overall, we can successfully conclude that the incorrect program has been fully repaired for
Figure 13: An example of a program with its corresponding model depicting use of variables
all test cases. This technique is used during testing and experimentation.

6  Graph Matching

One of CLARA’s main limitations is that it requires both the correct and the incorrect programs to have the same control flow in order to proceed with the repair process. It is uncommon to find many programs containing the same control flow, thereby greatly reducing the number of programs CLARA can repair. Hu et al. [8] carried out experiments using CLARA where they reported that the repair process could not run on 35.5% of incorrect programs. Therefore, we propose a graph matching algorithm that creates an approximate matching between the correct and incorrect programs. This matching takes into consideration both semantic and topological information of the graphs. Additionally, the graph matching technique is centered around eliminating the control flow problem, taking into account semantic similarity to eliminate useless feedback. However, since this is an approximate matching, it is possible to get feedback that may suggest deleting and then adding the same variable, but that would always be to a different location. Figure 14 shows the code for a correct program and an incorrect program we will be using throughout this section. These programs take in as input, an array containing integers between 0 ans 100 and aim to find the minimum element as well as add all the elements in a. However, rather than finding the minimum element, the incorrect program never changes the value of m, meaning that the if condition on line 10 is never executed.

6.1  Graph Creation

Each model created by CLARA is based on the program’s control flow. Hence, we decided to utilize the information available to us in the model to create control flow graphs for both the correct and incorrect programs. Each location is a node and the location’s transitions are the edges. Since many transitions pointed to None, indicating that no transition exists, we
```python
def f(a):
    x = 0
    m = 101
    s = 0
    while (x < len(a)):
        s += a[x]
        if (m > a[x]):
            m = a[x]
        x+=1
    print(s + "," + m)
```

```python
def g(a):
    i = 0
    m = 101
    s = 0
    while (i < len(a)):
        s += a[i]
        if (m < a[i]):
            m = a[i]
        i+=1
    if (m == 0):
        i-=1
    print(s + "," + m)
```

Figure 14: Correct and incorrect programs used in graph matching

Figure 15: Control flow graphs derived from the programs in Figure 14
a = [1, 5, 6] → {ListInit, 1, 5, 6}
b = a[0] → {GetElement, 0}
c = b + 1 → {Add, 1}

\{ListInit, 1, 5, 6, GetElement, 0, Add, 1\}

Figure 16: A node’s expressions with their corresponding labels

decided to create a node for None. Since CLARA assigns locations starting from 1, to always avoid conflicts, we assigned the None node to location 0.

Each node of the control flow graph contains a location number, the corresponding location’s expressions, the starting line number of that location, a description, and a label. This label contains all the semantic information we can extract from the expressions of the node. Each expression contains semantic information in terms of constants and operation semantics. Operation semantics allow us to identify the type of statement, e.g., addition, subtraction. Unlike expressions, which can hold different values due to variables, we want the elements of a label to always remain constant for a particular expression. Therefore, variables are not added to the multiset of labels. Since expressions can be nested, each expression contains a multiset of labels. Therefore, the label of a node is a multiset of the labels of its expressions.

Figure 16 contains examples of Python expressions in a node and their corresponding labels. The left side shows the expressions and the right side shows the labels for each expression. Since a contains the initialization of a list with the elements 5 and 6, it contains the label \{ListInit,1,5,6\}. b contains a expression where we get the first element of a, so it has the label \{GetElement,0\}. Similarly, since c is made of the sum of b and 1, it contains the label, \{Add,1\}. Therefore, the label for the node is a union of all of these sets, i.e, \{ListInit,1,5,6,GetElement,0,Add,1\}. We exclude the variables a, b and c from the label of the node as the variables do not necessarily evaluate to the same value at every point in the program.
We iterate through the model of a function to create the nodes with the information mentioned above. After the nodes have been created, we connect the graph’s nodes with edges representing the transitions between the model locations. Each edge is annotated with a True or False label based on the transition. Figure 15 shows the control flow graphs for the source code of the correct program (Figure 15a) and incorrect program (Figure 15b) found in Figure 14. Each node in the figure contains the location number and its corresponding label. The expressions, line numbers, and descriptions have been removed of the images of the graphs to improve readability. The dotted arrows represent the False transitions, and the solid arrows represent the True transitions. The node with the label Empty represents a location in the program model which does not contain any expressions.

6.2 Graph Matching

Our graph matching process aims to find a mapping between the correct and the incorrect programs such that there are no control flow conflicts. Hu et al. [8] addressed this issue too by refactoring the correct program such that its control flow matched the incorrect program. They achieved this by defining a set of transformations, which, when applied, can modify the control flow of a program. These transformations are applied to the correct program until its control flow matches the incorrect program. Using program refactoring, it is possible to modify a program so thoroughly that it no longer resembles itself, and can potentially cause a correct program to become incorrect. Furthermore, the goal is to suggest changes for the incorrect program such that it is now correct. Because of these issues, we hypothesize that changing only the incorrect program is more appealing. Hence, we opted to modify the incorrect program only as those changes could be recorded and mentioned as feedback to the user.

In the interest of creating a mapping for the correct program graph and the incorrect program graph, we need to find a matching between the nodes of each graph. This matching takes into consideration both the semantic and topological information of the graph, where
Algorithm 2: Our Graph Matching

Input: $G_C = (U, E)$ control flow graph (correct program)  
       $G_I = (V, F)$ control flow graph (incorrect program)  

Output: bestMatch $= U \rightarrow V$ program matching

// Initialize all possible permutations and others
bestMatch $\leftarrow \{\}$

score $\leftarrow 0$

// We find the best possible matching from every permutation
for $\phi \in \text{Permutations}(U, V)$ do
  // For a possible match we initialize a temporary currScore of 0
  currScore $\leftarrow 0$

  for $(u, v) \in \phi$ do
    // We find the Jaccard index of the labels of u and v
    labelDist $\leftarrow \text{Jaccard}(\text{Label}(u), \text{Label}(v))$

    // We find the similarity between u and v
    // u' and u'' are the neighbors of u
    Let $u \xrightarrow{\text{True}} u', u \xrightarrow{\text{False}} u''$

    // v' and v'' are the neighbors of v
    Let $v \xrightarrow{\text{True}} v', v \xrightarrow{\text{False}} v''$

    edgeDist $\leftarrow 0.5$

    // We check if the True neighbors and False neighbors match
    with each other
    if $v' = \phi(u') \land v'' = \phi(u'')$ then
      edgeDist $\leftarrow 1$
    else if $v' \neq \phi(u') \land v'' \neq \phi(u'')$ then
      edgeDist $\leftarrow 0$
    end

    currScore $\leftarrow$ currScore + $0.5 \times (\text{labelDist} + \text{edgeDist})$

    // We update the best match if $\phi$ gives us the maximum score
    if currScore $> \text{score}$ then
      score $\leftarrow$ currScore
      bestMatch $\leftarrow \phi$
    end
  end
end

36
the topological information represents the control flow information. Algorithm 2 presents the pseudocode for our algorithm. It takes as input the two control flow graphs, $G_c$ and $G_t$. We explore all possible permutations between the nodes of both graphs. Since the number of locations in an introductory programming assignment model, typically range from 10 to 20, it is possible to have more than 3 million permutations. As that is a large number, in practice, we aim to find permutations with high scores first. To find the permutations, we start by creating a dictionary, where the keys are nodes from the smaller graph (containing fewer nodes), and values are the list of nodes from the bigger graph (containing more nodes). The list of values is sorted by the semantic similarity between the labels of the key and the value. Using the dictionary of sorted nodes, we create permutations between the two graphs. We then iterate through the list, assign a score to each matching and finally choose the mapping with the maximum score. The score is assigned based on the semantic and topological information, where both are assigned equal importance.

The semantic similarity is based on the Jaccard index between two labels. The Jaccard index is calculated using the Weighted Jaccard Similarity formula, where weight is the number of times the element appears in a multiset. This formula for the multisets $U$, and $V$ is defined as:

$$Jaccard (U, V) = \frac{\sum_{x \in U \cup V} min(W(x, U), W(x, V))}{\sum_{x \in U \cup V} max(W(x, U), W(x, V))}$$

where $W(x, S)$ is the number of times $x$ appears in $S$. This Jaccard index lies between 0 and 1, where 0 indicates low similarity and 1 indicates high similarity. Table 4 shows examples of labels and their corresponding Jaccard indexes.

<table>
<thead>
<tr>
<th>Label 1</th>
<th>Label 2</th>
<th>Jaccard Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetElement, 0, 0</td>
<td>GetElement, 0</td>
<td>0.667</td>
</tr>
<tr>
<td>GetElement, 0</td>
<td>GetElement, 0</td>
<td>1</td>
</tr>
<tr>
<td>cond, Lt, ind#0, len, iter#0</td>
<td>cond, Gt, ind#1, len, iter#1</td>
<td>0.25</td>
</tr>
<tr>
<td>ListInit, 5, 6</td>
<td>ListInit, 8, 9</td>
<td>0.2</td>
</tr>
</tbody>
</table>

Table 4: Labels and their corresponding Jaccard indexes
To calculate the topological information, we analyze the edges coming out of each node that we are trying to match. Since we have a node pointing to `None`, we know that there will always be two edges coming out of each node, a True edge and a False edge. Hence, to calculate the topological similarity, we analyze both the True edges together and both the False edges together by comparing whether the nodes the edges are pointing to match according to the current matching we are analyzing.

Figure 17 shows a matching between the control flow graphs shown in Figure 15. The correct program graph is on the left side, and the incorrect program graph is on the right side. The solid red lines indicate the matching between the nodes. As one can see in the figure, nodes \( v_8 \), \( v_9 \) and \( v_{10} \) did not match with any nodes in the correct program graph. These locations represent the second if condition found in Figure 14b on line 10.

To understand how the final mapping is chosen lets consider nodes \( u_5 \), \( u_6 \), and \( u_7 \) from the correct program. We have the two mappings, \( \{ \phi_1(u_5) = v_5, \phi_1(u_6) = v_6, \phi_1(u_7) = v_7 \} \), and \( \{ \phi_2(u_5) = v_8, \phi_2(u_6) = v_6, \phi_2(u_7) = v_7 \} \). In \( \phi_2 \), we match the node \( u_8 \), with the label, `cond, Gt, GetElement` to the node \( v_8 \), of the incorrect program graph, with the label, `cond, Eq, 0`.

Figure 17: Matching between the control flow graphs from Figure 15.
cond. Eq. 0. We start by matching the labels of $u_5$ and $v_8$ and find the Jaccard index, which comes out to be 0.2. After the node labels, we match the edges. The True edge of $u_5$ goes to $u_6$ and the True edge of $v_8$ goes to $v_9$. However, $\phi_2(u_6) = v_6$, and not $v_9$. Next, we have to analyze the False edges. $u_5$ points to $u_7$ with the label and $v_8$ points to $v_{10}$. However, $\phi_2(u_7) = v_7$, and not $v_{10}$. Since neither of the edges match with each other, edgeDist is assigned to 0, making the score, $0.5 \times (0.2 + 0) = 0.1$.

In $\phi_1$, we match $u_5$ with $v_5$ from the incorrect program, with the label cond, Lt, GetElement. The Jaccard index for the labels of $u_5$, and $v_5$ is 0.5. The True edge of $v_5$ points to $v_6$, and $\phi_1(u_6) = v_6$, so we have a match. Similarly, the False edge of $v_5$ points to $v_7$, and $\phi_1(u_7) = v_7$. Therefore, edgeDist is assigned to 1. So the similarity weight for $u_5$, and $v_5$ is $0.5 \times (0.5 + 1) = 0.75$. Since the matching $u_5$ with $v_5$ has a greater similarity score than $v_8$, we pick $\phi_1$ over $\phi_2$ for the overall matching.

### 6.3 Recreating the Model

Now that we have a possible matching between the control flow graphs of the correct program and the incorrect program, we have to recreate the input that CLARA’s repair and matching processes require. Since both the processes receive models for each program, we utilize the result of the node mapping and the correct program model to recreate the model for the incorrect program. There are three possibilities while recreating the model. The first is that the correct program graph is smaller, meaning that it has fewer nodes than the incorrect program graph. In this case, locations need to be removed from the incorrect program model in the new model. Such is the case for the example we have followed so far. The second case is where the correct program graph is larger, meaning it has more nodes, implying that locations need to be added to the model of the new incorrect program. The third is that both graphs are the same size, i.e., they have the same number of nodes. In this case, nodes will neither be removed nor added to the new model, but the edges or nodes might need rearrangement. Therefore, based on the matching and number of nodes in the graph of the
correct program, we recreate the model of the incorrect program.

Algorithm 3 contains the pseudocode used to recreate the incorrect program model. It receives as input the control flow graphs for the correct \( G_C \), the incorrect \( G_I \) programs, the matching between the nodes \( \phi \), where the keys are nodes from the correct program, and the values are nodes from the incorrect program. We return \( M_I \), the updated version of the incorrect program model. If the correct program graph is smaller, we delete the locations not mapped in \( \phi \). For every location that is deleted, its corresponding expressions and transitions are deleted too. On the contrary, if the correct program is larger, we create locations for every node that exists in the correct program graph but is not mapped in \( \phi \), and add them to both \( \phi \) and the incorrect program model. The new locations do not have any expressions added to them yet. CLARA will suggest any addition of new expressions during the repair process because we do not know the corresponding variables associated with those expressions. We then iterate through \( \phi \) and update the transitions of the incorrect program model to match the correct program.

Figure 18 shows the control flow graphs for the incorrect program found in Figure 14b.
Algorithm 3: Recreating the Incorrect Model

**Input:** \( G_C = (U, E) \) control flow graph (correct program)  
\( G_I = (V, F) \) control flow graph (incorrect program)  
\( \phi = U \rightarrow V \) program matching  

**Output:** \( M_I = (W, J) \) updated incorrect program model

\( W \leftarrow V, J \leftarrow \emptyset \)

// We remove the locations that were not mapped from the incorrect model if the \( G_C \) is smaller than \( G_I \)

if \( |U| < |V| \) then

    for \( v \in V \setminus U \) do
        \( W \leftarrow W \setminus \{v\} \)
    end

end

// We add the locations that were not mapped to the incorrect model if the \( G_C \) is bigger than \( G_I \)

if \( |U| > |V| \) then

    for \( u \in U \land u \notin \text{dom}(\phi) \) do
        \( v' \leftarrow \text{CreateNewLocation()} \)
        \( W \leftarrow W \cup \{v'\} \)
        \( \phi(u) \leftarrow v' \)
    end

end

// We update the transitions to reflect the edges in \( G_C \)

for \( (u, v) \in \phi \) do

    // \( u' \) and \( u'' \) are the neighbors of \( u \)

    Let \( u \xrightarrow{\text{True}} u', u \xrightarrow{\text{False}} u'' \)

    \( J \leftarrow J \cup \{v \xrightarrow{\text{True}} \phi(u'), v \xrightarrow{\text{False}} \phi(u'')\} \)

end
Figure 18a represents the old control flow graph and Figure 18b represents the new graph we construct after running Algorithm 3. The model can be extracted from this graph. As one can see, the new graph is smaller i.e contains fewer locations as the edges and locations of the extra if condition have been removed, i.e, 8, 9 and 10 represented by $v_8$, $v_9$, and $v_{10}$. Because we map edges from the correct graph to the new model, one new edge is added, indicated using a green arrow from location 7 to location 2, which represents a True transition. No other edges needed reordering.

7 Feedback

Since one of the main features of CLARA is that it provides feedback to users about how to repair a program, it is paramount that a novice programmer can understand this feedback. Even though CLARA formats its expressions while printing them out to the console, one needs to be very familiar with the CLARA’s AST processor’s jargon. While some terms may be easy to understand, such as Add, Sub, StrAppend, others, such as ite, AssAdd, #iter can be difficult to understand. Therefore, in our attempt to make CLARA more flexible and user-friendly, we changed CLARA’s feedback processor.

We wrote a pretty printer for the feedback that would take an expression and return what the expression would look like in Python. Since expressions can be heavily nested, it was very challenging to process them so that a novice programmer understands them, follows general programming practices, and results in the original expression when modeled. Let us consider the simple example shown on the left side of Figure 19. CLARA returns the expression, $x := \text{GetElement}(	ext{ListInit}(5, 6), 0)$, which translates to $x = [5, 6][0]$. If we print out the translated expression without further processing, a student can still get confused. In general programming practices, if we wanted to assign a value 6 to x, we would never construct a list, add two elements, and then extract an element out of it. Additionally, even if the student did understand the statement $x = [5, 6][0]$, it might give him or her the incorrect idea that this
**Original Expression:**
x := GetElement(ListInit(5,6),0)

**Translated Expression:**
x = [5, 6][0]

**Our Version:**
x = [5, 6]
x = x[0]

**Original Expression:**
y := ite(Eq(Add(Sub(x, c), 1), 0), int(Mult(x, 2)), USub(1))

**Translated Expression and Our Version:**
if ((x + c + 1) == 0):
y = int(x * 2)
else:
y = (-1)

Figure 19: Examples of CLARA’s feedback, their translations and our feedback

is how code is written because the statement is returned by a programming tool. We prefer to return its evaluated expression, \( x = 5 \). However, \( x = 5 \) models to \( x := 5 \), which is not the same as the expression suggested by CLARA. Therefore, we further process this statement by breaking up the expression into two and returning both of them, as that is readable and will always result in the expression, \( x := \text{GetElement(ListInit(5,6),0)} \), when modeled.

The expression of the right side of figure 19 corresponds to an if condition. If the body of both the if and else conditionals are defining the same value, or if we have an if condition without the else conditional, CLARA converts the conditional into an \textit{ite} expression with three arguments, i.e, conditional, if expression body and else expression body. Since \textit{ite} is a CLARA term, it is necessary to print it as an if condition.

Our graph matching technique, which includes adding and removing locations, resulted in processing and providing more feedback. However, since locations depend on different control flow statements, specialized feedback must be provided. Since this feedback will need to consider conditionals, loops, and changes in location transitions, processing this feedback is out of the scope of this thesis. For now, we are outputting the number of locations added or deleted. Additionally, in the case of deletions, all expressions inside the locations which would also be deleted, are added to the feedback.
8 Results

This section will highlight the experiments we ran and quantitatively analyze those results. In our experiments, we utilized valid submissions for the programming problem 1560B, Who’s Opposite?, problem 1554A, Cherry, and problem 977C, Less or Equal from the online programming website CodeForces. For the incorrect submissions, we considered a submission to be valid if it is incorrect due to failure to pass a test case, not if it is considered incorrect for other reasons, such as syntax errors and timeouts. These programming problems were chosen as they fulfilled the condition of having at least one for/while loop, if condition, standard input, standard output, and had a sufficient number of user submissions. For problem 1560B, there were 835 correct user submissions and 510 incorrect user submissions; for problem 1554A, there were 613 correct user submissions and 319 incorrect user submissions, and for problem 977C, we had 814 correct user submissions and 2721 incorrect user submissions. The incorrect submissions were further filtered out based on the availability of valid test cases, resulting in 223 valid incorrect submissions for problem 1560B, 137 valid incorrect submissions for problem 1554A, and 162 valid incorrect submissions for problem 977C.

8.1 Test Cases

Since CLARA uses test-based repair, we must extract as many test cases as possible and repair the incorrect program with the test case that fails. While evaluating whether a program is correct or incorrect, CodeForces runs the program on several test cases. If a program does not pass a particular test case, the process stops, and the program is declared incorrect. Therefore, we only have access to the first test case that is not passed for every incorrect program. Furthermore, because CLARA uses variable tracing to find repairs in the program, we only need the test case input, not the output it is supposed to produce. For test cases that
are longer than 50 lines, rather than displaying the entire test case, CodeForces shows ... at
the end, leading to the test case being incomplete, inaccessible, and therefore, invalid. Thus,
we have to filter test cases based on validity. Problem 1560B has 16 test cases, of which 15
are valid. Similarly, problem 1554A has 28 test cases, of which 3 are valid. Therefore, all
incorrect programs that failed on an invalid test case were removed from the data set of the
experiments.

8.2 Experiments

All of the incorrect programs taken into account failed on valid test cases during the experi-
ments. They did not contain any language constructs that could not be parsed by CLARA,
e.g., lambda expressions or functions involving the sys library. These programs were all
unique and were made by different users. Since the number of nodes in each control flow
graph of a program could range from 5 to 25, while testing the graph matching technique,
we set a permutation limit of 1000. The best node mapping was chosen from these 1000
permutations. Because the possible values for each node were sorted by semantic similarity
before creating the permutations, after experimentation, we found that the best mapping
existed within the first 1000 permutations and was computed in under a minute. While
calculating the similarity score, if the score fell below 0.6, we halted the comparison. We did
not repair that specific incorrect program with the corresponding correct program as a low
score indicates that the programs are very different. This score limitation helped us avoid a
repair where the correct program would replace the entire incorrect program. Additionally,
we set a timeout of 5 minutes.

We ran three experiments, all involving CLARA’s repair process. The first experiment
(CLARA) included running CLARA with the added modifications described in Section 4 but
without the addition of the graph matching technique. The other two experiments involved
the graph matching technique. In the second (AGM(L)) experiment, we made use of only the
semantic information during the graph matching process to select the best mapping, and in
the third experiment (\textsc{AGM}(L + E)) we took both semantic and topological information equally into consideration during the graph matching process.

For every experiment, we analyzed the data using five metrics.

1. Control flow conflicts: the number of structure mismatches or the number of times we found a control flow conflict between a pair of correct and incorrect programs.

2. Comparisons repaired: the number of comparisons repaired out of the total number of comparisons. We consider a program repaired if, after the repairs suggested by CLARA are applied, no other repairs are suggested for any of the other test cases.

3. Total comparisons: The total number of comparisons run for the problem.

4. Incorrect programs repaired: the number of incorrect programs repaired.

5. Total incorrect programs: the total number of valid incorrect programs available to repair the problem.

6. Correct programs needed to minimize modification percentage: We find the correct program that minimizes its repair modification percentage for every incorrect program repaired. This metric gives us the total number of the correct programs needed such that the modification percentage is minimized for every incorrect program. Every problem used a total of 50 correct programs.

7. Minimum number of correct programs needed for all repairs: the minimum number of unique correct programs overall that repaired all incorrect programs. Every problem used a total of 50 correct programs.

We also noted the percentage of the incorrect program modified by a specific correct program for every incorrect program and the earlier metrics. The percentage is calculated based on the number of repairs suggested and the number of expressions in the correct program, and the original incorrect program. These percentages are only calculated for
Table 5: Results for Problems 1560B using 50 correct programs sorted by date based on experimental settings CLARA, AGM(L), AGM(L + E). CLARA is original CLARA with the modifications described in Section 4. AGM(L) and AGM(L + E) are respectively our approximate graph matching approach using only labels and labels and edges.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>CLARA</th>
<th>AGM(L)</th>
<th>AGM(L+E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control flow conflicts</td>
<td>9135</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Comparisons repaired</td>
<td>373</td>
<td>1478</td>
<td>1461</td>
</tr>
<tr>
<td>Total comparisons</td>
<td>11150</td>
<td>9228</td>
<td>9304</td>
</tr>
<tr>
<td>Incorrect programs repaired</td>
<td>79</td>
<td>163</td>
<td>164</td>
</tr>
<tr>
<td>Total incorrect programs</td>
<td>223</td>
<td>223</td>
<td>223</td>
</tr>
<tr>
<td>Correct programs needed to minimize modification percentage</td>
<td>19</td>
<td>18</td>
<td>20</td>
</tr>
<tr>
<td>Minimum number of correct programs needed for all repairs</td>
<td>10</td>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

programs that were successfully repaired. A higher percentage indicates more changes are required to repair the incorrect program; therefore if the percentage is 100%, every expression in the incorrect program is replaced by an expression from the correct program. Thus, we found the minimum number of unique correct programs needed to repair all of the programs such that the percentage for each repair is minimized, and the minimum number of unique correct programs overall that repaired all of the incorrect programs. We also found the associated average percentage of modifications in both cases, given in box and whisker plots. We also analyzed the similarity between correct and incorrect programs using our graph matching technique for each problem. We used both AGM(L), and AGM(L + E) to generate the similarity scores. These similarity scores are also displayed using box and whisker plots.

8.3 Quantitative Analysis

8.3.1 Problem 1560

We choose all 223 valid incorrect programs and 50 most recent correct programs sorted by date. However, while running the graph matching algorithm, since we added a score limit restriction, not all comparisons fully ran in AGM(L) and AGM(L + E). A summary of the results
Figure 20: Repair modification percentages for Problem 1560B for experiments CLARA, AGM(L) and AGM(L + E). The left contains box plots for percentages whilst minimizing the number of correct programs used and the right contains box plots for overall minimum percentages for each repair using any incorrect program.

is given in Table 5. The columns represent the experiment number, and the rows represent the corresponding values of that experiment. Overall, AGM(L) and AGM(L + E) performed the best, indicating that the graph matching technique aided in repairing many more incorrect programs overall. Both experiments entirely overcame the problem of differences in control flow and successfully repaired a minimum of 163 out of 223 incorrect programs, resulting in a minimum 37% improvement in the success rate. We analysed the minimum number of correct programs needed to repair all programs and its corresponding minimum percentage and the minimum percentage of modification overall. Both percentage box and whisker plots are given in Figure 20. As one can see, the average percentage difference between minimizing the number of correct programs and the percentage itself is 5%. However, the number of correct programs between the two techniques decreased by more than half for both AGM(L) and AGM(L + E).

We also calculated the similarity score between all the correct programs and the incorrect programs. There are at least 2 incorrect programs that are exactly the same but none in the case of correct programs. However, the medians for both the correct and the incorrect programs are similar, around 0.7.
8.3.2 Problem 1554A

To create the data set, we sorted all the correct programs by date and chose the 50 most recent correct programs, where the same person wrote no more than one correct program and 137 valid incorrect programs. Table 6 contains the results from running the experiments. As one can see, $\text{AGM}(L)$ and $\text{AGM}(L + E)$ had no control flow conflicts, therefore solving the control flow problem. They repaired a minimum of 77 incorrect programs out of 137, which is a 28% improvement over CLARA.

Using the modification percentages, we generate the box and whisker plots in Figure 22. The right contains box plots for the minimum modification percentage for each incorrect program repaired. The left contains box plots for the modification percentage of each repaired incorrect program, given the minimum set of correct programs that repair all the incorrect programs. As one can see $\text{AGM}(L)$ produced better percentages than $\text{AGM}(L + E)$. Additionally, there is a maximum 7% increase in the maximum percentage whilst trying to minimize the number of correct programs needed. While this increase is not high, less than half the number of correct programs are needed to repair all 81 incorrect programs. We also analysed the similarity between all the correct programs and all the incorrect programs using both $\text{AGM}(L)$
Table 6: Results for Problems 1554A using 50 correct programs sorted by date based on experimental settings CLARA, AGM(L), AGM(L+E). CLARA is the original CLARA with the modifications described in Section 4; AGM(L) and AGM(L+E) are respectively our approximate graph matching approach using only labels and labels and edges.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>CLARA</th>
<th>AGM(L)</th>
<th>AGM(L+E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control flow conflicts</td>
<td>5688</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Comparisons repaired</td>
<td>318</td>
<td>599</td>
<td>648</td>
</tr>
<tr>
<td>Total comparisons</td>
<td>6850</td>
<td>5472</td>
<td>6231</td>
</tr>
<tr>
<td>Incorrect programs repaired</td>
<td>39</td>
<td>81</td>
<td>77</td>
</tr>
<tr>
<td>Total incorrect programs</td>
<td>137</td>
<td>137</td>
<td>137</td>
</tr>
<tr>
<td>Correct programs needed to minimize modification percentage</td>
<td>13</td>
<td>17</td>
<td>20</td>
</tr>
<tr>
<td>Minimum number of correct programs needed for all repairs</td>
<td>6</td>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

Figure 22: Repair modification percentages for Problem 1554A for experiments CLARA, AGM(L) and AGM(L+E). The left contains box plots for percentages whilst minimizing the number of correct programs used and the right contains box plots for overall minimum percentages for each repair using any incorrect program.
and AGM(L+E), shown in Figure 23. The figure indicates that the correct programs were similar but not of them were exactly the same, as in the case of the incorrect programs. Additionally AGM(L+E) seemed to produce higher scores overall, indicting somewhat similarity in the control flow, while not necessarily in the labels. The experimental results indicate that AGM(L) performed slightly better than AGM(L+E) in terms of unique incorrect programs repaired, minimum unique correct programs needed, and the average of the percentage of programs modified.

8.3.3 Problem 977C

The data set for this problem consisted of 50 correct programs sorted by date and 162 valid incorrect programs. This problem had 14 valid test cases. AGM(L) repaired 22% more programs than CLARA and AGM(L+E) repaired 34% more programs than CLARA. Similar to the other problems, AGM(L), and AGM(L) produced no structure mismatches, indicating that the control flow limitation has been completely overcome. However, unlike the other two problems, the difference in the number of correct programs needed to minimize the modification percentage, versus the minimum number of correct programs overall.

Figure 24 contains box plots for the percentage modified for each repair while minimizing
Table 7: Results for Problems 977C using 50 correct programs sorted by date based on experimental settings CLARA, AGM(L), AGM(L + E). CLARA is original CLARA with the modifications described in Section 4. AGM(L) and AGM(L + E) are respectively our approximate graph matching approach using only labels and labels and edges.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>CLARA</th>
<th>AGM(L)</th>
<th>AGM(L+E)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Control flow conflicts</td>
<td>6668</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Comparisons repaired</td>
<td>12</td>
<td>65</td>
<td>105</td>
</tr>
<tr>
<td>Total comparisons</td>
<td>8100</td>
<td>7482</td>
<td>7332</td>
</tr>
<tr>
<td>Incorrect programs repaired</td>
<td>10</td>
<td>46</td>
<td>65</td>
</tr>
<tr>
<td>Total incorrect programs</td>
<td>162</td>
<td>162</td>
<td>162</td>
</tr>
<tr>
<td>Correct programs needed to minimize modification percentage</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>Minimum number of correct programs needed for all repairs</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Figure 24: Repair modification percentages for Problem 977C for experiments CLARA, AGM(L) and AGM(L + E). The left contains box plots for percentages whilst minimizing the number of correct programs used and the right contains box plots for overall minimum percentages for each repair using any incorrect program.
Figure 25: Similarity scores for Problem 977C for experiments AGM(L) and AGM(L + E)

the overall percentage(right) and minimizing the number of correct programs needed(left). As you can see both AGM(L), and AGM(L + E) produced similar modification percentages. AGM(L + E) produced more outliers than AGM(L), indicating that AGM(L) performed better at aligning the correct and incorrect program for this problem. Figure 25 contains box plots for the similarity between all the correct programs and the incorrect programs. Unlike the problems above, both the correct and the incorrect programs had at least two of the same programs.

8.4 Qualitative Analysis

This section will analyze four different incorrect programs, where the first three are from Problem 1560B and the last incorrect program is from Problem 1554A. The first three incorrect programs were repaired by the same correct program\textsuperscript{5}. We will refer to this correct program as C. The code for C is given in Figure 26a. It contains a for loop that runs for every input. It extracts the three input values, a, b, and c and uses a, and b to calculate the total number of people n. It then checks if any input value is out of bounds, and if so, outputs -1 to the console. Otherwise, it calculates the position of the person opposite c and

\footnote{https://codeforces.com/problemset/submission/1560/130276184}
tc = int(input())
for i in range(tc):
t = input().split()
x = int(t[0])
y = int(t[1])
z = int(t[2])
a = abs(x - y)
if a * 2 < max(x, y, z):
    print(-1)
else:
z += a
if z > a * 2:
z -= a * 2
print(z)

(a) Correct Program

(b) Incorrect Program 1

Figure 26: Correct and Incorrect Program 1 from Codeforces solving problem 1560B

We will be using C as it repaired many programs throughout all 4 of the experiments. C repaired 34 programs in CLARA, 74 programs in AGM(L + E), and 70 programs in AGM(L). We will be highlighting the repairs suggested by C and three different incorrect programs, along with the number of locations added and removed. We assume the feedback for location addition/removal and transition changes has already been given. Only the feedback suggested by CLARA for the modified incorrect program will be shown. This is because the feedback for location addition and removal must be processed before it is provided to students. We will be looking at processing this feedback in our future work. After implementing the feedback suggested by CLARA, all incorrect programs mimic the correct program in terms of both code and functionality.

8.4.1 Incorrect Program 1

Figure 26b contains the source code for incorrect program 1. C repaired this incorrect
tc = int(input())
for i in range(tc):
t = input().split()
x = int(t[0])
y = int(t[1])
z = int(t[2])
a = abs(x - y)
if a * 2 < max(x, y, z):
    print(-1)
else:
z += a
if z > a * 2:
z -= a * 2
print(z)

import math
for _ in range(int(input())):
a, b, c = map(int, input().split())
if (abs(a - b) == 1) or a < b:
    print(-1)
elif c > min(a, b) and c < max(a, b):
    print(c + abs(b - a))
else:
    print(-1)

(a) Correct Program  
(b) Incorrect Program 2

Figure 27: Correct and Incorrect Program 2 from Codeforces solving problem 1560B

The answer calculation and the boundary check are incorrect in this incorrect program. Since there was no structure mismatch, no locations were added or deleted. The feedback for AGM(L + E) and AGM(L) is the same. All sets of feedback are in Figure 30 of Appendix A. This is because the print statements have been rearranged by the graph matching technique, resulting in a lower cost. CLARA suggests to replace lines 4, 5, 7, and 11 of Figure 26b with lines 3, 7, 8, and 14 of Figure 26a respectively. It also suggests adding the if condition on line 12 of Figure 26a.

8.4.2 Incorrect Program 2

Figure 27b contains the source code for incorrect program 2. This incorrect program has more locations than C due to the extra if condition and the print statement. The incorrect program also calculates its final answer based on the absolute difference between a, and b but does not account for the answer being greater than the total number of people. Hence, experiment 1 produced a structure mismatch and did not successfully run. The feedback for
1  tc = int(input())
2  for i in range(tc):
3      t = input().split()
4      x = int(t[0])
5      y = int(t[1])
6      z = int(t[2])
7      a = abs(x−y)
8      if a * 2 < max(x, y, z):
9          print(−1)
10     else:
11          if z > a * 2:
12              z += a
13          else:
14              if z == a * 2:
15                  print(z)

(a) Correct Program

(b) Incorrect Program 3

Figure 28: Correct and Incorrect Program 3 from Codeforces solving problem 1560B

experiments 2 and 3 is given in Figure 31 of Appendix A. Both sets of feedback differ by
one statement, which is an extra delete suggestion produced because of the graph matching
technique. During the graph matching process, 5 locations were deleted for both experiments.
Both experiments had a modification percentage of 22%. In AGM(L + E), the graph matching
algorithm deletes lines 4 and 5 from the incorrect program, and in AGM(L), only line 4 is
suggested. CLARA suggests to add lines 1, 3, 7, 12, and 13 from Figure 27a to the incorrect
program. In addition to the suggestions mentioned above, it suggests replacing lines 6, 7,
and 9 of Figure 27b with lines 8, 9, and 14 of Figure 27a.

8.4.3 Incorrect Program 3

Figure 28b contains the source code for incorrect program 3. Incorrect Program 3 has
incorrect conditionals for boundary checks and the final answer. This incorrect program has
one less location than C due to an empty location added by CLARA after the if condition
while creating the model for the correct program. The difference in locations causes a structure
mismatch while running CLARA’s repair process during experiment 1. In experiment 3, the program was not repaired due to a timed out. However, experiment 2 successfully repaired the program and modified it by 16%. The feedback is given in Figure 32 of Appendix A. One location was added during the graph matching process, and 1 location was added which is an empty location. CLARA suggests adding lines 1, 12, and 13 of the correct program. Additionally it suggests to replace line 9 of Figure 28b by line 14 from Figure 28a.

8.4.4 Incorrect Program 4

This incorrect program and its corresponding correct program are from Problem 1554A. The code for both programs is given below. Both the correct and incorrect programs are very similar. The only difference between the programs is the print statement on line 8. Rather than printing after the for loop, the incorrect program contains the statement inside the for loop, printing at every iteration. This incorrect program was not repaired by CLARA but was repaired by both AGM(L) and AGM(L + E). Since print statements have their location, the only difference between the models was one edge. Due to the edge difference, CLARA could not repair the incorrect program. Since the graph matching algorithm adjusts edges according to the correct program, after recreating the new incorrect model, no further changes were required by CLARA to repair the program. Therefore, CLARA did not return any feedback, resulting in a modification percentage of 0%.

9 Future Work

Currently, our work only supports Python. Since CLARA works on C++, Java, and Python, we hope to extend our work to all three languages. All three languages in CLARA have different AST processes and interpreters, so our work will not automatically be extended. Additionally, due to the addition of the graph matching process, we need to process the feedback returned by CLARA before it is provided to students. Based on the locations added
or deleted and the edges modified, we need to be able to inform the user to add/delete the corresponding expressions. Another limitation of the graph matching process is that while removing locations from the incorrect program model, it is possible to remove a variable with no other definition in the rest of the program. Therefore, causing the program to crash if it is used. We hope to address this issue in the future by checking if deleting a variable can cause the program to crash and, if so, adding that variable and its expression to a different location. Finally, we hope to make CLARA available online, allowing it to take in programs and test cases and return the repairs that need to be made. This will make CLARA easily accessible to both students and faculty.

10 Conclusion

This thesis aimed to extend the source code of CLARA, enabling it to work better with real-world programs. Therefore we extended CLARA by adding support for common Python language constructs such as standard input, output, nested functions, import statements, and Boolean variables. We also propose a graph matching technique between the correct and incorrect programs to aid in surpassing the limitation imposed due to control flow. However,
with the addition of this technique, the current feedback to students needs further processing and can not currently be delivered as-is. In the future, we hope to post-process this feedback and extend our work to the other languages supported by CLARA.
A Feedback from Experiments

* Change 'k = (2 * abs((a - b)))' to 'k = split(input_val_)' at line 4 of the incorrect program and at line 3 of the correct program

* Change
  if (c <= (k / 2)):
    d = (c + abs((a - b)))
  else:
    d = (c - abs((a - b)))
  to 'd = abs((a - b))' at line 5 of the incorrect program and at line 7 of the correct program

* Change '$cond = ((d in [a, b]) or (d > k))' to '$cond = ((d * 2) < max(a, b, c))' the condition of the if-statement at line 7 of the incorrect program and at line 8 of the correct program

* Add assignment
  if ((c + d) > (d * 2)):
    c = ((c + d) - (d * 2))
  else:
    c = (c + d)
inside the else-branch starting at line 10 of the incorrect program and inside the else-branch starting at line 11 of the correct program

* Change 'print(d)' to 'print(c)' at line 10 of the incorrect program and at line 14 of the correct program

(a) Feedback for Incorrect Program 1 for Experiment 1

(b) Feedback for Incorrect Program 1 for Experiment 2 and 3

Figure 30: Feedback for Incorrect Program 1
* Delete `$cond = ((abs((a - b)) == 1) or (a < b))` the condition of the if-statement at line 4 of the incorrect program
* Delete `print((-1))` at line 9 of the incorrect program
* Add assignment `$new_tc = int(input_val)` around the beginning of function main
* Add assignment `$new_a = abs((a - b))` inside the body of the 'for' loop beginning at line 3
* Add assignment `$new_t = split(input_val_)` inside the body of the 'for' loop beginning at line 3
* Change `$cond = ((c > min(a, b)) and (c < max(a, b)))` to `$cond = (($new_a * 2) < max(a, b, c))` at line 6 of the incorrect program and at line 8 of the correct program
* Change `print((c + abs((b - a))))` to `print($out)` at line 7 of the incorrect program and inside the else-branch starting at line 11 of the correct program
* Add assignment
  if (($c + $new_a) > ($new_a * 2)):
    c = (($c + $new_a) - ($new_a * 2))
  else:
    c = ($c + $new_a)
the print function at line 7 of the incorrect program and inside the else-branch starting at line 11 of the correct program
* Change `print($out)` to `print($out)` after the if-statement beginning at line 6 of the incorrect program and at line 14 of the correct program

(a) Feedback for Incorrect Program 2 for Experiment 2

* Delete `$cond = ((abs((a - b)) == 1) or (a < b))` the condition of the if-statement at line 4 of the incorrect program
* Add assignment `$new_tc = int(input_val)` around the beginning of function main
* Add assignment `$new_a = abs((a - b))` inside the body of the 'for' loop beginning at line 3
* Add assignment `$new_t = split(input_val_)` inside the body of the 'for' loop beginning at line 3
* Change `$cond = ((c > min(a, b)) and (c < max(a, b)))` to `$cond = (($new_a * 2) < max(a, b, c))` at line 6 of the incorrect program and at line 8 of the correct program
* Change `print((c + abs((b - a))))` to `print($out)` at line 7 of the incorrect program and inside the else-branch starting at line 11 of the correct program
* Add assignment to
  if (($c + $new_a) > ($new_a * 2)):
    c = (($c + $new_a) - ($new_a * 2))
  else:
    c = ($c + $new_a)
the print function at line 7 of the incorrect program and inside the else-branch starting at line 11 of the correct program
* Change `print((-1))` to `print(c)` at line 5 of the incorrect program and at line 14 of the correct program

(b) Feedback for Incorrect Program 2 for Experiment 3

Figure 31: Feedback for Incorrect Program 2
* Add assignment `new_tc = int(input_val)` around the beginning of function main
* Add assignment `y = split(input_val_)` inside the body of the 'for' loop beginning at line 2 of the incorrect program and at line 3 of the correct program
* Change
  
  ```
  if (((x - c) + 1) == 0):
      y = int((x * 2))
  else:
      y = ((x - c) + 1)
  ```
  to `y = y` after the if-statement beginning at line 4 of the incorrect program and inside the else-branch starting at line 11 of the correct program
* Add assignment
  
  ```
  if ((c + x) > (x * 2)):
      c = ((c + x) - (x * 2))
  else:
      c = (c + x)
  ```
  after the if-statement beginning at line 4 of the incorrect program and inside the else-branch starting at line 11 of the correct program
* Change `print(y)` to `print(c)` at line 9 of the incorrect program and at line 14 of the correct program

Figure 32: Feedback for Incorrect Program 3
References


