Autoval: Towards Automatic Evaluation of Introductory Programming Assignments

Manas Mhasakar

Under the supervision of Dr. Anup Mathew & Dr. Mattias Ulbrich

Abstract

Grading and providing feedback on assignments in introductory programming courses is a time-consuming task for the instructor. Existing tools generate feedback based on failing test cases, which is both incomplete and insufficient for students looking for more guided feedback. Moreover, even after an incorrect program is identified, fixing the error is non-trivial for students in an introductory programming class. This paper presents an approach that determines the correctness of a programming assignment without requiring a test suite and a program-repairing method for introductory programming assignments. The incorrect program is repaired, and the modifications in the program are then provided as feedback to the student who can use it as guidance in the course. The instructor is also provided with information on common semantic errors made by their students that they can analyze to determine the learning preference, strengths, and weaknesses of the overall class.

Chapter 1

Introduction

1.1 Overview

In programming education, automatic assignment evaluation and feedback generation are highly useful. Students who are part of the introductory programming course offered by most technical universities have to submit multiple programming assignments throughout this course. These assignments are usually graded against pre-defined testcases. Often, these testcases have to be carefully picked by the instructor to ensure coverage of the entire code. This high-quality test suite generation is a tedious and expensive task requiring substantial effort from the instructor. Furthermore, due to limited resources, it might not be practicable to provide a full test suite for a programming problem.

The students' assignments are then verified against these testcases, and in current techniques, feedback is given with a failing testcase. This type of feedback is not sufficient as it does not help students understand their errors and improve their solutions. A better feedback system should allow the students to find and repair the incorrect statements in their submissions.

Another problem with this system is the lack of feedback for the instructor. The information from the failing testcases may not be sufficient for the instructor to understand the common semantic errors made by the students. To understand such common errors, the instructor may have to go through the students' submissions. This itself is time-consuming as the students' submissions may be syntactically different and it may require time and effort to interpret them. A better feedback system for the instructor should allow the instructor to understand the common semantic errors made by the students.

1.2 Problem Statement

Given a set of student submissions to a programming assignment and a reference solution, the goal of our work is to

- Automatically evaluate student programming submissions according to a reference implementation without requiring a predefined test suite
- Provide feedback to students with incorrect submissions irrespective of their program's control flow graph by making modifications that make their program correct
- Provide feedback to the instructor by providing information on common semantic errors made by students without having the instructor to analyze each program
- Is sufficiently fast to be used in a classroom setting of a large public university

1.3 Existing Approaches and their limitations

Table 1.1 provides a summary of some prominent automated program evaluation and repair tools.

Tool	Automated Evaluation of Programming Assignments	Generate feedback and repair programs beyond identical Control Flow Graph	Generates Feedback for Instructors using Reference Solution
Clara	✓	×	×
Autograder	✓	×	×
Our Approach	✓	✓	✓

Table 1.1: Programming assignment evaluation tool comparison

Autograder [6] automatically determines the correctness of a student solution by checking equivalence with a reference solution. However, it cannot repair the incorrect program or/and provide guided feedback for students to progress towards a correct solution. Another state-of-the-art automated evaluation and repair tool, Clara [4] cannot repair the incorrect student program and has a different control flow graph from the instructor's reference solution.
In this work, we propose an approach to automatically evaluate and verify the correctness of a student program, provide guided feedback to the student to make progress toward the correct solution, and provide general feedback to the instructor to understand common semantic errors.

Chapter 2

Background

2.1 Equivalence Checking

Equivalence Checking [3] is the problem of formally proving two functions or programs are semantically equivalent. To prove a pair of programs are semantically equivalent, two steps are required

- Product Program Construction
- Proving invariant of the constructed program

```
function1(*)
{
    return a;
}
```

Listing 2.1: Program1

```
function2(*)
{
    return b;
}
```

Listing 2.2: Program2

```
function1(*)
{
    return a;
}

function2(*)
{
    return b;
}

composition() {
    assert(function1() == function2());
}
```

Listing 2.3: Product Program

For example, A simple product program for programs 2.1 and 2.2 can be 2.3. In the product program, the assertion condition is used to prove the invariant; the output of two programs should be equal. The main goal is to find property violations and limitations in the program's behavior using counterexamples. This can be proved using

- Symbolic Model Algorithms
- Bounded Model Checking Algorithms

2.1.1 Symbolic Model Checking

Symbolic model checking is a technique used to verify the temporal logic properties of finite state machines formally. Instead of explicitly constructing the graph for FSMs, by using a formula in quantified propositional logic, symbolic model-checking algorithms express the graph implicitly. We use **Klee** [2] - a dynamic symbolic execution engine built on top of the LLVM compiler infrastructure. Klee provides the ability to verify equivalence for a pair of programs for each path just by feeding a symbolic argument and asserting they return the same value. For every path that reaches an assertion, Klee checks if any value that exists on the path violates the assertion condition. If so, a counterexample is generated for the same.

2.1.2 Bounded Model Checking

Bounded Model Checking [1] is a SAT procedure-based alternative to symbolic model checking. This is done by unrolling the FSM of product program for a fixed number of unwindings and checking if a violation occurs in that many steps or less. The problem is encoded as an instance of SAT and the process is repeated with larger and larger unwindings untill all possible violations have been ruled out. We use **CBMC** [5] - a Bounded Model Checker for C and C++ programs. Since its development and upkeep more than ten years ago, CBMC has been used to create a bit-precise bounded model checking for C programs. Under a specified loop unwinding bound, CBMC confirms the lack of claims that have been violated.

2.2 Mutant and Supermutant Generation

A mutant [7] or mutated program is the original program with a small difference in one of the statements. The small difference is introduced by modifying expressions, changing, adding or removing operators or statements. Mutant generation is a technique to generate such mutants from a source program according to a mutant schemata. The schemata ensures that the mutant is syntactically correct and will not result in any compile time errors if the original program is syntactically correct too.

We call a program supermutant if it contains one or more such possible mutations inserted at compile time and can be activated or deactivated one by one at run time.

```
int solve(int a, int b, int c) {
    int x = b * b;
    int y = 4 * a * c;
    int d = x - y;
    return d;
}
```

Listing 2.4: Source Program

```
int solve(int a, int b, int c) {
    int x = b + b; // '*' changed to '+'
    int y = 4 * a * c;
    int d = x - y;
    return d;
}
```

Listing 2.5: Mutant

```
int solve(int a, int b, int c, bool flag1, bool flag2, bool flag3) {
    int x = (flag1) ? b * b : b + b;
    int y = (flag3) ? (4 * (flag2 ? a * c : (a - c))) : (4 + (flag2 ? a * c : (a - c)));
    int d = x - y;
    return d;
}
```

Listing 2.6: Super Mutant

For example, One of the possible mutants for the source program 2.4 is the program 2.5. Program 2.6 denotes a supermutant for the same source program. Note that the mutations can be activated and deactivated using boolean flags presented in the function definition. Initially all flags are set to true and the super mutant is semantically equivalent to the source program.

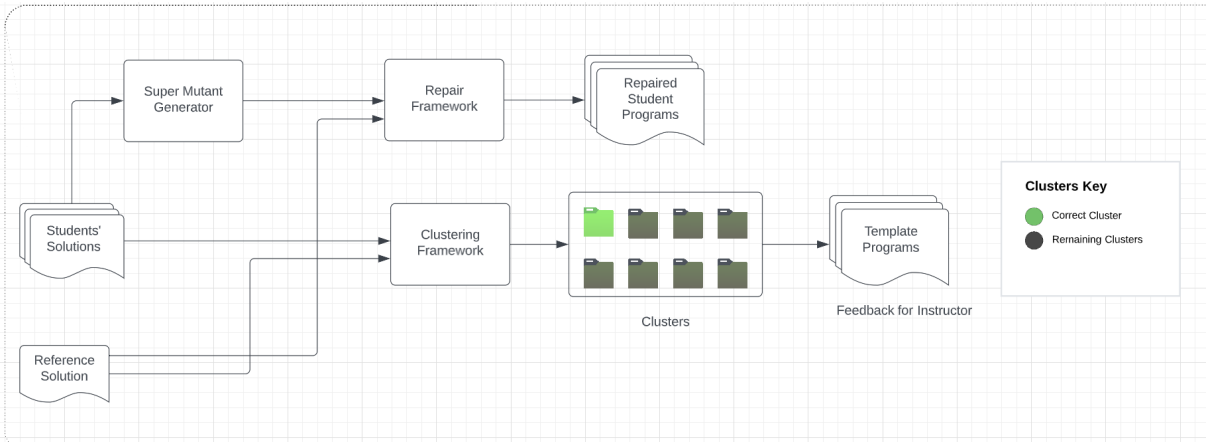
Chapter 3

Design and Methodology

3.1 Overview

Figure 3.1 gives a high-level overview of our entire approach.
Given a set of student submissions to a programming assignment and an instructor's reference program, our approach works as follows.

FIGURE 3.1: Overview



First, we use our clustering algorithm to cluster the student submissions and the reference program based on the notion of semantic equivalence. The clustering algorithm itself is enough to evaluate student programs and verify their correctness. The cluster containing the reference program is called the **correct cluster**.
Our clustering Algorithm generates counterexamples that separate clusters from each other. We then present a submission from each cluster, called **template program**, along with the counterexamples that separate it from the correct cluster as feedback to the instructor.
We then use our repair algorithm on the super mutants of incorrect student programs to localize

the error and repair the program. The output from the repair algorithm is then provided to the student as guided feedback to make progress toward the correct solution.

3.2 Clustering Algorithm

In order to provide feedback to the instructor, we cluster semantically equivalent student programs together.

FIGURE 3.2: Clustering Framework

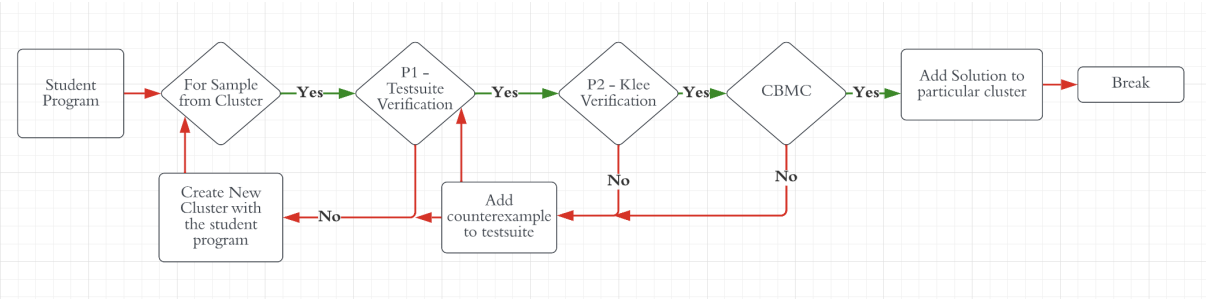


Figure 3.2 provides a high-level summary of our clustering algorithm. Our clustering algorithm asserts two preconditions before using a bounded model checker to verify the semantic equivalence of any two programs.
Initially, a cluster is created and the first program directly goes to this cluster. From the second program onwards, the steps are as follows

3.2.1 Step 1: Picking Sample Program

Choose one program from a cluster not chosen yet for the current student program. The choice can be made randomly. We then create a product program to verify the semantic equivalence of the student program with this program(or this cluster). If all the clusters have been chosen, we create a new cluster for the current program and rerun this algorithm for another student program.

3.2.2 Step 2: P1 - Test suite verification

As will be noticed in the later steps, the clustering algorithm generates counterexamples for program pairs that are found to be semantically inequivalent. These counterexamples form our basis for the first precondition. We execute the produce program for the entire test suite. In case of failure, we declare the two programs to be semantically inequivalent and return to step 1. In case the test suite passes, we move forward to our next precondition or step 3.

3.2.3 Step 3: P2 - Klee Verification

The goal of this precondition is to find one path of the product program which leads to an assertion failure within a specific time limit. KLEE [2], being a symbolic engine, provides the ability to exit execution as soon as an assertion error along one of the paths is encountered. We exploit this ability to terminate our verification early, and we do not move forward to the next step in case of an assertion failure. KLEE also generates a counterexample for the path which leads to assertion failure. As discussed in the previous step, we include this counterexample, or more formally, this test case to our test suite. For further iterations, the test case further helps in early terminating our verification at Step 2 itself.
In case the product program executes without an assertion failure, we move forward to the next step.

3.2.4 Step 4: Bounded Model Verification

Once the product program passes the two preconditions, we use CBMC [5] to verify the semantic equivalence. In this step, the product program is un-rolled to some fixed, pre-determined finite steps k, and the model checker will be used to show that the two programs are semantic equivalent for this pre-decided reachability. If the two programs diverge, the corresponding counterexample is recorded and added to the test suite. If not, the two programs are judged to be equivalent, and the student program joins the cluster.

3.2.5 Feedback for the Instructor

The clustering algorithm itself is enough to evaluate the student programs and provide feedback to the instructor. The instructor can analyse the template program from all but the correct cluster to analyze program each representing a different semantic error. Grades can be awarded based on the semantic error with the correct cluster receiving the maximum score. Additionally, the teacher may find out each student's preferred method of learning, their areas of strength and weakness, and data to assist plan lessons.

3.3 Automated Program Repair

In order to guide students to make progress towards the correct solution, we aim to localize the error(s) in the student program and assist them to repair their solution.

FIGURE 3.3: Program Repair Framework

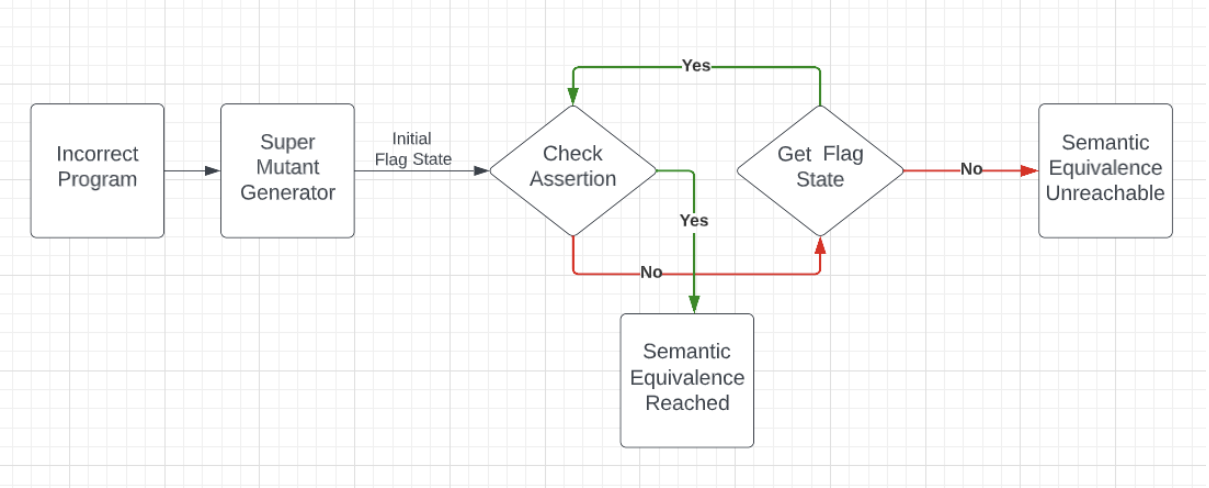


Figure 3.3 provides a high-level state diagram of our program repair algorithm.
The algorithm works as follows:

3.3.1 Step 1: SuperMutant Generator

The supermutant generator is used to generate supermutants of the incorrect student program. In our work, we will skip the design of supermutant generator and assume supermutants of incorrect solutions are already available to us. The mutations in supermutants can be activated or deactivated at runtime.

3.3.1.1 Flag State

This activation or deactivation is done by assigning values to a list of boolean flags. This assignment is termed the flag state. Each boolean flag maps to one mutant. Initially, all mutants are deactivated.

3.3.2 Step 2: Assertion Checker

A product program is created using the super mutant and the instructor's reference solution by taking a flag state as input. The assertion checker verifies for any assertion error in the execution of the product program. Listing 3.1 shows one such implementation.

```
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
#include<string.h>
```

```
void klee_assume();

void klee_make_symbolic();

// reference solution
int solve1 ( int a , int b , int c ) {
int x = b * b ;
int y = 4 * a * c ;
int d = x - y ;
return d ;
}

// supermutant
int solve2 ( int a , int b , int c , bool flag1 , bool flag2 , bool flag3 ) {
    int x = ( flag1 ) ? b * b : b + b ;
    int y = ( flag3 ) ? (4 * ( flag2 ? a * c : ( a - c ))) : (4 + ( flag2 ? a * c : ( a
    - c )));
    int d = x - y ;
    return d ;
 }

int main()
{
        int a, b, c;
        klee_make_symbolic(&a,sizeof(a),"a");

        klee_make_symbolic(&b,sizeof(b),"b");

        klee_make_symbolic(&c,sizeof(c),"c");

    int return_value_1 = solve1(a,b,c);
        int return_value_2 = solve2(a,b,c, true, false, true); // Flag state true false
    true is taken as input

        assert(return_value_1 == return_value_2);

        return 0;
 }
```

Listing 3.1: Assertion Checker's Implementation

This verification is done using KLEE.

If no assertion error is found, the super mutant with the current flag state is judged to be equivalent to the reference solution. The current flag state defines the mutations required in the incorrect program to repair it.

If an assertion error is obtained, one or more counterexamples are recorded, and we move to the next step.

3.3.3 Step 3: Getting the flag state

This step aims to find a flag state, i.e., an instance of the super mutant for which no counterexample observed so far fails. If such a flag state is obtained, we return to step 2 to check if the product program succeeds with this flag state.Listing 3.2 shows one such implementation.
For every instance, whenever the assertion checker in step 2 fails, one or more new counterexample are observed. Hence, the flag state obtained with each iteration at this step is better than the previous one.
If we're unable to find a flag state at any iteration, that signifies semantic equivalence can not be obtained, and/or the incorrect program cannot be repaired. In this case, we terminate the algorithm and return the result as inconclusive.

```
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
#include<string.h>

void klee_assume();

void klee_make_symbolic();

// reference solution
int solve1 ( int a , int b , int c ) {
int x = b * b ;
int y = 4 * a * c ;
int d = x - y ;
return d ;
}

// supermutant
int solve2 ( int a , int b , int c , bool flag1 , bool flag2 , bool flag3 ) {
    int x = ( flag1 ) ? b * b : b + b ;
    int y = ( flag3 ) ? (4 * ( flag2 ? a * c : ( a - c ))) : (4 + ( flag2 ? a * c : ( a
    - c )));
    int d = x - y ;
    return d ;
 }

int main()
{
    bool flag1, flag2, flag3;
        klee_make_symbolic(&flag1,sizeof(flag1),"flag1");

        klee_make_symbolic(&flag2,sizeof(flag2),"flag2");

        klee_make_symbolic(&flag3,sizeof(flag3),"flag3");

    int r_c1_1 = solve1(1, 1, 0 );
        int r_c1_2 = solve2(1, 1, 0, flag1, flag2, flag3);

    int r_c2_1 = solve1(1, 0, 1);
        int r_c2_2 = solve2(1, 0, 1, flag1, flag2, flag3);
```

```
        assert(!(r_c1_1 == r_c1_2 && r_c2_1 == r_c2_2)); // Aim is to throw error for a
    flag state for which both inputs (1,1, 0) and (1, 0, 1) return the correct output

        return 0;
}
```

Listing 3.2: Getting the flag state

3.3.4 Feedback for the student

The above approach aims to generate a flag state for which no inputs fail. This flag state maps to the mutations activated in the super mutant. Activating these mutations generate a program similar to the student's incorrect solution and semantically equivalent to the reference solution. This information is then returned to the student, either all together or in steps, guiding the student to progress towards repairing the solution themself.

Chapter 4

Results

4.1 Benchmarks

We collected the submitted programs for a programming problem from the Introduction to Computer Programming Course at Birla Institute of Technology and Science, Pilani. The problem selected is a beginner-level problems with clear specifications to solve them. These programs are coded in C Programming language. The problem is as follows

- Digit Sum - a program that accepts a number as input and recursively finds the sum of digits

Considering our research context, all programs which were not compilable and executable were eliminated. Out of the remaining submissions, 50 programs, including the reference solution for the problem, were taken for our analysis. The first program (1.c) is the instructor's reference solution. This constituted our benchmark and is available at `https://github.com/manasmhasakar/autoval-benchmarks`
Before these programs were fed to our framework, we customized them and encapsulated them in a new method that directly accepts test arguments as input.

4.2 Evaluation Metric

We evaluated our two main frameworks, i.e., the Clustering framework and the Program Repair framework. The results are as follows

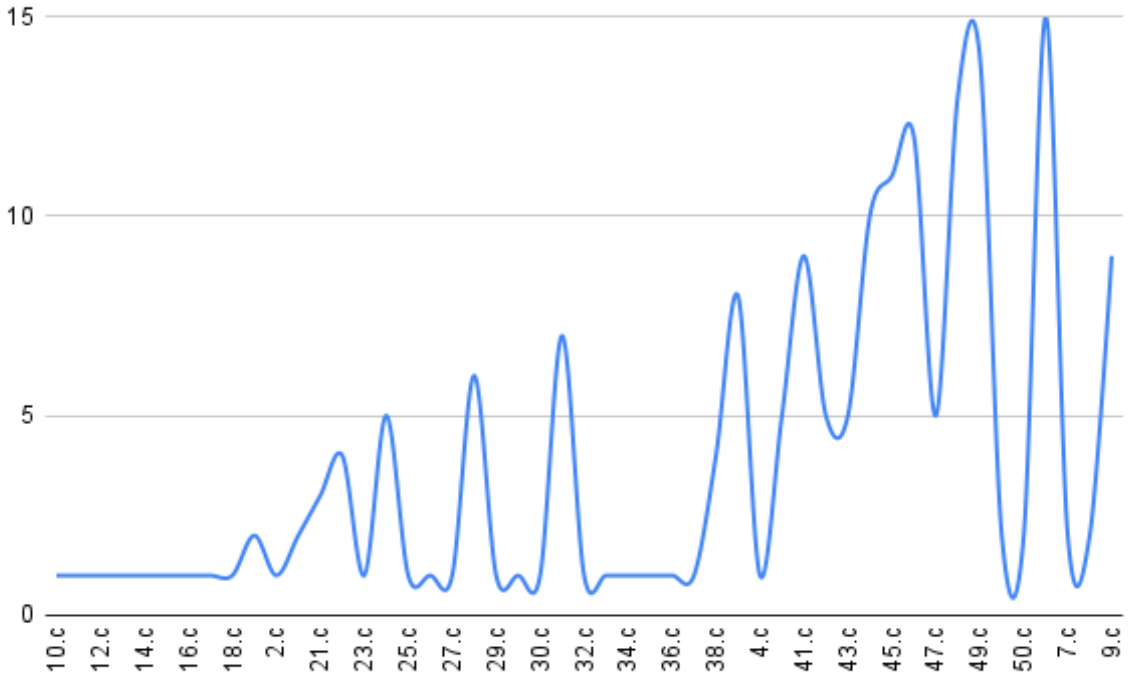
4.2.1 Clustering Framework

TABLE 4.1: Benchmarks

Benchmark	Clustering Time
Digit Sum	16.36 minutes

Table 4.2 gives a detailed result of the time each component takes individually to cluster each program. It is worth noting that the time the clustering framework takes per program is proportional to the number of clusters present in the system. Hence, the programs observed at a later stage during the analysis will require more comparisons in the average case and will require more time to cluster. Figure 4.1 shows how the total comparisons increase with the number of programs analyzed.

FIGURE 4.1: Program vs Total Comparisons



4.2.2 Program Repair Framework

Considering our research context, we assume the super mutants of the incorrect programs are already available to us. Listing 4.1 shows one such execution of our program repair framework.

Table 4.2: Clustering Framework Result

Program	Testcases Time	Klee Time	CBMC Time	Total Time (seconds)
10.c	0.001909	2.012243	30.888626	32.902778
11.c	0.000834	10.892066	12.598603	23.491503
12.c	0.000803	1.904156	4.388981	6.29394
13.c	0.000653	1.725014	36.000769	37.726436
14.c	0.000645	10.60332	14.285988	24.889953
15.c	0.000752	1.130819	3.489457	4.621028
16.c	0.001158	1.143407	29.680825	30.82539
17.c	0.000653	10.850466	12.384638	23.235757
18.c	0.000696	0.462344	0	0.46304
19.c	0.216584	1.372043	0	1.588627
2.c	0.115043	1.630751	4.415152	6.160946
20.c	0.174729	0.330428	0.515033	1.02019
21.c	0.321878	0	0	0.321878
22.c	0.377954	11.24523	0	11.623184
23.c	0.094915	2.004474	5.39493	7.494319
24.c	0.502769	4.076987	0	4.579756
25.c	0.104381	10.98071	66.553291	77.638382
26.c	0.118176	10.703998	12.311466	23.13364
27.c	0.093826	7.888729	11.597904	19.580459
28.c	0.577551	11.766532	0	12.344083
29.c	0.09269	0.955395	2.409972	3.458057
3.c	0.090644	1.752164	31.034739	32.877547
30.c	0.088397	13.330389	6.728231	20.147017
31.c	0.63906	13.099078	4.563632	18.30177
32.c	0.096725	10.845853	41.963165	52.905743
33.c	0.088207	10.38442	14.359246	24.831873
34.c	0.139741	10.865854	20.248592	31.254187
35.c	0.098994	12.355784	45.686463	58.141241
36.c	0.089847	0.857626	14.456525	15.403998
37.c	0.090956	10.674171	8.463875	19.229002
38.c	0.394225	0.478141	0.692561	1.564927
39.c	0.773417	18.476233	0	19.24965
4.c	0.115151	11.321895	6.489066	17.926112
40.c	0.445685	69.633856	34.310357	104.389898
41.c	0.826538	0	0	0.826538
42.c	0.448425	10.669848	29.731127	40.8494
43.c	0.459819	0.497988	0.751869	1.709676
44.c	1.066797	19.394972	4.027591	24.48936
45.c	1.174111	0	0	1.174111
46.c	1.202188	0	0	1.202188
47.c	0.443772	1.444311	4.234493	6.122576
48.c	1.199952	0	0	1.199952
49.c	1.458097	21.69686	4.298209	27.453166
5.c	0.327001	1.506053	4.622993	6.456047
50.c	0.194509	13.847357	6.692542	20.734408
6.c	1.354841	0	0	1.354841
7.c	0.17828	12.854948	32.668517	45.701745
8.c	0.177016	10.716453	20.035259	30.928728
9.c	0.797597	0.499237	0.736339	2.033173

```
// Reference Solution - DigitSum

int solve(int n)
{
    if(n > 9) {
        return solve(n % 10 + solve(n / 10));
    } else {
        return n;
    }
}

// Super Mutant

int solve(int n, bool flag1, bool flag2, bool flag3, bool flag4, bool flag5)
{
    if ((flag1 ? n != 0 : ((flag2)  ? (n > 9) : (n > 0) )))
    {
        int val = (flag3) ? (n % 10 + solve(n / 10, flag1, flag2, flag3, flag4, flag5))
    : (solve(n % 10 + solve(n / 10,  flag1, flag2, flag3, flag4, flag5),  flag1, flag2,
    flag3, flag4, flag5));

        return val;
    }
    else
    {
       return flag4 ? n : 0;
    }
}


python3 main.py

Current Step :  Step.GET_COUNTER_EXAMPLE
For Flag State [1, 1, 1, 1, 1], Counter Example =  [19]


Current Step :  Step.GET_FLAG_STATE
For Counter Examples [[19]], Good Flag = [0, 1, 0, 1, 0]


Current Step :  Step.GET_COUNTER_EXAMPLE
For Flag State [0, 1, 0, 1, 0], Counter Example =  []
Found Result in 0.239919 seconds
[0, 1, 0, 1, 0]
```

Listing 4.1: Program Repair Execution

Chapter 5

Limitations and Future Work

5.1 Overview

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

- First, our approach only determines the correctness of a program. The number of testcases that pass in a programming issue is typically used as a criterion by conventional automatic graders to determine how well an assignment is graded. In order to allow automatic grading of assignments, we plan to devise a grading system that calculates the distance between two programs.
- Secondly, our approach suffers current limitations of bounded model checkers and symbolic execution tools such as the path explosion problem and lack of support for nonlinear arithmetic operations.
- Thirdly, In the future we aim to devise a method to create super mutants and do so intelligently by learning from student submissions and predicting repair patches.

Bibliography

[1] Armin Biere et al. “Bounded Model Checking”. In: vol. 58. Dec. 2003, pp. 117 –148. ISBN: 9780120121588. DOI: 10.1016/S0065-2458(03)58003-2.

[2] Cristian Cadar, Daniel Dunbar, and Dawson Engler. “KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs”. In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI’08. San Diego, California: USENIX Association, 2008, 209–224.

[3] Berkeley Churchill et al. “Semantic Program Alignment for Equivalence Checking”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, 1027–1040. ISBN: 9781450367127. DOI: 10.1145/3314221.3314596. URL: https://doi.org/10.1145/3314221.3314596.

[4] Sumit Gulwani, Ivan Radiček, and Florian Zuleger. “Automated Clustering and Program Repair for Introductory Programming Assignments”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2018. Philadelphia, PA, USA: Association for Computing Machinery, 2018, 465–480. ISBN: 9781450356985. DOI: 10.1145/3192366.3192387. URL: https://doi.org/10.1145/3192366.3192387.

[5] Daniel Kroening and Michael Tautschnig. “CBMC – C Bounded Model Checker”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Erika Ábrahám and Klaus Havelund. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 389–391. ISBN: 978-3-642-54862-8.

[6] Xiao Liu et al. “Automatic Grading of Programming Assignments: An Approach Based on Formal Semantics”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. 2019, pp. 126–137. DOI: 10.1109/ICSE-SEET.2019.00022.

[7] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. “Mutation Analysis Using Mutant Schemata”. In: *SIGSOFT Softw. Eng. Notes* 18.3 (1993), 139–148. ISSN: 0163-5948. DOI: 10.1145/174146.154265. URL: https://doi.org/10.1145/174146.154265.