TÉCNICO LISBOA

FeedFix: Generating Abstract and Personalized Feedback for Introductory Programming Assignments Through Automated Program Repair

Tiago Miguel Rodrigues da Costa e Silva Vaz

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Vasco Miguel Gomes Nunes Manquinho
Doctor Pedro Miguel Orvalho Marques da Silva

Examination Committee

Chairperson: Prof. Rui Filipe Fernandes Prada
Supervisor: Prof. Vasco Miguel Gomes Nunes Manquinho
Member of the Committee: Prof. João Fernando Peixoto Ferreira

October 2025

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

I would like to thank my parents for their unwavering support. From both a financial and an emotional stand point, I would not be where I am today without their help. It was through their help that I managed to overcome the challenges that arose over my years at Instituto Superior Técnico, especially throughout the COVID pandemic. I would also like to thank my grandfather, whose work ethic and drive still inspires me to improve everyday.

I would also like to acknowledge my dissertation supervisors Prof. Vasco Manquinho and Doctor Pedro Orvalho for their insight, support, sharing of knowledge and patience that have made this Thesis possible.

I would also like to thank the people I came to call friends during my university journey, for always keeping me company, during both highs and lows.

Lastly, I must thank my best friend of several years for reading and re-reading several portions of my Thesis to check for errors and help me with phrasing. Despite having no technical knowledge, his help was indispensable whenever I was in a writing rut.

To each and every one of you – Thank you.

This work was supported by Portuguese national funds through FCT, under projects UID/50021/-2025, UID/PRR/50021/2025, and 2023.14280.PEX (DOI: 10.54499/2023.14280.PEX).

Abstract

The increasing number of students that enroll in higher education programming courses, along with the significant increase in the availability of Massive Online Open Courses, have made it nearly impossible for personalized feedback to be given to students who are now starting to learn programming. This reduction in the amount of personalized feedback leads to frustration and a slower rate of learning from the students, whose work and progression suffers from the lack of guidance. In this Thesis, we present `FeedFix`, a tool designed to tackle this problem, by taking advantage of Fault Localization to pinpoint errors in the program and leveraging the recent improvements in the realm of Large Language Models, when it comes to Automated Program Repair, to create personalized solutions based on the students' own buggy submissions. `FeedFix` then takes these solutions and its corresponding buggy student submission to generate a detailed Sketch of the solution, revealing the structure of the program, while abstracting away any specific portions of code of the student submission that were incorrect or missing. `FeedFix` also allows the faculty members to alter the level of feedback given, giving them the freedom to provide more or less detail in the sketch, as they see fit. With this in mind and considering that `FeedFix's` best configurations have over a 50% repair rate, `FeedFix` shows tremendous promise when it comes to tackling this feeedback problem.

Keywords

Automated Program Repair; Introductory Programming Assignments; Personalized Feedback; Code Sketches.

Resumo

O aumento de estudantes que se inscrevem em cadeiras de programação universitárias, em conjugação com o aumento significativo da disponibilidade de Cursos Abertos Online Massivos, tornaram a tarefa de dar feedback personalizado aos estudantes quase impraticável. Esta redução na quantidade de feedback personalizado, leva a um acréscimo nos sentimentos de frustração dos estudantes e a uma diminuição na sua progressão. Nesta tese, apresentamos `FeedFix`, uma ferramenta criada para enfrentar este problema, tirando vantagem de Localização de Falhas para encontrar erros no programa de forma precisa e utilizando os desenvolvimentos recentes no domínio de Modelos Linguagem de Grande Escala, no que toca a Reparação Automática de Programas, para criar soluções personalizadas baseadas nas submissões dos alunos. `FeedFix` utiliza estas soluções, em conjunto com as submissões dos estudantes com bugs para gerar um esboço do programa solução, revelando a estrutura do programa, mas abstraindo porções específicas do código que a submissão do estudante tinha em falta ou tinha errado. `FeedFix` também permite que membros do corpo docente alterem o nível de feedback da ferramenta, dando-lhes a liberdade de escolher se querem que o esboço mais ou menos detalhado. Tendo isto em mente e considerando que as melhores configurações de `FeedFix` têm uma taxa de reparação superior a 50%, `FeedFix` demonstra grande potencial no que toca a enfrentar este problema de fornecer feedback personalizado a cada estudante.

Palavras Chave

Reparação Automática de Programas; Trabalhos de Programação Introdutórios; Feedback Personalizado; Esboços de Código.

Contents

List of Figures

List of Tables

Listings

Acronyms

APR	Automated Program Repair
AST	Abstract Syntax Tree
CEGIS	Counterexample Guided Inductive Synthesis
FIM	Fill in the Middle
FL	Fault Localization
GPU	Graphics Processing Unit
IPA	Introductory Programming Assignment
LLM	Large Language Model
MOOC	Massive Open Online Course
PG	Patch Generation
PV	Patch Validation

1

Introduction

Contents

1.1 Problem Overview

With the ever-changing landscape of technology and the tremendous strides it has taken, along with the promise of higher pay and a higher degree of freedom in relation to work hours and remote work, every year, millions of new Computer Science and Engineering students worldwide embark on their universitarian journeys by learning programming [3, 4].

According to an article by W. M. To et al [5], in the 1980's, the worldwide ratio of higher education professors to students was about 1:13, whereas in 2018 the ratio had deteriorated to 1:17. Something quite noticeable in the findings is also the fast increase in the number of students year-on-year, compared to the relative stagnation of the number of teachers. This problem has been severely exacerbated in recent years, with the spread of Massive Open Online Courses (MOOCs), which feature little to no human interaction. This is due to both the vast numbers of students that these courses often have

and the fact that these courses tend to be available internationally, meaning that scheduling time for a meeting to try to clear up doubts could be almost impossible due to timezone differences, for example. As such, MOOCs rely almost entirely on automated responses, with feedback being extremely limited when it comes to more practical courses, such as introductory programming courses [6, 7].

These facts lead to an interesting problem, where each student has less direct contact with the teachers, due to teachers being forced to stretch their time over a significantly larger amount of students. This consequence means personalized feedback becomes unfeasible as the teachers do not have enough time to provide individual criticism for each of their student's work, being relegated to providing basic feedback, such as indicating whether a given exercise is correct or incorrect, based solely on testing [1].

This is particularly damaging when practical exercises, such as programming projects and exercises, are put on the curriculum, as the students will often lack the attention needed from faculty members to understand where they went wrong, leading to a lot of wasted time. This limits the number of practical exercises that can be given to students, as the workload would increase significantly.

When it comes to programming, this is a significant problem, as practicing with any given programming language will significantly increase the students' proficiency with it. However, the lack of practical exercises, besides the projects, has a detrimental impact on the overall speed at which new students become comfortable and develop their skills with a given programming language and its rules as a whole. The lack of personalized feedback other than, for instance, `"Test 2: Failed"`, makes the student iterate through their entire program, trying to find what could be the cause of the issue.

For instance, let us use a program designed to calculate the n^{th} number in the Fibonacci sequence. In this case, a student submits his solution, shown in Listing 1.1, whose mistake lies in the lack of an equal sign in the conditional statement of the `for` loop (fixed program shown in Listing 1.2). The student, without proper feedback, might easily overlook the lack of the equal sign in the loop condition, thinking the issue lies within the logic of the program, leading to a lot more submissions and wasted time for the student. This is extremely unproductive, as the mistake is very easy to fix and does not represent a valuable learning opportunity for the student other than being a reminder to pay closer attention to the code, while it is being written.

With the recent advancements in the realm of Automated Program Repair (APR) [3, 6–9], an opportunity to remedy some of the problems with the lack of feedback presents itself. As seen in Figure 1.1, the simplified way for students at *Instituto Superior Técnico* to currently interact with their practical submissions is by submitting it to a `Git` repository (either for the individual student or the student group). This submission is then tested with the test suite and feedback on the success or failure of the test is added to the student's repository, along with hints on fault localization, memory related issues and complexity analysis. A more detailed description of the inner workings of `GitSEED` can be found in its paper [1]. These repositories are stored by the University, leading to a large set of different archived solutions for

Listing 1.1: Buggy Submission

```c
#include <stdio.h>

int main(){
    int n, fibonacci;
    int prev1 = 1;
    int prev2 = 0;

    scanf("%d", &n);

    if (n <= 1) return n;

    for (int i = 2; i < n; i++){
        fibonacci = prev1 + prev2;
        prev2 = prev1;
        prev1 = fibonacci;
    }

    printf("%d", fibonacci);

    return 0;
}
```

Listing 1.2: Correct Submission

```c
#include <stdio.h>

int main(){
    int n, fibonacci;
    int prev1 = 1;
    int prev2 = 0;

    scanf("%d", &n);

    if (n <= 1) return n;

    for (int i = 2; i <= n; i++){
        fibonacci = prev1 + prev2;
        prev2 = prev1;
        prev1 = fibonacci;
    }

    printf("%d", fibonacci);

    return 0;
}
```

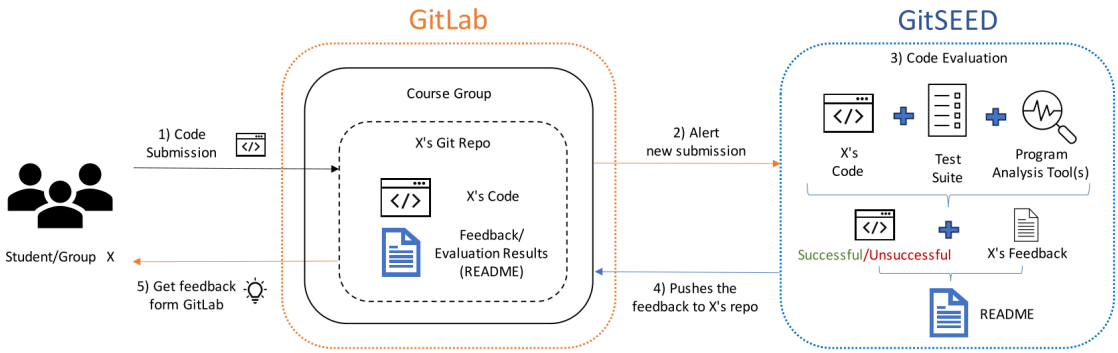

Figure 1.1: GitSEED Overview [1].

each programming exercise.

Currently, few tools exist that are tailored specifically to help with the feedback that is provided to the students. While there are some tools which aim to help programmers, such as *AnalyseC* [10], that try to provide feedback to the student by basing their feedback on the intersection between the student submission and a reference solution, there is a clear lack of available tools that can give personalized feedback to students based around their own code and intended solution instead [11].

In this thesis, we present `FeedFix`, a tool whose aim it is to tackle these problems within the Computer Science sphere of higher education, by providing a tool for the `C` programming language that automatically corrects a given buggy program submitted by a student and provides an identification of where

the problems lie within the submission. This was achieved by evaluating FeedFix with C-Pack-IPAs [4], a benchmark of 25 programming assignments in C, consisting in a total of 1431 faulty programs drawn from three laboratory classes at Instituto Superior Técnico. This tool is intended to provide a similar level of personalized feedback as faculty would, identifying faulty segments of a program and pointing the student in the correct direction. `FeedFix` achieves this level of feedback by making use of Large Language Model based Automated Program Repair to generate a correction of the student's submission and use it as a baseline to generate a detailed sketch of the solution. This sketch provides the structure of the program to the student, retaining the sections of code that the student got correct, while abstracting away the portions that are missing or incorrect in the student's submission.

By providing a toggle to allow professors to tweak the functionality of the tool, the desired degree of precision for the tool can be achieved. Whether it be a direct identification of the exact place where the problem lies, using the Listing 1.1 example `for(int i = 0; i /* Op */ n; i++;)`, or just identifying said portion of code as buggy, i.e., `for(int i = 0; /* BinaryOp */; i++;)`, `FeedFix` would provide faculty with control over the feedback. This toggle is intended to let each lecturer either reduce or increase the amount of information given by `FeedFix`, depending on whether they deem a certain exercise to be worth a higher level of detail on its feedback or not. As an example, early exercises in the semester could have more detailed feedback, whereas later exercises would provide more abstract feedback in `FeedFix's` response. Another example is increasing the level of detail given by `FeedFix` based on the number of submissions made by the student.

`FeedFix` leverages a new method of generating feedback, based around APR, along with a purpose built sketch tool which makes use of both the student submission along with its corrected version, to generate a program sketch that attempts to mimic the feedback that a faculty member would provide to the students.

Thus, `FeedFix` is intended to be used on introductory programming courses where students learn how to program in `C`, in different institutions, and as such, it was built with consideration for the computational and financial limitations using Open Source software to not encumber the host institution with financial burdens. In the case of Instituto Superior Técnico, `FeedFix` is planned to be incorporated into `GitSEED` [1] complementing its capabilities by adding this new sketch to its already existing repertoire of feedback information, like hints on fault localization, memory related issues and complexity analysis.

1.2 Document Overview

This thesis structure is organized as follows: **Chapter 2 - Background:** Provide the background information necessary to have a baseline understanding of the underlying characteristics of the problem and the tools meant to tackle it. **Chapter 3 - Related Work:** Provide an in-depth look at the concepts

that will be utilized for our solution and the work that has been done to forward those areas. **Chapter 4 - FeedFix:** Give an in-depth look at the implementation of `FeedFix`, accompanied by explanations as to why certain approaches were chosen over other available options and why certain options were abandoned. **Chapter 5 - Results:** Share the results of our work, along with a critical analysis of the performance of our tool and its successes and shortcomings. **Chapter 6 - Conclusion:** Give a short summary of the problem, our implemented tool `FeedFix`, as well as what possible steps could be taken to further improve `FeedFix`.

2

Background

Contents

2.1 Automated Program Repair

Automated Program Repair (*APR*) is typically composed of three main phases: *Fault Localization*, *Patch Generation* and *Patch Validation* [12]. A small code example of the APR process is shown in Listing 2.1 and 2.2, where a function meant to return the largest of 3 numbers has been coded. This function has a bug in line 2, where `(a >= b && a <= c)` should be `(a >= b && a >= c)`. Using this example while following Figure 2.1 the APR process would receive Listing 2.1 along with the test suite and their respective Expected Outputs, shown in Table 2.1. Additionally, in that same table, a column of the current responses of the program is provided to help showcase the error in the program. It would then find the bug in line 2 using Fault Localization, generate the correct `(a >= b && a >= c)` code patch for line 2 using Patch Generation, and, ultimately, verify if the program, using the newly generated patch,

Listing 2.1: Buggy Code

```
int bigger(int a, int b, int c) {
    if (a >= b && a <= c){
        return a;
    }
    else if (b >= a && b >= c){
        return b;
    }
    else{
        return c;
    }
}
```

Listing 2.2: Code after Patch Validation

```
int bigger(int a, int b, int c) {
    if (a >= b && a >= c){
        return a;
    }
    else if (b >= a && b >= c){
        return b;
    }
    else{
        return c;
    }
}
```

Tests	Inputs a	b	c	Expected Output	Given Output
Test 1	0	1	1	1	1
Test 2	3	5	2	5	5
Test 3	2	1	3	3	2

Table 2.1: Tests and their Expected and Given Output

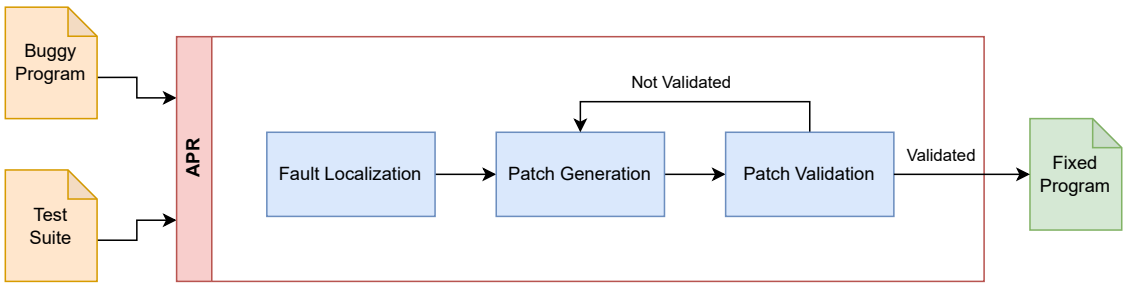

Figure 2.1: APR Process

passes the tests in the test suite. If all the outputs match their respective expected output, the fixed code is returned.

2.1.1 Fault Localization

Fault Localization (FL) serves to identify the portions of the code that are more likely to be responsible for the program's faulty behavior. There are a few different approaches to FL, with the main ones being:

1. **Spectrum-based** fault localization aims to iterate through the different lines of the program, trying to find failure points in each of the tests given and attributing to each line of code a "suspicion score". This suspicion score acts as a weight that determines how likely each line is to be faulty, within the given code snippet. The higher said score, the higher the likelihood that the line is faulty [13]. Given that this method relies on a set of functions that attribute a score to a given line,

it does not give exact identifications of issues apart from very specific edge cases (i.e. a line of code that is executed in every test case that fails), but rather giving an overview of which lines are more likely to be faulty. As such, it requires several executions since the bug can be given a lower suspicion score and, consequently, be treated as less "urgent" in the fixed priority, which can lead to several good fixes being rejected since the buggy part of the program has yet to be addressed.

2. **Formula-based** fault localization (*FBFL*) encodes the several test scenarios with failing cases into different optimization problems, in order to identify the minimal set of faulty statements within the program. Some FBFL tools make use of Satisfiability Module Theory (*SMT*) Solvers [14]. These solvers utilize an SMT formula, consisting of a conjunction of various logical clauses. These clauses correspond to logical operations between literals, real numbers, arrays, vectors, etc., which can be in expressions of their own (i.e., n + m = 0), with each clause representing different portions of the optimization problem. Upon encoding the problem, an SMT Solver tries to find a set of values that, when attributed to the literals in the formula, return a value of `True`, if a solution is possible [15]. Unlike SMT formula clauses, Boolean Satisfiability Problem (*SAT*) formula clauses can only use Boolean values for their literals. As such, the different clauses are only comprised of disjunctions of literals, instead of the various different data types or expressions that could be used in an SMT formula. Similarly to SMT Solvers, SAT Solvers try to find a set of values for the different literals (with these values now being forced to be Boolean values) for the conjunction of these clauses to equal `True`. A Maximum Satisfiability (*MaxSAT*) solver adds an additional layer to SAT Solvers, by allowing each of the different clauses of a SAT formula to be defined as a "soft-clause", which carry a weight. This weight is used to discern which of the different clauses are more important than others, meaning that if two of the clauses can not possibly coexist while both equaling to `True`, the lower weighted clause will be chosen to equate to `False` [16]. Of these, MaxSAT tends to be the option that is most used, since it allows these tools to find a minimal set of faulty lines. An example of a tool which utilizes a MaxSAT solver in its approach is `CFaults` [17], where the soft clauses correspond to the different program statements, with lines being weighted differently, depending on how "important" they are to be kept intact. Given that the intent of this work is to help introductory programming students who are trying to learn to code, both the input and output lines are given a much heavier weight, ensuring that those lines will only be flagged as problematic, *iff* there is no set of changes to the other lines that would ensure that the program would be able to pass the tests. This technique also ensures that the minimal number of faulty statements considering all failing test cases are changed, as each added statement that is flagged as buggy will increase the "cost" of repair, which we are trying to minimize. Given the inherent properties of this type of approach, this also guarantees that we only require a single execution, as the FL program will identify every line that is necessary to be changed to achieve the correct

results, rather than a set of possibilities.

2.1.2 Patch Generation

The second main phase of APR, Patch Generation (PG), consists of creating new sets of patches of code which, when used as a replacement for the buggy sections of code found during the FL phase, manage to fix the bugs encountered in the test cases. Several methods of generating patches exist, with three specific ones being the most common: *(1) Heuristics-Based*, *(2) Constraint-Based* and *(3) Pattern-Based* [12] (see Chapter 3). Depending on which implementation of the PG techniques are used, there might be a Patch Ranking phase, where the different generated patches are compared with each other and ranked based on how likely they are to be correct.

Outside of these patch generation methods, the rise in the capabilities of Large Language Models (LLMs) have put them in the spotlight in the realm of APR [18–21]. These new approaches centered around LLMs, are usually divided into either a refinement based around fine-tuning, via the use of large amounts of data to train the models to better handle the problem, or prompt manipulation, based around different prompting strategies to improve repair abilities [18, 22–24].

2.1.3 Patch Validation

As a final step, to ensure proper repair, there is a Patch Validation (PV) process, where the new "fixed" code is tested and compared, using the provided test suite, in order to gauge if the repair managed to correct the errors, or if it has failed to do so. In case of the latter, some tools try the next most likely implementation given by the Patch Ranking, or if the technique used does not use a Patch Ranking, we return to the PG step, where a new piece of code will be generated to be validated again. Usually, a limit is provided to prevent infinite generation, given the fact that the process might have failed during the FL phase, and, as such, the PG is trying to find a solution when there is none to be found due to faulty FL. In the case that the limit is reached, it is assumed that the APR process failed, and no solution could be found.

2.2 Abstract Syntax Trees

Abstract Syntax Trees (ASTs) are a data structure meant to represent the structure of a program [25]. These trees are made up of nodes, with each one being divided into one of two categories:

1. Parent - Node which contains "children" nodes at a lower depth than it. It is a "branch" of the AST.
2. Terminal - Node with no "children" nodes. It is a "leaf" of the AST.

Each node of the tree serves as a representation of something within the code, whether it be a `While` loop, a `Function Call`, a `Variable`, etc. These nodes store the code's information and their placement in the AST is directly influenced by how the code is structured, with the way in which the nodes are stored keeping track of certain important details, i.e., what is to the left and to the right of a `Binary Operation`, or the order in which a block of code is executed in.

As such, these data structures are crucial in the context of program repair and giving feedback to students (see Chapter 3), as it abstracts much of the text based clutter while retaining all of the relevant information, leading to an easier analysis of the programs.

3

Related Work

Contents

This Chapter will cover some of the work that has been done in relation to both Automatic Program Repair (*APR*) and of available feedback tools for students.

Sections 3.1 through 3.4, will mostly cover the work related to the Patch Generation (*PG*) step of APR, while Section 3.5 will focus on the different automated feedback tools that are currently available.

3.1 Heuristic-Based Repair Techniques

Heuristic-Based strategies rely on using a set of heuristics, dependent on the algorithm, that will create a search space based on previous patches and then explore it, in order to find suitable code patches that may fix the problem. Most applications based on these types of algorithms are “light” or “fast”

when compared to the other options available. This is owed to the fact that the Heuristic models are based around fixed parameters, which can provide a more streamlined search space generation and code patch search. A shining example of this is the utilization of a heuristics-based approach with other methods of APR, such as with `SearchRepair` [8], a Constraint-Based tool. As described in a paper by Liam Schramm [26], when a Heuristics-Based approach was added to `SearchRepair` its speed increased dramatically, as a sufficiently accurate classifier can easily identify the perceived bad patches and remove them from the search far faster, than a non-heuristics approach. The light requirements of this approach come with the inherent benefit that the methods by which the generated code patch is achieved are rather simple and therefore efficient. However, this leads to a lack of "depth" in its search as the heuristics will explore the search space top-down based on the standards set by the heuristics given, meaning that any potential code repairs that might be viable but are outside the purview of the assigned heuristics are not explored. This leads to a fundamental problem, as the ceiling of the usefulness of these tools is highly limited. This is due to the methods not adapting to suit the situation, giving way to tools that are extremely compelling when targeting very common mistakes that can be planned accordingly while designing and choosing the heuristics to use. However, these strategies are extremely frail and brittle when the given bug falls outside of the limited range that the heuristics can cover. These factors result in a set of tools which can not reach the heights other techniques might achieve, as they are not limited by the heuristics that provide this method its edge in both speed and computational weight but also hinder its repair ability. They also inherit some issues from the other techniques, given the fact that these methods tend to require other techniques to actually generate the search space from which to derive the solution.

For example, the tool `SimFix` [27] requires patterns, much like Pattern-Based generation techniques, to generate the search space. These patterns are generated in what is referred to as the "Mining Stage", by utilizing pairs of buggy and fixed pieces of code. In the case of `SimFix`, a Code Change Statistics Mining process is used to generate the patterns (an in-depth overview of Code Change Statistics is given in Section 3.3.3). This fact means that the tool will suffer from both of the technique's faults, resulting in a compounding of problems which could be averted if a purer approach was used.

Some other tools, such as `GenProg` [28], utilize a genetic algorithm to apply mutations to the code and fix the given patch. Mutations can vary between: *insertions of code, swaps of operands within a given statement or deletions of sections of code*. Using Listing 2.1 as an example, the mutation algorithm could choose to change line 2, by changing it to `if (a >= b && a == c)`. In this example, the generated mutation is wrong, so it would end up being discarded during Patch Validation. The mutation algorithm is achieved by utilizing different weights within the program to increase or decrease the breadth of changes being made at any given iteration and the probability of each of the different code alteration options. While this does increase the effective area that is being explored for the new

code patch, it can be a far slower process, when compared to tools even within its own category, such as `SimFix`, as the heuristic approach only expands or diminishes the range in which the piece of code can be altered, rather than actively reducing the exploration time. Another issue is the probabilistic nature of the model being used, since even for simple fixes, there is no guarantee that the mutations will generate the correct result or that it will be generated in a reasonable amount of time.

3.2 Constraint-Based Repair Techniques

Constraint-based strategies are methods which rely on the use of conditional statements, usually condensed down to a single formula, that is solved using SMT or MaxSAT solvers to find a code patch that satisfies the condition established [12, 29]. To do this, the tools generate the expression based on certain building blocks that are assumed to be viable. These building blocks tend to be similar from tool to tool, as they tend to equate to `if` statements. As such, these generation methods are ideal while trying to fix problems with the internal logic of a program since they are focused on the Boolean statements of the given buggy program, but they often lack the overall reach to find errors that may occur when the bugs are not found in the conditional statements or lack thereof, but rather in the algorithm of the code. These methods, for example, would be extremely efficient at finding and resolving bugs similar to the one showcased in Listing 2.1, whereas bugs that are more dependent on each individual operation within the code rather than the conditional statements, would pose a more intricate challenge. Let us look into a different example, where a function is trying to calculate the factorial of a given number. For simplicity's sake, we assume that all of the checks for negative numbers, null pointers and other issues have been done prior to entering the function itself. As we can see in Listing 3.1, on the second line of the function we have a bug where the line `int result = 0;` is wrong, given that we are multiplying by this value at every iteration of the for loop, so every test scenario will lead to a `return` value of 0. The corrected code, seen in Listing 3.2, has that same line now corrected to `int result = 1;`. This example serves to illustrate a type of bug that would very likely go unfixed by these types of tools since the bug has nothing to do with the conditional statements in the program, but rather a fault with the assigned value to a given variable. In this case, the mistake is far too simple and would most likely be easily fixed by these models as well. However, one can see how these types of bugs could generate further complications when distributed over a bigger section of code, where multiple assignments are wrong or the underlying algorithm is being employed incorrectly.

An example of a tool using constraint-based methods which showcases both the upsides and downsides of this approach, is `NOPOL` [30]. `NOPOL` focuses on the strengths of this approach, by targeting conditional statements specifically, whether it be adding or modifying any existing ones. This allows for a much more efficient tracking of issues and patch searches. However, it is far more limited, since it relies

Listing 3.1: Buggy Code

```
1 int factorial(int a){
2     int result = 0;
3     for (int i = 1; i <= a; i++){
4         result = result * i;
5     }
6     return result;
7 }
```

Listing 3.2: Fixed Code

```
1 int factorial(int a){
2     int result = 1;
3     for (int i = 1; i <= a; i++){
4         result = result * i;
5     }
6     return result;
7 }
```

on the problem being found within the conditional statements themselves. Therefore, this tool serves as an excellent standard for this type of methodology, easily being able to handle and correct the bug in Listing 2.1, but being unable to effectively target and fix the bug shown in Listing 3.1.

A different tool `PMBugAssist` [31], focused specifically on Persistent Memory bugs, utilizes constraint-based repair techniques to correct bugs related to memory violation and curb the problems that it might entail. This tool is based on the fact that Persistent Memory bugs can be encoded into logical formulas, granting SMT and MaxSAT a significant advantage when it comes to correctly finding fixes for these problems. This tool also reiterates an interesting problem with this approach. Similar to what was mentioned with `NOPOL` and its inability to perform assignment repairs, `PMBugAssist` is highly specific. This is a repeat issue for constraint-based approaches, as the encoding process gets increasingly more complex and time-consuming the more different types of bugs are accounted for.

Another tool that makes use of constraint-based methods is `VeriFix` [7]. This tool was developed to tackle the student feedback problem identified in Section 1 and it revolves around utilizing the constraint-based approach to verify and produce an alignment in their Automatons, which serve as a representation of the control flow of the different programs. These Control Flow Automata, rely on MaxSMT solvers to identify edge differences between the reference solution and the student submission and generate the corresponding missing or mistaken patches of code. Although this method does circumvent some of the common problems of the constraint-based methods, such as the ability to change assignments, this tool is still highly gated by the backlog of data that it has stored and its quality. While similar to the method of Automatic Mining[1], this tool does not utilize patterns, but rather a set of example solutions from which to derive the Control Flow Automata. This difference leads to a significant drop in the scope of the repairs, as the number of examples is limited, reducing the number of available repairs possible.

Overall, when compared with the other approaches, Constraint-Based generation techniques strike a middle ground in terms of the power and breadth of the repairs and the computational resources and time required to achieve its results. It tends to serve as a faster and lighter option when compared to Pattern-Based repair options since it does not require the long learning processes that these methods entail. However, these methods lack the depth that Pattern-Based repair techniques benefit from, as they

[1] Explained in detail in Section 3.3.2.

are confined to finding problems within conditional statements, lacking the ability to modify assignments and other non-conditional problems that might be causing the bugs. Their inherent reliance on logical formulas also reduces some of the breadth of repairs available to these methods, as the tools are forced into being problem-specific far more often than with other methods that could be used. Comparatively to the Heuristics-Based approach, a constraint-based technique ends up being slower on average, as the search through the search space based on Heuristics allows for a much faster patch generation, upon the creation of the search space. Although Heuristic-Based approaches have a bigger overall memory requirement on average, since they tend to make use of Pattern-Based techniques within their processes, their computational resources are not as strained since the heuristics used do not require the computational power that an SMT-Solver would require, leading to a lighter tool overall, despite the heavier memory requirement.

3.3 Pattern-Based Repair Techniques

Pattern-based repair techniques are built upon the idea that common code changes can be abstracted into a pattern, which, when applied to the faulty piece of code, are capable of generating a new patch of code that corrects the bug [32]. Tools that make use of this method rely on the context of the faults in order to better approximate the faulty code to the patterns that are being used, utilizing different characteristics, such as abstract syntax tree nodes, which are, in essence, representations of all the syntactical elements of the programming language [33]. These context clues are then matched with the different available patterns to try to find suitable options that might lead to a possible fix.

To generate these patterns, code examples have to undergo a process called *pattern mining*. This process consists of taking the given code snippets and abstracting them into the “rules” that are used in the patch generation. The pattern mining process is divided into three categories: *(1) Manual Design*, *(2) Automatic Mining* and *(3) Code Change Statistics* [32].

An example to give an overview of the pattern mining process is presented in Figure 3.1. `TIFPs` [2] takes in the Bug set, extracts the relevant information from it and its test execution and starts to cluster the information. These clusters are then used to extract the set of rules from which it creates the different patterns that can be fed into a system to add the potential fixes to the APR tool’s repertoire.

3.3.1 Manual Design

Manual Design is a method of pattern mining consisting of having a human create or curate a set of different cases and bugs that will be used by the given tool. This process requires the person (or people) arranging the patterns to either build different programs that highlight particular problems of interest, like the security bugs mentioned in the study by E. Pinconschi et al [34] or curate a list of examples that have

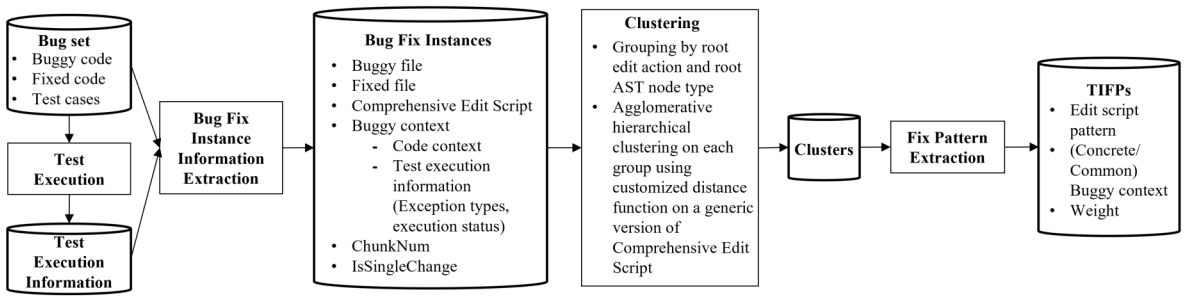

Figure 3.1: Pattern Mining Example [2].

to be addressed. The manual design approach is highly beneficial when it comes to common problems, as the code that the tool receives is made or edited with the sole reason of being the most accurate and widely applicable instance of that bug, meaning it is extremely valuable in the learning process of the tool. It can also be extremely helpful when it comes to hyper-specific edge cases that the development team might want the tool to tackle, as the tool will have an almost perfect example of how to correct said type of bug. This comes with the added benefit that the bug is so specific that minor changes to the structure of the code will mean it never pops up, facilitating the use of a single pattern.

As a whole, the main draw of this approach is the fact that the patterns are highly curated, ensuring that only quality patterns are learned and that the most important bugs are addressed efficiently. However, this approach is inherently limited due to manpower constraints. Given that it requires a human to physically go through and either build new examples or curate a list of examples while having to verify each one to ensure their quality, this approach will always be limited in terms of scope since no developer or group of developers will ever be able to build every possible bug [2, 35].

3.3.2 Automatic Mining

Automatic mining is a method that relies on utilizing a large quantity of “random” examples in order to infer the correct patterns, utilizing a “hands-off” approach [2, 36]. This approach takes advantage of the wide range of different repositories by several different people from which to infer the patterns. This guarantees a much more extensive array of bugs, as the sheer amount of data will far surpass anything that a small team might be able to create, as is the usual case with the Manual Design method. It also means that far more bugs that are deemed uncommon will show up, given that the large amounts of examples all but ensures the presence of rare bugs that might have gone unnoticed if the examples were to be created from scratch. This approach does lack the focus and precision of a manual method since the amount of data being used in the process is far too large to be properly verified and guaranteeing the quality of each example. This leads to a much more strenuous mining process, as each individual example is assumed to be of far less quality, as to prevent low-quality examples from negatively affecting

the mining process. As a consequence, this method tends to take significantly longer in the mining process since the number of examples being processed is far larger than in the Manual Design approach.

A Pattern Mining tool that utilizes this approach is `FixMiner` [33]. This tool starts by collecting all the relevant bug-fixing patches, abstracts the code utilizing ASTs, then searches for similar trees and groups them into clusters. This tool exemplifies the problems with Automatic Mining on a large scale, as the tool had extreme computational requirements, specifically when it comes to RAM, to generate the patterns for around 1.1 million pairs of trees.

Another tool that utilizes a process extremely similar to Automatic Mining is `Clara` [3]. Similar to `Verifix`, this tool was developed to tackle the same student feedback problem. `Clara` utilizes the large repositories available from Massive Open Online Courses (*MOOCs*) to cluster the different student solutions and abstract them. As such, it is essentially utilizing these clusters as the patterns that will be used in the different repairs. While this approach does differ slightly from normal Automatic Mining methods in the way it generates its patterns, the main principles still apply. It does present an interesting difference when compared to tools that follow the approach more closely like `FixMiner`, as the differences in the methods mean that small, discreet, changes might be overlooked. This is due to the fact that these changes may be close enough to be clustered next to other options, instead of generating a completely different pattern. This fundamental difference presents itself as a double-edged sword as `Clara` can easily remove redundancy, by effectively compressing multiple solutions into a single cluster, at the cost of the finer details that could be kept utilizing a purer approach.

Overall, this method is akin to casting an extremely large net. The amount of examples allows for this approach to have almost every common, uncommon and rare bug in its patterns, bar extremely rare bugs that have barely been seen. This means it is far more applicable on average, given that the tool will be far more knowledgeable of less common mistakes. However, this advantage, inherent to this method, also has two major drawbacks. The first is the lack of precision in the fixes since the examples are not properly vetted, which leads to the possibility that some of the patterns learnt are not correct and, therefore, the resulting code patches generated by the tool might be susceptible to more errors, as opposed to the Manual approach. The second and far more limiting problem is the sheer amount of computational power and memory space required to properly utilize this method. The extremely large amount of data necessary to effectively employ this method places it in a very precarious position, as this method is effectively relegated to much bigger tools rather than smaller tools that a team with a small amount of resources could create.

3.3.3 Code Change Statistics

The Code Change Statistics approach utilizes the *top-n* most common changes to the code (at the Abstract Syntax Tree level) to infer the pattern [32]. This method heavily leans on the given quality

of each change, as the mining process is based on the changes that occur between each iteration, so the quality of each change is extremely important in the learning process. This method is extremely similar to the Automatic approach, sharing similar strengths and weaknesses. The main difference is the computational time and the quality of the repairs since the differences between the different iterations of the code ease the inference process, allowing for a much faster learning process per any given example. This approach is far more susceptible to errors in its inference process when compared to the other options, though, as the quality of the different commits is of the utmost importance for a correct pattern to be constructed. The number of common changes necessary (*top-n*) is an important balancing point, allowing for the changes to be "diluted" and regress to the mean, reducing the errors.

As mentioned in Section 3.1, `SimFix` [27] utilizes a Code Change Statistics approach to its Pattern Mining. In their work, they refer to the *top-n* threshold as *k*, which is mentioned to be set to a value that allows them to cover 80% of the search space. Although this reduces the number of outliers, which would probably be populated by faulty or unnecessary examples that could "pollute" the pattern mining process, it also eliminates potentially valuable information on rarer bugs, that wouldn't normally present themselves.

Another tool that made use of a threshold for their Pattern Mining is `Repatt` [37], which utilizes this approach to better group the different patterns mined during the extraction process. This allows them to have more redundancy, by being able to have significantly more examples per bug, while removing stray "fixes" that might not be applicable in the vast majority of situations. Again, this creates the same problem as `SimFix`, where potentially interesting edge-case fixes that could be learnt and utilized by the APR tool are discarded, in favor of more consistently usable patterns.

Overall, the inherent characteristics of utilizing a threshold hinder the use of this method for more uncommon problems. This is owed to the fact that there might not be enough similar code changes to satisfy the *top-n* requirement, which either might require the change to be completely discarded, throwing away data that could lead to a pattern that is useful in those rare instances, or adding a bypass to generate patterns from code changes that do not satisfy the *top-n* condition. This causes problems within the mining process since lowering the number of similar changes will allow for significantly more patterns to be created. However, they might be more prone to errors and create doubled patterns that would have otherwise been considered as part of the same type of change with a bigger *top-n*.

3.4 Large Language Models

Over the recent years, Artificial Intelligence has made tremendous strides. Tools such as `ChatGPT-4`, which are widely available to the masses, have reached extremely impressive capabilities, being very useful for a variety of tasks, from education to summarizing reports, or giving in-depth advice for cod-

ing [38–41]. This last example is of particular interest, given its implications for APR. Due in part to these rapid and extreme developments, Large Language Models (*LLMs*) have now become the focal point in the realm of APR [18]. LLMs benefit from a large amount of characteristics, the most important of which are the ability to memorize information and to generate responses. These two factors make these models extremely enticing for APR, as the tools can both improve their current understanding of a certain topic due to their memory, while also using their response generation to generate new patches of code. While not fitting neatly in any of the former categories, LLMs tend to utilize an implicit Pattern-Based structure. This is due to the principle of *Self-Attention*, which serves as a weight representative of the importance of a given word [42]. These weights are relative to the context that said word was utilized in and, as such, they are bound by an inferred pattern, leading to a learning process similar to normal Pattern-Based approaches for APR. As such, LLMs have become prime candidates for Pattern-Based APR tools since they can be trained on the data to improve their patch-generation capabilities and provide better solutions.

These factors make LLMs extremely enticing, as their capabilities and ease of use far exceed other options available. This is not without flaw since these powerful tools either incur a steep financial cost, as is the case with `ChatGPT-4` [2], or require powerful machines on which to run them [43]. These limitations can hinder using these tools on a large scale for APR, as the costs, either computational or monetary, could spiral out of control. As such, "out-of-the-box" options of LLM have become more interesting for institutions, as they do not come with the financial burdens that other tools might carry, and their significantly reduced computational weight allows for "weaker" setups to be able to run said tools effectively at large scale. In the context of education, these options are especially important for universities that do not have access to the hardware or financial requirements to be able to provide these services to students and faculty.

These circumstances have led to the development of several different tools with the express purpose of utilizing LLMs to better attack the APR problem [44]. These tools either utilize LLMs for Fault Localization (`Toggle` [45], `AGENTFL` [46], `CrashTracker` [47], etc.) or use them in Patch Generation to generate the new snippets of code to be inserted in the faulty code. We will focus on the second category since it is the most relevant to this work. Tools such as `GAMMA` utilize Pattern-Based methodology to improve code creation by the LLM, by providing it with various fix templates as training data [48]. A tool such as `Repilot` utilizes the suggestions from the LLM in combination with a Completion Engine in order to generate patches [49], akin to the methods used in a Heuristic-based model. `MORepair` fine-tunes the LLMs, to improve its code patch generation, by training the LLMs to better pick up on context cues such as the syntactic nuances of any given code alteration and the logical code changes underpinning them [50].

[2] As of the time of writing, pricing on ChatGPT-4 is currently $2.50 per 1 million tokens. Due to the iterations required for `FeedFix's` algorithm, this is not economically viable.

These examples showcase the different methods used by LLMs and their versatility when confronted with the APR problem. From LLM tools specifically tailored for FL to tools that employ direct Pattern or Heuristic-Based approaches, the variety of approaches demonstrates the personalization that can be implemented with the different LLMs, highlighting the power these tools have when properly trained and employed.

3.4.1 Different Prompting Strategies

Outside of fine-tuning with the use of a large amount of training data, LLMs can also vastly improve in their results via different prompting strategies [9, 51–53]. In the realm of APR, utilizing these different strategies has started to get explored, as the modifications to prompting strategies do not require the extensive datasets and expensive training process that is required when fine-tuning LLMs.

These different prompting strategies can employed in a variety of ways, with a paper by X. Cal et al [52] using both "Hard" and "Soft" prompts to try to tune them to the best of their ability. While "Hard" prompts use natural language, "Soft" prompts make use vectors that are learnt by the LLM during the tuning process and are not human-interpretable. These prompts were then sub-divided further into "Basic" and "Knowledge" prompts, with the Basic prompts having no additional information and the Knowledge prompts containing extra information in regards to the bugs in the program. Of these, the clear stand outs were Soft Knowledge prompts, which relayed bug information to the different LLMs to better identify and fix the issues. Despite this, a warning is made when it comes to providing "too much" unnecessary information, as the models might become "confused" and result in a reduction of performance, instead.

3.4.1.A Counterexample Guided Inductive Synthesis Loop

A different prompt approach was employed in the `MENTOR` [6, 9] tool, which used a Counterexample Guided Inductive Synthesis (CEGIS) loop to improve the repair capabilities of several different LLMs, testing various prompt configurations to ascertain which would produce the best results. Some of these configurations made use of the sketching strategy outlined in Solar-Lezama et al [54], utilizing fault localization tools (see Section 2.1.1) to build a sketch from the original buggy program and provide the LLM with added information on the program to improve its repair ability.

The CEGIS loop is divided into two steps: The Synthesis Step and the Validation Step. The Synthesis step can consist of Patch Generation (*PG*) processes described prior. The Patch Validation (*PV*) process, as was described in Section 4.2, first verifies if the generated program successfully passes every test scenario and in the event that it does not, we return to the PG step. The CEGIS loop alters the prompt before asking for a new solution, now adding a counterexample of a test case where the given

generated program fails. This reduces the cases where the LLM repeats the same mistakes, steering it to a better fix.

3.5 Automated Feedback Tools

With the advent of the increasing numbers of student in programming university courses and MOOCs, the challenge to give each student personalized feedback has become harder to tackle every year. Due to this, there has been a significant push to create automated tools which can provide feedback to the students, without the need for direct human intervention [11]. While this concerted effort has yielded some results, the vast majority of tools still only focus on providing the student with the minimum amount of feedback, by returning whether the student submission successfully passes every test or not.

Despite this, some tools have begun to emerge that try to give the students more information. These tools can be broadly separated into two main categories: (1) Abstract Syntax Tree Based Automated Feedback Tools and (2) Large Language Model Based Automated Feedback Tools. With the prior making use of the AST of the student program to generate its feedback and the later using LLMs to generate the feedback for the student.

3.5.1 Abstract Syntax Tree Based Automated Feedback Tools

An example of an AST based Automated Feedback Tool is `AnalyseC` [10], a tool meant to help students by comparing their submission with a model solution and providing an assessment based on the AST of the two programs. The tool works by taking the ASTs of both programs and conducting a structural and a semantic analysis of both programs and then deriving the Program Dependence Graph (*PDG*) of both programs from their semantic analysis. These graphs, which contain both the data and control dependencies of the programs, are compared to one another and the assessment is created based on the similarities and differences between the two. This approach uses the fact that the generated PDGs of two different codes that want to solve the same problem have similar control and data dependencies. Considering this, the model solutions used as a launch point should not have as much influence over the resulting assessment, given that both codes should have similar PDGs.

While this does occur often enough to be somewhat reliable, the vast numbers of students who are just starting to learn programming means that the number of unorthodox solutions that are tried makes it near impossible for a model solution to match every possible solution that a student might come up with. Furthermore, the use of a model solution restricts the feedback to the baseline solution, being unable to adapt to the student's own version. While the feedback might help identify the problem, the lack of a personalized program to compare with, results in a feedback that does not retain the structure of the original code, trying to abstract it down to the model solution, instead.

Listing 3.3: Student Submission

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int n;
6
7      scanf("%d", &n);
8      int factorial(int a){
9          if (a == 1) return 0;
10         return a * factorial(a - 1);
11     }
12     printf("%d", factorial(n));
13
14     return 0;
15 }
```

Listing 3.4: Example Solution

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int n, result = 1;
6
7      scanf("%d", &n);
8      for (int i = 1; i <= n; i++){
9          result = result * i;
10     }
11     printf("%d", result);
12
13     return 0;
14 }
```

Another tool that tries to combat this problem is FLIP [55], which was a tool designed to help programming beginners learn JavaScript. After performing syntax checks within the code, this tool takes the AST of the buggy submission and transforms it into a vector. Upon transformation, this vector is then given to a rule-based reasoning system, which identifies the different patterns in the code, flagging the ones that appear to demonstrate bad coding practices or mistakes. FLIP then uses this information to give the student help by providing them with either small tips, references to the language documentation, code refactoring, etc.

Despite not using a model solution, this approach is also flawed when it comes to unorthodox solutions. This is because the vector that the submission AST is distilled down to is being compared to the norm, rather than being taken as an individual solution. As such, certain possible solutions which are not wrong, will be deemed "bad" practices and the feedback given will not be on the mistakes of the student itself, but rather the fact that the structure of the program is completely different.

To demonstrate the issues that these tools might come into conflict with, in Listing 3.3 we show a possible student submission for the problem of calculating the factorial of a given number `n`. This code is wrong, due to the `return 0;` in line 9, which should be `return 1;`, instead. Listing 3.4, shows what an example solution could look like. Here we can see why these tools might fall short in helping the student. The student's approach is based on recursion, with the structure of the program being correct and only the base case having an error. Meanwhile, the example/average implementation uses a `for` loop, to calculate the factorial. These tools will pick up on the completely different code structure and signal it as incorrect, in spite of the fact that the only thing wrong with the program is the assignment. This would lead the student to wrongly believe that recursion is wrong by default, rather than just pointing out the small assignment mistake that was done. As such, these methods still fall short when it comes to giving students properly personalized feedback.

3.5.2 Large Language Model Based Automated Feedback Tools

As alluded to in Section 3.4, LLMs have become increasingly interesting in the realm of education [38–41]. Their capabilities are of particular interest in the realm of APR, where their versatility can be applied both during the Fault Localization (*FL*) phase and Patch Generation (*PG*) phase. This versatility extends to the feedback as well, with several models, such as `ChatGPT-4`, providing explanations for most of the steps taken during the repair/creation process of the new code [56, 57]. Given this fact, LLMs have started to become increasingly more attractive when it comes to automated feedback generation.

As shown in several studies [58, 59], certain LLMs like `ChatGPT-4` can provide accurate feedback in close to 90% of cases, with the feedback given being similarly as helpful as what Faculty members provide [59]. While these results are extremely promising, LLM based automated feedback still has severe drawbacks.

The first is the lack of validation of the generated feedback that the LLM provides the student. Given that the feedback is automated, the generated feedback is not reviewed by a human, meaning that miss-identifications of problems, along with errors in the suggestions of the LLMs go unnoticed. These problems can severely impact the performance of students and lead to more confusion, since the miss-identification of the problems will cause the students to "correct" sections of code that were already correct and make unnecessary changes.

Another point of contention related to the lack of supervision, is the fact that several instructions have to be given in order to prevent the models from returning the solutions, to improve the way the feedback is phrased and to improve the efficiency of the model overall. While this can be achieved via the "roles" and header instructions provided to the model, prior to its interaction with the student, the vast majority of these models can be broken rather easily with simple prompting techniques, such as Direct Prompt Injections. This allows the students to override the initial instructions and alter the output of the tools, which could lead to cheating by removing the output modifiers that had been initially introduced and allowing the LLM to return the solution in full. While completely preventing cheating is impossible, regardless of methods, a more significant concern that this override creates is in regards to security breaches. The fact that these models can be overridden consistently, while also having access to information regarding the machines and network they are run on could open the door to malicious actors [60]. As such, this method would require significant scrutiny, as to minimize these security concerns.

Finally, the models capable of in-depth feedback generation tend to be significantly more computationally intensive than the purpose built LLMs that are used for APR and other more specific tasks. This means that non high end hardware would struggle to maintain a system that makes use of these types of tools, as the amount of interactions with students on a reasonably sized institution would be too big to be feasible. With this in mind, non open source options would also be problematic as, in addition to the

computational requirements, the financial requirements would be too great to be put into practice.

4

FeedFix

Contents

In this Chapter, we present `FeedFix`, along with detailing the how each of the modules was implemented, as well as the reasoning behind the choices taken when deciding on the different possible approaches. Throughout this Chapter, we will use an example of a student submission for the problem of printing the biggest of any 3 given numbers. This example will serve to demonstrate the result of the choices that were made throughout the development process.

4.1 FeedFix Architecture

`FeedFix` is comprised of three major modules:

1. An LLM, which is responsible for generating the potential fixes.

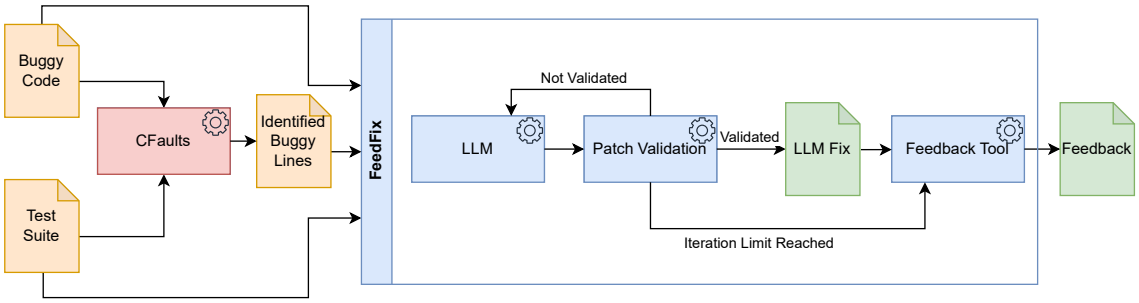

Figure 4.1: FeedFix's Architecture

2. The CEGIS loop that was described in Section 3.4.1.A, responsible for the Patch Validation of the generated code.

3. The Feedback Tool itself, which will take the generated correct code and the original buggy submission and use them to create a sketch, where the buggy part of the student's program is abstracted away, retaining the structure of the program, but not giving details that they might otherwise not have access to, e.g., wrong arguments in a given function call.

Figure 4.1 shows an overview of `FeedFix`'s architecture. `FeedFix` starts by taking the output of *CFaults* [17], which utilizes the student buggy submission and test suite to identify and output the buggy lines in the program and feeding them to an LLM. Next, an LLM attempts to fix the program. Upon each attempt, we check if the potential fix manages to pass every test in the test suite, or if it still fails to do so. In the case that the PV fails, we ask the LLM to try again. After hitting a given time limit (**TIMEOUT**) or reaching a certain number of attempts, we skip this step and instead use the output of *CFaults* [17] as the feedback. Otherwise, in the case that the fix was successfully validated, we use this fixed program, along with the buggy program to create the feedback (see Section 4.4) that will then be given to the student.

4.2 Automated Program Repair

`FeedFix` is divided into three modules with the first two being responsible for the repair of the faulty program.

Fault Localization (FL)

Firstly, we have the FL tool. Considering that we are trying to tackle Introductory Programming Assignments (IPAs) which are written in the `C` language, we elected to use *CFaults* [17], a tool created with the express intent of locating bugs in the `C` programming language. While not a part of `FeedFix` itself,

Listing 4.1: Original Student Submission

```
1  #include <stdio.h>
2
3  int main()
4  {
5          int n,i,z;
6          scanf("%d",&n);
7          scanf("%d\n",&i);
8          scanf("%d\n",&z);
9          if (n>i && n>z)
10         printf("%d\n",n);
11         if (i>z)
12         printf("%d\n", i);
13         else
14         printf("%d\n",z);
15         return 0;
16 }
```

CFaults' role in the tool's process cannot be understated, as the localization files are directly responsible for a significant increase in the performance of the APR tools, as will be discussed in Chapter 5 [1]. While `CFaults` was extremely helpful in identifying the faults within the programs and significantly improving the performance of the different prompting strategies that will be discussed in Section 4.3, this tool is not without its flaws. Although extremely accurate, `CFaults` is unable to identify every single bug, owed to the fact that it is a bounded process and that the tool lacks guarantees when it comes to proper pointer manipulation (specifically when non-allocated memory space is accessed). In spite of this, `CFaults` still performs at a high enough level to where we can disregard these drawbacks.

While `FeedFix` was built with `CFaults` [17] in mind, it can easily be adapted to utilize a different FL tool instead, only requiring modifications to the parsing of information from the file where the information is stored. Considering this, the **FB** configurations would also have to be slightly adjusted, since these configurations interact with the feedback files directly (see Section 4.3). Outside of these two modifications, all of the other parts and configurations of the tool would be unaffected.

Listing 4.1, shows the original student submission for the running example. Here, it would be the task of the FL tool to identify lines 7, 8 and 11 as faulty (highlighted in red). Please note that the yellow highlight serves to showcase a portion of code that is not necessarily wrong, but will be changed by the LLM during the patch generation itself. Lines 7 and 8 are incorrect, as the formatting of the `scanf()` clashes with the way in which the tests input their values, causing issues reading the values. The other main problem is the fact that the `if` statement in line 11 should instead be an `else if`, as not having the exclusionary effect of an `else` can cause duplicate prints, in the case that `n > i && n > z` is true and `i > z` is true, the student's program would output both the value of `n` and the value of `i`.

[1] A brief overview behind the principles and inner workings of *CFaults* can be found in Section 2.1.1.

Patch Generation (PG) and Patch Validation (PV)

The other major component of `FeedFix` is the portion responsible for the patch generation (*PG*) and patch validation (*PV*). This component is divided into two parts: 1) The LLM responsible for the PG for the different fixes and 2) The CEGIS loop responsible for the PV and seeing whether or not we have reached the iteration limit. We follow the CEGIS approach used in prior work [9] (see Section 3.4.1.A).

4.2.1 Large Language Models

For this tool, we wanted to focus on LLMs that would be open source and best function in "lower" end specs and with minimal resources, given the intended use case of helping students with their IPAs. In our case, the Graphics Processing Units (GPUs) used had 16GB of memory, which meant that any LLM we might want to use was limited to that maximum. It was considered whether or not to distribute the LLMs per various GPUs, in order to have access to larger models. However, this idea was abandoned as the distribution would be extremely dependent on the architecture of the servers and would require a far greater use of resources. Given that University resources are limited as is, the added strain this would cause, effectively doubling the amount of GPUs required, would significantly reduce the practicality of `FeedFix`, as it would end up requiring far too many resources to be put into use.

Several different LLMs were considered and tested, with different LLMs being run through different tests, but with the vast majority being abandoned for differing reasons.

1. **DeepSeek** - At the time we began this project, *DeepSeek* [61] had become international news, given the immense breakthroughs that occurred in early 2025. As such, we considered and tried to test *DeepSeek*, given its popularity and the buzz around this model. However, the available models were far too heavy for the GPUs we were using, having to be separated into at least 2 different GPUs even on the lightest models. This, combined with the fact that we would require a complete overhaul of how we interfaced with the model, was deemed to not be worth the effort and was scrapped shortly thereafter.

2. **Llama3** - Another LLM that was heavily considered was *Llama3* [62]. We discarded this model due to the fact that the *Llama3* version we were using was relatively recent and utilizes far more recent software[2], as a result. This made it so that several required dependencies within the environment would clash with the dependencies required for the LLM to work properly. This is a problem that we faced several times, as the newer models tended to clash with the older dependencies we needed to properly run the various LLMs.

3. **Gemma3** - Similarly to *Llama3* [62], *Gemma3* [63] also faced the same issues of clashing software dependencies. As such, it was abandoned shortly after.

[2]The version of Llama we tested was released in the back half of 2024

4. **Phi4** - Similarly to *Deepseek* [61], *Phi4* [64] also suffered from being a model that was far too large for the used GPUs. As such, we wound up trying to test *Phi4-Mini* [65], but the problems of the dependencies clashing with one another wound up being too vast to overcome, once more.

5. **Magicoder** - Similar to the ones mentioned prior, *Magicoder* [66] suffered from the models being too large, along with the conflicting software dependencies, which caused us to abandon it.

6. **Starcoder** - Similar to *DeepSeek* [61], *Starcoder's* [67] available models were far too big to be able to run on the available GPUs.

7. **Mistralai** - Unlike the ones before it, *Mistralai* [68] was heavily considered and tested extensively, only being dropped at the end of the refinement of this portion of the solution. This was due to the fact that we intended to test Fill in the Middle (FIM) requests and their effects on the solve ratio of the different LLMs, which are not supported by *Mistralai*.

8. **CodeGemma** - The first of our selected LLMs, *CodeGemma* [69] was selected due to the fact that it was light enough to be used in the available GPUs, along with having all the functionality that we required.

9. **CodeLlama** - The second LLM we selected was *CodeLlama* [70], since it also met all the requirements while not having any of the problems that the other LLMs faced.

10. **QwenCoder-2.5** - Likewise, *QwenCoder* [71] was also chosen due to the fact that it met all the requirements, without having dependency or size issues.

4.3 Configurations Used

Instead of trying to fine tune the different LLMs around the problem, we elected to use different prompting strategies to find and tune said prompts to the best outcome. As such, several different configurations of the prompts were used to try and assess which of the different prompt strategies would result in the best possible performance. Ideally, these configurations would improve both the repair rate and the similarity with the original submitted code. Each of these configurations aimed to alter small sections of the prompt, increasing the amount of information that was provided to the LLMs and to see whether that would augment the repair tools' capabilities. As mentioned in Section 3.4.1.A the configurations we used were based around the CEGIS loop and we followed the prompting style previously used in other works [6, 9, 53]. The different configurations and the rationale behind each of these configurations being picked is as follows:

1. **De:** Gives the description of the IPA, along with the buggy program and instructions to not add unnecessary explanations. This configuration serves to give a baseline for each of the different tools, as it is the simplest possible prompt and each reply gives no further indications.

2. **De-TS-CE:** Along with the description of the programming assignment, we also send the test suite to the LLM. In the case that the attempted fix fails, we now also reply to the model with a counter example of a test case which the newly generated program fails. This configuration was chosen as it adds a small amount of back and forth, providing additional context whenever the LLM gives a wrong reply, hopefully improving the overall capabilities of the model.

3. **De-TS-CE-EX:** Expanding on the former configuration, this option adds two significant of changes to the prompt. The first is FL, as the buggy code is now altered, prior to being sent to the LLM, and the text `/* FIXME */` added to the end of the lines that were identified as having faults. The second change is that three code examples are given to the LLM on how to handle this addition. These two changes are meant to provide the tool with better guidance, by giving the model a location of the mistake and how the code should be returned.

4. **De-TS-CE-EX-FB:** In addition to all of the capabilities of the `De-TS-CE-EX` configuration, this option handles some of the problems that the prior FL options might have. In some cases, the FL tool is not able to pin point the exact line that the mistake occurs at, either due to wrong formatting of the answer or certain other edge cases. When this occurs, the report needed to add the `/* FIXME */` text is not available and therefore, it can not be added. This option allows the prompt generator to access the feedback file generated by the FL tool and, in turn, add the feedback given by it to the LLM prompt.

5. **De-TS-CE-FB:** Unlike the prior options, this configuration removes the three examples on how the LLM should deal with the `/* FIXME */` lines. This configuration was mainly chosen to see if the increased information given by the examples had a positive effect on how the LLMs managed to resolve the issues, or if it had a detrimental or negligible effect.

6. **If_De-TS-CE:** This option diverges from the others, as it utilizes the infilling/FIM abilities of the different LLMs to repair the different programs. To achieve this, FL is used to locate the faults and then the code between the first and last fault is removed, replaced by the infilling keyword of the given LLM. When FL is not available, it behaves identically to `De-TS-CE`. Listing 4.2, shows an example of how this configuration operates, utilizing Listing 4.1 as the start point. As we can see, any lines between the initial fault and the final one were removed, being replaced by the, in this case, *QwenCoder* [71] token for the FIM process.

7. **Sk_De-TS-CE:** This option, like all the others starting in `Sk`, utilizes sketches. This means that

Listing 4.2: Infilling Example Portion

```
int main()
{
        int n,i,z;
        scanf("%d",&n);
<|fim_suffix|>
        else
        printf("%d\n",z);
        return 0;
}
```

instead of adding the `/* FIXME */` suffix to every faulty line, the lines are deleted and replaced with `/* HOLE N */`. This allows us to remove the bias towards the original mistake from the repair tool, essentially giving it a blank slate to repair that line of the program. Outside of this difference, it has the IPA description and counter examples present in the other options, being functionally the same as `De-TS-CE` otherwise.

8. **Sk_De-TS-CE-EX:** Functionally the same as `De-TS-CE-EX`, with the only difference being the use of the sketch approach. As such, the examples given were modified to accommodate the Sketch instead.

9. **Sk_De-TS-CE-EX-FB:** Functionally the same as `De-TS-CE-EX-FB`, with the only difference being the use of the sketch approach. As such, the examples given were modified to accommodate the Sketch instead.

10. **Sk_De-TS-CE-FB:** Functionally the same as `De-TS-CE-FB`, with the only difference being the use of the sketch approach.

Of these different configurations, **De**, **De-TS-CE** and **Sk_De-TS-CE** were already implemented and evaluated in [6, 9].

Iteration Limit

The last thing we implemented, relative to the APR process, was an iteration limit that is responsible for stopping the repair process, if the LLM does not find the correct solution after some time. This was done via a **TIMEOUT** of 90 seconds, where if the tool has not managed to return a valid fix, we declare it as a **TIMEOUT** and move onto the next submission. The other limiter that was added was in relation to the number of iterations themselves.

Initially, we had thought of adding a limiter on the raw number of iterations themselves. However, depending on how fast each of the LLMs was at responding, the number of iterations that could happen in the 90 second window was wildly different. As such, given that we had already imposed a time

Listing 4.4: Problem Description and Test Suite

```
### Problem Description ###
(Maximum between 3 Numbers)
Write a program that
determines and prints
the largest of three integers
given by the user.

### Test Suite
#input:
6 2 1
#output:
6

#input:
-1 3 1
#output:
3

#input:
1 2 3
#output:
3
```

Listing 4.5: LLM Prompt Program

```
### Incomplete Program <c> ###

#include <stdio.h>

int main()
{
        int n,i,z;
        scanf("%d",&n);
        /* HOLE 1 */
        /* HOLE 2 */
        if (n > i && n > z )
        printf("%d\n",n);
        /* HOLE 3 */
        printf("%d\n", i);
        else
        printf("%d\n",z);
        return 0;
}
```

limit, there was not an appealing reason for making the iteration limiter based around the raw number of iterations. Instead, we observed that many of the timed out fix attempts occurred because the LLM would reach a point where it would ignore any new instructions and would simply repeat the same potential fix over and over again. This is due to the LLMs poor understanding of formatting, which would cause them to be unable to fix bugs where additional prints are being made. With this in mind, we made an iteration limiter where, if we had received this same exact potential fix 3 times in a row, we would consider the repair as having failed. We did not store fixes that had already been tried that were not done back-to-back, as, if this happened, it meant that the LLM was still advancing its state, despite making the same mistake again.

This change significantly improved the speed at which we could test the dataset, as the limiter helped us improve the performance of our APR module.

Returning to the example on Listings 4.1, an example of the prompt utilized while using the **Sk_De-TS-CE-EX-FB** configuration with *QwenCoder* [71] can be seen in Listings 4.3 through 4.5. Each of these Listings showcase different portions of the prompt used to fix this program. Listing 4.3 starts by explaining that we are using sketches and explaining how it works. Then, we start providing examples of a program sketch and how we want the LLM to behave when trying to fix said sketch. Important to note that we offered 3 different examples, with different exercises in each one, but omitted the other 2 from these Listings.

After providing the different examples, we continue by giving the description of the IPA and the test suite, as seen in Listing 4.4. Finally, we add the buggy program to the prompt, as seen in Listing 4.5.

Listing 4.3: Sketch and Example Portion

```
Complete all the '/* HOLES N */'
in the incomplete program below.

Modify the code as little as possible.
Do not provide any explanation.

To exemplify the desired behaviour,
here are 3 examples.

In the first example we want to get
the biggest of 3 numbers.

The program:
int main(){
        int f,s,t;
        scanf("%d %d %d",&f,&s,&t);
        /* HOLE 1 */
                printf("%d",f);
        else if (s > f && s >= t)
                printf("%d",s);
        /* HOLE 2 */
                printf("%d",t);
        return 0;
}
Should be corrected to:
int main(){
        int f,s,t;
        scanf("%d %d %d",&f,&s,&t);
        if (f >= s && f >= t)
                printf("%d",f);
        else if (s > f && s >= t)
                printf("%d",s);
        else
                printf("%d",t);
        return 0;
}
```

After the prompt is completed, we wait on the response from the LLM. In this case, we did not get the correct fix on the first try, so we send back the response shown in Listing 4.6. In this prompt we inform the LLM that its fix was not correct and provide it with an example input that causes the new program to fail. After a few failed attempts, Listing 4.7, shows the fix that managed to be validated.

4.4 FeedBack Tool

The last main component of `FeedFix` is the Feedback Tool itself. This tool is meant to take the solutions achieved by the LLMs during the Automated Program Repair (see Section 4.2) and the original student submission and create a sketch of the repaired program. This sketch takes the repaired program and abstract away the various different sections of code that were edited or added to the original submission. Ideally, this new sketch will serve as a guideline that allows the student to have a rough draft of the correct approach, while sticking as close as possible to the student's own submission.

Listing 4.6: Response Prompt

```
### Feedback ###
Your previous suggestion was incorrect!
Try again. CODE ONLY.
Provide no explanation.

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

Listing 4.7: LLM Fix

```
#include <stdio.h>

int main()
{
        int n,i,z;
        scanf("%d",&n);
        scanf("%d",&i);
        scanf("%d",&z);
        if (n>=i && n>=z)
        printf("%d\n",n);
        else if (i>=z)
        printf("%d\n", i);
        else
        printf("%d\n",z);
        return 0;
}
```

A simple overview of the Feedback Tool is given in Section 4.4.1 to establish the main points that will be expanded upon in further Sections.

4.4.1 FeedBack Tool Overview

This tool utilizes the *pycparser*[3] library to parse and store the different ASTs of the original submission and solution to then traverse them and construct the sketch. This approach is intended to remove several of the downsides of using a text change based option, as the ASTs retain more information and allow for finer handling of the errors that are found.

This approach has its downsides, however, as traversing the two trees at the same time while trying to match the different nodes to one another is not an ideal approach (further in-depth explanation as to why, can be found in Section 4.4.3). As such, a combination of the two was chosen. This will allow us to skip a lot of the iterations and recursion necessary on a solely AST based approach, while retaining most of the advantages that this method provides when compared to the purely text based approach. It also avoids several of the downfalls that only using the AST approach could be subject to.

The Feedback tool is divided into three different processes, each serving to improve the results of the tool while trying to minimize the impact on its performance. These processes are as follows:

1. Normalize both the original submission and the LLMs solution.
2. Generate a `diff` file between the newly normalized codes.
3. Generate the sketch based on the normalized solution and original student submission and their respective `diff` file.

[3]Repository is available at: https://github.com/eliben/pycparser

The first step consists of normalizing each of the two codes. This step is required, as the approach used is a mixed one, which leads to certain problems in the text-based portion of the method, when compared to a pure AST approach.

The second step consists of generating the `diff` file of the two codes. This `diff` file will provide the information to better locate the mistakes that occur within the students' code.

The final step consists of taking the normalized files, as well as the `diff` file between them, to traverse the solution AST and generate the final feedback sketch. Many different approaches for this final step were prototyped and tested. The various approaches as well as the reasoning behind each of them and the causes behind abandoning each one can be found in Section 4.4.3.

4.4.2 Normalizer and Diff Generator

The first step in the generation of the feedback is to normalize the two different codes. While this step is not required when utilizing the AST based approach that we are mainly leveraging, it is required to prevent problems that might arise while using the text-based portion of it. This is due to the `diff` file that is needed to locate the changes that occurred during the automated repair process.

Listings 4.8 and 4.9, showcase an example as to why this process is necessary.

In this example, we want to get the maximum number between the three numbers given by the user. The original student submission (Listing 4.8) has a mistake in line 12 where the `printf("%d", c);` (highlighted in red) should, instead, be `printf("%d", b);`. The fix that the LLM managed to arrive at is displayed in Listing 4.9. As we can see, the change is rather simple, modifying the `c` to the correct `b`. However, this is not the only change that occurs. In line 11, we can see that the original code had parenthesis identifying each of the clauses on the `else if` condition. This was modified in the LLM fix, where the parenthesis were removed. This change does not impact the program in any way, amounting to, at most, a clarity or preference issue. Despite not presenting a problem to the AST based approach, this change in formatting would be identified when the `diff` command is run between the two different codes. This would then lead to a loss of performance, as the sketch generator would have to spend extra time trying to find what was wrong in said line, despite no errors existing.

While in this case, the tool would easily be able to identify that there were no errors, a case where mistakes and confusion might arise is when we take things like multiple declarations or multiple statements in the same line. In these same two Listings, line 5 is also highlighted. These yellow highlights are meant to indicate portions of the code that the *Normalizer* is intended to normalize.

This declaration stands out as a major point that needs normalization, as the multiple declarations lead to interesting problems when utilizing the AST approach. Mainly, the fact that the multiple declarations get stored differently within the ASTs, when compared to declarations in different lines. As an example, we can observe a different possible fix in Listing 4.11. While this is also the result of normal-

Listing 4.8: Original Student Submission

```c
#include <stdio.h>

int main()
{
    int a, b, c;
    scanf("%d %d %d", &a, &b, &c);

    if (a >= b && a >= c){
        printf("%d", a);
    }
    else if ((b >= a) && (b >= c)){
        printf("%d", c);
    }
    else if (c >= a && c >= b){
        printf("%d", c);
    }

    return 0;
}
```

Listing 4.9: LLM fix

```c
#include <stdio.h>

int main()
{
    int a, b, c;
    scanf("%d %d %d", &a, &b, &c);

    if (a >= b && a >= c){
        printf("%d", a);
    }
    else if (b >= a && b >= c){
        printf("%d", b);
    }
    else if (c >= a && c >= b){
        printf("%d", c);
    }

    return 0;
}
```

ization of Listing 4.9 (as will be elaborated on further ahead), this code could possibly be one of the fixes that any given LLM might arrive at. As we can see, the multiple declarations that occur in line 5 in the original submission were turned into three different lines, each having a declaration for each of the variables. Despite having no impact on the code itself, this change actually affects the AST, as well as the resulting `diff` file. The reason behind why this change alters the AST is that the resulting tree from the original submission will have a `Compound` (Code Block) node which will store the declarations, whereas the fix would have three separate declaration nodes and no `Compound` node. As such, when the Sketch generator arrived at the declarations, it would find that there are mistakes in that section. This problem would arise both in a pure AST approach, as well as when using a text-based method. This problem could even be exacerbated on the text-based approach due to the fact that we can have several expressions in a single line.

Listing 4.10 shows the normalized version of the buggy program that was submitted (Listing 4.8). As we can see, the declarations were separated as well, meaning that the `diff` file would not find any conflicts in the declaration lines between Listings 4.10 and 4.11, as would have been the case had we not first normalized them.

Let us imagine a different program that takes two numbers and prints either `True` or `False` depending on whether or not the first number is divisible by the second number. Listing 4.12 show a student submission for this programming exercise. In it, we can see the clear mistake on line 9, where the `if` clause is `a % b == 0` instead of the correct `a % b != 0`. This program is quite simple, as such it is reasonable to assume that some students might choose to essentially collapse their code, by not opening and closing parenthesis and by putting everything on the same line. While this does not directly impact the code, when FL is used the entire line is deemed as "wrong", despite the fact that the error

Listing 4.10: Normalized Student Submission

```c
#include <stdio.h>

int main()
{
    int a;
    int b;
    int c;
    scanf("%d %d %d", &a, &b, &c);

    if (a >= b && a >= c)
    {
        printf("%d", a);
    }
    else if (b >= a && b >= c)
    {
        printf("%d", c);
    }
    else if (c >= a && c >= b)
    {
        printf("%d", c);
    }

    return 0;
}
```

Listing 4.11: Normalized LLM fix

```c
#include <stdio.h>

int main()
{
    int a;
    int b;
    int c;
    scanf("%d %d %d", &a, &b, &c);

    if (a >= b && a >= c)
    {
        printf("%d", a);
    }
    else if (b >= a && b >= c)
    {
        printf("%d", b);
    }
    else if (c >= a && c >= b)
    {
        printf("%d", c);
    }

    return 0;
}
```

rests solely on the condition, rather than the `printf` that follows it.

As was explained in Section 4.2, several different ways of prompting the various LLMs were tried, with one of the options being the sketch option, which would create holes within the program according to the FL diagnosis. A direct consequence of utilizing the sketch method is that the entire line would be deleted and replaced by the comment `/* HOLE */`. As such, the LLM could arrive at a solution such as the one shown in Listing 4.13. As we can see, the fix was rather simple, with the only change required being the change in the operator. However, since the entire line was deleted in the original sketch fed to the LLM, the fix spread out the line to three different lines, instead of retaining the structure of the original which only utilized a single line.

Ideally, we want to isolate the mistakes as much as possible, in order to not have multiple different complex nodes that could be flagged as "wrong" when using the `diff` file. This also demonstrates the necessity behind normalizing both of the codes, as the added lines and the splitting of the `if` clause and `printf` function call would be alerted as a mistake, in spite of it being functionally the same. In the case of the AST based approach, this would only have an impact on performance, whereas when utilizing a text based method, these false positives would create a lot of problems to be dealt with.

In Listings 4.14 and 4.15 we can see the resulting normalized codes. Several small changes took place, with both the partitioning of the `if, else` clauses and the addition of the brackets to their own respective lines. While this changes nothing in terms of the functionality of either of the codes, this change will allow for the `diff` command to only find the portions of the code that impact functionality,

Listing 4.12: Original Student Submission

```c
#include <stdio.h>

int main()
{
    int a;
    int b;
    scanf("%d %d", &a, &b);

    if (a % b == 0) printf("False");
    else printf("True");

    return 0;
}
```

Listing 4.13: LLM fix

```c
#include <stdio.h>

int main()
{
    int a;
    int b;
    scanf("%d %d", &a, &b);

    if (a % b != 0) {
        printf("False");
    }
    else printf("True");

    return 0;
}
```

Listing 4.14: Normalized Student Submission

```c
#include <stdio.h>

int main()
{
    int a;
    int b;
    scanf("%d %d", &a, &b);

    if (a % b == 0)
    {
        printf("False");
    }
    else
    {
        printf("True");
    }

    return 0;
}
```

Listing 4.15: Normalized LLM fix

```c
#include <stdio.h>

int main()
{
    int a;
    int b;
    scanf("%d %d", &a, &b);

    if (a % b != 0)
    {
        printf("False");
    }
    else
    {
        printf("True");
    }

    return 0;
}
```

i.e., the initial `if` clause. This normalization will also lead to ignoring the change to the amount of lines that were taken up as would have been the case if non normalized codes were used.

As such, we normalize both of the programs in order to remove these formatting or presentation preferences, to ensure that both the ASTs and the text are formatted in the same way, preventing these mistakes.

For this normalization process, we are making use of some of the built-in functionality of *pycparser* for writing `C` programs based on their ASTs. Despite serving as a solid launchpad, we had to alter the behavior of the tool when it came to the `If` node, as we had to change the way in which we handled `Else` statements, as will be further elaborated on in Section 4.4.3.C.

Turning our attention once again to the example of Listing 4.1 and 4.7, in Listing 4.16 and 4.17, we can see the normalized programs for both the student submission and the LLM fix. As we can see, the

Listing 4.16: Normalized Student Submission

```
1  int main()
2  {
3    int n;
4    int i;
5    int z;
6    scanf("%d", &n);
7    scanf("%d\n", &i);
8    scanf("%d\n", &z);
9    if (n > i && n > z)
10     printf("%d\n", n);
11   if (i > z)
12     printf("%d\n", i);
13   else   printf("%d\n", z);
14   return 0;
15 }
```

Listing 4.17: Normalized LLM Fix

```
1  int main()
2  {
3    int n;
4    int i;
5    int z;
6    scanf("%d", &n);
7    scanf("%d", &i);
8    scanf("%d", &z);
9    if (n >= i && n >= z)
10     printf("%d\n", n);
11   else   if (i >= z)
12     printf("%d\n", i);
13   else   printf("%d\n", z);
14   return 0;
15 }
```

Listing 4.18: Diff File of Both Programs

```
1  7,9c7,9
2  <    scanf("%d", &i);
3  <    scanf("%d", &z);
4  <    if (n >= i && n >= z)
5  ---
6  >    scanf("%d\n", &i);
7  >    scanf("%d\n", &z);
8  >    if (n > i && n > z)
9  11c11
10 <    else   if (i >= z)
11 ---
12 >    if (i > z)
```

formatting underwent some significant changes, especially in the declaration of the variables in lines 3 through 5, where the multiple declarations were separated into three different lines.

Diff File Generation

After normalizing both of the codes, we run the command:

```
diff -w llm_fix.c student_submission.c > diff_file.diff
```

This command produces a file that is then used in the sketch generation process. It is important to note that despite the normalization, the program we use to create this normalization retains certain white spaces, such as empty lines. Considering this, the `-w` flag was used when creating the resulting `diff` file, in order to prevent said lines and added white spaces from interfering with the results and having false positives.

Returning to the example of the three biggest numbers problem we have been using throughout this Chapter, we take the normalized codes shown in Listings 4.16 and 4.17, to generate the `diff` file shown in Listing 4.18. As will be discussed in Section 4.4.3.C, the behavior of the `else` statement in line 11

Inserted Hole	What was Sketched Out
/* Constant */	String, Int, Float, etc...
/* ID */	Identifier of a Variable or Function.
/* FuncCall */	Function Call
/* Op */	Operator (Not a Node, only used in Sketches)
/* UnaryOp */	Unary Operation
/* BinaryOp */	Binary Operation
/* TernaryOp */	Ternary Operation
/* Assignment */	Assignment
/* IdentifierType */	Type of a Node, i.e. int, float, str, etc..
/* TypeDef */	Type During a Definition, i.e. int, float, str, etc...
/* Decl */	Declaration
/* FuncDef */	Function Definition
/* Return */	Return
/* If */	If
/* While */	While
/* DoWhile */	Do While Structure
/* Switch */	Switch
/* Case */	Case
/* Default */	Default

Table 4.1: Sketch Caption

is extremely relevant, as, had we not removed the additional new line, the following `if` would be a line down and it would have caused matching errors during the sketching process, given the fact that the `diff` command would have both the `else` and `if` lines, were these statements to be in different lines.

4.4.3 Sketch Generator

The last step in generating the feedback is the sketch generator itself. It is responsible for taking the normalized student submission and normalized LLM fix, as well as the `diff` file between them and creating a sketch of the solution.

The sketch generator had a lot of different prototypes, each utilizing either entirely different approaches, or with certain tweaks that drastically altered the output of the generator itself. This Section will focus on providing an overview of the different choices that were tried, as well as the tweaks that were made to improve the output to be as close as possible to the ideal.

Despite differing approaches, every iteration and tweak done was based around the *pycparser* library, using the standard frameworks that the library uses as a building block for the sketch generator.

Throughout this section, we will refer to several types of nodes within the ASTs. Table 4.1 shows what each of the node types represent as well as the comments used during the sketching process.

4.4.3.A Pure Dual Abstract Syntax Tree Traversal Approach

Initially, we attempted to create a tool that only utilized the AST method to traverse both trees at once and design the sketch as the trees are being traversed. While this method seemed promising, its implementation proved to be extremely challenging. This is mainly due to the fact that the structure of

Listing 4.19: Normalized Student Submission

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int aux;
6      int max;
7
8      scanf("%d", &max);
9
10     for (int i = 0; i < 3; i++)
11     {
12         scanf("%d", &aux);
13         if (aux > max) max = aux;
14     }
15
16     printf("%d", max);
17
18     return 0;
19 }
```

Listing 4.20: Normalized LLM fix

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int a;
6      int b;
7      int c;
8      int max;
9      scanf("%d %d %d", &a, &b, &c);
10
11     if (a >= b && a >= c)
12     {
13         max = a;
14     }
15     if (b >= a && b >= c)
16     {
17         max = b;
18     }
19     if (c >= a && c >= b)
20     {
21         max = c;
22     }
23
24     printf("%d", max);
25
26     return 0;
27 }
```

the abstract trees themselves is intended to be traversed recursively. As such, visiting the nodes of two different trees at once generates significant friction, in the case that the two trees do not line up perfectly with one another.

While this does not make it impossible to traverse both trees at once, it does create major issues when the abstract tree structure does not match. Shown in Listings 4.19 and 4.20, are examples of a program intended to calculate the biggest of three numbers given by the user. While this example might seem far fetched, it is a slightly modified version of a real student submission and its respective LLM fix pair. In it, we can see that the original code is extremely different from a “normal” solution for this problem. The mistake lies in line 10, where the for loop should only have two iterations, rather than the three that were used. Despite being a simple change to a human, in several of the LLM fixes, the models were unable to find fixes that retained the structure of the original student submission, opting instead for a solution that was closer to what would be classified as an “average” solution. As a result, the structure of the AST of the fix is vastly different from the structure of the original buggy submission. This is evident within the code itself, as the `if` clauses that exist in the LLM fix are almost entirely absent on the original submission, but the two biggest differences are the `for` loop and the second `scanf` function call that is called during the iteration. Another smaller issue is the number of variables, which increased from the original three (counting the iterator `i` of the `for` loop) to four.

The ASTs of both codes are shown in Listings 4.21 and 4.22. As we can see, their structure is

Listing 4.21: Student Submission AST

```
FileAST:
  FuncDef:
    Decl: main, [], [], [], []
      FuncDecl:
        TypeDecl: main, [], None
          IdentifierType: ['int']
    Compound:
      Decl: aux, [], [], [], []
        TypeDecl: aux, [], None
          IdentifierType: ['int']
      Decl: max, [], [], [], []
        TypeDecl: max, [], None
          IdentifierType: ['int']
      FuncCall:
        ID: scanf
        ExprList:
          Constant: string, "%d"
          UnaryOp: &
            ID: max
      For:
        DeclList:
          Decl: i, [], [], [], []
            TypeDecl: i, [], None
              IdentifierType: ['int']
            Constant: int, 0
        BinaryOp: <
          ID: i
          Constant: int, 3
        UnaryOp: p++
          ID: i
        Compound:
          FuncCall:
            ID: scanf
            ExprList:
              Constant: string, "%d"
              UnaryOp: &
                ID: aux
          If:
            BinaryOp: >
              ID: aux
              ID: max
            Assignment: =
              ID: max
              ID: aux
      FuncCall:
        ID: printf
        ExprList:
          Constant: string, "%d"
          ID: max
      Return:
        Constant: int, 0
```

Listing 4.22: LLM fix AST

```
FileAST:
  FuncDef:
    Decl: main, [], [], [], []
      FuncDecl:
        TypeDecl: main, [], None
          IdentifierType: ['int']
    Compound:
      Decl: a, [], [], [], []
        TypeDecl: a, [], None
          IdentifierType: ['int']
      Decl: b, [], [], [], []
        TypeDecl: b, [], None
          IdentifierType: ['int']
      Decl: c, [], [], [], []
        TypeDecl: c, [], None
          IdentifierType: ['int']
      Decl: max, [], [], [], []
        TypeDecl: max, [], None
          IdentifierType: ['int']
      FuncCall:
        ID: scanf
        ExprList:
          Constant: string, "%d %d %d"
          UnaryOp: &
            ID: a
          UnaryOp: &
            ID: b
          UnaryOp: &
            ID: c
      If:
        BinaryOp: &&
          BinaryOp: >=
            ID: a
            ID: b
          BinaryOp: >=
            ID: a
            ID: c
        Compound:
          Assignment: =
            ID: max
            ID: a
      If:
        BinaryOp: &&
          BinaryOp: >=
            ID: b
            ID: a
          BinaryOp: >=
            ID: b
            ID: c
        Compound:
          Assignment: =
            ID: max
            ID: b
      If:
        BinaryOp: &&
          BinaryOp: >=
            ID: c
            ID: a
          BinaryOp: >=
            ID: c
            ID: b
        Compound:
          Assignment: =
            ID: max
            ID: c
      FuncCall:
        ID: printf
        ExprList:
          Constant: string, "%d"
          ID: max
      Return:
        Constant: int, 0
```

markedly different, with very little overlap, besides for the initial few nodes, the return node, and the nodes representing the `int max;` and `printf("%d", max);` expressions. These differences heavily impact this approach, as the recursive approach would require the tool to mix and match each of the nodes from both trees to try to find the similarities. Using this approach, the recursion would go as follows:

1. The recursion would start by checking the `FileAST` node. Given that they both "match", it would then proceed to the next node on both trees.

2. It would then move onto the `FuncDef` (Function Definition) node, which exists in both trees. It would then proceed with the traversal on both trees.
3. We would then visit the `Decl` (Declaration) node and its "children" and reach both the `IdentifierType` (Type of Declaration, i.e., int, str, etc.) nodes without issues.
4. After successfully checking both the `Decl` nodes on line 3, we would move onto the `Compound` (Code Block) node on line 7. The tool would then enter the node, to see the declaration node, which does not match, as the variable names are different.
5. Considering the recursive nature of the ASTs, the only way to achieve the desired outcome would be to do a depth-first search on either of the trees to see if any node matches. In the prototype, we elected to do this search on the student submission (Listing 4.21 tree), since we wanted to accumulate the LLM fix's code string for the final feedback.
6. Given this fact, the search would traverse the entirety of the student submission's AST, before finding no match. This lack of match would then be propagated to the upper node (in this case the `Compound` node in line 7) which would then move on to the next child, a second `Decl` node in line 11 of Listing 4.22 .
7. It would repeat step 5 on both this `Decl` node and the one responsible for declaring the variable `c` too.
8. Only once it reached the `Decl` node for the variable `max`, would it find a match and proceed.
9. It would then reach the `FuncCall` (Function Call) node, responsible for calling the `scanf` function. This node would match, as both of the calls are made to the same function.
10. Once it reached the `ExprList` (Expression List) node, the traversal would not find a match.
11. This process would repeat for each of the AST nodes of the fix tree, failing to find any match, up until the final `printf` and `return 0` expressions.

This approach had some glaring flaws, as can be seen by the description of the process. One of the main drawbacks was the fact that the amount of backtracking necessary on a worst case scenario was quadratic. This is due to not being aware of the location of the changes, requiring us to have to traverse the submission tree fully and try to match each of the nodes to their respective fixed counterparts. In the event that no match is found, we would move on to the next node in the fixed code tree. However, we would have to return to the last non-matched node in the submission tree. This is because the fixed line we were unable to find a match for might be a completely new line that was added by the LLM. As such, we cannot discard the entirety of the student submission and have to return to the node in the student submission we tried to originally match.

A potential fix to the performance issue would be to effectively "lock" the search to the current "depth". Essentially, the search would only be allowed to occur at the same level within the AST. While this would significantly reduce the number of nodes that need to be searched, it could cause other problems. The first has to do with how the nodes are represented within the AST itself. Since a `Compound` node (among others) might exist in one of the trees, while not existing in the other, even if a portion of the contents of the `Compound` node match the desired node, said contents would be a level deeper than that of the one it is trying to match. As such, it would not find the match, despite the correct node match existing within the `Compound` node. While most of these issues would be solved during the normalization phase (described in Section 4.4.2), if the problem was the lack of certain expressions or the existence of additional, unnecessary, expressions which could generate a `Compound` node, this problem would appear and we would not be able to tackle it.

Besides the performance issues, this matching strategy had another major problem, which was the fact that the nodes can be matched at wrong points. Listings 4.23 and 4.24 illustrate this problem. In Listing 4.23 there is a possible student submission for a program meant to print all the numbers in a Fibonacci sequence up to a user given number `n`. The two mistakes can be clearly found, with the first being the initial variable declaration in line 6, where instead of the intended `int prev1 = 1;` the variable is initialized to 0 and the `printf()` in line 17, which should be done before the addition in line 14. A possible, valid fix can be seen in Listing 4.24, where the variable initialization was corrected and the `printf()` call was changed to its intended position. While this should cause this function call to be diagnosed as an error and sketched out, this approach would treat this case differently.

Following the logic explained previously, the nodes would traverse each of the ASTs at the same time, reaching the mistake in line 6 and correctly identifying it as such. It would then continue until it reached the `FuncCall` node representing the `printf()` in line 14 of the LLM fix. Upon reaching this point, the tool would see that it could not find a matching node on the student submission AST, since it would currently be at the `fibonacci = prev1 + prev2 ;` node. Given this, it would then skip over this node, until it reached the `FuncCall` node in line 17, which would be a match to the node in line 14. This fact would effectively cause the portion of code that was skipped over to be eliminated, despite it being correct. Given this fact, this approach could be prone to sketching out far more information than what would be desired. Considering that the order in which expressions appear in the code might be crucial for the result (as is the case in this example), we can not backtrack to check whether or not the portions of code we skipped over are matches, as this could generate further false positive matches that could turn the sketch more confusing.

Due to these issues, as well as the difficulty and challenges presented during implementation, this approach was scrapped in favor of other options.

Listing 4.23: Normalized Student Submission

```
#include <stdio.h>

int main()
{
    int n, fibonacci;
    int prev1 = 0;
    int prev2 = 0;

    scanf("%d", &n);

    if (n <= 1) fibonacci = n;

    for (int i = 2; i <= n; i++){
        fibonacci = prev1 + prev2;
        prev2 = prev1;
        prev1 = fibonacci;
        printf("%d\n", fibonacci);
    }

    printf("%d\n", fibonacci);

    return 0;
}
```

Listing 4.24: Normalized LLM fix

```
#include <stdio.h>

int main()
{
    int n, fibonacci;
    int prev1 = 1;
    int prev2 = 0;

    scanf("%d", &n);

    if (n <= 1) fibonacci = n;

    for (int i = 2; i <= n; i++){
        printf("%d\n", fibonacci);
        fibonacci = prev1 + prev2;
        prev2 = prev1;
        prev1 = fibonacci;
    }

    printf("%d\n", fibonacci);

    return 0;
}
```

4.4.3.B Hash Based Sub-tree Matching

The second approach that was tested was an approach based on the number of matching hashes in the sub-trees of the two different codes.

This approach based itself around the fact that, ideally, the hashes of different portions of code would differ, whereas when the codes are the same, the hashes would match as well.

This method was chosen for our second prototype, as it would be able circumvent a lot of the issues that we faced when utilizing the Dual AST approach described in the previous section. Mainly, this would severely reduce the impact on performance and cut down on the wrong matching and recursive searching that plagued the previous approach.

Figure 4.2 showcases a simple overview of the process that was used on this prototype.

To begin this process, we would take the AST of the student submission and traverse the tree, with the intent of storing the hashes of the various nodes. Each node hash was comprised of the hash of its class name (i.e. a `FuncCall` or `ID` node, seen in Listing 4.21) and the hash of its children's hashes. An immediate problem becomes evident, as the order in which the children nodes are given is highly important. This is because a different order within the children of the node that we are visiting would result in a different hash. To avoid this, we sorted the hashes of the children node, prior to hashing the entire string. This would mean that the different hashes are ordered the same way, regardless of the order that they are written in the code, as long as its parents are the same. To demonstrate this process, let us use the `FuncCall` node for the `scanf()` in line 14 of Listing 4.21 and its corresponding node in

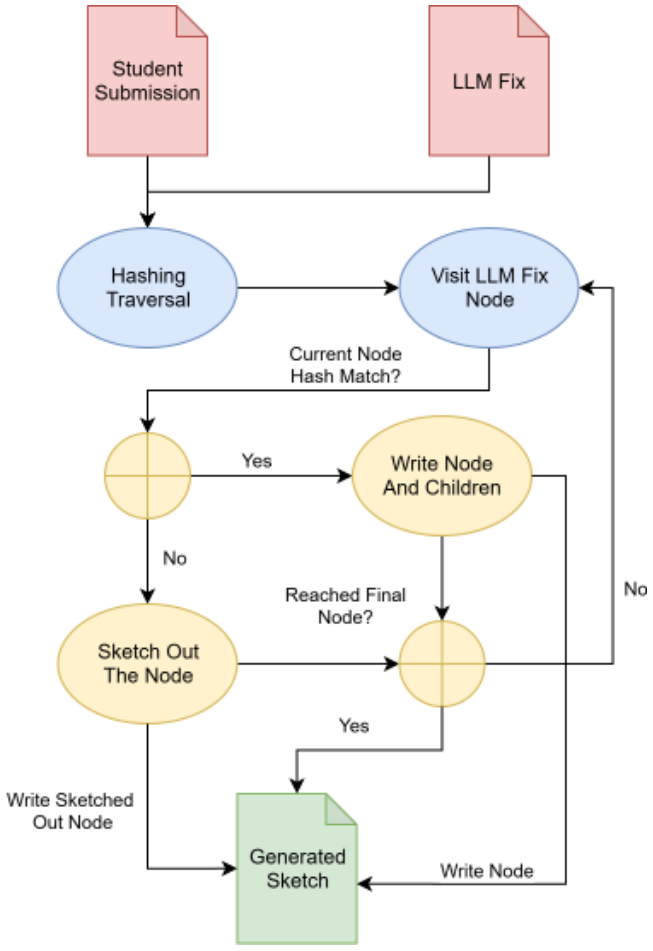

Figure 4.2: Hash Based Sub-tree Matching Process

Listing 4.22 in line 20. The hashing from the student submission `FuncCall` node will be as follows:

1. Visitor arrives at `FuncCall` (Function Call) node, visits its children and reaches the `ID` (Identifier) node in line 15.
2. Upon reaching said `ID` node, we hash the node, returning the hash(”ID”). This hash would then return the string “I”[4] to its parent.
3. We would then continue through the children, reaching the `ExprList` (Expression List) node and deeper into the `Constant` node in line 17.
4. From here, we would hash the string “Constant”, returning the hash “C”.
5. Following that, we would visit the `UnaryOp` (Unary Operation) node and then the `ID` node.
6. The `ID` node would return the hash “I”, since its the same node type as the one in line 15.
7. The `UnaryOp` node would then hash the following string “UnaryOpI”. This hash would return the string “UI”.
8. Upon the return of the hash, the `ExprList` node would sort the two strings. Therefore, it would take the “C” string as well as the “UI” string and sort between the two of them. As such, the next hash would be “ExprListCUI”. This would result in the string “ECUI” being returned.
9. Now that both the children of the `FuncCall` node have returned their hashes, we take the two hashes “I” and “ECUI” and sort the two, resulting in the string “FuncCallECUII”, which, upon hashing, would give us the hash “FECUII”.

After the traversal through the student submission AST, we traverse the LLM fix to store its hashes. This is done prior to the sketch generation itself, as it will allow us to have the hashing of a given node prior to visiting it during sketch creation, meaning that upon entering a node, we will know whether or not there is a match from the onset. If a match exists, then we write both the node and its children, since we can be sure that the entire sub-tree is matched correctly. Returning to the `FuncCall` example we used and following this same method, we can calculate that the hash for the `FuncCall` in line 20 of Listing 4.22 would result in the string “FECUIUIUII”. As we can see, the hashes of the two nodes are different, owing to the fact that the `ExprList` node is significantly different from one code to another. Given that the sub-trees do not match, we would then start the sketching process, to block out the parts that do not match.

It is in this step that we start seeing the drawbacks of utilizing this method. The hashing process simply tells us whether or not any given sub-tree appears within the AST of the code. It does not tell us

[4] To simplify this example, assume that the hashing for each of the different nodes is the first letter in its class name.

Listing 4.25: Normalized Student Submission

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int a;
6      int b;
7      scanf("%d %d", &a, &b);
8
9      if (a % b == 0)
10     {
11         printf("False");
12     }
13     else
14     {
15         printf("True");
16     }
17
18     return 0;
19 }
```

Listing 4.26: Normalized LLM fix

```
1  #include <stdio.h>
2
3  int main()
4  {
5      int a;
6      int b;
7      scanf("%d %d", &a, &b);
8
9      if (a % b != 0)
10     {
11         printf("False");
12     }
13     else
14     {
15         printf("True");
16     }
17
18     return 0;
19 }
```

which nodes appear, their order or their attributes. Therefore, when a sub-tree is not an exact match, we have no idea at which point within the sub-tree it stops matching. In the example we used, the function being called is the same, despite the arguments being different. Ideally, we would want the sketch to keep the `scanf()`, while abstracting away the arguments. However, in contrast to what we would want from this approach, the result is that any given sub-tree can only appear if the entirety of the sub-tree exists.

Another major issue resulting directly from this approach is the fact that each of the nodes is not unique. None of the different nodes have their unique identifiers (in the case of this `FuncCall` node the "scanf" ID). This was, in part, due to the fact that the sub-tree matching used was built to try to find matching section of code to find the most structurally similar codes, instead of actively trying to match the code to one another.

To try to reduce the impact that this might have on the sketch, we introduced a way to identify each of the nodes, by taking the identifiers of basic nodes and adding them to their hashes (i.e. an `ID` node would add its name to the hash and a `Constant` node would add the value being returned). While this helped with the creation of unique sub-trees, it still had some problems, mainly that the non-simple nodes would not have their own identifiers. Returning to Listings 4.25 and 4.26, we can see that the `if` statements, despite having different conditions, would generate the same hash. This is due to the fact that the constants and identifiers of the nodes are the same, with the only difference being the operator in line 9, which we did not take into account when hashing the program.

This could be solved by adding these identifiers to the non-simple nodes as well, with each of the different node types having different attributes that could be used in the hashing process. In the case of the `BinaryOp` nodes, for example, we can use the operator of the node to identify it. While it will not be

unique, it will reduce the frequency with which the nodes are wrongly matched with one another. With this change, we would be able to avoid the error in line 9. However, this change highlights an inherent problem with this approach: it is incapable of knowing whether a parent node exists or not.

Due to the way that the hashing process works, the closer we get to an "end" node, the simpler its hash will be. Recalling the example of the ASTs, the hash of the `ID` node in line 19 of Listing 4.21 and keeping in mind the changes we made thus far, the hash would only consist of "Imax", whereas its parent node `UnaryOp` would be "U&Imax". This hash becomes more and more complex as we go "up" the tree. The ramifications of this is that, if something does not match at a "deeper" level, everything before it can not be guaranteed to exist. As such, when generating the sketch, the non-terminal nodes are not able to know whether or not they exist, unless the hash matches exactly.

To try to tackle this issue, our next tweak was to store the "height" that each of the nodes can be found at. This height would then be used to see whether or not we had reached the bottom of the tree. Each node `Parent`, with `N` children would have their height set as follows:

$$Height_{Parent} = max(Height_{C0}, ..., Height_{CN}) + 1$$

In the case that the node does not have children (terminal node), its height is set at 0.

This height would then be used as we traverse the tree to see if we have reached the bottom of the tree or not. If we had not yet reached the bottom, we would represent the node, despite no matching hash being found. If we reached the bottom, we would see if its hash existed or not and represent it only in the case that it existed. While this approach removed the problem of not representing the higher nodes that did not have a perfect hashing match, this approach meant that basically everything that was not the bottom nodes was represented. As a consequence, the sketches became far too detailed, giving far more information than what was intended. To combat this, we added a "minimum height" parameter while traversing the tree. This minimum height was meant to ensure that we would not just stop at the terminal nodes, but rather a given number of parent nodes before them. With this, we could try to fine tune the height at which it produced the best results. The main problem with this was that the height of a given node is not consistent.

Let us say we set `Minimum Height = 1`. Using Listing 4.21 once again as an example, this minimum height would allow us to match anything that had a height greater than 1. The biggest difference between the AST in Listing 4.21 and 4.22 is that the student submission utilizes a `for` loop, whereas the LLM fix uses a group of `if/else` statements. This means that unless we set the minimum height to the height of the `for` loop, we are unable to sketch out the loop. It also causes everything that is not the for loop to be sketched out, in the case that we do not have an exact hash match. This is because, due to the way that we calculate the height for any given node, the height of the parent node will always be 1 above the height of its "tallest" child. In this case, the `For` node is the "tallest" child of the `Compound` node of line 7,

with them having a height of 5 and 6, respectively. As such, every other child of the `Compound` node will be sketched out completely, unless the hash of that node finds a match.

However, this is not the main issue with utilizing this height based limiter to try to curtail the problem of giving too much information. Let us now imagine that instead of the `scanf()` function call that appears in line 32, was instead replaced by a `printf("Hello World!")`. This new node would be 1 height shorter, than the one that we currently have, meaning that the `for` node would have a height of 4, instead of the 5. Therefore, the optimal height to sketch out the program, would not be 5, but 4.

The heights from the portions of code we want to abstract away are not known to us, unless we manually try to figure out which parts of the code we want to remove. As such, there is no way of knowing for certain what minimum height will provide the best results, since it fluctuates from code to code. Given that this occurs, the generated sketch could not be consistent in the way in which it hid certain portions of code and showed others. After tinkering with the minimum height to see whether or not we could find a point that was somewhat consistent, we ended up moving away from this approach, as anything above 1 would start hiding things we did not intend to be hidden, while retaining certain portions of code that should not be retained.

4.4.3.C Abstract Syntax Tree Node Coordinate and Diff Based Approach

Our final prototype, and the one that we managed to hone to a point of satisfaction, was based around utilizing a `diff` file to identify the changes between the student submission and the LLM fix and save the lines that those changes occurred at. This information would then be used during the traversal of the AST of the fixed submission to better pinpoint the nodes at which the faults appeared in.

An overview of this process can be seen in Figure 4.3. As we can see from the figure, we first generate the `diff` file. Following this, we begin the sketching process itself, by storing the nodes of the student submission for easier access, during the matching process. At the same time, we parse the `diff` file to store all the lines that contain changes between the student submission and the LLM fix. Next, we traverse the corrected AST and check if the node we are visiting is located in a line we identified as faulty. In the case that the node is not in a faulty line or we find a correct matching node in the corresponding line of the student submission, we write the node. However, if we are unable to find a fitting match, we abstract the node away. If we have yet to reach the final node, we continue the traversal, otherwise we return the final sketch.

As alluded to previously, the first step in this process is taking the previously normalized codes (both the student submission and the LLM fix) and generating a `diff` file of them[5]. After generating the `diff` file, we begin the sketch generation process itself.

Next, we traverse the student submission AST in order to store the nodes in a python dict, where its

[5]The command used to generate this file can be found in Section 4.4.2

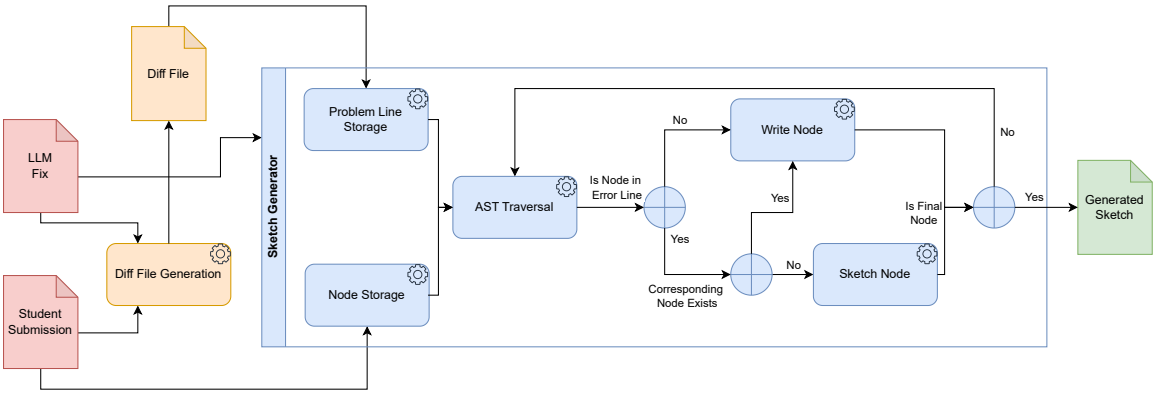

Figure 4.3: Abstract Syntax Tree Node Coordinate and Diff Based Process

key is the line in which the node appears in and the value stored is a list of the nodes that appear in said line. An important detail when it comes to the organization of the AST, is that, due to the recursive nature of its traversal, we can guarantee that the first node we enter in a given line will either be at the same "depth" or a higher "depth" than every other node in this line. This can be seen in Listings 4.21 and 4.22 where, when entering a new node, the first child is always either at the same level or higher than any other node below its parent.

Next, we parse the generated `diff` file, storing the lines in which changes occurred. There are three possible change types, with each one getting treated as follows:

1. **Addition (a)** - When an addition is found, we store the line at which the addition should be inserted in the LLM fix tree and line it occurs at in the student submission tree.

2. **Deletion (d)** - When a deletion is found, we store the line at which this deletion occurred and we "connect" it to a "non-correspondent" node. This means that the line we are storing is guaranteed to not have a matching line/node in the student submission.

3. **Change (c)** - When a change is found, we match each of the lines to each other. Listing 4.27 shows a `diff` file for a student submission and its fixed counterpart. As we can see, there are multiple lines which have a corresponding or matching line in the student's submission. We parse this information with the premise that the LLM will try to alter as little as possible from the submission. As such, we assume that, on average, the positioning of the lines that are altered will be retained. This case serves as a perfect example of this, since each of the lines has a perfect match with the same order. Given this fact, we match the changes to the line that is in the same "place" as it. In this case, line 2 (7 on the fixed program) will be matched with line 6 (7 on the Student Submission), line 3 will be matched with line 7 and line 4 with line 8. In the case that the solution has more lines than the initial submission, we match each line that was altered one by one until we

Listing 4.27: Example Diff File

```
7,9c7,9
<    scanf("%d", &i);
<    scanf("%d", &z);
<    if (n >= i && n >= z)
---
>    scanf("%d\n", &i);
>    scanf("%d\n", &z);
>    if (n > i && n > z)
11c11
<    else     if (i >= z)
---
>    if (i > z)
```

Listing 4.28: Base CGenerator Diff File

```
7,9c7,9
<    scanf("%d", &i);
<    scanf("%d", &z);
<    if (n >= i && n >= z)
---
>    scanf("%d\n", &i);
>    scanf("%d\n", &z);
>    if (n > i && n > z)
11,12c11
<    else
<      if (i >= z)
---
>    if (i > z)
```

reach the end of the lines in the submission. The lines left with no match are treated as not having a correspondent and will be deemed as wrong when abstracting the code. This style of matching is also why we had to alter the *Normalizer's* behavior when it came to the `if` node. Listing 4.28 showcases why this change was necessary, as when an `else` was added to the code, it would add a new line, where previously there was none. This would make the `else` line, which is not represented as node within the AST structure we are using, match the `if` node, causing matching problems. With this in mind, we removed the new line character that was added upon normalizing these nodes, allowing the matching to occur correctly, despite the addition of the `else` to the code.

Upon parsing the `diff` file and storing the different lines where errors occurred, we start traversing the solution tree. At each node we verify two different things: first, we check whether or not we need to add a portion of code that the student had in his code; second, whether or not the line coordinate of the node we are currently visiting matches the line number in the list of changes that occurred.

For the first point, whenever we visit a node, we check if there is any addition, that is yet to be added, that needs to be made before the coordinate of this node. A common mistake that is made by students is adding an unnecessary `printf()` to their submission. Let us take Listing 4.14 as a baseline and imagine that between line 6 and 7, a `printf("Write 2 numbers:\n");` was written. In this scenario, we would want to warn the student that this section needs to be removed. For this, upon reaching line 7 in Listing 4.15 we add a `/* Remove this Section\n`, before visiting and exploring the `scanf()` node. We also add an ellipsis before finishing the comment to show that a portion of what was removed was omitted.

After checking if there is missing code and dealing with it appropriately, we check to see if the line coordinate of the node we are currently visiting has an associated error or not. If no error is found, we proceed normally to next node. If an error is found, we try to match this node to a possible node in the student's submission. To achieve this, we go to the nodes we had stored during the beginning of this process and search for the line that we matched with during the parsing of the `diff` file. If the line is not

found, we can assume there was no corresponding node and, therefore, we are free to sketch out said node. If the line is found, we need to search for the corresponding node. For this, we cycle through the nodes in that line and see if there is an exact match. If there is one, then we write the node, otherwise, we sketch out the node. This process is then repeated until we reach the final node in the AST.

This approach was by far the most reliable of the ones that were tried, behaving as intended for the vast majority of cases. However, it still had a couple of major drawbacks that we had to overcome:

1. The first major drawback is the way in which the `diff` command presents the differences between the two files. For example, in the case where two lines had been changed and a single line was added between them, the output of the command was not the ideal "line 1 was changed, line 2 was added, line 3 was changed", but rather "line 1, 2, 3 were changed". This would cause errors in matching that we were not able to overcome, as we do not have access to the node when parsing the file and, therefore, cannot try to find the corresponding node in the other file. This problem did not occur frequently enough to justify scrapping this tool, since the only effect it had on the generated sketch was the sketching out of an extremely small number of lines, when it did occur.

2. The second major drawback was the matching of nodes themselves. To try to retain a level of disconnect between each of the "depths" of each node, we only utilized the class name, along with the non-node attributes of said node to check whether or not they were the same. This means that multiple nodes in the same line could be mistaken for one another, if their class and attributes were the same. This was surprisingly common in the `BinaryOp` (Binary Operation) nodes, since the only attribute they have which is not a node is their operator. As such, it would often be the case that in non-simple binary operations, there would be a matching symbol, despite it not being the correct match. This caused several issues, chief among them being that far more was being shown and corrected in the conditions of `if` nodes than what was intended. This is mainly due to the fact that the conditions of `if` nodes usually have multiple checks, rather than a single operation. To showcase why this is problematic, the expression `if (a == b || a >= c && b < c)` contains five different binary operations. If a student were to submit a faulty program where this line is instead written as `if (a >= b && a < c || b == c)`, no portion of the code would be abstracted, since there are suitable matching nodes, given that all of the operations used in the fixed code appear on the student's submission. To tackle this problem with the `BinaryOp` nodes, we took advantage of the fact that the storage of nodes happens in order of their precedence. This fact allowed us to guarantee that upon entering the first `BinaryOp` node, its intended match would be the first `BinaryOp` we find in the corresponding line of the student submission. With this in mind, we completely overhauled the way we handled the `BinaryOp` nodes to be similar to our first prototype in Section 4.4.3.A, where we would traverse each of the nodes at the same time recursively. Since we knew beforehand that the nodes are intended to be the match, we can traverse them at

the same time without wasting time searching for a match like we were forced to do before. This way, we could compare if either side was correct and keep the correct portions while sketching out the portions that were incorrect. This change significantly improved the consistency with which the tool performed at the level that was desired of it.

The reason why we did not apply this same solution to certain other nodes such as the `FuncCall` (Function Call) node, despite them having similar issues, had mostly to do with the fact that this approach would remove a lot of the information that we might want to keep. Using Listing 4.19 and 4.20 as an example, the `scanf()` function call in line 8 and 9 respectively has different arguments. As we can see in Listing 4.21 the `ExprList` node for its `scanf()` function in line 16 has a size of 2, having only the `Constant` and `UnaryOp` node within it. In contrast, the `ExprList` node that we find in Listing 4.22 (line 22) has a size of 4, having 2 extra `UnaryOp` nodes. In this case, since both the constant and the `UnaryOp` nodes are all different from one another, this approach would have worked. However, let us imagine that instead of the `ID` node in line 19 of the students submission being called "max", it was instead called "b". If this were the case, we would want to keep the "b" in the function call arguments, abstracting away only the unnecessary "a" and "c" `ID` nodes and the wrong `Constant` node. In this new scenario, this approach would completely delete every argument, since the `ExprList` node did not match. Considering this, we deemed the extra precision to not be worth it, discarding this option for other nodes.

Another issue of our approach is the way in which we dealt with the change in the variable names. Whenever the LLM fix would change a variable name, the tool would not be able to recognize it and, as such, we would sketch out the instances where that variable appeared. This is mainly due to the fact that we utilize the attributes of the node to compare and see if they match to one another or not, given that if the name is changed the equality does not hold and, therefore, we do not have a match. To combat this somewhat, we made it so that, when given the "Below" help setting (explained in Section 4.4.3.D) a declaration would always show the name of the variable being declared. Even though this does not completely fix the problem that we have, giving the name of the declared variables appeared to be enough for the rest of the sketch to be understandable. A real example of this can be seen in Listings 4.29 through 4.31. Where we have a real student submission, its LLM fix and the resulting abstracted code. The error with the program lies in line number 26, where the `printf()` is missing a new line character after printing the values. We can see that the fix managed to correct the output. However, the `x` variable name was changed to `temp` during the fixing process. Listing 4.31 shows the resulting sketch, where every instance of the `x`/`temp` variable was replaced by an `/* ID */` comment, with the sole exception of the initial declaration in line 6. It also correctly changed the argument in the `printf()` function call in line 26. While this is not the ideal result, we can see that the comments clearly map out the exact location of the `x` variable, so, upon investigating the generated sketch, we would be able to assume that those comments refer to the newly name changed variable and not to some other error.

Listing 4.29: Original Student Submission

```c
int main()
{
  int a;
  int b;
  int c;
  int x;
  scanf("%d%d%d", &a, &b, &c);
  if (a > b)
  {
    x = a;
    a = b;
    b = x;
  }
  if (a > c)
  {
    x = a;
    a = c;
    c = x;
  }
  if (b > c)
  {
    x = b;
    b = c;
    c = x;
  }
  printf("%d %d %d", a, b, c);
  return 0;
}
```

Listing 4.30: LLM Fix

```c
int main()
{
  int a;
  int b;
  int c;
  int temp;
  scanf("%d%d%d", &a, &b, &c);
  if (a > b)
  {
    temp = a;
    a = b;
    b = temp;
  }
  if (a > c)
  {
    temp = a;
    a = c;
    c = temp;
  }
  if (b > c)
  {
    temp = b;
    b = c;
    c = temp;
  }
  printf("%d %d %d\n", a, b, c);
  return 0;
}
```

The final issue which was alluded to during the discussion on how we utilized and parsed the `diff` file, was the way in which we had to deal with the `else` statements. Within the tool we are using, the `else` statements are not classified as a distinct node type, but rather a set of expressions (which may or may not contain another `if` in the event of an `else if` statement) that happen in case the corresponding `if` node is `False`. While this difference may seem small, it clashes with the way that we handle every other node. This is because the `else` node type does not exist, so we cannot compare the nodes properly. This is especially problematic when taking into account the fact that if the `else` line is identified as being the faulty line, we have no way of reaching it, since it is not a node in itself. As such, the decision to not add a new line character after the `else` when normalizing the program or writing the sketch, served not only to try to minimize false matches, but it also serves to effectively guarantee that a node exists in an `else` line. The removal of the new line character, managed to reduce these problems significantly. However, the fact that the `else` is not a node still causes a major issue which unfortunately we could not solve, which is the fact that, as a result of not being a node, every time the solution has an `else` statement, it will always appear in the sketch, regardless of whether or not that statement appeared in the students submission. Despite this not being ideal, this occurrence was deemed to be small enough to overlook, especially given the fact that the idea of this feedback tool is to give the structure of the program to the student. Considering the fact that an additional `else` can completely modify the entire

Listing 4.31: Generated Sketch

```
1  int main()
2  {
3    int a;
4    int b;
5    int c;
6    int temp;
7    scanf("%d%d%d", &a, &b, &c);
8    if (a > b)
9    {
10     /* ID */ = a;
11     a = b;
12     b = /* ID */;
13   }
14   if (a > c)
15   {
16     /* ID */ = a;
17     a = c;
18     c = /* ID */;
19   }
20   if (b > c)
21   {
22     /* ID */ = b;
23     b = c;
24     c = /* ID */;
25   }
26   printf(/* Constant */, a, b, c);
27   return 0;
28 }
```

program, we elected to keep it as is, since the added structure it shows was considered more important than trying to remove the extra piece of information.

4.4.3.D Level of Feedback

A major setting that we wanted to implement was the idea of different levels of feedback to the student. Essentially, we wanted to give the ability to either add or remove information from the sketch. This would present itself in the granularity of the feedback that was to be given to the student, with it being split into two different options: Help Depth "Below" or "Current Node"[6].

To create this functionality, we leverage the fact that every node in an AST is either a terminal node, or a parent node. This is relevant, since the information stored within each node can pertain to their children (for example, a `BinaryOp` (Binary Operation) node can have an `ID` node to each side of the operation). This fact means that we can always either choose to remain at the level of the current node or continue to its children.

Functionally, this setting allows us to decide whether or not to keep giving information in the case that a parent node is incorrect, with the "Current Node" setting limiting itself to the node we are currently on and the "Below" setting allowing us to continue the search, in spite of already finding a mistake in

[6]Below is called using "-hd 1", and Current Node using "-hd 2".

Listing 4.32: Generated Sketch with “Below” Help Depth

```
1  int main()
2  {
3    int n;
4    int i;
5    int z;
6    scanf("%d", &n);
7    scanf(/* Constant */, &i);
8    scanf(/* Constant */, &z);
9    if (n /* Op */ i && n /* Op */ z)
10     printf("%d\n", n);
11   else    if (i /* Op */ z)
12     printf("%d\n", i);
13   else    printf("%d\n", z);
14   return 0;
15 }
```

Listing 4.33: Generated Sketch with “Current Node” Help Depth

```
1  int main()
2  {
3    int n;
4    int i;
5    int z;
6    scanf("%d", &n);
7    scanf(/* Constant */, &i);
8    scanf(/* Constant */, &z);
9    if (/* BinaryOp */ && /* BinaryOp */)
10     printf("%d\n", n);
11   else    if (/* BinaryOp */)
12     printf("%d\n", i);
13   else    printf("%d\n", z);
14   return 0;
15 }
```

the program. Using Listing 4.14 as an example, the `BinaryOp` node in line 9 would not find a match, as the operator in the higher tier of precedence (the == sign) does not have an equal node to match with. As such, if we were using the “Current Node” setting, the entire condition would be omitted. However, using the “Below” level, we could “force” a match between the == and the `!=` nodes, and then continue our search for more matches. In this second case, the only thing that would be omitted would be the operator `!=`.

This setting will allow for a more tailored approach, which can either be incorporated as an automated process, by changing the “help depth” level depending on the number of attempts or by being manually set for any given lab or exercise. We believe that this feature is one of the main highlights of this approach, as the AST based approach allowed us to implement this in a way that avoided complicated setups that might be unreliable.

To complete the example from Listing 4.1, we take the normalized codes, along with their diff file, that we had generated (shown in Listings 4.16 through 4.18) and create the final generated sketch. For this step we traverse the tree of the solution to verify whether or not a matching node exists in the other tree. Listings 4.32 and 4.33 demonstrate the resulting sketches, depending on the given “Help Depth”. In this case, we can see that the generated sketch is almost identical. This is because the structure of the “taller” nodes within the tree is extremely similar, with the only difference being found in line 9 and 11. The difference in operators means that with the “Below” mode we represent the children of the `BinaryOp` node, since they are the same, while with the “Current Node” mode we abstract them away, instead.

5

Results

Contents

This Chapter will focus on showcasing and analyzing the results of the different LLMs, when it came to the program repair module of `FeedFix`, as well as the performance of the feedback tool module. We will also discuss which of the prompt configurations and LLM pairing we believe to be the best.

5.1 C-Pack-IPAs Dataset

To conduct our analysis, we used the `C-Pack-IPAs` dataset [4], which contains student submissions to programming exercises from laboratory classes. The `C-Pack-IPAs` dataset comprises 1,431 incorrect submissions across 25 different `C` programming language IPAs, collected from three distinct laboratory classes over three academic years. The exercises cover a diverse range of topics, including integer and input–output operations, loops and character manipulation, and vector processing.

Labs	#IPAs	#Correct Submissions	#Wrong Submissions
Lab 02	10	789	486
Lab 03	7	363	699
Lab 04	8	468	246
Total	**25**	**1620**	**1431**

Table 5.1: Dataset Breakdown

Abreviation	Meaning
De	IPA Description
Sk	Sketch Approach - Uses Fault Localization
TS	Test Suite
CS	Counterexample
EX	Example on how to handle Fault Localization - Uses FIXME when not a Sketch
FB	If FL fails, uses generated tips to steer LLM
If	Infilling Implementation - Uses Fault Localization

Table 5.2: Configuration Abbreviation Caption

This dataset has been successfully employed in several prior works across various program analysis tasks [9, 17, 72, 73], demonstrating its reliability and versatility. For this dataset, we specifically selected submissions that compiled successfully, since compilation errors are already identified by the compiler and therefore do not require additional feedback.

A simplified overview of the dataset is presented in Table 5.1.

5.2 Analysis of Performance Results

Table 5.3 showcases the results of program repair that each of the different LLMs reached under the different configurations utilized (refer to Subsection 4.3 for an overview of each configuration), whereas Table 5.5 showcases the tree edit distances between the ASTs of the original buggy program and the fixed program [74]. As mentioned in Section 4.3, configurations **De, De-TS-CE** and **Sk_De-TE-CS** had already been tested before in the AAAI paper by Pedro Orvalho et al. [9]. The remaining seven configurations were newly built for this thesis, not having been tested before.

5.2.1 Analysis of Different Prompt Configurations

We will start by analyzing the different prompt configurations and their impact on the performance of the different evaluated LLMs. This metric is the most important and, as such, will carry the majority of the weight for any decisions taken moving forward. Table 5.2 shows what the abbreviations in each of the different configurations stand for and what they add to the LLM prompt.

The first thing to take notice of is the fact that *Qwen Coder* [71] had, by far, the best performance

	Entire Benchmark					
	Configurations					
LLMs	**De**	**De-TS-CE**	**De-TS-CE-EX**	**De-TS-CE-EX-FB**	**De-TS-CE-FB**	**Portfolio (All Configurations)**
CodeGemma	553 (38.6%)	571 (39.9%)	571 (39.9%)	561 (39.2%)	562 (39.3%)	878 (61.4%)
CodeLlama	509 (35.6%)	548 (38.3%)	602 (42.1%)	620 (43.3%)	617 (43.1%)	869 (60.7%)
Qwen	770 (53.8%)	823 (57.5%)	780 (54.5%)	747 (52.2%)	763 (53.3%)	1052 (73.5%)
Portfolio (All LLMs)	856 (59.8%)	912 (63.7%)	950 (66.4%)	920 (64.3%)	925 (64.6%)	1079 (75.4%)
LLMs	**If_De-TS-CE**	**Sk_De-TS-CE**	**Sk_De-TS-CE-EX**	**Sk_De-TS-CE-EX-FB**	**Sk_De-TS-CE-FB**	**Portfolio (All Configurations)**
CodeGemma	549 (38.4%)	**618 (43.2%)**	597 (41.7%)	605 (42.3%)	580 (40.5%)	878 (61.4%)
CodeLlama	**713 (49.8%)**	610 (42.6%)	616 (43.0%)	621 (43.4%)	605 (42.3%)	869 (60.7%)
Qwen	**833 (58.2%)**	814 (56.9%)	814 (56.9%)	790 (55.2%)	804 (56.2%)	1052 (73.5%)
Portfolio (All LLMs)	**959 (67.0%)**	944 (66.0%)	957 (66.9%)	919 (64.2%)	927 (64.8%)	1079 (75.4%)

Table 5.3: Results of Different Configurations.

out of the chosen LLMs as it outperformed in every single category. This even extends to the **De** configuration (which managed to resolve 53.8% of the wrong submissions), that only utilized the IPA descriptions of the problems, which out performed both *CodeGemma* [69] and *CodeLlama* [70] in all of the different configurations (with them being 43.2% and 49.8% respectively in their best performing configurations).

Another interesting pattern that emerges is that the sketch and infilling based implementations tended to outperform their non-sketch counter-parts. This could be due to a variety circumstances, especially when compared to the `/* FIXME */` approaches (configurations **De-TS-CE-EX, De-TS-CE-EX-FB** and **De-TS-CE-FB**), but a significant one that emerged rather often is how the different configurations handle extra `printf();` lines that students often add by mistake.

This mistake was repeated throughout the various different exercises, where the students would add prints meant to serve as guidance to the user, which would then muddle the output, causing them to fail the test cases. Let us use code that calculates the Fibonacci sequence up to a given number `N`, as an example. In Listing 5.1, we can see this type of mistake on display. The additional `printf("Provide a N:");`, meant to give guidance to the user, causes every test to fail, as the program will always print the phrase, before printing out the correct result. This type of mistake, while extremely easy to fix when given context since the program itself is correct, gives the LLMs significant issues, as they are unable to understand what is incorrect with the code. Considering the fact that, unlike the `/* FIXME */` implementations, which add the comment to the end of the line, the *Sketch* and *Infilling* based implementation remove the line completely (as seen in Listing 5.2), it is no surprise that these implementations manage to handle these mistakes far better than the other configurations. Given that the faulty part is removed, the LLM would understand that the program is correct and simply return back the program, without the added `Hole` or their respective *Infilling* identifier.

This case does highlight a big flaw within the different LLMs: they are unable to comprehend what the formatting of the answers should be. While the tools are mostly able to correctly identify and ac-

Listing 5.1: Buggy Submission

```c
#include <stdio.h>

int main(){
    int n, fibonacci;
    int prev1 = 1;
    int prev2 = 0;

    printf("Provide a N:");
    scanf("%d", &n);

    if (n <= 1) fibonacci = n;

    for (int i = 2; i <= n; i++){
        fibonacci = prev1 + prev2;
        prev2 = prev1;
        prev1 = fibonacci;
    }

    printf("%d", fibonacci);

    return 0;
}
```

Listing 5.2: Sketch Example

```c
#include <stdio.h>

int main(){
    int n, fibonacci;
    int prev1 = 1;
    int prev2 = 0;

    /* HOLE 1 */
    scanf("%d", &n);

    if (n <= 1) fibonacci = n;

    for (int i = 2; i <= n; i++){
        fibonacci = prev1 + prev2;
        prev2 = prev1;
        prev1 = fibonacci;
    }

    printf("%d", fibonacci);

    return 0;
}
```

knowledge the test cases and inputs and outputs that are provided to the tools in their prompts, they have difficulty understanding the desired formatting of the answers. In the case of the aforementioned Fibonacci example, given the input `3`, the desired output is `6`. However, to the LLM, an output of `6\n` would still be a correct output, even though it would clash with the intended formatting of the answer and result in the test cases being wrong. This is a severe limiting factor, as these mistakes should be easily identifiable to the student, but they wind up being too precise for the different LLMs to be able to detect and correct.

This issue is something that the **FB** configurations ideally would have addressed and mitigated, as they would give more information to the LLMs specifying issues in formatting of answers, hinting at the fact that the mistakes lie somewhere within the `printf()` statements in the code. However, as we can see from the results, the **FB** configurations, both the ones using `/* FIXME */` and `/* HOLE N */`, were almost always outperformed by those that did not utilize this additional set of directions. While it is impossible to know exactly why the performance was poorer when using the **FB** configurations, we can see that the extra information did not positively impact the results.

In contrast to the **FB** configurations, we can observe that both the **TS-CE** and **TS-CE-EX** configurations managed to, in a majority of cases, vastly outperform the one that solely relied on the IPA descriptions (**De**). It is interesting to note that the added examples, intended to serve as guidance to the LLMs on how to "attack" the solving process, often had a negative impact on the performance of these tools, with the exception of *CodeLlama* [70], whose performance increased. This could be due

Qwen Repair per Lab					
	Configurations				
Labs	**If_De-TS-CE**	**Sk_De-TS-CE**	**Sk_De-TS-CE-EX**	**Sk_De-TS-CE-EX-FB**	**Sk_De-TS-CE-FB**
lab02	**388 (79.8%)**	374 (77.0%)	376 (77.4%)	365 (75.1%)	377 (77.6%)
lab03	**299 (42.8%)**	290 (41.5%)	296 (42.3%)	293 (41.9%)	291 (41.6%)
lab04	146 (59.3%)	**150 (61.0%)**	142 (57.7%)	132 (53.7%)	136 (55.3%)

Table 5.4: Repair Rate of QwenCoder on Most Promising Configurations by Lab

to a variety of different factors, but the most likely one is that the provided examples steered the LLM in the wrong direction, confusing it and having a negative impact on its solving abilities. Nevertheless, these configurations have some promising aspects, as will be discussed in Subsection 5.2.2, since they appear to follow the original program more closely than the other configurations.

Unsurprisingly, the highest performing configuration for both *CodeLlama* [70] and *QwenCoder* [71] was the Infilling based implementation, which showed a marked increase in performance, when compared to their other configurations. This is especially noticeable with *CodeLlama* [70] as it showed a 15% increase in the amount of corrected programs, when compared to the second highest performing configuration (**Sk_De-TS-CE-EX-FB**). This is to be expected, given that these models were made with FIM requests in mind and so, their performances being above the other options is in congruence with what was to be expected from them. The more interesting result of these *Infilling* based implementations is related to the *CodeGemma* [69] results, which are comparatively much worse than those that were achieved by the other two LLMs, achieving a solve rate inferior to even the **De** configuration. This is extremely unexpected, as the FIM capabilities are built-in to the model, and, as such, it would be expected for it to perform at a higher level when making use of them.

Overall, *QwenCoder* [71] was, by far, the model with the best performance, managing to correct 73.5% of the entire set of problems, spanning all its configurations. An interesting detail to point out is the fact that while the individual configurations of *CodeGemma* [69] had a lower efficiency when fixing the different programs than *CodeLlama* [70], *CodeGemma* managed to fix a broader range of problems with its aggregate solve ratio (total Portfolio) being 61.4%, whereas *CodeLlama* sits at a lower 60.7%. This could be indicative of some predisposition of *CodeGemma*, as it appears to be more random with the problems it can solve, when compared to the other two models. If so, the added variance could explain the discrepancy between the range of coding problems it solved on the aggregate and the limited amount of problems it managed to solve with each individual configuration. This factor might make it undesirable, as the inconsistency in its output might lead to a more frustrating experience, since the model ends up being unreliable.

Considering how *QwenCoder's* [71] performance was significantly better than that of the other models, we wanted to examine its most promising configurations, and how they differ between labs. As we can see in Table 5.4, the *Infilling* configuration outperforms the other options in lab02 and lab03,

whereas the **Sk_De-TS-CE** implementation slightly edges it out in lab04, managing to fix four more programs than its Infilling counterpart. While the best configuration when it comes the repair rate of the tool is clearly the **If_De-TS-CE**, having achieved the best results for two out of the three labs and having the highest repair rate overall, the discussion for the number two is a lot more interesting. When all labs are taken into account, both **Sk_De-TS-CE** and **Sk_De-TS-CE-EX** managed to solve the exact number of programs (see Table 5.3), with them both being able to solve 56.9% of the dataset. However, as is shown in Table 5.4, the **Sk_De-TS-CE-EX** configuration performed better in the initial labs, while taking a performance hit on lab04. This difference could be an interesting push-and-pull, as if we value the initial labs more, the **Sk_De-TS-CE-EX** configuration might seem more appealing, whereas if we believe lab04 to be more valuable to the learning process, the **Sk_De-TS-CE** configuration might become more interesting.

5.2.2 Analysis of Abstract Syntax Trees Results

Keeping the efficiency of the different LLMs in mind, we now move to analyze the similarity between the ASTs of the original buggy code and the final solved code. These values can be seen in Table 5.5, where the highlighted values are the smallest AST edit [74] distance between the original code and the corrected code returned by the model. It is extremely important to note that these values need to be viewed through the lens that they only make sense in the context of the percentage of solved problems and that taking them at face value is extremely dangerous.

As an example, the lowest distance in the entire table is in *CodeLlama's* [70] **De** implementation, which sits at *2.1*. This does not mean that the implementation was better as a whole, but rather that the generated AST for the fixed program was closer to the original. This is an important distinction, given the fact that the **De** configuration was the worst performing option for *CodeLlama*. The reason behind this discrepancy is that the only ASTs that are taken into account are the ones from scenarios where the model managed to fix the buggy portion of the code. This means that the harder the solve and the more modifications are necessary, the more the AST will diverge from the original. As such, configurations with a lower solve ratio will tend towards having more similar ASTs as the more complex solves end up falling to the wayside, leading to a skew in the results towards lower AST distances.

Despite these pitfalls, this metric is still useful when evaluating similarly performing configurations, as it provides insight as to how the different configurations might alter the way in which the models tackle the problem.

The first thing to take notice of, is that the Infilling implementations are near the upper echelon of distance for every LLM. This makes sense intuitively, considering the way the buggy code is provided to the models when using this approach. As explained in the Subsection 4.3, the Infilling implementations removed the chunk of code from the first fault, down to the last one, meaning any line that might

have been correct between two different faulty lines was completely removed. As an example, in Listing 5.3 we can see a program meant to calculate the Fibonacci sequence again, but in this case there are two mistakes. The first being the initial `printf("Provide a N:");` in line 8, and the second being the `prev2 = fibonacci;` in the assignment in line 15. In Listing 5.4 it is shown how the Infilling implementation would present the buggy code to the model[1]. In this example, we can see that every line between line 9 and 14, which are correct, are also deleted when preparing the code to be sent to the LLM. This removal is necessary for the Infilling process, as the models used do not support multiple Infilling tokens. With this in mind, it is expected that this approach would trend towards a larger distance between the ASTs, as portions of the code that are correct are completely removed. In the example of Listing 5.4, the FIM request could easily produce a `while` loop, instead of the `for` loop that was used. It could also alter the way in which the original code handled the case when `n <= 1`. These differences, while still able to fix the program, would generate a bigger disparity between the resulting ASTs, as the resulting code would be further apart from the original submission.

This metric presents an interesting question for *QwenCoder* [71] specifically, as its repair capabilities between **If_De-TS-CE, Sk_De-TS-CE, Sk_De-TS-CE-EX** and **Sk_De-TS-CE-FB** are similar with less than 30 problems being the difference between the highest and lowest performing configurations (2% of the entire set). Utilizing this metric we can also rule out the **De-TS-CE** configuration for *QwenCoder*, as the average distance between the ASTs is far greater than any other alternative, along with its performance being worse than the **If_De-TS-CE** implementation. It is almost impossible to know the reason behind the fluctuations from just the numbers, given that the Infilling implementation managed to solve a broader range of problems that might have required bigger changes to the ASTs as a whole. Considering this, and similarly to what we did when analyzing the repair rate of the different LLMs, we chose to isolate *QwenCoder's* [71] best performing configurations and see how they perform in relation to the AST distance by the different labs. These values can be seen in Table 5.6, where we highlighted the biggest average AST edit distance. Note that, as the programs increase in difficulty, the more the AST has to be altered for a successful repair, which causes the average tree edit distance to increase with each lab. However, it is also clear that the Infilling implementation alters the AST of the programs, significantly more, especially in the early labs. Here, we can also see that both **Sk_De-TS-CE** and **Sk_De-TS-CE-EX** configurations share a similar amount of average edits. Despite this, we can see that the differences between the two configurations in lab02 and lab03 are smaller than that of lab04. This higher average could be a result of the **Sk_De-TS-CE** configuration being able to repair a larger number of programs, so while **Sk_De-TS-CE-EX** appears to be slightly better, there can be no conclusive answer.

We will delve deeper into which of these configurations ends up being the most desirable when we analyze the number of attempts and time required by each configuration (Section 5.2.3), along with the

[1] Note that the "INFILL" in line 8 is a placeholder and the token for the specific model is used instead.

Entire Benchmark					
Metric: **avg(Tree Edit Distance between fixed and original incorrect program)**					
Configurations					
LLMs	**De**	**De-TS-CE**	**De-TS-CE-EX**	**De-TS-CE-EX-FB**	**De-TS-CE-FB**
CodeGemma	4.9	**4.5**	5.6	5.5	4.7
CodeLlama	**2.1**	2.7	4.0	4.2	4.8
Qwen	15.3	17.1	15.0	13.4	14.5
LLMs	**If_De-TS-CE**	**Sk_De-TS-CE**	**Sk_De-TS-CE-EX**	**Sk_De-TS-CE-EX-FB**	**Sk_De-TS-CE-FB**
CodeGemma	6.1	7.2	7.3	6.7	6.5
CodeLlama	6.7	5.1	4.3	4.7	4.4
Qwen	15.9	14.9	14.8	14.4	**13.3**

Table 5.5: Distance in ASTs from the original program to the corrected version

Listing 5.3: Buggy Submission

```c
#include <stdio.h>

int main(){
    int n, fibonacci;
    int prev1 = 1;
    int prev2 = 0;

    printf("Provide a N:");
    scanf("%d", &n);

    if (n <= 1) fibonacci = n;

    for (int i = 2; i <= n; i++){
        fibonacci = prev1 + prev2;
        prev2 = fibonacci;
        prev1 = fibonacci;
    }

    printf("%d", fibonacci);

    return 0;
}
```

Listing 5.4: Infilling Code Prompt

```c
#include <stdio.h>

int main(){
    int n, fibonacci;
    int prev1 = 1;
    int prev2 = 0;

    <|INFILL|>
        prev1 = fibonacci;
    }

    printf("%d", fibonacci);

    return 0;
}
```

resulting code generated by `FeedFix` (Section 5.3).

5.2.3 Analysis of Number of Tries on Successful Repairs

Similarly to the prior metric of the AST edit distance between the different trees, analyzing the time or the number of tries required to fix a given problem needs to be contextualized by the fact that whenever a model configuration is unable to correct the buggy code, its number of tries is not taken into account. As such, just like with the metric revolving around the ASTs, the numbers cannot be taken at face value.

Table 5.7 displays the average number of attempts that were required to successfully repair the program, where the bolded values are the lowest average number of tries required for each LLM. The values that immediately standout are those of *CodeGemma* [69], with 1.0s in every configuration except

Qwen's AST Distance from Original to Fixed by Lab				
	Configurations			
Labs	**If_De-TS-CE**	**Sk_De-TS-CE**	**Sk_De-TS-CE-EX**	**Sk_De-TS-CE-FB**
lab02	**12.5**	10.5	10.6	9.8
lab03	**19.1**	17.0	18.0	15.4
lab04	18.5	**22.0**	19.5	18.7

Table 5.6: QwenCoder AST Distance from Original to Fixed Tree on Best Performing Configurations

Entire Benchmark					
Metric: **avg(Number of chat messages of Successful Repairs)**					
	Configurations				
LLMs	**De**	**De-TS-CE**	**De-TS-CE-EX**	**De-TS-CE-EX-FB**	**De-TS-CE-FB**
CodeGemma	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**
CodeLlama	1.3	**1.2**	1.3	1.4	1.5
Qwen	**1.1**	**1.1**	**1.1**	**1.1**	**1.1**
LLMs	**If_De-TS-CE**	**Sk_De-TS-CE**	**Sk_De-TS-CE-EX**	**Sk_De-TS-CE-EX-FB**	**Sk_De-TS-CE-FB**
CodeGemma	**1.0**	**1.0**	**1.0**	1.1	**1.0**
CodeLlama	1.3	**1.2**	**1.2**	**1.2**	**1.2**
Qwen	**1.1**	1.2	**1.1**	1.2	1.2

Table 5.7: Average Number of Tries when Successfully Repairing

for the **SK_De-TX-CE-EX-FB** option. While this might seem like an appealing thing, this betrays the lack of adaptation that the model suffers from. These values showcase the fact that *CodeGemma* either gets it right the first time, or it will flounder around until the timeout eventually stops it. This means that a lot of time is being wasted, since the back-and-forth that was intended to occur to improve the capabilities of the model has a negligible effect on the outcome of the code repair. As such, in this specific case, this metric serves as a condemnation of the model, showcasing its inability to handle cases that it is unable to fix out of the gate.

Unlike *CodeGemma* [69], both *QwenCoder* [71] and *CodeLlama* [70] have much more interesting results. *CodeGemma* shows every configuration at 1.2 attempts or higher, whereas *Qwen Coder* sits at 1.1 tries or higher. These results are significantly more interesting to analyze, as they showcase that the models are indeed having a back-and-forth, with *CodeLlama* reaching a 1.5 in the **De-TS-CE-FB** configuration.

5.3 Analysis of Sketch Generator

Unlike the other metrics, the analysis of the sketch is, fundamentally, a subjective task. Not only is the best way to present the abstractions completely subjective, but the process is highly reliant on the LLM being able to closely follow the student submission. This is because a significantly different solution would result in a confusing sketch. An example of this can be clearly seen with the **If_De-TS-CE** configuration, as its removal of a significant portion of the code leads it to create a large amount of differences. Looking back at the example shown in Listing 4.1, we can see how this configuration might

Listing 5.5: Infilling Fixed Program

```
1  int main()
2  {
3    int num1;
4    int num2;
5    int num3;
6    scanf("%d %d %d", &num1, &num2, &num3);
7    if (num1 >= num2 && num1 >= num3)
8    {
9      printf("%d\n", num1);
10   }
11   else   if (num2 >= num1 && num2 >= num3)
12   {
13     printf("%d\n", num2);
14   }
15   else   {
16     printf("%d\n", num3);
17   }
18   return 0;
19 }
```

Listing 5.6: Generated Diff File

```
1  3,17c3,13
2  <    int num1;
3  <    int num2;
4  <    int num3;
5  <    scanf("%d %d %d", &num1, &num2, &num3);
6  <    if (num1 >= num2 && num1 >= num3)
7  <    {
8  <      printf("%d\n", num1);
9  <    }
10 <    else   if (num2 >= num1 && num2 >= num3)
11 <    {
12 <      printf("%d\n", num2);
13 <    }
14 <    else   {
15 <      printf("%d\n", num3);
16 <    }
17 ---
18 >    int n;
19 >    int i;
20 >    int z;
21 >    scanf("%d", &n);
22 >    scanf("%d\n", &i);
23 >    scanf("%d\n", &z);
24 >    if (n > i && n > z)
25 >      printf("%d\n", n);
26 >    if (i > z)
27 >      printf("%d\n", i);
28 >    else   printf("%d\n", z);
```

affect the results. Listing 5.5, shows the normalized fixed program that *QwenCoder* [71] managed to create utilizing this configuration. As we can see, we have several differences, between the complete name change of every variable, to the fact that the multiple `scanf()` statements were condensed down to a single one. This results in a `diff` file whose contents indicate a fault in almost every line, as seen in Listing 5.6. The results of these flaws within the process can be seen in the sketch itself, as shown in Listing 5.7 (see Table 4.1 for a caption for each comment). This sketch, utilizing the “Below” mode, is significantly inferior to the one provided in both Listings 4.32 and 4.33, despite supposedly giving more help than the latter. This is mainly due to the wrong matching occurring because of the major differences detected by the `diff` command. As such, the traversal of the AST abstracted away far more nodes than what was intended.

This occurrence was a core aspect of the **If_De-TS-CE** configuration, mainly due to how the configuration operates as a whole. As such, this configuration ends up falling short of expectations, since the resulting fixes generate significantly worse and more convoluted sketches, which directly clash with the intended purpose of helping the students. Unlike the infilling configuration, the other configurations tended to follow the students submissions much closer, which resulted in better sketches.

The best possible option to properly judge the sketches would have been to roll out a demo of this project to the students and gather their thoughts on the tool and the sketch as a whole. However, due to

Listing 5.7: Generated Feedback Sketch

```
int main()
{
  int num1;
  int num2;
  int num3;
  scanf(/* Constant */, &/* ID */, &/* ID */, &/* ID */);
  if (/* BinaryOp */)
  {
    /* ID */(/* Constant */, /* ID */);
  }
  else    if (/* BinaryOp */ /* Op */ /* BinaryOp */)
  {
    printf("%d\n", /* ID */);
  }
  else  {
    /* ID */(/* Constant */, /* ID */);
  }
  return 0;
}
```

the timeline of the development of the tool, we were unable to test this tool in a real life scenario, as the classes which would make use of `FeedFix` were held before the start of this tool's development.

Given that the best option was not available, a manual "sanity check" was performed on every exercise of the first academic year from the IPA dataset on the programs from the second lab for every configuration. This manual check revealed that the main problem was the matching of the lines, where when the matching failed, the output was significantly worse, as seen in Listing 5.7. Despite this, the overall behavior of the tool was in line with what was intended and generated what we deemed to be suitable sketches that could help the students. As such, we are planning on deploying `FeedFix` in February 2026, to provide feedback to students and properly evaluate `FeedFix's` performance based on its impact on the students.

5.4 Result Discussion and Best Configuration/LLM Pairing

Throughout this Chapter we analyzed the different possible configurations and LLMs for both their performance and similarity to the original code. Of the various possible pairings, we can conclude that the standout performer across the board, when it comes to the different LLMs, is *QwenCoder* [71]. The model was light in the context of large language models, having only 7B parameters, while retaining the best performance by a considerable margin. While this is not shown, due to the way it interacts with the timeouts and how the used time is registered, this model was also the quickest, which is a very big advantage for a tool such as `FeedFix`, as a faster response time means that the student will get their hands on the feedback sooner, reducing idle time while waiting for feedback.

Of the various configurations, we can see that four of them standout as the best overall perform-

ers, those being: **If_De-TS-CE, Sk_De-TS-CE, Sk_De-TS-CE-EX** and **De-TS-CE**. The first three are not surprising, as these configurations provide ample information to the tool, while removing the faulty portions of the code, reducing the bias towards the already written code that might confuse the LLM, as is the case with the `/* FIXME */` approaches. However, it is interesting to note that, while not using a Sketch implementation, the **De-TS-CE** configuration had an impressive repair rate, beating out both **Sk_De-TS-CE** and **Sk_De-TS-CE-EX** configurations.

Of these four configurations, the first we can safely discard is the infilling implementation (**If_De-TS-CE**). Despite having the best repair rate, as discussed in Section 5.2, this implementation suffers from the fact that the formatting of the fixed program can differ wildly from the original submission. This causes major problems when it comes to the sketch generation process, leading to significant portions of code being abstracted away. Given that the final sketch is what the students will have to interact with, the added repair rate seems to be too small to justify such a big drop in the quality of the sketches.

By this same metric and, as mentioned in Section 5.2.2, we can rule out the **De-TS-CE** configuration, as it had an higher average AST edit distance than the infilling implementation.

As such, we are left with the other two configurations, **Sk_De-TS-CE** and **Sk_De-TS-CE-EX**, which both had the exact same repair rate. Between these two, the metrics are too close to be able to definitively say which one is the best. However, the average AST distance appears to be slightly in favor of the **Sk_De-TS-CE-EX** configuration. This makes sense intuitively, as the **EX** tag helps guide the LLM on how to handle the holes left in the code and trying to limit the changes made by the model to said holes. In theory, this would then lead to a more similar code than the non **EX** option. Despite this, no definitive conclusion can be drawn about these configurations, as the generated feedback appears to be extremely similar on manual inspection.

As discussed during Section 5.3, the best way to arrive at a conclusion for this would be to roll out a demo of the project with each of these two configurations and do A/B testing on the satisfaction of the students with the tool's performance. Unfortunately, this was not possible, so we can only speculate as to which of these configurations is best. As it stands, both **Sk_De-TS-CE** and **Sk_De-TS-CE-EX** configurations seem like valid approaches with merit, so either one could serve as a suitable building block for `FeedFix`. To clear up these doubts and to evaluate `FeedFix` objectively, we will deploy `FeedFix` in February 2026 to get the opinions of students on the overall performance of the tool and which configuration they prefer.

6

Conclusions and Future Work

Contents

6.1 Conclusion

Automated Program Repair (*APR*) for Introductory Programming Assignments (*IPAs*) is mainly driven by the large influx of new students every year to programming courses. This increase in students, despite the relative stagnation in the number of professors and teaching assistants, places a heavy burden on faculty members, who are now unable to personally help every student [1]. The lack of support results in a worse learning experience, which can lead to frustration. To address these issues, we propose `FeedFix`, which we believe represents a novel approach that can help tackle this problem by providing individually tailored feedback. By using APR in combination with the Abstract Syntax Tree (*AST*) and text-based sketch generation, we can create detailed sketches of solutions which we believe provides the students far better insights than currently available options. This new tool receives a student's buggy submission, alongside the test suite and the fault localization information provided by `CFaults` [17] and

attempts to correct the program using a Large Language Model (*LLM*) and the CEGIS loop described in Section 3.4.1.A. Upon the termination of this loop, we either provide the student with the fault information from `CFaults'` output if the patch generation was unsuccessful, or we generate a sketch program based on the student's buggy submission and the LLM's fix to return to the student as feedback. Given the capabilities of the APR module of `FeedFix`, this combination allows us to provide students with a personalized sketch in over 50% of cases, based on the submitted buggy code's structure. Consequently, the generated sketch will be far more similar to the student's own code than if a given "example solution" was used in its place. Another advantage of this approach is the fact that we are only obfuscating the sections of code that the students are not privy to, not giving away the answer, despite providing the complete structure of the corrected version of the code. That, combined with the "Help Depth" toggle mentioned in Section 4.4.3.D, which allows the faculty to tailor the tool to each exercise, to remove or add more of the structure of the program, depending on their preferences, is a clear improvement over the current approaches to feedback.

6.2 Future Work

Despite the overall good performance of `FeedFix`, it is important to note parts of the tool that might require some further attention and development in the future.

The first and most glaring is the A/B testing mentioned in Section 5.4, which is necessary to properly evaluate the usefulness of `FeedFix` on providing tailored feedback to students. This is going to be addressed in the future, as we plan on launching this experimental version in the next iteration of *IAED* (class for which this tool was built), in February 2026.

Another point that could be expanded on is the way in which we find the differences between the two programs, post normalization. While the `diff` command is usually good enough for the task at hand, the fact that changes that might be deemed additions/removals are often considered as "changes" instead means that the line matching of the sketch generation process can be wrong more often than we would want. As such, a program tailored to this task that more precisely identifies the text changes acting as a replacement for the `diff` command that is used could provide a noticeable positive impact on the performance of the sketch generator.

Lastly, new and improved models are still being released at an extremely fast pace. These new models might prove to be better than what we used for `FeedFix` and the tool might significantly improve with those new models.

Bibliography

[1] P. Orvalho, M. Janota, and V. Manquinho, "GitSEED: A Git-backed Automated Assessment Tool for Software Engineering and Programming Education," in *Proceedings of the 2024 ACM Virtual Global Computing Education Conference V. 1, SIGCSE Virtual 2024, Virtual Event, NC, USA, December 5-8, 2024*, M. Dorodchi, M. Zhange, and S. Cooper, Eds. ACM, 2024.

[2] P. T. T. Huyen, K. Yasuda, and S. Itoh, "Mining fix patterns with context information for automatic program repair," in *IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2023, Melbourne, Australia, May 16, 2023*. IEEE, 2023, pp. 1–8. [Online]. Available: https://doi.org/10.1109/APR59189.2023.00007

[3] S. Gulwani, I. Radicek, and F. 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, June 18-22, 2018*, J. S. Foster and D. Grossman, Eds. ACM, 2018, pp. 465–480. [Online]. Available: https://doi.org/10.1145/3192366.3192387

[4] P. Orvalho, M. Janota, and V. Manquinho, "C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments," in *IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2024, Lisbon, Portugal, April 20, 2024*. IEEE, 2024, pp. 14–21.

[5] W. M. To and B. Yu, "Rise in higher education researchers and academic publications," *Emerald Open Research*, vol. 2, p. 3, 01 2020.

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

[7] U. Z. Ahmed, Z. Fan, J. Yi, O. I. Al-Bataineh, and A. Roychoudhury, "Verifix: Verified repair of programming assignments," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 4, pp. 74:1–74:31, 2022. [Online]. Available: https://doi.org/10.1145/3510418

[8] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (T)," in *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015,*

Lincoln, NE, USA, November 9-13, 2015, M. B. Cohen, L. Grunske, and M. Whalen, Eds. IEEE Computer Society, 2015, pp. 295–306. [Online]. Available: https://doi.org/10.1109/ASE.2015.60

[9] P. Orvalho, M. Janota, and V. M. Manquinho, "Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault Localization," in *AAAI-25, Sponsored by the Association for the Advancement of Artificial Intelligence, February 25 - March 4, 2025, Philadelphia, PA, USA*, T. Walsh, J. Shah, and Z. Kolter, Eds. AAAI Press, 2025, pp. 649–657.

[10] W. Wu, G. Li, Y. Sun, J. Wang, and T. Lai, "Analysec: A framework for assessing students' programs at structural and semantic level," in *2007 IEEE International Conference on Control and Automation*, 2007, pp. 742–747.

[11] H. Keuning, J. Jeuring, and B. Heeren, "A systematic literature review of automated feedback generation for programming exercises," *ACM Trans. Comput. Educ.*, vol. 19, no. 1, pp. 3:1–3:43, 2019. [Online]. Available: https://doi.org/10.1145/3231711

[12] Q. Zhang, C. Fang, Y. Ma, W. Sun, and Z. Chen, "A survey of learning-based automated program repair," *ACM Trans. Softw. Eng. Methodol.*, vol. 33, no. 2, pp. 55:1–55:69, 2024. [Online]. Available: https://doi.org/10.1145/3631974

[13] F. Y. Assiri and J. M. Bieman, "Fault localization for automated program repair: effectiveness, performance, repair correctness," *Softw. Qual. J.*, vol. 25, no. 1, pp. 171–199, 2017. [Online]. Available: https://doi.org/10.1007/s11219-016-9312-z

[14] Y. Ishii and T. Kutsuna, "Effective fault localization using dynamic slicing and an smt solver," in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2016, pp. 180–188.

[15] D. Ramos, J. Pereira, I. Lynce, V. Manquinho, and R. Martins, "UNCHARTIT: an interactive framework for program recovery from charts," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*. IEEE, 2020, pp. 175–186. [Online]. Available: https://doi.org/10.1145/3324884.3416613

[16] A. P. Guerreiro, J. Cortes, D. Vanderpooten, C. Bazgan, I. Lynce, V. Manquinho, and J. Figueira, "Exact and approximate determination of the pareto front using minimal correction subsets," *Computers & Operations Research*, vol. 153, p. 106153, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0305054823000175

[17] P. Orvalho, M. Janota, and V. M. Manquinho, "CFaults: Model-Based Diagnosis for Fault Localization in C with Multiple Test Cases," in *Formal Methods - 26th International Symposium, FM 2024*,

ser. Lecture Notes in Computer Science, A. Platzer, K. Y. Rozier, M. Pradella, and M. Rossi, Eds., vol. 14933. Springer, 2024, pp. 463–481.

[18] F. Zubair, M. Al-Hitmi, and C. Catal, "The use of large language models for program repair," *Computer Standards & Interfaces*, vol. 93, p. 103951, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S092054892400120X

[19] S. Wang, L. Lu, S. Qiu, Q. Tian, and H. Lin, "DALO-APR: llm-based automatic program repair with data augmentation and loss function optimization," *J. Supercomput.*, vol. 81, no. 5, p. 640, 2025. [Online]. Available: https://doi.org/10.1007/s11227-025-07102-3

[20] H. Hu, C. He, H. Zhang, X. Xie, and Q. Zhang, "APRMCTS: improving llm-based automated program repair with iterative tree search," *CoRR*, vol. abs/2507.01827, 2025. [Online]. Available: https://doi.org/10.48550/arXiv.2507.01827

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

[22] J. Li, F. Rabbi, C. Cheng, A. Sangalay, Y. Tian, and J. Yang, "An exploratory study on fine-tuning large language models for secure code generation," 2025. [Online]. Available: https://arxiv.org/abs/2408.09078

[23] K. Huang, J. Zhang, X. Bao, X. Wang, and Y. Liu, "Comprehensive fine-tuning large language models of code for automated program repair," *IEEE Transactions on Software Engineering*, vol. 51, no. 4, pp. 904–928, 2025.

[24] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "sk_p: a neural program corrector for moocs," 2016. [Online]. Available: https://arxiv.org/abs/1607.02902

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

[26] L. Schramm, "Improving performance of automatic program repair using learned heuristics," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 1071–1073. [Online]. Available: https://doi.org/10.1145/3106237.3121281

[27] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International*

Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018, F. Tip and E. Bodden, Eds. ACM, 2018, pp. 298–309. [Online]. Available: https://doi.org/10.1145/3213846.3213871

[28] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 54–72, 2012. [Online]. Available: https://doi.org/10.1109/TSE.2011.104

[29] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, L. K. Dillon, W. Visser, and L. A. Williams, Eds. ACM, 2016, pp. 691–701. [Online]. Available: https://doi.org/10.1145/2884781.2884807

[30] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, p. 34–55, Jan. 2017. [Online]. Available: https://doi.org/10.1109/TSE.2016.2560811

[31] Z. Huang and C. Wang, "Constraint based program repair for persistent memory bugs," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 91:1–91:12. [Online]. Available: https://doi.org/10.1145/3597503.3639204

[32] K. Liu, J. Zhang, L. Li, A. Koyuncu, D. Kim, C. Ge, Z. Liu, J. Klein, and T. F. Bissyandé, "Reliable fix patterns inferred from static checkers for automated program repair," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 4, pp. 96:1–96:38, 2023. [Online]. Available: https://doi.org/10.1145/3579637

[33] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. L. Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empir. Softw. Eng.*, vol. 25, no. 3, pp. 1980–2024, 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09780-z

[34] E. Pinconschi, R. Abreu, and P. Adão, "A comparative study of automatic program repair techniques for security vulnerabilities," in *32nd IEEE International Symposium on Software Reliability Engineering, ISSRE 2021, Wuhan, China, October 25-28, 2021*, Z. Jin, X. Li, J. Xiang, L. Mariani, T. Liu, X. Yu, and N. Ivaki, Eds. IEEE, 2021, pp. 196–207. [Online]. Available: https://doi.org/10.1109/ISSRE52982.2021.00031

[35] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA,*

USA, May 18-26, 2013, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds. IEEE Computer Society, 2013, pp. 802–811. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606626

[36] F. Long, P. Amidon, and M. C. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 727–739. [Online]. Available: https://doi.org/10.1145/3106237.3106253

[37] J. Jiang, Z. Zhao, Z. Ye, B. Wang, H. Zhang, and J. Chen, "Enhancing redundancy-based automated program repair by fine-grained pattern mining," *CoRR*, vol. abs/2312.15955, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2312.15955

[38] Y. A. Ahmed and A. Sharo, "On the education effect of chatgpt: Is ai chatgpt to dominate education career profession?" in *2023 International Conference on Intelligent Computing, Communication, Networking and Services (ICCNS)*, 2023, pp. 79–84.

[39] Y. Xue, H. Chen, G. R. Bai, R. Tairas, and Y. Huang, "Does chatgpt help with introductory programming? an experiment of students using chatgpt in cs1," in *2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, 2024, pp. 331–341.

[40] N. K. Gupta, A. Chaudhary, R. Singh, and R. Singh, "Chatgpt: Exploring the capabilities and limitations of a large language model for conversational ai," in *2023 International Conference on Advances in Computation, Communication and Information Technology (ICAICCIT)*, 2023, pp. 139–142.

[41] H. Ali, J. Qadir, T. Alam, M. Househ, and Z. Shah, "Chatgpt and large language models in healthcare: Opportunities and risks," in *2023 IEEE International Conference on Artificial Intelligence, Blockchain, and Internet of Things (AIBThings)*, 2023, pp. 1–4.

[42] Y. Liu, H. He, T. Han, X. Zhang, M. Liu, J. Tian, Y. Zhang, J. Wang, X. Gao, T. Zhong, Y. Pan, S. Xu, Z. Wu, Z. Liu, X. Zhang, S. Zhang, X. Hu, T. Zhang, N. Qiang, T. Liu, and B. Ge, "Understanding llms: A comprehensive overview from training to inference," *Neurocomputing*, vol. 620, p. 129190, 2025. [Online]. Available: https://doi.org/10.1016/j.neucom.2024.129190

[43] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: memory optimizations toward training trillion parameter models," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, C. Cuicchi, I. Qualters, and W. T. Kramer, Eds. IEEE/ACM, 2020, p. 20. [Online]. Available: https://doi.org/10.1109/SC41405.2020.00024

[44] Q. Feng, X. Ma, J. Sheng, Z. Feng, W. Song, and P. Liang, “Integrating various software artifacts for better llm-based bug localization and program repair,” *arXiv preprint arXiv:2412.03905*, 2024.

[45] S. B. Hossain, N. Jiang, Q. Zhou, X. Li, W. Chiang, Y. Lyu, H. A. Nguyen, and O. Tripp, “A deep dive into large language models for automated bug localization and repair,” *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, pp. 1471–1493, 2024. [Online]. Available: https://doi.org/10.1145/3660773

[46] Y. Qin, S. Wang, Y. Lou, J. Dong, K. Wang, X. Li, and X. Mao, “Agentfl: Scaling llm-based fault localization to project-level context,” *CoRR*, vol. abs/2403.16362, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2403.16362

[47] J. Yan, J. Huang, C. Fang, J. Yan, and J. Zhang, “Better debugging: Combining static analysis and llms for explainable crashing fault localization,” *CoRR*, vol. abs/2408.12070, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2408.12070

[48] Q. Zhang, C. Fang, T. Zhang, B. Yu, W. Sun, and Z. Chen, “GAMMA: revisiting template-based automated program repair via mask prediction,” *CoRR*, vol. abs/2309.09308, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2309.09308

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

[50] B. Yang, H. Tian, J. Ren, H. Zhang, J. Klein, T. F. Bissyandé, C. Le Goues, and S. Jin, “Multi-objective fine-tuning for enhanced program repair with llms,” *CoRR*, vol. abs/2404.12636, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2404.12636

[51] R. Wu, H. Zong, E. Wu, J. Li, Y. Zhou, C. Zhang, Y. Zhang, J. Wang, T. Tang, and B. Shen, “Improving large language models for mirna information extraction via prompt engineering,” *Computer Methods and Programs in Biomedicine*, vol. 271, p. 109033, 2025. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016926072500450X

[52] X. Cai and L. Jiang, “Adapting knowledge prompt tuning for enhanced automated program repair,” 2025. [Online]. Available: https://arxiv.org/abs/2504.01523

[53] H. Joshi, J. P. C. Sánchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radicek, “Repair is nearly generation: Multilingual program repair with llms,” in *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence,*

IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023, B. Williams, Y. Chen, and J. Neville, Eds. AAAI Press, 2023, pp. 5131–5140. [Online]. Available: https://doi.org/10.1609/aaai.v37i4.25642

[54] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu, "Programming by sketching for bit-streaming programs," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 281–294. [Online]. Available: https://doi.org/10.1145/1065010.1065045

[55] S. Karkalas and S. Gutiérrez-Santos, "Enhanced javascript learning using code quality tools and a rule-based system in the flip exploratory learning environment," in *2014 IEEE 14th International Conference on Advanced Learning Technologies*, 2014, pp. 84–88.

[56] J. Y. Khan and G. Uddin, "Automatic code documentation generation using GPT-3," in *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 2022, pp. 174:1–174:6. [Online]. Available: https://doi.org/10.1145/3551349.3559548

[57] J. Leinonen, P. Denny, S. MacNeil, S. Sarsa, S. Bernstein, J. Kim, A. Tran, and A. Hellas, "Comparing code explanations created by students and large language models," in *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1, ITiCSE 2023, Turku, Finland, July 7-12, 2023*, M. Laakso, M. Monga, Simon, and J. Sheard, Eds. ACM, 2023, pp. 124–130. [Online]. Available: https://doi.org/10.1145/3587102.3588785

[58] M. Mueller, C. List, and M. Kipp, "The power of context: An llm-based programming tutor with focused and proactive feedback," in *Proceedings of the 6th European Conference on Software Engineering Education, ECSEE 2025, Seeon, Germany, June 2-4, 2025*, J. Mottok and G. Hagel, Eds. ACM, 2025, pp. 1–10. [Online]. Available: https://doi.org/10.1145/3723010.3723034

[59] H. Nguyen, N. Stott, and V. Allan, "Comparing feedback from large language models and instructors: Teaching computer science at scale," in *Proceedings of the Eleventh ACM Conference on Learning @ Scale, L@S 2024, Atlanta, GA, USA, July 18-20, 2024*, D. Joyner, M. K. Kim, X. Wang, and M. Xia, Eds. ACM, 2024, pp. 335–339. [Online]. Available: https://doi.org/10.1145/3657604.3664660

[60] F. Wu, N. Zhang, S. Jha, P. D. McDaniel, and C. Xiao, "A new era in LLM security: Exploring security concerns in real-world llm-based systems," *CoRR*, vol. abs/2402.18649, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2402.18649

[61] DeepSeek-AI, A. Liu, B. Feng, B. Xue, B. Wang, B. Wu, C. Lu, C. Zhao, C. Deng, C. Zhang, C. Ruan, D. Dai, D. Guo, D. Yang, D. Chen, D. Ji, E. Li, F. Lin, F. Dai, F. Luo, G. Hao, G. Chen, G. Li, H. Zhang, H. Bao, H. Xu, H. Wang, H. Zhang, H. Ding, H. Xin, H. Gao, H. Li, H. Qu, J. L. Cai, J. Liang, J. Guo, J. Ni, J. Li, J. Wang, J. Chen, J. Chen, J. Yuan, J. Qiu, J. Li, J. Song, K. Dong, K. Hu, K. Gao, K. Guan, K. Huang, K. Yu, L. Wang, L. Zhang, L. Xu, L. Xia, L. Zhao, L. Wang, L. Zhang, M. Li, M. Wang, M. Zhang, M. Zhang, M. Tang, M. Li, N. Tian, P. Huang, P. Wang, P. Zhang, Q. Wang, Q. Zhu, Q. Chen, Q. Du, R. J. Chen, R. L. Jin, R. Ge, R. Zhang, R. Pan, R. Wang, R. Xu, R. Zhang, R. Chen, S. S. Li, S. Lu, S. Zhou, S. Chen, S. Wu, S. Ye, S. Ma, S. Wang, S. Zhou, S. Yu, S. Zhou, S. Pan, T. Wang, T. Yun, T. Pei, T. Sun, W. L. Xiao, and W. Zeng, "Deepseek-v3 technical report," *CoRR*, vol. abs/2412.19437, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2412.19437

[62] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, A. Yang, A. Fan, A. Goyal, A. Hartshorn, A. Yang, A. Mitra, A. Sravankumar, A. Korenev, A. Hinsvark, A. Rao, A. Zhang, A. Rodriguez, A. Gregerson, A. Spataru, B. Roziere, B. Biron, B. Tang, B. Chern, C. Caucheteux, C. Nayak, C. Bi, C. Marra, C. McConnell, C. Keller, C. Touret, C. Wu, C. Wong, C. C. Ferrer, C. Nikolaidis, D. Allonsius, D. Song, D. Pintz, D. Livshits, D. Wyatt, D. Esiobu, D. Choudhary, D. Mahajan, D. Garcia-Olano, D. Perino, D. Hupkes, E. Lakomkin, E. AlBadawy, E. Lobanova, E. Dinan, E. M. Smith, F. Radenovic, F. Guzmán, F. Zhang, G. Synnaeve, G. Lee, G. L. Anderson, G. Thattai, G. Nail, G. Mialon, G. Pang, G. Cucurell, H. Nguyen, H. Korevaar, H. Xu, H. Touvron, I. Zarov, I. A. Ibarra, I. Kloumann, I. Misra, I. Evtimov, J. Zhang, J. Copet, J. Lee, J. Geffert, J. Vranes, J. Park, J. Mahadeokar, J. Shah, J. van der Linde, J. Billock, J. Hong, J. Lee, J. Fu, J. Chi, J. Huang, J. Liu, J. Wang, J. Yu, J. Bitton, J. Spisak, J. Park, J. Rocca, J. Johnstun, J. Saxe, J. Jia, K. V. Alwala, K. Prasad, K. Upasani, K. Plawiak, K. Li, K. Heafield, K. Stone, K. El-Arini, K. Iyer, K. Malik, K. Chiu, K. Bhalla, K. Lakhotia, L. Rantala-Yeary, L. van der Maaten, L. Chen, L. Tan, L. Jenkins, L. Martin, L. Madaan, L. Malo, L. Blecher, L. Landzaat, L. de Oliveira, M. Muzzi, M. Pasupuleti, M. Singh, M. Paluri, M. Kardas, M. Tsimpoukelli, M. Oldham, M. Rita, M. Pavlova, M. Kambadur, M. Lewis, M. Si, M. K. Singh, M. Hassan, N. Goyal, N. Torabi, N. Bashlykov, N. Bogoychev, N. Chatterji, N. Zhang, O. Duchenne, O. Çelebi, P. Alrassy, P. Zhang, P. Li, P. Vasic, P. Weng, P. Bhargava, P. Dubal, P. Krishnan, P. S. Koura, P. Xu, Q. He, Q. Dong, R. Srinivasan, R. Ganapathy, R. Calderer, R. S. Cabral, R. Stojnic, R. Raileanu, R. Maheswari, R. Girdhar, R. Patel, R. Sauvestre, R. Polidoro, R. Sumbaly, R. Taylor, R. Silva, R. Hou, R. Wang, S. Hosseini, S. Chennabasappa, S. Singh, S. Bell, S. S. Kim, S. Edunov, S. Nie, S. Narang, S. Raparthy, S. Shen, S. Wan, S. Bhosale, S. Zhang, S. Vandenhende, S. Batra, S. Whitman, S. Sootla, S. Collot, S. Gururangan, S. Borodinsky, T. Herman, T. Fowler, T. Sheasha, T. Georgiou,

T. Scialom, T. Speckbacher, T. Mihaylov, T. Xiao, U. Karn, V. Goswami, V. Gupta, V. Ramanathan, V. Kerkez, V. Gonguet, V. Do, V. Vogeti, V. Albiero, V. Petrovic, W. Chu, W. Xiong, W. Fu, W. Meers, X. Martinet, X. Wang, X. Wang, X. E. Tan, X. Xia, X. Xie, X. Jia, X. Wang, Y. Goldschlag, Y. Gaur, Y. Babaei, Y. Wen, Y. Song, Y. Zhang, Y. Li, Y. Mao, Z. D. Coudert, Z. Yan, Z. Chen, Z. Papakipos, A. Singh, A. Srivastava, A. Jain, A. Kelsey, A. Shajnfeld, A. Gangidi, A. Victoria, A. Goldstand, A. Menon, A. Sharma, A. Boesenberg, A. Baevski, A. Feinstein, A. Kallet, A. Sangani, A. Teo, A. Yunus, A. Lupu, A. Alvarado, A. Caples, A. Gu, A. Ho, A. Poulton, A. Ryan, A. Ramchandani, A. Dong, A. Franco, A. Goyal, A. Saraf, A. Chowdhury, A. Gabriel, A. Bharambe, A. Eisenman, A. Yazdan, B. James, B. Maurer, B. Leonhardi, B. Huang, B. Loyd, B. D. Paola, B. Paranjape, B. Liu, B. Wu, B. Ni, B. Hancock, B. Wasti, B. Spence, B. Stojkovic, B. Gamido, B. Montalvo, C. Parker, C. Burton, C. Mejia, C. Liu, C. Wang, C. Kim, C. Zhou, C. Hu, C.-H. Chu, C. Cai, C. Tindal, C. Feichtenhofer, C. Gao, D. Civin, D. Beaty, D. Kreymer, D. Li, D. Adkins, D. Xu, D. Testuggine, D. David, D. Parikh, D. Liskovich, D. Foss, D. Wang, D. Le, D. Holland, E. Dowling, E. Jamil, E. Montgomery, E. Presani, E. Hahn, E. Wood, E.-T. Le, E. Brinkman, E. Arcaute, E. Dunbar, E. Smothers, F. Sun, F. Kreuk, F. Tian, F. Kokkinos, F. Ozgenel, F. Caggioni, F. Kanayet, F. Seide, G. M. Florez, G. Schwarz, G. Badeer, G. Swee, G. Halpern, G. Herman, G. Sizov, Guangyi, Zhang, G. Lakshminarayanan, H. Inan, H. Shojanazeri, H. Zou, H. Wang, H. Zha, H. Habeeb, H. Rudolph, H. Suk, H. Aspegren, H. Goldman, H. Zhan, I. Damlaj, I. Molybog, I. Tufanov, I. Leontiadis, I.-E. Veliche, I. Gat, J. Weissman, J. Geboski, J. Kohli, J. Lam, J. Asher, J.-B. Gaya, J. Marcus, J. Tang, J. Chan, J. Zhen, J. Reizenstein, J. Teboul, J. Zhong, J. Jin, J. Yang, J. Cummings, J. Carvill, J. Shepard, J. McPhie, J. Torres, J. Ginsburg, J. Wang, K. Wu, K. H. U, K. Saxena, K. Khandelwal, K. Zand, K. Matosich, K. Veeraraghavan, K. Michelena, K. Li, K. Jagadeesh, K. Huang, K. Chawla, K. Huang, L. Chen, L. Garg, L. A, L. Silva, L. Bell, L. Zhang, L. Guo, L. Yu, L. Moshkovich, L. Wehrstedt, M. Khabsa, M. Avalani, M. Bhatt, M. Mankus, M. Hasson, M. Lennie, M. Reso, M. Groshev, M. Naumov, M. Lathi, M. Keneally, M. Liu, M. L. Seltzer, M. Valko, M. Restrepo, M. Patel, M. Vyatskov, M. Samvelyan, M. Clark, M. Macey, M. Wang, M. J. Hermoso, M. Metanat, M. Rastegari, M. Bansal, N. Santhanam, N. Parks, N. White, N. Bawa, N. Singhal, N. Egebo, N. Usunier, N. Mehta, N. P. Laptev, N. Dong, N. Cheng, O. Chernoguz, O. Hart, O. Salpekar, O. Kalinli, P. Kent, P. Parekh, P. Saab, P. Balaji, P. Rittner, P. Bontrager, P. Roux, P. Dollar, P. Zvyagina, P. Ratanchandani, P. Yuvraj, Q. Liang, R. Alao, R. Rodriguez, R. Ayub, R. Murthy, R. Nayani, R. Mitra, R. Parthasarathy, R. Li, R. Hogan, R. Battey, R. Wang, R. Howes, R. Rinott, S. Mehta, S. Siby, S. J. Bondu, S. Datta, S. Chugh, S. Hunt, S. Dhillon, S. Sidorov, S. Pan, S. Mahajan, S. Verma, S. Yamamoto, S. Ramaswamy, S. Lindsay, S. Lindsay, S. Feng, S. Lin, S. C. Zha, S. Patil, S. Shankar, S. Zhang, S. Zhang, S. Wang, S. Agarwal, S. Sajuyigbe, S. Chintala, S. Max, S. Chen, S. Kehoe, S. Satterfield, S. Govindaprasad, S. Gupta, S. Deng, S. Cho, S. Virk,

S. Subramanian, S. Choudhury, S. Goldman, T. Remez, T. Glaser, T. Best, T. Koehler, T. Robinson, T. Li, T. Zhang, T. Matthews, T. Chou, T. Shaked, V. Vontimitta, V. Ajayi, V. Montanez, V. Mohan, V. S. Kumar, V. Mangla, V. Ionescu, V. Poenaru, V. T. Mihailescu, V. Ivanov, W. Li, W. Wang, W. Jiang, W. Bouaziz, W. Constable, X. Tang, X. Wu, X. Wang, X. Wu, X. Gao, Y. Kleinman, Y. Chen, Y. Hu, Y. Jia, Y. Qi, Y. Li, Y. Zhang, Y. Zhang, Y. Adi, Y. Nam, Yu, Wang, Y. Zhao, Y. Hao, Y. Qian, Y. Li, Y. He, Z. Rait, Z. DeVito, Z. Rosnbrick, Z. Wen, Z. Yang, Z. Zhao, and Z. Ma, "The llama 3 herd of models," 2024. [Online]. Available: https://arxiv.org/abs/2407.21783

[63] G. Team, A. Kamath, J. Ferret, S. Pathak, N. Vieillard, R. Merhej, S. Perrin, T. Matejovicova, A. Ramé, M. Rivière, L. Rouillard, T. Mesnard, G. Cideron, J. bastien Grill, S. Ramos, E. Yvinec, M. Casbon, E. Pot, I. Penchev, G. Liu, F. Visin, K. Kenealy, L. Beyer, X. Zhai, A. Tsitsulin, R. Busa-Fekete, A. Feng, N. Sachdeva, B. Coleman, Y. Gao, B. Mustafa, I. Barr, E. Parisotto, D. Tian, M. Eyal, C. Cherry, J.-T. Peter, D. Sinopalnikov, S. Bhupatiraju, R. Agarwal, M. Kazemi, D. Malkin, R. Kumar, D. Vilar, I. Brusilovsky, J. Luo, A. Steiner, A. Friesen, A. Sharma, A. Sharma, A. M. Gilady, A. Goedeckemeyer, A. Saade, A. Feng, A. Kolesnikov, A. Bendebury, A. Abdagic, A. Vadi, A. György, A. S. Pinto, A. Das, A. Bapna, A. Miech, A. Yang, A. Paterson, A. Shenoy, A. Chakrabarti, B. Piot, B. Wu, B. Shahriari, B. Petrini, C. Chen, C. L. Lan, C. A. Choquette-Choo, C. Carey, C. Brick, D. Deutsch, D. Eisenbud, D. Cattle, D. Cheng, D. Paparas, D. S. Sreepathihalli, D. Reid, D. Tran, D. Zelle, E. Noland, E. Huizenga, E. Kharitonov, F. Liu, G. Amirkhanyan, G. Cameron, H. Hashemi, H. Klimczak-Plucińska, H. Singh, H. Mehta, H. T. Lehri, H. Hazimeh, I. Ballantyne, I. Szpektor, I. Nardini, J. Pouget-Abadie, J. Chan, J. Stanton, J. Wieting, J. Lai, J. Orbay, J. Fernandez, J. Newlan, J. yeong Ji, J. Singh, K. Black, K. Yu, K. Hui, K. Vodrahalli, K. Greff, L. Qiu, M. Valentine, M. Coelho, M. Ritter, M. Hoffman, M. Watson, M. Chaturvedi, M. Moynihan, M. Ma, N. Babar, N. Noy, N. Byrd, N. Roy, N. Momchev, N. Chauhan, N. Sachdeva, O. Bunyan, P. Botarda, P. Caron, P. K. Rubenstein, P. Culliton, P. Schmid, P. G. Sessa, P. Xu, P. Stanczyk, P. Tafti, R. Shivanna, R. Wu, R. Pan, R. Rokni, R. Willoughby, R. Vallu, R. Mullins, S. Jerome, S. Smoot, S. Girgin, S. Iqbal, S. Reddy, S. Sheth, S. Põder, S. Bhatnagar, S. R. Panyam, S. Eiger, S. Zhang, T. Liu, T. Yacovone, T. Liechty, U. Kalra, U. Evci, V. Misra, V. Roseberry, V. Feinberg, V. Kolesnikov, W. Han, W. Kwon, X. Chen, Y. Chow, Y. Zhu, Z. Wei, Z. Egyed, V. Cotruta, M. Giang, P. Kirk, A. Rao, K. Black, N. Babar, J. Lo, E. Moreira, L. G. Martins, O. Sanseviero, L. Gonzalez, Z. Gleicher, T. Warkentin, V. Mirrokni, E. Senter, E. Collins, J. Barral, Z. Ghahramani, R. Hadsell, Y. Matias, D. Sculley, S. Petrov, N. Fiedel, N. Shazeer, O. Vinyals, J. Dean, D. Hassabis, K. Kavukcuoglu, C. Farabet, E. Buchatskaya, J.-B. Alayrac, R. Anil, Dmitry, Lepikhin, S. Borgeaud, O. Bachem, A. Joulin, A. Andreev, C. Hardin, R. Dadashi, and L. Hussenot, "Gemma 3 technical report," 2025. [Online]. Available: https://arxiv.org/abs/2503.19786

[64] M. Abdin, J. Aneja, H. Behl, S. Bubeck, R. Eldan, S. Gunasekar, M. Harrison, R. J. Hewett,

M. Javaheripi, P. Kauffmann, J. R. Lee, Y. T. Lee, Y. Li, W. Liu, C. C. T. Mendes, A. Nguyen, E. Price, G. de Rosa, O. Saarikivi, A. Salim, S. Shah, X. Wang, R. Ward, Y. Wu, D. Yu, C. Zhang, and Y. Zhang, "Phi-4 technical report," 2024. [Online]. Available: https://arxiv.org/abs/2412.08905

[65] Microsoft, :, A. Abouelenin, A. Ashfaq, A. Atkinson, H. Awadalla, N. Bach, J. Bao, A. Benhaim, M. Cai, V. Chaudhary, C. Chen, D. Chen, D. Chen, J. Chen, W. Chen, Y.-C. Chen, Y. ling Chen, Q. Dai, X. Dai, R. Fan, M. Gao, M. Gao, A. Garg, A. Goswami, J. Hao, A. Hendy, Y. Hu, X. Jin, M. Khademi, D. Kim, Y. J. Kim, G. Lee, J. Li, Y. Li, C. Liang, X. Lin, Z. Lin, M. Liu, Y. Liu, G. Lopez, C. Luo, P. Madan, V. Mazalov, A. Mitra, A. Mousavi, A. Nguyen, J. Pan, D. Perez-Becker, J. Platin, T. Portet, K. Qiu, B. Ren, L. Ren, S. Roy, N. Shang, Y. Shen, S. Singhal, S. Som, X. Song, T. Sych, P. Vaddamanu, S. Wang, Y. Wang, Z. Wang, H. Wu, H. Xu, W. Xu, Y. Yang, Z. Yang, D. Yu, I. Zabir, J. Zhang, L. L. Zhang, Y. Zhang, and X. Zhou, "Phi-4-mini technical report: Compact yet powerful multimodal language models via mixture-of-loras," 2025. [Online]. Available: https://arxiv.org/abs/2503.01743

[66] Y. Wei, Z. Wang, J. Liu, Y. Ding, and L. Zhang, "Magicoder: Empowering code generation with oss-instruct," 2024. [Online]. Available: https://arxiv.org/abs/2312.02120

[67] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" 2023.

[68] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, "Mistral 7b," 2023. [Online]. Available: https://arxiv.org/abs/2310.06825

[69] H. Zhao, J. Hui, J. Howland, N. Nguyen, S. Zuo, A. Hu, C. A. Choquette-Choo, J. Shen, J. Kelley, K. Bansal, L. Vilnis, M. Wirth, P. Michel, P. Choy, P. Joshi, R. Kumar, S. Hashmi, S. Agrawal, Z. Gong, J. Fine, T. Warkentin, A. J. Hartman, B. Ni, K. Korevec, K. Schaefer, and S. Huffman, "Codegemma: Open code models based on gemma," *CoRR*, vol. abs/2406.11409, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2406.11409

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

[71] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang, A. Yang, R. Men, F. Huang, X. Ren, X. Ren, J. Zhou, and J. Lin, "Qwen2.5-coder technical report," *CoRR*, vol. abs/2409.12186, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2409.12186

[72] P. Orvalho, J. Piepenbrock, M. Janota, and V. Manquinho, "Graph neural networks for mapping variables between programs," in *ECAI 2023 - 26th European Conference on Artificial Intelligence*, ser. Frontiers in Artificial Intelligence and Applications, K. Gal, A. Nowé, G. J. Nalepa, R. Fairstein, and R. Radulescu, Eds., vol. 372. IOS Press, 2023, pp. 1811–1818.

[73] P. Orvalho, M. Janota, and V. Manquinho, "InvAASTCluster: On Applying Invariant-Based Program Clustering to Introductory Programming Assignments," *J. Syst. Softw.*, vol. 230, p. 112481, 2025.

[74] K.-C. Tai, "The tree-to-tree correction problem," *J. ACM*, vol. 26, no. 3, p. 422–433, Jul. 1979. [Online]. Available: https://doi.org/10.1145/322139.322143