IMPACT OF PATCH PRIORITIZATION ON AUTOMATED PROGRAM REPAIR

MOUMITA ASAD
Exam Roll: 0710

A Thesis
Submitted to the Master of Science in Software Engineering Program Office
of the Institute of Information Technology, University of Dhaka
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE IN SOFTWARE ENGINEERING

Institute of Information Technology
University of Dhaka
DHAKA, BANGLADESH

© MOUMITA ASAD, 2020

IMPACT OF PATCH PRIORITIZATION ON AUTOMATED PROGRAM REPAIR

MOUMITA ASAD

Approved:

Signature *Date*

Supervisor: Dr. Kazi Muheymin-Us-Sakib

To *Dr. Mahfuza Khanam*, my mother
who has always been there for me and inspired me

Abstract

Automated program repair aims at finding the correct patch of a bug using a specification such as test cases. Expanding the search space of a program repair technique increases the probability of generating the correct patch. However, it also increases the chance of finding incorrect plausible patches before the correct one. To prevent these problems, existing program repair approaches either avoid or limitedly focus on expression level bugs such as method invocation or assignment expression. Nevertheless, an existing study found that almost 82.40% repair actions are associated with expressions. Due to the importance of handling expression type bugs, an automated program repair approach needs to be devised that extensively deals with these types of bug. Besides, the devised technique should incorporate a strategy to handle the enlarged search space and prioritize the correct patch over incorrect plausible ones.

To find potentially correct patches earlier, recent program repair approaches have incorporated patch prioritization using either syntactic or semantic similarity between faulty code and fixing ingredient. However, it has not been analyzed yet whether patch prioritization can be further improved by combining similarities. For this purpose, two patch prioritization methods that integrate syntactic and semantic similarities are proposed. For calculating semantic similarity, genealogical and variable similarity are used since these are good at differentiating between correct and incorrect patches. To measure syntactic similarity, two widely used metrics namely normalized longest common subsequence and token similarity are

considered individually. To understand the impact of combining similarities, the proposed approaches are compared with patch prioritization techniques that use only semantic or syntactic similarity. For comparison, replacement mutation bugs from the historical bug fixes dataset are used. The proposed methods outperform semantic or syntactic similarity based patch prioritization approaches, in terms of median rank of the correct patch and search space reduction. Furthermore, the combined approaches rank the correct patch at the first position in 11.79% and 10.16% cases. It indicates that combining similarities has the potential to identify the correct patch before incorrect plausible patches.

Based on the result of the empirical study, two automated program repair techniques are proposed that incorporate syntactic and semantic similarities to work at the expression level. At first, the proposed techniques identify faulty expression type nodes and assign those a suspiciousness score using the execution traces of the test cases. Next, these faulty nodes are replaced with fixing ingredients for generating patches. To validate potentially correct patches earlier, the generated patches are sorted using suspiciousness score and similarity score. Lastly, the correctness of a patch is examined by executing test cases. To evaluate the proposed techniques, these are compared with baseline program repair approaches that use either semantic or syntactic similarity. For comparison, these approaches are executed on single line bugs from Defects4J and QuixBugs benchmark respectively. Result reveals that the proposed techniques can correctly repair 22 and 21 expression level bugs from these benchmarks which are higher than approaches using only semantic or syntactic similarity. Furthermore, the devised approaches obtain a precision of 64.71% and 61.76% and outperform the baseline program repair techniques.

Acknowledgments

"All praises are due to Allah"

Firstly, I express my gratitude to Allah for giving me the opportunity and granting me the ability to continue my research work properly.

I want to express my appreciation and utmost respect to my supervisor, Professor Dr. Kazi Muheymin-Us-Sakib, Institute of Information Technology, University of Dhaka. Without his motivation, support, effort and guideline, this thesis could not be successful.

I would like to appreciate the faculty and thesis committee members of Institute of Information Technology, University of Dhaka for their valuable feedback, which improved my thesis.

My sincere thanks also goes to He Ye from KTH Royal Institute of Technology and Ali Ghanbari from University of Texas at Dallas for sharing their experimentation data with me.

I am grateful to my classmates for their support. I am immensely indebted to my parents who have always been there whenever I need.

I am also thankful to the Ministry of Posts, Telecommunications and Information Technology, Government of the Peoples Republic of Bangladesh for granting me ICT Fellowship No - 56.00.0000.028.33.093.19-427;Dated 20.11.2019. Last but not least, my special thanks to the Bangladesh Research and Education Network (BdREN) for providing me with virtual machine to carry out my thesis.

List of Publications

1. "Impact Analysis of Syntactic and Semantic Similarities on Patch Prioritization in Automated Program Repair" in *Proceedings of the 36th International Conference on Software Maintenance and Evolution (ICSME)*, pp. 328-332, 2019.

2. "Impact of Combining Syntactic and Semantic Similarities on Patch Prioritization" in *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE)*, pp. 170-180, 2020.

3. "Impact of Similarity on Repairing Small Programs: A Case Study on QuixBugs Benchmark" in *Proceedings of the 42nd International Conference on Software Engineering Workshops (ICSEW)*, pp. 21-22, 2020.

4. "Combined Similarity Based Automated Program Repair Approaches for Expression Level Bugs" in *Communications in Computer and Information Science (CCIS) (Revised Selected Papers of ENASE)*, 2020. (Submitted)

Contents

Approval ii

Dedication iii

Abstract iv

Acknowledgements vi

List of Publications vii

Table of Contents viii

List of Tables xi

List of Figures xii

Listings xiii

1 Introduction 1
- 1.1 Motivation . . . 5
- 1.2 Research Questions . . . 9
- 1.3 Contribution and Achievement . . . 10
- 1.4 Organization of the Thesis . . . 12

2 Background Study 14
- 2.1 Concepts Related to Software Bug . . . 14
  - 2.1.1 Software Bug . . . 15
  - 2.1.2 Test Case . . . 15
  - 2.1.3 Faulty Code . . . 17
  - 2.1.4 Patch . . . 17
  - 2.1.5 Fixing Ingredient . . . 19
- 2.2 Concepts Related to Source Code . . . 19
  - 2.2.1 Statement and Expression . . . 19
  - 2.2.2 Abstract Syntax Tree . . . 20
- 2.3 Automated Program Repair . . . 21
  - 2.3.1 Importance of Automated Program Repair . . . 24
  - 2.3.2 Steps of Automated Program Repair . . . 25
  - 2.3.3 Challenges of Automated Program Repair . . . 28

2.4 Summary 32

3 Literature Review on Automated Program Repair 33
3.1 Stochastic Patch Selection Based Approaches 34
3.1.1 GenProg 34
3.1.2 PAR 35
3.1.3 RSRepair 37
3.1.4 HDRepair 38
3.2 Patch Prioritization Based Approaches 40
3.2.1 Approaches Using Characteristics of Generated Patches . . . 40
3.2.1.1 ssFix 41
3.2.1.2 SimFix 42
3.2.2 Approaches Using Similarity Between Faulty Code and Fixing Ingredient 44
3.2.2.1 ELIXIR 44
3.2.2.2 CapGen 46
3.2.2.3 LSRepair 48
3.3 Summary 50

4 Impact of Combining Syntactic and Semantic Similarities on Patch Prioritization 52
4.1 Introduction 53
4.2 Methodology of the Empirical Study 54
4.2.1 Dataset Preprocessing 55
4.2.2 Proposed Approach 57
4.3 Experiment and Result Analysis 65
4.3.1 Implementation 65
4.3.2 Evaluation 66
4.4 Threats to Validity 71
4.5 Summary 72

5 Combined Similarity Based Automated Program Repair Approaches 74
5.1 Introduction 75
5.2 Methodology of the Proposed Approaches 77
5.3 Experiment 82
5.3.1 Implementation 82
5.3.2 Dataset 83
5.3.3 Evaluation Metrics 86
5.4 Result Analysis 88
5.5 Threats to Validity 95
5.6 Summary 96

6 Conclusion 98
6.1 Impact of Combining Syntactic and Semantic Similarities on Patch Prioritization 99
6.2 Combined Similarity Based Automated Program Repair Approaches 100
6.3 Future Work 102

A Selected Bugs of Defects4J Benchmark **104**

Bibliography **105**

List of Tables

2.1 Calculation of Suspiciousness Score for Buggy *gcd()* Program 26

4.1 Genealogy Contexts of Faulty Node *Character.isDigit(next)* and Fixing Ingredient *Character.isDigit(next)* ‖ *next == '.'* 61
4.2 Genealogical Similarity between Faulty Node *Character.isDigit(next)* and Fixing Ingredient *Character.isDigit(next)* ‖ *next == '.'* 62
4.3 Token Set of Faulty Node *Character.isDigit(next)* and Fixing Ingredient *Character.isDigit(next)* ‖ *next == '.'* 64
4.4 Differences between Mean Ranking of Correct Patch 70

5.1 Test Suite for Buggy *levenshtein()* Method 79
5.2 Line Number wise Suspiciousness Score 79
5.3 Projects of Defects4J Benchmark 85
5.4 Results of Automated Patch Correctness Assessment for 5 Buggy Programs of QuixBugs 87
5.5 Results of Different Approaches on Defects4J Benchmark 89
5.6 Results of Different Approaches on QuixBugs Benchmark 89
5.7 Differences between Mean Repairing Time (In Minutes) of Different Approaches 94
5.8 Overall Result of Different Approaches 95

A.1 Selected Bugs of Defects4J Benchmark 104

List of Figures

1.1 Validating Numerous Patches Before the Correct One 3
1.2 Incorrect Plausible Patch Occurs Before the Correct One 3
1.3 Ranking the Correct Patch Higher Through Patch Prioritization . . 3

2.1 Numerous Patches for the Buggy *max()* Method 18
2.2 Abstract Syntax Tree for *max()* Method 21
2.3 Steps of Automated Program Repair 25
2.4 Activities to Constrain the Search Space 27

3.1 Graph-based Representation of A Bug Fix 39
3.2 Classification of Patch Prioritization Based Approaches 40

4.1 Overview of the Methodology 55
4.2 Steps of Dataset Preprocessing 56
4.3 Overview of the Proposed Technique 58
4.4 Comparison of Correct Patch Rank among *SSBA*, *Com-L*, *Com-T*, *LBA* and *TBA* 67
4.5 Comparison of Average Space Reduction among *SSBA*, *Com-L*, *Com-T*, *LBA* and *TBA* 68
4.6 Comparison of Perfect Repair among *SSBA*, *Com-L*, *Com-T*, *LBA* and *TBA* 69

5.1 Overview of the Proposed Automated Program Repair Approaches 76
5.2 Ranked List of Sample Patches for the Buggy *levenshtein()* Method 81
5.3 Steps for Selecting Bugs from Defects4J Benchmark 84

Listings

1.1 Expected Code 2
1.2 Actual Code 2
1.3 Sample Patch 1 for *sum()* Method 2
1.4 Sample Patch 2 for *sum()* Method 2
1.5 Sample Patch 3 for *sum()* Method 2
1.6 Sample Patch 4 for *sum()* Method 2
1.7 Sample Patch 5 for *sum()* Method 2
1.8 Sample Patch 6 for *sum()* Method 2
1.9 A Sample Bug *Chart_9* from Defects4J Benchmark 7
1.10 A Correct Sample Patch from Project Apache Lucene 8
1.11 A Correct Sample Patch from Project Apache Derby 8
1.12 A Correct Sample Patch from Project Apache Ant 8
2.1 Buggy Version of *max()* Method 15
2.2 A Passing Test Case for Method *max()* 16
2.3 A Failing Test Case for Method *max()* 16
2.4 Fixed Version of *max()* Method 17
2.5 Statements of *max()* Method 20
2.6 Buggy Version of *gcd()* Method 22
2.7 Test Cases for *gcd()* Method 22
2.8 Fixed Version of *gcd()* Method 23
2.9 Buggy Version of *flatten() Method* 28
2.10 Fixed Version of *flatten() Method* 29
2.11 An Incorrect Plausible Patch for Buggy *flatten() Method* 30
2.12 Test Suite for Repairing Buggy *flatten() Method* 31
2.13 A Test Case that Differentiates between Correct and Incorrect Plausible Patch of *flatten() Method* 32
4.1 Buggy Statement, Fixed Statement and Fixing Ingredient of Bug *fasseg_exp4j_4* 59
4.2 A Correct Sample Patch for Bug *hornetq_hornetq_70* 67
4.3 A Correct Sample Patch for Bug *Cervator_Terasology_2* 69
4.4 A Correct Sample Patch for Bug *spring-projects_spring-roo_10* . . . 70
4.5 A Correct Sample Patch for Bug *broadgsa_gatk_2* 71
5.1 A Sample Patch from CodRep Dataset 75
5.2 Buggy *levenshtein()* Method 78
5.3 A Correct Sample Patch for Bug *Math_63* 88
5.4 An Incorrect Plausible Patch for Bug *Math_63* 88
5.5 A Correct Sample Patch for Bug *Math_59* 88

5.6 A Correct Sample Patch for Bug *Lang_59* 90
5.7 An Incorrect Plausible Patch Generated by *LBA* and *TBA* for Bug *Lang_59* 90
5.8 An Incorrect Plausible Patch Generated by *ComFix-T* for Bug *flatten* 91
5.9 A Correct Sample Patch for Bug *flatten* 91
5.10 An Incorrect Plausible Patch Generated by *ComFix-L* and *ComFix-T* for Bug *quicksort* 92
5.11 A Correct Sample Patch for Bug *next_palindrome* 92
5.12 A Correct Sample Patch for Bug *mergesort* 92

Chapter 1

Introduction

A patch is the modifications applied to a program for fixing a bug. For example, a developer wants to write a code that adds two numbers, as shown in Listing 1.1. By mistake, she writes the code, depicted in Listing 1.2. Here, the patch for fixing the bug in Listing 1.2 is replacing a-b with a+b. Automated program repair finds the patch of a bug based on a specification, for example, test cases [1]. It takes a buggy program and test cases with at least one failing case as input. It outputs a patch that passes all the test cases, known as the plausible patch. Automated program repair works in three steps namely fault localization, patch generation and patch validation [2], [3]. Fault localization identifies the faulty code where the bug resides. Patch generation modifies the faulty code to fix the bug. Finally, patch validation examines whether the bug has been fixed or not, by executing test cases. Since the search space is infinite, numerous patches can be generated [4]. For example, 6 sample patches for Listing 1.2 are shown in Listing 1.3-1.8. Similarly, many more patches can be generated. Besides, a plausible patch can be incorrect, which is known as overfitting problem [5].

Listing 1.1: Expected Code

```
int sum(int a, int b) {
    return a+b; //correct code
}
```

Listing 1.2: Actual Code

```
int sum(int a, int b) {
    return a-b; //faulty code
}
```

Listing 1.3: Sample Patch 1 for *sum()* Method

```
int sum(int a, int b) {
    return a*b;
}
```

Listing 1.4: Sample Patch 2 for *sum()* Method

```
int sum(int a, int b) {
    return a/b;
}
```

Listing 1.5: Sample Patch 3 for *sum()* Method

```
int sum(int a, int b) {
    return a+b;
}
```

Listing 1.6: Sample Patch 4 for *sum()* Method

```
int sum(int a, int b) {
    return (a-b)/3;
}
```

Listing 1.7: Sample Patch 5 for *sum()* Method

```
int sum(int a, int b) {
    return (a*b)/10;
}
```

Listing 1.8: Sample Patch 6 for *sum()* Method

```
int sum(int a, int b) {
    return a/4;
}
```

To limit the search space, most of the program repair techniques rely on redundancy assumption, which states that the patch of a bug can be found elsewhere in the application or other projects [6], [7]. The existing code element that is reused for generating the patch, is called the fixing ingredient [5]. The redundancy assumption has already been validated by existing studies such as [8], [9]. Martinez

et al. found that 3-17% of the commits are redundant at the line level, whereas it is 29-52% at the token level [8]. Another study on 15,723 commits reported that approximately 30% fixing ingredients exist in the same buggy file [9]. Although the redundancy assumption limits the search space, in practice it is too large for exploring exhaustively [6]. In Figure 1.1, total 130 candidate patches are generated and the correct patch occurs at position 93. Hence, 92 patches need to be compiled and validated before finding the correct patch, which is time consuming [6], [10]. Furthermore, if an incorrect plausible patch occurs prior to the correct one, as illustrated in Figure 1.2, automated program repair will output the incorrect patch since it is the first plausible patch. Consequently, it minimizes the precision (the ratio of correct and incorrect plausible patches) of an automated program repair technique. To tackle these problems, potentially correct patches need to be validated earlier.

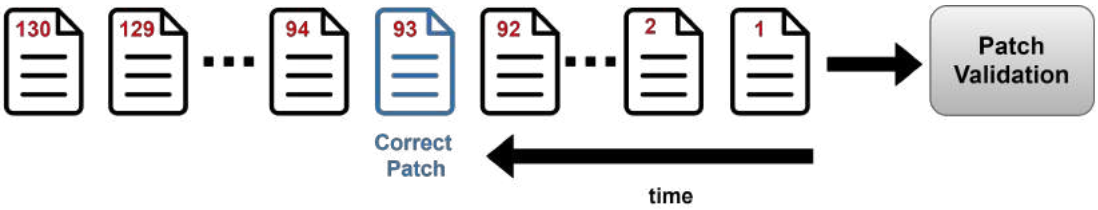

Figure 1.1: Validating Numerous Patches Before the Correct One

Figure 1.2: Incorrect Plausible Patch Occurs Before the Correct One

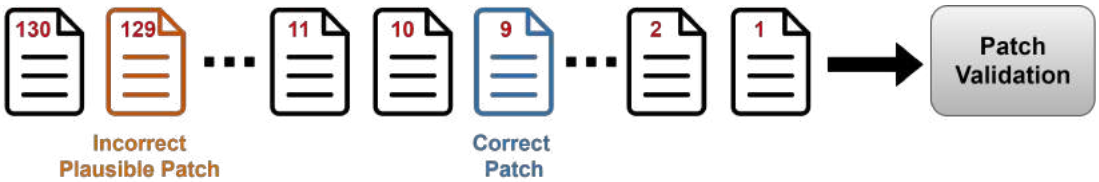

Figure 1.3: Ranking the Correct Patch Higher Through Patch Prioritization

Sorting candidate patches, based on its probability of correctness, is called patch prioritization [11]. It can help to maximize the precision of a program repair technique and fix bugs within the given time limit by ranking the correct patch higher. For example, in Figure 1.3, the correct patch is now ranked higher and before the incorrect plausible patch due to prioritization. To prioritize patches, some information need to be considered such as similarity between faulty code and fixing ingredient or patterns derived from existing patches [4], [5], [10]. The information should be able to minimize the overfitting problem by finding the correct patch before incorrect plausible ones. Furthermore, it should rank the correct solution higher so that it can be found within the allocated time [5].

To the best of the knowledge, for patch prioritization, existing approaches use either patch related attributes or similarity between faulty code and fixing ingredient. SimFix [4] and ssFix [12] use patch related attributes such as number of modifications, whereas ELIXIR [10], CapGen [5] and LSRepair [13] consider similarity between faulty code and fixing ingredient. ELIXIR uses contextual and bug report similarities for capturing syntactic similarity [10]. CapGen uses three models based on genealogical structures, accessed variables and semantic dependencies to measure semantic similarity [5]. LSRepair considers syntactically or semantically similar methods to perform code search [13]. However, all of these techniques limitedly focus on expression level bugs such as method invocation or assignment, although these types of bug are prevailing [14]. The reason is to prevent search space explosion and incorrect plausible patch generation. Additionally, none of these approaches analyze whether patch prioritization can be further improved by integrating the strengths of both similarities. In this context, the current research examines the impact of combining syntactic and semantic similarities on patch prioritization. Next, it incorporates this impact for developing two automated program repair approaches to handle the enlarged search space of expression level bugs as well as identify the correct patch over incorrect plausible ones.

This chapter first discusses the motivation of the research. Based on this discussion, the research questions are formulated and a guideline towards answering these questions is provided. After that, the contribution and achievement of this research are presented. Lastly, the organization of this thesis is mentioned for providing a guideline to the readers.

1.1 Motivation

An expression is a syntactic unit of any programming language that consists of variables, operators or method invocations and evaluates to a single value [15]. The execution behavior of a program is determined by the expressions and therefore bugs are mostly associated with expressions [14]. A study on 16,450 bug fix commits from 6 open-source projects such as Apache Mahout, Solr, etc reported that around 82.40% repair actions are related to expressions [14]. To understand the importance of handling expression level bugs, the current research also analyzes the CodRep dataset [6]. This dataset contains 58,069 one-line replacement bugs and corresponding fixes from 29 projects, for example, Apache Log4j, Spring Framework, etc. For each sample from this dataset, the differences between the buggy and the fixed version files are identified at the expression level. Result demonstrates that 42,856 out of 58,069 bugs (73.80%) are fixed by replacing an expression with another. All these evidences highlight that automated program repair techniques should rigorously deal with expression level bugs.

Although it is important to handle expression level bugs, existing automated program repair approaches [5], [10], [16], [17] either avoid or restrictively work on expression level. To keep the search space tractable, GenProg [17] and RSRepair [18] only work at the statement level [5]. Similarly, fix pattern based program repair approaches PAR [16], ELIXIR [10] and CapGen [5] define only a few templates related to expressions for preventing search space explosion [19]. Although

PAR [16] can modify *conditional expressions*, it cannot solve other frequently occurring expression type bugs such as *class instance creation* expression or *variable name* modification [14]. ELIXIR [10] extensively deals with method invocation related bugs, however, it can not repair *assignment* or *class instance creation* expression bugs, which are also prevalent [14]. Likewise, CapGen [5] cannot handle bugs related to *class instance creation* or *number literal* expression [19]. Other approaches SimFix [4] and LSRepair [13] operate at a coarse granularity, which are a code snippet of 10 lines and method level respectively. Hence, the searching strategy used by these techniques are suitable for identifying only coarse-grained fixing ingredients [20], [21].

It can be seen that none of these existing techniques extensively deal with expression level bugs. This is because working at expression level drastically enlarges the search space [22]. For example, there is only one buggy line in Listing 1.9, however, existing fault localization techniques mark 187 statements from 7 source files as buggy. If fixing ingredients are collected from the corresponding buggy files, 1081 statements are retrieved. Hence, the number of patches generated will be almost 187*1081=202147. On the other hand, if the expression level granularity is used, 591 expressions are labeled as buggy and 4451 expressions are obtained as fixing ingredients. In this case, the number of patches generated will be approximately 591*4451=2630541, which is 13 times larger compared to the statement level granularity. Each of these patches needs to be compiled and validated by executing test cases, which is time consuming [6, 10]. Besides, an existing study found that enhancing the search space increases the probability of generating incorrect plausible patches rather than correct ones [23]. To operate at the expression level granularity, a strategy is required so that the correct patch can be found from the enlarged search space within the allocated time and ranked before incorrect plausible patches.

Listing 1.9: A Sample Bug *Chart_9* from Defects4J Benchmark

```
public TimeSeries createCopy(RegularTimePeriod start,
                             RegularTimePeriod end) {
-    if (endIndex < 0)
+    if ((endIndex < 0) || (endIndex < startIndex))
}
```

Recent program repair approaches [4], [5], [10], [12] have used either syntactic or semantic similarity to effectively navigate the search space. Syntactic similarity focuses on textual alikeness such as similarity in method or variable names. On the other hand, semantic similarity focuses on code meaning such as data type of variables [24]. Approaches such as ELIXIR [10], ssFix [12] and SimFix [4] use syntactic similarity between faulty code and fixing ingredient. ELIXIR uses contextual and bug report similarities for capturing syntactic similarity [10]. To calculate syntax-similarity score, ssFix uses TF-IDF model [12]. For finding top fixing ingredients, syntactically similar to the faulty code, SimFix uses three metrics - structure, variable name and method name similarity [4]. On the other hand, CapGen uses three models based on genealogical structures, for example, ancestors of an abstract syntax tree node, accessed variables and semantic dependencies to measure semantic similarity [5]. Nevertheless, techniques using syntactic similarity such as ELIXIR [10] and ssFix [12] obtain low precision. On the other hand, some semantic similarity metrics such as semantic dependency, suffer from scalability problem [25].

The above discussion indicates that in spite of having limitations, syntactic or semantic similarity is effective in automated program repair. However, the impact of combining the strengths of both similarities has not been explored till now. Listing 1.10, 1.11 and 1.12 show three sample bug fixes from the CodRep dataset. Here, the lines of code started with "+" and "-" indicate the added and

deleted line respectively. It can be seen that the inserted line (fixing ingredient) and the deleted line (faulty code) are syntactically and semantically similar. For example, both the faulty code and the fixing ingredient in Listing 1.11 access the same variable *se*. Therefore, combining syntactic and semantic similarities may improve the patch prioritization and thereby contribute to handle the enlarged search space caused by expression level.

Listing 1.10: A Correct Sample Patch from Project Apache Lucene

```
public void testFilterIndexReader() throws Exception {
-    IndexReader reader = new TestReader(IndexReader.open(directory,
                                true));
+    IndexReader reader = new TestReader(SlowMultiReaderWrapper
                .wrap(IndexReader.open(directory, true)));
}
```

Listing 1.11: A Correct Sample Patch from Project Apache Derby

```
public boolean updateCurrentValueOnDisk( Long oldValue,
           Long newValue ) throws StandardException {
-    if(!se.getMessageId().equals( SQLState.LOCK_TIMEOUT))
+    if(!se.isLockTimeout())
}
```

Listing 1.12: A Correct Sample Patch from Project Apache Ant

```
public Path createClasspath() throws TaskException {
-    this.classpath = new Path(project);
+    this.classpath = new Path(getProject());
}
```

1.2 Research Questions

Based on the discussion presented in Section 1.1, this research aids in automated program repair by answering the following question:

- **RQ:** How does the combination of syntactic and semantic similarity impact automated program repair?

  This research question will be answered by the following sub-questions:

  - **SQ1:** What is the impact of combining syntactic and semantic similarity on patch prioritization?

    For answering this question, a patch prioritization approach that integrates syntactic and semantic similarities can be formulated. To understand the impact of combining similarities, the devised approach can be compared with patch prioritization techniques that use either semantic or syntactic similarity. For comparison, these approaches can be executed on a well-known bug dataset that contains diverse projects. Since the goal is to rank the correct patch higher among all the generated patches, the results can be inspected using patch ranking related metrics. For example, median rank of the correct patch, how much space needs to be searched before finding the correct patch or perfect repair (the percentage of bug fixes for which the correct solution is ranked at first position) [6].

  - **SQ2:** How does the combination of syntactic and semantic similarity contribute to deal with the enlarged search space caused by expressions?

    To answer this question, an automated program repair technique can be devised that works at the expression level. Both syntactic and semantic similarity can be integrated with the approach to limit the search space as well as prioritize the generated patches. Next, the technique

can be executed on various benchmarks that contain large and small buggy projects, for example, Defects4J [26], QuixBugs [27] or Bugs.jar [28]. Lastly, the results can be examined based on the number of bugs correctly fixed, precision and repairing time, as followed by other automated program repair techniques [5], [12], [29].

1.3 Contribution and Achievement

This thesis proposes two automated bug fixing approaches that combine syntactic and semantic similarities to address the research question raised in the previous section. At first, an empirical study on patch prioritization is conducted to analyze the impact of integrating syntactic and semantic similarities. Based on this empirical study, the proposed program repair approaches incorporate both similarities to tackle the enlarged search space caused by expressions and identify the correct patch over incorrect plausible ones. In a nutshell, the major contributions of this thesis are given below:

- **Analyzing the impact of combining syntactic and semantic similarities on patch prioritization:** To the best of the knowledge, the impact of combining syntactic and semantic similarities on patch prioritization has not been explored yet. As a result, this thesis develops two patch prioritization approaches that incorporate both syntactic and semantic similarity between faulty code and fixing ingredient. To measure semantic similarity, genealogical and variable similarity are used since these are good at differentiating between correct and incorrect patches [5]. For capturing syntactic similarity, two popular metrics namely normalized longest common subsequence and token similarity are considered individually [30]. The approaches take source code and faulty line as input and provide a sorted list of patches as output. The patches are sorted using similarity score, calculated by combining ge-

nealogical, variable similarity with normalized longest common subsequence or token similarity.

To observe the combined impact of similarities, the proposed approaches are compared with patch prioritization techniques that use either semantic or syntactic similarity. For comparison, 246 replacement bugs from historical bug fixes dataset [29] are used. Results demonstrate that combining syntactic and semantic similarities helps to rank the correct patch higher compared to techniques using only syntactic or semantic similarity. Using the combination of similarities, almost 96.57% of the total search space can be eliminated to find the correct patch. In 11.79% and 10.16% cases, the combined approaches rank the correct solution at first position. The empirical study is described in detail in Chapter 4.

- **Proposing automated program repair approaches working at the expression level:** In this thesis, two automated program repair approaches are proposed that work at the expression level. Based on the result of the empirical study, the devised techniques integrate syntactic and semantic similarities for constraining the search space and ranking the correct patch over incorrect plausible ones. The approaches take a buggy program, a set of test cases as input and provides a program passing all the test cases as output. At first, faulty *Expression* type nodes are identified and assigned a suspiciousness score using the execution traces of the test cases [31], [32]. Next, patches are generated by replacing the faulty nodes with fixing ingredients that are collected from the corresponding buggy file. These patches are sorted using suspiciousness score of the faulty node and similarity between faulty node and fixing ingredients. The calculation of similarity score is the same as the empirical study. Lastly, the correctness of a patch is examined by executing test cases.

Existing studies [33], [34] found that most of the program repair techniques are evaluated using only Defects4J and biased towards this benchmark. Unlike these approaches, the proposed techniques are executed on both Defects4J [26] and QuixBugs [27] benchmark to ensure generalizability. In addition, the patches generated by the proposed techniques are examined both manually and automatically to assess their correctness, as suggested by [35]. Experimentation result reveals that these techniques can correctly repair expression level bugs from both large and small projects belonging to Defects4J and QuixBugs benchmark. Furthermore, these approaches obtain a precision of 64.71% and 61.76% on these benchmarks, which is higher than baseline program repair techniques that use only syntactic or semantic similarity. The details of the proposed approaches and their evaluation are presented in Chapter 5.

1.4 Organization of the Thesis

This section provides an overview of the subsequent chapters. The chapters are organized in the following way:

- **Chapter 2 Background Study:** This chapter creates the knowledge base for understanding automated program repair. The chapter starts with presenting software bugs and associated terms, for example, faulty code, patch, test cases. Next, terminologies related to source code such as statement, expression, abstract syntax tree, are explained. The chapter ends with describing the importance, steps and challenges of automated program repair.

- **Chapter 3 Literature Review of Automated Program Repair:** In this chapter, existing program repair techniques are classified into two categories namely stochastic patch selection based approaches and patch prioritization based approaches. Based on this classification, the methodology,

strength and weakness of existing automated program repair approaches are discussed.

- **Chapter 4 Impact of Combining Syntactic and Semantic Similarities on Patch Prioritization:** This chapter presents an empirical study on patch prioritization to analyze the impact of integrating syntactic and semantic similarities. At first, the methodology of the study is presented. Next, the implementation details, evaluation criteria and result analysis are discussed. Followed by this discussion, the threats to validity of the study are described. Lastly, the study is summarized.

- **Chapter 5 Combined Similarity Based Automated Program Repair Approaches:** In this chapter, two automated program repair techniques are proposed that focus on expression level. Firstly, the working procedure of these approaches is described. Next, the implementation details, experimentation dataset and evaluation metrics are discussed. Based on this discussion, the obtained results of the study are examined. After that, the threats to validity of the study are reported. At the end of the chapter, the study is summarized.

- **Chapter 6 Conclusion:** In this chapter, the whole thesis is summarized as well as the future work is presented.

Chapter 2

Background Study

Automated program repair is a technique of finding the solution of a bug based on a specification, for example, test cases [1]. To minimize the development and maintenance effort of a software, automated program repair is required [36], [37]. This chapter contains the background information to understand automated program repair. Since automated program repair deals with software bug, at first, bug and its related terms such as faulty code, patch, test cases, are described. Next, terminologies associated with source code, for example, statement, expression, abstract syntax tree, are presented because bugs mostly occur at source code [38]. Lastly, the importance, steps and challenges of automated program repair are discussed.

2.1 Concepts Related to Software Bug

Since automated program repair deals with software bug, a clear concept of bug and associated terminologies (test case, faulty code, patch and fixing ingredient) is necessary before understanding automated program repair. The definitions and examples of those terms as well as their relevance with automated program repair are presented in this section.

2.1.1 Software Bug

Bug is a deviation between the expected and the actual behavior of a program execution [1]. Listing 2.1 shows a method that finds maximum between two integers. It is expected that if a and b are set to 3 and 4 respectively, the method will output 4. However, the method returns 3, which indicates there is a bug. Bugs can occur due to unclear or constantly changing requirements, complexity of a software, errors in programming or documentation, unrealistic timeline, communication gap, etc [39]. Bugs can cause financial loss as well as loss of human lives [38], [40]. For example, in 2001, 28 patients received overdose of radiation due to incorrect calculation provided by a treatment-planning software [41]. Therefore, it is necessary to fix bugs.

Listing 2.1: Buggy Version of *max()* Method

```
int max(int a, int b)
{
    if(a>b)
      return a;
    else
      return a; // buggy statement
}
```

2.1.2 Test Case

Test case is a specification that states a set of inputs, execution conditions and expected outputs [42], [43]. Test cases are developed for a particular objective such as executing a specific program path or detecting bugs. Listing 2.2 presents a sample test case for the method *max()* (shown in Listing 2.1). It will check whether the output of *max(6,5)* is equal to the expected output 6. If these are not

equal, the test case will fail. Test cases can be divided into two categories based on the outcome of a test execution [44]. The categories are:

1. **Passing test case:** A test case for which a program's actual output matches with the expected one, is called a passing test case [45]. For the test case in Listing 2.2, the output of *max(6,5)* is 6, which is the same as expected output. Hence, it is a passing test case.

   Listing 2.2: A Passing Test Case for Method *max()*

   ```
   @Test
   public void test() {
       assertEquals(6, max(6,5));
   }
   ```

2. **Failing test case:** A test case for which a program's actual output is different from the expected output, is called a failing test case [45]. The test case in Listing 2.3 is a failing test case since the expected and actual output do not match. The actual output of *max(3,4)* (shown in Listing 2.1) is 3, whereas the expected output is 4.

   Listing 2.3: A Failing Test Case for Method *max()*

   ```
   @Test
   public void test2() {
       assertEquals(4, max(3,4));
   }
   ```

Apart from buggy source code, automated program repair takes test cases with at least one failing test as input. These test cases are used for two purposes: (1) to identify the buggy source code location and (2) to check the correctness of a generated solution (details are presented in Section 2.3.2).

2.1.3 Faulty Code

Faulty code refers to the source code fragment where the bug resides. For example, in Listing 2.1, line 6 represents the faulty code. The line should be *return b;*, as shown in Listing 2.4. Faulty code can be detected in a number of ways such as program slicing [46] or execution results of a test suite (collection of test cases), which is itself another research area [40].

In automated program repair, faulty code is identified by the execution results of test cases [47]. If a program element (statement or predicate) is frequently executed by the failing test cases and rarely executed by the passing test cases, it is likely to be faulty [48], [49]. Based on this assumption, a suspiciousness score is assigned to the program elements (details are given in Section 2.3.2).

Listing 2.4: Fixed Version of *max()* Method

```
1: int max(int a, int b)
2: {
3:     if(a>b)
4:       return a;
5:     else
6:       return b; // fixed statement
7: }
```

2.1.4 Patch

A patch refers to the modifications applied to a program for fixing a bug [50]. Patches can be generated in numerous ways such as inserting a new statement, replacing or deleting the faulty code fragment. For the bug in Listing 2.1, the correct patch is replacing variable a in line 6 with variable b, as illustrated in Listing 2.4.

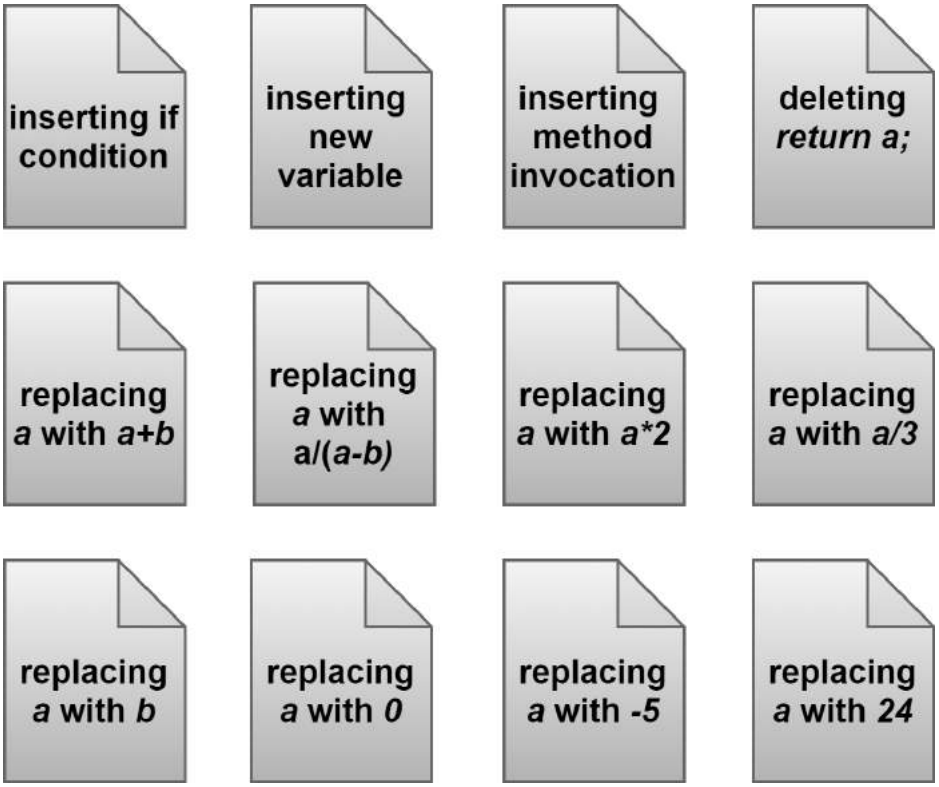

Figure 2.1: Numerous Patches for the Buggy *max()* Method

Automated program repair aims at finding the correct patch of a bug. Since the search space is huge, numerous patches can be generated [4]. Even if the faulty line in Figure 2.1 is known, many patches can be produced, as depicted in Figure 2.1. To limit the number of generated patches, most of the automated program repair techniques rely on redundancy assumption [6]. According to the redundancy assumption, the solution of a bug can be found elsewhere in the application or other projects [6], [7]. This assumption has already been validated by existing studies such as [8], [9]. By conducting a study on six open-source projects such as JUnit, Log4j, Martinez et al. found that 3-17% of the commits are redundant at the line level, whereas it is 29-52% at the token level [8]. Another study on 15,723 commits reported that approximately 30% fixing ingredients (existing code used to fix the bug) exist in the same buggy file [9].

2.1.5 Fixing Ingredient

Fixing ingredient refers to existing code elements that can be reused to generate the patch of a bug [5], [51]. In Listing 2.4, variable a in line 6 is replaced with variable b for generating the correct patch. Here, variable b is the fixing ingredient.

Fixing ingredients can be collected from the buggy source file or the whole project [52]. Considering the whole project increases the chance of including the correct fixing ingredient in the search space. However, it enlarges the search space and thereby makes it difficult to find the correct fixing ingredient earlier [23]. On the other hand, considering only the buggy source file helps to minimize the bug fixing time by finding the correct fixing ingredient earlier. Nevertheless, it may cause unavailability of the correct fixing ingredient in the search space [13].

2.2 Concepts Related to Source Code

Almost 90% of the reported bugs reside in source code [38]. Hence, knowledge of source code related terminologies (statement, expression and abstract syntax tree) is needed before understanding automated program repair. This section describes the definitions and examples of those terms as well as their relevance with automated program repair.

2.2.1 Statement and Expression

Statement is the fundamental unit of execution that denotes some action to carry out [15]. Statements are of various types such as if statement, while statement, break statement, etc. Listing 2.5 displays the statements of *max()* method by marking those as comments.

An expression is a construct made up of variables, operators, or method invocations that evaluates to a single value [15]. Expressions are used in a statement or as part of another expression. In Listing 2.5, $a>b$, *return* a, a, *return* b and b

all are expressions. Automated program repair mostly collects fixing ingredient at statement or expression level to generate patches [5].

Listing 2.5: Statements of *max()* Method

```
1: int max(int a, int b)
2: {
       // if-else statement
3:     if(a>b)
         // return statement
4:       return a;
5:     else
         // return statement
6:       return b;
7: }
```

2.2.2 Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a tree that represents syntactic structure of a source code while hiding details such as parentheses or semicolons [53], [54]. The corresponding AST of the method *max()* (shown in Listing 2.4) is presented in Figure 2.2.

In automated program repair, AST is a way of representing the source code [55], [56]. This AST is traversed to collect fixing ingredients (statements or expressions) [5]. Additionally, patches are generated by modifying the AST [16].

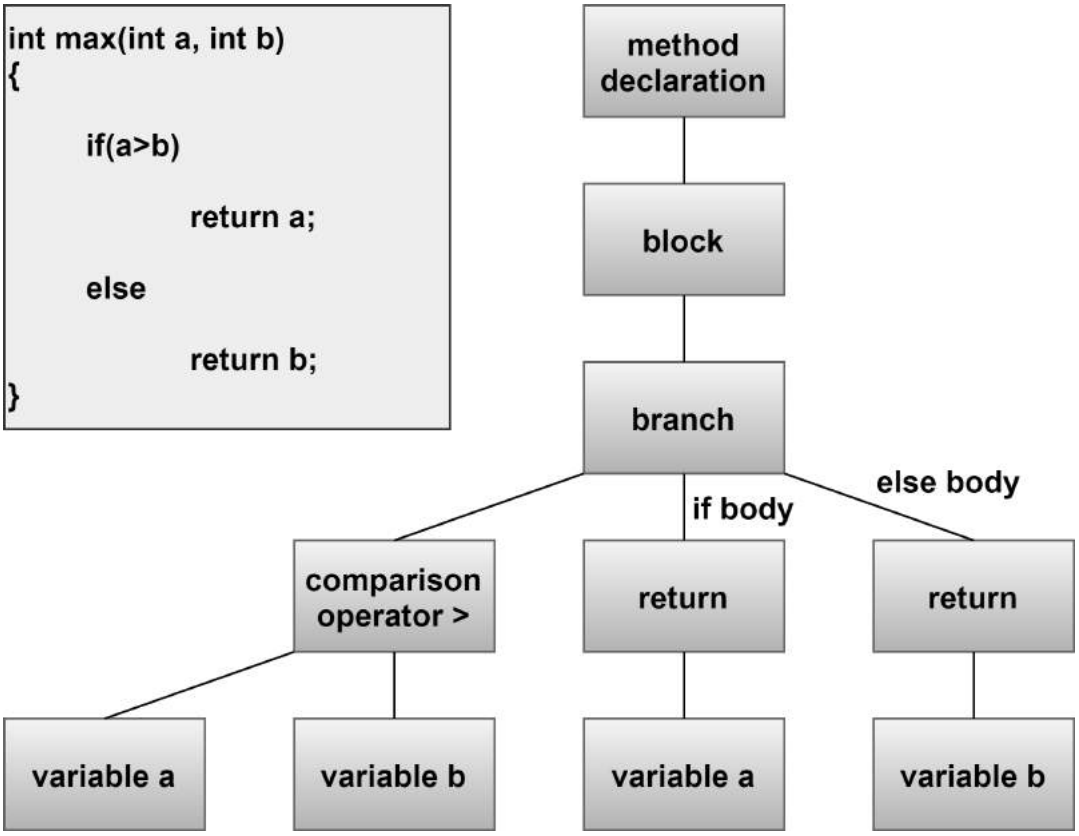

Figure 2.2: Abstract Syntax Tree for *max()* Method

2.3 Automated Program Repair

Automated program repair is a technique of finding the correct patch of a bug based on a specification, for example, test cases [1]. It takes a buggy program and test cases with at least one failing test case as input and generates a program passing all the test cases as output [57], [58]. A sample input of automated program repair, taken from [59], is presented in Listing 2.6 and Listing 2.7. Listing 2.6 shows a buggy code of greatest common denominator (gcd) and Listing 2.7 shows the corresponding test cases. Based on these inputs, automated program repair aims to find the correct patch (presented in Listing 2.8). This section discusses the importance, steps and challenges of automated program repair.

Listing 2.6: Buggy Version of *gcd()* Method

```
int gcd(int a, int b) {
    if(a==0) {
        return a; // buggy statement
    }
    while(b != 0) {
        if(a > b)
            a = a - b;
        else
            b = b - a;
        }
        return a;
}
```

Listing 2.7: Test Cases for *gcd()* Method

```
@Test
public void test() {
    assertEquals(0, gcd(0, 0));
}

@Test
public void test2() {
    assertEquals(2, gcd(0, 2));
}

@Test
public void test3() {
    assertEquals(3, gcd(3, 0));
```

```
}

@Test
public void test4() {
    assertEquals(5, gcd(10, 15));
}

@Test
public void test5() {
    assertEquals(5, gcd(15, 10));
}
```

Listing 2.8: Fixed Version of *gcd()* Method

```
int gcd(int a, int b) {
    if(a==0) {
        return b; // fixed statement
    }
    while(b != 0) {
        if(a > b)
            a = a - b;
        else
            b = b - a;
        }
        return a;
}
```

2.3.1 Importance of Automated Program Repair

The importance of automated program repair is given below:

- **Decreasing the cost of debugging:** Debugging is the process of identifying and resolving bugs [43], [60]. An existing study reported that globally around 312 billion dollars are spent in a year for debugging [61]. Automated program repair has the potential to decrease this cost by reducing the manual effort [62].

- **Saving time:** To solve a bug, at first, a developer needs to understand the problem and identify the faulty source code [22]. Next, he develops patch accordingly and checks whether the test cases pass. However, this process is challenging and time consuming since the developer may need to understand code written by others or third party libraries [22]. In fact, the median time to solve a bug manually is around 200 days [63]. In addition, developers spend approximately 50% of their time on repairing bugs [61]. Automated program repair can greatly reduce this time. Thus, developers can engage themselves in adding new features [64].

- **Fulfilling the inadequacy of human resources:** It has been found that human resources are not enough to solve even known bugs. For example, Windows 2000 was shipped with more than 63000 known bugs due to inadequacy of human resources [16]. A triager from Mozilla stated that everyday almost 300 bugs appear, which is far too many for only the Mozilla programmers to handle [65]. Therefore, automated program repair can be used to fulfill the inadequacy of human resources.

2.3.2 Steps of Automated Program Repair

Most of the automated program repair techniques work in three steps [3], [66], as shown in Figure 2.3. These steps are: fault localization, patch generation and patch validation. The details of these steps are given below:

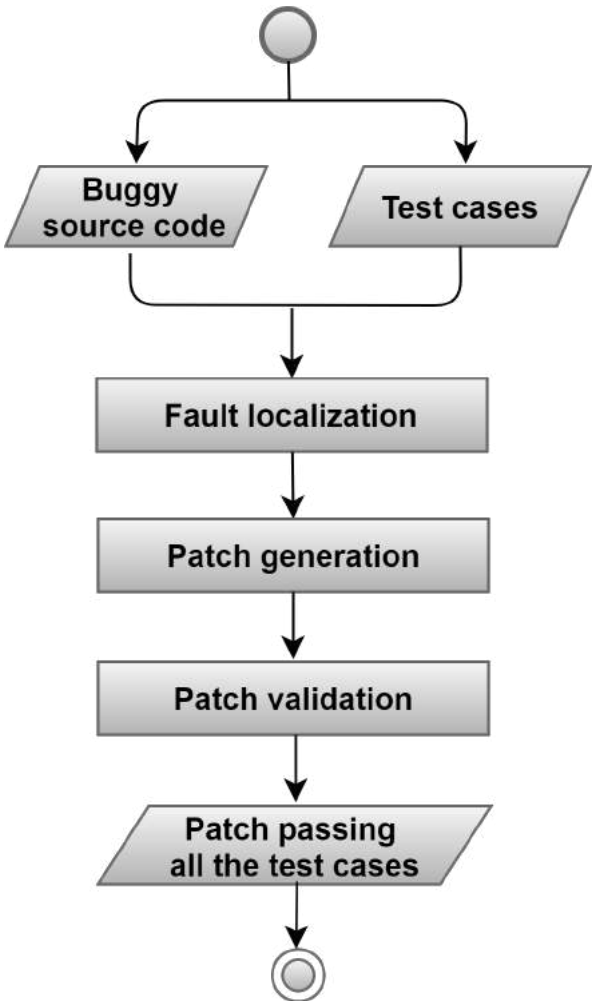

Figure 2.3: Steps of Automated Program Repair

1. **Fault localization:** It identifies the faulty code where the bug resides [3], [67]. The success of a repair depends largely on localizing the fault correctly. Most of the automated program repair approaches use spectrum-based fault localization [31], [40]. A program spectrum is a collection of data, indicating the parts of a program that are active during an execution [68]. Using this spectrum, a ranked list of program elements (statement or predicate) is

Table 2.1: Calculation of Suspiciousness Score for Buggy *gcd()* Program

Line No	**Failing Test**	**Passing Test**				**Suspiciousness Score**
	(0,2)	**(0,0)**	**(3,0)**	**(10,15)**	**(15,10)**	
2	✓	✓	✓	✓	✓	0.45
3	✓	✓				0.71
5			✓	✓	✓	0
6				✓	✓	0
7				✓	✓	0
8				✓	✓	0
9				✓	✓	0
11			✓	✓	✓	0

created. The order of elements in this list denotes their likelihood of being faulty. To calculate the likelihood of faultiness, also called the suspiciousness score, mostly Ochiai metric is used [69], [70]. Ochiai metric outputs suspiciousness score of a statement based on the number of passing and failing tests that execute it. It particularly uses equation (2.1) to compute the suspiciousness score ($suspiciousness(s)$) of a statement s.

$$suspiciousness(s) = \frac{failed(s)}{\sqrt{total\ failed * (failed(s) + passed(s))}} \tag{2.1}$$

Where, $total\ failed$ denotes the total number of failing test cases in the test suite. $failed(s)$ and $passed(s)$ represent the number of failing and passing test cases that execute the statement s.

Table 2.1 shows the suspiciousness score calculation for the buggy *gcd()* program in Listing 2.6 and the corresponding test suite in Listing 2.7. Here, all the statements are represented using line number and their executions are shown as tick marks. For example, the *if statement* in line no 2, is executed by all the test cases. In this example, (0,2) is the failing test case, whereas the other 4 are passing ones. Therefore, the suspiciousness score of the *if statement* is $= 1/\sqrt{1 * (1 + 4)} = 0.45$.

2. **Patch generation:** This step modifies the faulty code identified by fault localization to generate patches [66]. Since the search space is infinite, numerous patches can be generated (discussed in Section 2.1.4). To limit the number of generated patches, mainly two activities are performed, as shown in Figure 2.4. These activities are discussed below:

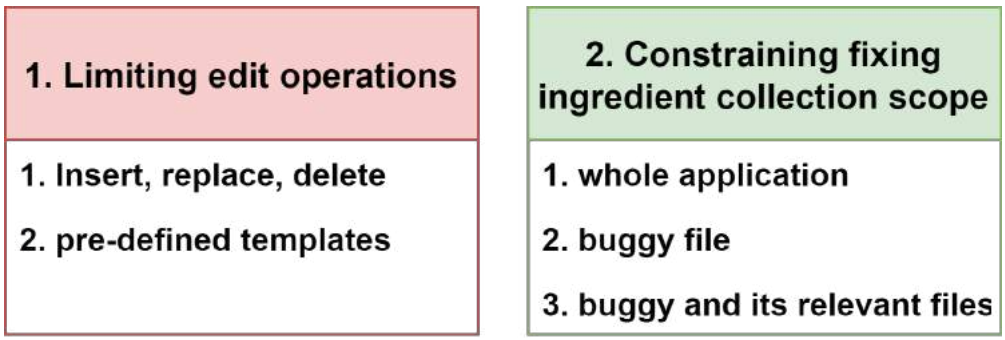

Figure 2.4: Activities to Constrain the Search Space

(a) **Limiting edit operations:** Some approaches such as [17], [18] limit edit operation to insert, replace and delete, whereas other approaches such as [10], [16], [19] use pre-defined fix templates. For example, insert null pointer checker, mutate conditional expression, etc.

(b) **Constraining fixing ingredient collection scope:** To collect fixing ingredients according to redundancy assumption, the following sources are mostly used [52]:

- **Whole application:** [17] and [18] use the whole application for gathering fixing ingredients.
- **Buggy file:** [5], [16] and [19] collect fixing ingredients from only the buggy source file.
- **Buggy and its relevant files:** [10] examines the buggy files as well as its relevant file to collect fixing ingredients. Files whose fields or methods are accessed by the buggy method, are called relevant files.

3. **Patch validation:** This step checks the correctness of a generated patch by executing test cases [66], [71]. If a patch passes all the test cases, it is considered as correct. This step continues until a patch, passing all the test cases, is found or a predefined time-limit (for example, 90 minutes) exceeds [5], [17], [72].

2.3.3 Challenges of Automated Program Repair

Based on prior literature, the challenges faced by automated program repair can broadly be divided into two categories [5], [12], [13]. These are: 1) determining the search space and 2) identifying the correct patch over incorrect plausible ones. These challenges are described below:

1. **Determining the search space:** Since the search space is infinite, automated program repair approaches can produce numerous patches [4]. Nevertheless, search space should be constrained for scaling automated program repair to large projects [73]. On the contrary, this may result into the unavailability of the correct patches in the search space. In such case, it becomes impossible to repair a bug successfully. For example, to repair the bug in Listing 2.9, a method invocation expression *flatten(arr)* in line 14 needs to be replaced with a simple name expression *arr*, as depicted in Listing 2.10. An automated program repair approach that uses only insertion and deletion of statement, cannot fix this bug correctly.

Listing 2.9: Buggy Version of *flatten() Method*

```
1: public static Object flatten(Object arr) {
2:     if (arr instanceof ArrayList) {
3:       ArrayList narr = (ArrayList) arr;
4:       ArrayList result = new ArrayList(50);
5:       for (Object x : narr) {
```

```
6:          if (x instanceof ArrayList) {
7:             result.addAll((ArrayList) flatten(x));
8:          } else {
9:             result.add(flatten(x));
10:           }
11:         }
12:         return result;
13:      } else {
14:         return flatten(arr); // buggy statement
15:      }
16: }
```

2. **Identifying the correct patch over incorrect plausible ones:** A patch that passes all the test cases is called a plausible patch [11], [74], [75]. In real world projects, most often the test cases are unable to completely specify the program behaviors [76], [77]. Due to the weaknesses of the test suite, incorrect plausible patches occur. These incorrect patches pass all the given test cases but fail to be generalized. It is known as overfitting problem [77], [78], [79].

Listing 2.10: Fixed Version of *flatten() Method*

```
1: public static Object flatten(Object arr) {
2:     if (arr instanceof ArrayList) {
3:       ArrayList narr = (ArrayList) arr;
4:       ArrayList result = new ArrayList(50);
5:       for (Object x : narr) {
6:          if (x instanceof ArrayList) {
7:             result.addAll((ArrayList) flatten(x));
```

```
8:          } else {
9:             result.add(flatten(x));
10:           }
11:         }
12:         return result;
13:      } else {
14:         return arr; // fixed statement
15:      }
16: }
```

Listing 2.11 displays an incorrect plausible patch for the bug in Listing 2.9. Both the correct and incorrect plausible patches pass the test suite shown in Figure 2.12. However, when additional test case is supplied, the incorrect plausible patch does not pass. For example, the incorrect plausible patch fails in the test case presented in Listing 2.13. It results in stack overflow error due to infinite recursion. Although correct patches are sparse in the search space, incorrect plausible patches are densely distributed [23]. Thus, it is challenging to find the correct patch over incorrect plausible ones.

Listing 2.11: An Incorrect Plausible Patch for Buggy *flatten() Method*

```
1: public static Object flatten(Object arr) {
2:     if (arr instanceof ArrayList) {
3:       ArrayList narr = (ArrayList) arr;
4:       ArrayList result = new ArrayList(50);
5:       for (Object x : narr) {
6:          if (x instanceof ArrayList) {
7:             result.addAll((ArrayList) flatten(x));
8:          } else {
```

```
9:              result.add(x); // modified statement
10:            }
11:          }
12:          return result;
13:       } else {
14:          return flatten(arr);
15:       }
16: }
```

Listing 2.12: Test Suite for Repairing Buggy *flatten() Method*

```
@Test(timeout = 60000)
public void test1() throws Exception {
    Object result = flatten(new ArrayList(Arrays.asList(1, 4, 6)));
    String resultFormatted = QuixFixOracleHelper.format(result,true);
    assertEquals("[1,4,6]", resultFormatted);
}

@Test(timeout = 60000)
public void test2() throws Exception {
    Object result = flatten(new ArrayList(Arrays.asList("moe", "curly",
                                                        "larry")));
    String resultFormatted = QuixFixOracleHelper.format(result,true);
    assertEquals("[moe,curly,larry]", resultFormatted);
}
```

Listing 2.13: A Test Case that Differentiates between Correct and Incorrect Plausible Patch of *flatten() Method*

```
@Test(timeout = 60000)
public void test3() throws Exception {
    Object object0 = flatten((Object) null);
    assertNull(object0);
}
```

2.4 Summary

This chapter creates the knowledge base for understanding automated program repair. It describes frequently used terminologies in automated program repair such as bug, patch, etc as well as the importance, steps and challenges of automated program repair. A bug denotes deviation between the expected and the actual behavior of a program execution [1]. The changes applied to a program for fixing a bug, is called a patch [50]. Automated program repair aims at finding the correct patch using a specification, for example, test cases. It has the potential to reduce the bug fixing time and cost [62].

Given a buggy program and a test suite, automated program repair outputs a program passing all the test cases [22]. Firstly, the faulty code is identified. Next, patches are generated by modifying the faulty code. Lastly, the correctness of a patch is validated by executing test cases. To increase the probability of including correct patches, the search space needs to be enlarged. However, enhancing the search space increases the chance of generating incorrect plausible patches [5]. Consequently, it becomes challenging for automated program repair to find the correct patch over incorrect plausible ones. Based on the concepts described in this chapter, next the related work on automated program repair will be reviewed.

Chapter 3

Literature Review on Automated Program Repair

Automated program repair aims at finding the correct patch of a bug using a specification, for example, test cases [1]. Recently, it has drawn the attention of researchers due to its potentiality of minimizing debugging effort [62], [80], [81]. Automated program repair techniques generate patches based on a heuristic such as random mutation and validate patches until a patch passing all the tests is found or resource limit is reached [82]. Existing automated program repair approaches can be broadly divided into two categories based on the patch selection mechanism. The first category is stochastic patch selection based approaches [16], [17], [18], [59]. These approaches generate and select patches for validation using randomized algorithm such as genetic programming [83]. The second category is patch prioritization based approaches [4], [5], [10]. These techniques rank and select patches for validation based on their likelihood of correctness [11]. Each of these approaches are described in detail in the subsequent sections.

3.1 Stochastic Patch Selection Based Approaches

Stochastic patch selection based approaches randomly choose the mutation operator and fixing ingredients for patch generation. Techniques such as GenProg [17], PAR [16], RSRepair [18], HDRepair [29] fall into this category, which are discussed in the following subsections.

3.1.1 GenProg

Le et al. proposed GenProg, which is the first generic automated program repair approach[17], [59], [84]. Techniques before GenProg deal with a specific type of bug such as buffer overflow, flaws exploited by worms, etc. To tackle their limitation, GenProg uses genetic programming to guide the repairing process. It takes a buggy program, a set of positive and negative test cases as input. It generates a program variant passing all test cases as output.

GenProg represents each program variant as a pair of an AST and a weighted path. The weighted path includes the statements that execute during negative test cases along with their weights. If a statement executes during negative test cases only, it is assigned a weight of 1. If a statement executes during both positive and negative test cases, it gets a weight of 0.1. At first, the initial population is generated by mutating the buggy program. Earlier, GenProg randomly performed insertion, deletion or swapping mutation on statements [17]. Later, it is modified to perform replacement mutation instead of swapping [84]. During mutation, it reuses statements from the buggy program.

After generating the initial population, the fitness of each variant is calculated to select population for the next generation. GenProg calculates fitness using the weighted average of the positive and negative test cases. Based on this fitness, 50% variants from the previous iteration are selected for the next generation. Next, the selected variants are crossed over to generate the other 50% members of the next

generation. During cross over, two variants exchange all the statements based on a cutoff point. The cutoff point is selected from the weighted path. After cross over, all the variants go through mutation. All these steps (calculating fitness, cross over, mutation) continue until a variant passing all test cases is found or a predefined time limit exceeds. The variant that passes all the test cases is known as primary repair. Due to genetic programming, the primary repair can contain additional changes that are not required to fix the bug. To eliminate those changes, GenProg applies structural differencing [85] and delta debugging techniques [86] on the primary repair.

GenProg is the first to introduce the redundancy assumption (bug can be repaired using existing code). Through this assumption, GenProg is able to constrain the infinite search space of automated program repair. However, GenProg collects fixing ingredients at statement level, which is too coarse-grained to find the correct ones in the search space [5]. Furthermore, GenProg randomly selects the mutation operator and fixing ingredients. As a result, most of the patches generated by GenProg are incorrect plausible patches [74]. Therefore, it obtains lower precision namely 15% on Defects4J benchmark [82].

3.1.2 PAR

Kim et al. proposed the first template-based automated program repair approach named PAR that uses fix patterns learned from existing human-written patches [16]. They found that GenProg can generate nonsensical patches due to randomness in mutation. For example, GenProg can replace a method invocation that causes the error with another random statement. To address this problem, the authors manually inspected 62,656 human-written patches from Eclipse JDT project and identified common fix patterns. At first, they divided the patches into three groups based on the modifications (insert/update/delete). Next, they examined the root cause of the bug and corresponding changes made to the program for fix-

ing the bug. From the analysis, they found ten common patterns such as adding a null checker, changing method parameters, that cover approximately 30% of all the patches. Based on these patterns, the authors defined ten templates such as "Null Pointer Checker", "Parameter Replacer", for generating patches.

Similar to GenProg [17], PAR uses genetic programming for repairing bugs. At first, PAR identifies the faulty statement using the same fault localization technique as GenProg. Next, it generates patches by modifying the faulty locations using those predefined templates. For a faulty location, PAR checks which templates can be applied to it by inspecting its context. For example, to verify whether "Null Pointer Checker" can be applied to a faulty location, PAR searches for an object reference in the faulty location context. If multiple templates can be applied to a faulty location, PAR randomly chooses one template and generates a patch using fixing ingredients from the corresponding buggy source file. If multiple fixing ingredients can be applied to the faulty location, PAR randomly selects one. Lastly, the fitness of the generated patch is calculated using the weighted average of the positive and negative test cases, as followed in GenProg. Based on the fitness, patches are selected for the next generation through tournament selection. These steps (patch generation, fitness calculation) continue until a program variant passing all test cases is found or predefined condition is met, for example, the maximum number of generations. If a program variant passes all the test cases, PAR considers it as the solution.

PAR is the first approach to incorporate existing human-written patches to solve bugs. The authors conducted an experiment on six open source projects such as Eclipse, Apache Log4j, etc and showed that PAR can repair 27 out of 119 bugs. Later, a study by Monperrus found that only 3 out of the 10 templates are prevailing [87]. Most of the bugs are fixed using the "Null Pointer Checker", "Expression Adder, Remover" and "Expression Replacer" template. Besides, PAR randomly selects the template and corresponding fixing ingredient. Consequently,

most of the patches generated by PAR are incorrect plausible ones. For example, the precision of PAR is only 15.9% on Defects4J benchmark [82].

3.1.3 RSRepair

Qi et al. proposed RSRepair, a random search based automated program repair technique [18]. After GenProg [84], researchers proposed various automated program repair techniques such as TrpAutoRepair [71], SemFix [88], and justified those by comparing with the baseline approach GenProg [1]. Both GenProg [84] and PAR [16] achieved promising results in repairing bugs by using genetic programming. However, to what extent genetic programming helps these two approaches in finding the correct patch, was still unknown. To address this question, RSRepair was developed and compared with GenProg.

RSRepair combines random search with test case prioritization technique to guide the bug fixing process. Similar to GenProg, it takes a buggy program and a set of positive and negative test cases as input. The test cases are sorted based on their ability to detect invalid patches. Initially, negative test cases are assigned a value of 1 (called the *index value*) and ranked first. On the other hand, the positive test cases get an *index value* of 0. These values are continuously incremented based on the number of invalid patches detected. At first, RSRepair localizes the faulty statements using the same method as GenProg. Next, a patch is generated by mutating the buggy program using redundancy assumption. For mutation, RSRepair uses the same operators as GenProg (insert, replace and delete statements), since those are proven to be effective in repairing programs. To validate a patch, it is sequentially run against the prioritized test cases. If a test case invalidates the patch, its *index value* is increased by 1 and the test cases are reordered. If two test cases have the same *index value*, RSRepair orders those randomly. This process is repeated until a plausible patch is found or the predefined time limit exceeds.

Through experimentation on 24 bugs such as wireshark, python bug, the authors showed that RSRepair generates fewer patches than GenProg before finding the correct patch. In addition, it can accelerate the patch validation step through test case prioritization. Nevertheless, Liu et al. found that it can correctly fix only 2 out of 395 bugs from Defects4J, whereas GenProg can correctly fix 3 bugs [82]. Similar to GenProg, it cannot solve bugs requiring fix at a finer granularity (expression level) as it works at statement level. Additionally, it obtains a precision of 4.9%, which indicates that it can not differentiate between correct and incorrect plausible patches.

3.1.4 HDRepair

Le et al. proposed HDRepair, the first repair technique to incorporate historical bug-fix patterns for patch selection [29]. These patterns help to filter-out illogical patches, for example, deleting throw statement to pass a test case. To generate these patterns, the authors collected fixes of 3000 bugs across 700+ large, popular GitHub projects such as Apache Tomcat, Google Guava, etc. Next, these bug-fixes are represented as graphs to obtain common and abstract formats. Figure 3.1 shows the graph-based representation of method call parameter replacement type bug. Here, the edge denotes the type of change (insert/update/delete) occurred. The parent and child node indicate the context and location of the change respectively. From these graphs, frequently occurring patterns were detected based on a threshold.

At first, HDRepair applies a single edit operation to the buggy program for generating the initial population. It uses twelve mutation operators taken from mutation testing or existing program repair techniques GenProg [84] and PAR [16] to generate patches. For example, replacing a statement, boolean negation, etc. At each iteration, HDRepair randomly selects the faulty locations and corresponding mutation operators for generating an intermediate pool of candidate

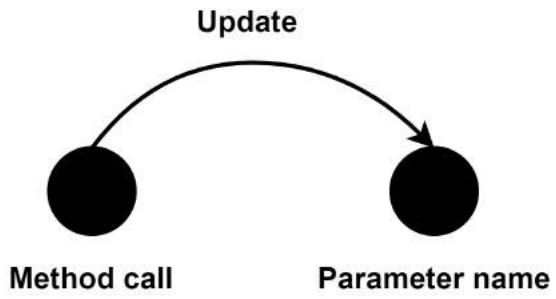

Figure 3.1: Graph-based Representation of A Bug Fix

patches. From this pool, patches that occur frequently in the bug-fix history are selected. To calculate the recurrence of a patch, HDRepair averages the frequencies of all the edit operations. Next, the selected patches are validated against the failing test cases. If a patch passes all the failing test cases, it is added to a set of possible solutions. Lastly, a predefined number of possible patches, ranked by their frequency in the historical bug-fixes, are presented to the developer.

By executing HDRepair on Defects4J benchmark [26], the authors reported that it can solve 23 out of 90 bugs, which was higher than prior techniques GenProg [84], PAR [16]. The reason is HDRepair uses a rich set of mutation operators and historical bug-fix patterns to guide the patch generation process. However, later, it was found that only 6 out of the 23 fixes are actually correct [19]. Thus, it obtains a precision of 26.09%, which indicates considering only historical bug-fix patterns is not enough to eliminate incorrect plausible patches. Furthermore, HDRepair assumes that the faulty method is known, which may not always be true [3]. Simlar to GenProg [17] and RSRepair [18], HDRepair cannot solve bugs requiring fix at a finer granularity, such as expression level, due to the design of the mutation operators.

3.2 Patch Prioritization Based Approaches

Patch prioritization based approaches rank and select patches based on their probability of correctness so that potentially correct patches can be validated earlier [11]. These techniques can be divided into two categories based on the information used for prioritization, as shown in Figure 3.2:

1. Approaches using characteristics of generated patches
2. Approaches using similarity between faulty code and fixing ingredient

In the following subsections, these two type of approaches are described in detail.

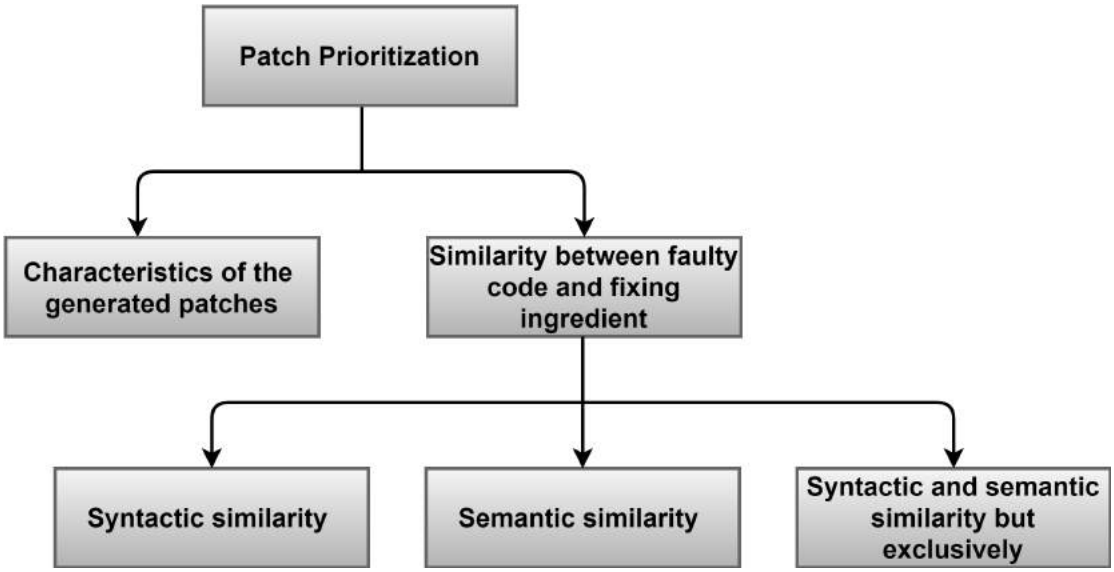

Figure 3.2: Classification of Patch Prioritization Based Approaches

3.2.1 Approaches Using Characteristics of Generated Patches

These techniques use various characteristics of the generated patches for prioritization, for example, number of modifications. ssFix [12] and SimFix [4] belong to this category. To generate patches, ssFix uses TF-IDF model, whereas SimFix considers three syntactic similarity metrics- structure, variable name and method

name similarity. Next, patches are ranked based on their characteristics. For example, the fewer modifications a patch contains, the higher the rank is. The details of these techniques are presented in the following subsections.

3.2.1.1 ssFix

Xin et al. proposed ssFix, the first technique to perform syntactic code search from a codebase containing the faulty program and other projects [12]. A prior study revealed that test cases alone are not enough to find the correct solution effectively. Another study found that 69% repairing code lines can be obtained from local or non-local programs. Two techniques namely Code Phage [89] and SearchRepair [45] perform semantic code search to obtain those repairing lines. However, semantic code search is found to be expensive. Hence, ssFix was proposed to analyse the feasibility and effectiveness of syntactic code search.

At first, ssFix identifies the faulty statement using both stack trace (if available) and spectrum-based fault localization [40]. Next, it extracts the faulty location along with its context using a LOC based algorithm (approximately 6 lines). It is called target chunk (*tchunk*). A similar process is followed to retrieve fixing ingredients and their contexts from the codebase. These are called candidate chunks (*cchunks*). The *tchunk* and *cchunks* are tokenized after masking project-specific code, for example, variable name, literals etc. Next, Lucene's TF-IDF model [1] is used to obtain *cchunks*, syntactically similar to the *tchunk*. These *cchunks* are sorted based on their syntax-relatedness to the *tchunk*. Currently, ssFix uses utmost top 100 *cchunks* for generating patches. For a *cchunk*, it maps and renames the identifiers to those in *tchunk*. Next, differences between these chunks are identified by matching their components (statements and expressions). Based on these differences, patches are generated by modifying the *tchunk*. These patches are next prioritized using the type and size of modifications. For exam-

[1] `https://lucene.apache.org/`

ple, patches generated by replacement or insertion are ranked higher than those generated by deletion.

ssFix can repair 20 out of 357 bugs from the Defects4J benchmark [26]. The median time to fix these bugs is 11 minutes, which is lower than most of the prior techniques GenProg [84], HDRepair [29]. Results further show that ssFix can repair multi-line bugs. However, it obtains only 33.33% precision which is lower compared to other techniques [4], [5], [10]. It indicates that the code search technique needs to be improved further to differentiate between correct and incorrect plausible solutions.

3.2.1.2 SimFix

Jiang et al. proposed SimFix, the first approach that combines both existing human-written patches and similar code to guide the bug fixing process [4]. Existing patches help to detect patches with the most likely bug fixing modifications. On the other hand, similar code helps to identify suitable patches based on the current project's perspective. SimFix uses the common patches derived from these two sources for repairing bugs.

SimFix consists of two stages namely mining and repairing. In the mining stage, frequently occurring abstract modifications are obtained from existing human-written patches. In the repairing stage, at first, the faulty statements are identified using Ochiai metric. To improve the accuracy of the fault localization step, test cases are purified. To include the context of the faulty statement, SimFix converts it to a code snippet of 10 lines, which is called *faulty snippet*. The same process is repeated on all the source files to retrieve fixing ingredients and their contexts, which are called *donor snippets*. Next, similarity between the *faulty snippet* and each *donor snippet* is measured using three metrics - structure, variable name and method name similarity. Structure similarity extracts a list of features related to AST node, such as number of *if* or *for* statements. Variable name simi-

larity tokenizes variable names, for example, splitting studentID into student and ID, and calculates similarity using Dice coefficient [90]. Method name similarity follows the same process as variable name similarity.

To generate patches, SimFix selects top 100 *donor snippets* based on the similarity score. The variables of the faulty snippet and the donor snippets are mapped based on name, type and usage similarity. Next, the faulty snippet and the donor snippets are compared to identify the modifications required to transform the faulty snippet into the donor snippets. To further limit the search space, only patches with frequently occurring modifications are considered. Lastly, these patches are sorted based on their characteristics such as number and type of modifications (insertion/replacement) and validated sequentially. For example, patches with fewer modifications are ranked higher and thereby validated earlier.

By evaluating SimFix on Defects4J benchmark [26], authors showed that it can correctly fix 34 of out 357 bugs, which is higher than all other techniques [5], [10], [12]. Besides, SimFix is capable of repairing multi-line bugs. However, SimFix yields low precision which is 60.70%. Besides, it sometimes fails to find fine-grained fixing ingredients such as *expressions* since it identifies donor snippets at a coarse-grain (10 lines) [20]. After executing SimFix on 192 additional bugs from 8 open-source projects such as Apache Commons CLI, jsoup, Ghanbari et al. found that SimFix can not repair any of the bugs since it can not find donor snippets [34]. All these demonstrate that the granularity and similarity metrics (structure, variable name and method name similarity) for identifying donor snippets need to be further improved.

3.2.2 Approaches Using Similarity Between Faulty Code and Fixing Ingredient

These techniques use similarity between faulty code and fixing ingredient for ranking patches. This category can further be classified into three groups based on the similarity used, as depicted in Figure 3.2. ELIXIR and CapGen respectively use syntactic and semantic similarity between faulty code and fixing ingredient. On the other hand, LSRepair exclusively considers syntactic and semantic similarity. These approaches are described in the following subsections.

3.2.2.1 ELIXIR

Saha et al. proposed ELIXIR, the first object-oriented program repair technique that makes use of extensive method invocation [10]. An empirical study on three popular Java projects revealed that around 77% of one-line bugs occurred during the lifetime of these projects were solved by modifying or inserting a method invocation. However, approaches before ELIXIR incorporated method invocation in very limited ways to avoid search space explosion problem. ELIXIR uses eight templates including modification and insertion of method invocation for generating patches. It uses four features to prioritize patches:

1. **Distance:** The closer a fixing element is to the repair location, the more relevant it is. ELIXIR measures the distance between an element (E) and the repair location (R) using equation (3.1).

$$distance = \begin{cases} 1 - \frac{ld(E,R)}{len(m)}, & \text{if } len(m) \geq ld(E,R). \\ 0, & \text{otherwise.} \end{cases} \tag{3.1}$$

where, $ld(E, R)$ denotes the minimum number of comment-free lines between E and R. $len(m)$ represents the number of comment-free lines in the faulty

method. If a fixing ingredient contains multiple elements, ELIXIR averages the distances.

2. **Contextual similarity:** The more a fixing ingredient is syntactically similar to the faulty location context (code surrounding the faulty location), the more relevant it is. ELIXIR sets the length of faulty location context to six lines. It extracts and tokenizes all the identifiers [91] used by the fixing ingredient and the faulty location context. Next, contextual similarity is measured using equation (3.2).

$$contextual\ similarity = \frac{|S1 \cap S2|}{|S1 \cup S2|} \tag{3.2}$$

   where, $S1$ and $S2$ represent tokens used by the fixing ingredient and the faulty location context respectively.

3. **Frequency in the context:** If a fixing ingredient consists of frequently used elements, it is more relevant in the context. Correlation-based Feature Subset Selection revealed that only variables and objects are co-related with frequency. Therefore, ELIXIR averages only the frequency of variables and objects for multi-element fixing ingredients.

4. **Bug report similarity:** It calculates the textual similarity between fixing element and bug report. ELIXIR measures bug report similarity using the same formula as equation (3.2), where, $S1$ and $S2$ represent tokens used by the fixing ingredient and the bug report respectively.

For assigning different weights to these features, logistic regression model is used. The model was trained using Bugs.jar dataset [28]. ELIXIR validates only the top 50 patches generated from each template. It can correctly repair 26 out of 82 bugs from Defects4J benchmark [26], which was the highest compared to contemporary techniques [12], [29], [78]. Although ELIXIR can handle *method invocation*

expression, it can handle other *expression* type bugs in a limited way due to the design of the templates [92]. For example, it can not repair *assignment* or *class instance creation* expression bugs, which are also prevalent in Java programs [14]. Similar to ssFix and SimFix, ELIXIR obtains a low precision of 63.41%, which indicates the ranking features need to be further improved to alleviate incorrect plausible patches.

3.2.2.2 CapGen

Wen et al. proposed CapGen that incorporates context information while selecting mutation operators and prioritizing fixing ingredients [5]. The context-aware mutation operators help to constrain the search space by limiting the number of patches generated. On the other hand, patch prioritization based on context similarity helps to detect correct patch over incorrect plausible ones. CapGen works on AST node level since finer granularity increases the probability of including the correct solution in the search space. After identifying the faulty nodes using GZoltar tool [93], CapGen selects the mutation operators that can be applied to the faulty nodes. It defines 30 mutation operators, for example, insert *Expression* statement under *if* statement to generate patches. These operators are derived from a dataset of 3000 real bugs from 700 open-source projects [29]. To prioritize patches, CapGen uses following three context similarity based models that mainly focus on semantic similarity between faulty node and fixing ingredient:

1. **Genealogy context:** It checks whether faulty node and fixing ingredient are frequently used with the same type of code elements. For example, inside *while* loop. To extract the genealogy contexts of a node, CapGen inspects the type of its ancestor and sibling nodes through AST traversal. The ancestors are traversed up to method declaration. For sibling nodes, *Expressions* or *Statements* type nodes within the same block of the specified node are examined.

2. **Variable context:** Variables accessed by a node provides useful information as these are the primary components of a code element. To measure variable similarity, CapGen generates two lists, θ_T and θ_S, containing names and types of variables used by the faulty node and the fixing ingredient respectively. Next, variable similarity is calculated using equation (3.3).

$$g(\theta_S, \theta_T) = |\theta_S| * \frac{|\theta_S \cap \theta_T|}{|\theta_S \cup \theta_T|} \tag{3.3}$$

CapGen considers two variables equal if their names and types are exact match. Here, multiplication by $|\theta_S|$ is done so that more priority can be given to complex elements (fixing ingredients with more variables). It helps to avoid generating plausible patches.

3. **Dependency context:** Dependency context helps to identify the nodes that are affected by a specified node and vice versa. To model dependency context, CapGen conducts intra-procedure forward and backward slicing [46] based on the variables accessed by a node. Through slicing, nodes of type *Expression* or *Statement* are collected. Next, the type of each node is analyzed and the frequency of different types of nodes are stored. The same procedure is repeated for the faulty node and the fixing ingredient. Lastly, dependency score is calculated based on equation (3.4).

$$f(\psi S, \psi T) = \frac{\sum_{t \in K} min(\psi T(t), \psi S(t))}{\sum_{t \in K} \psi T(t)} \tag{3.4}$$

Where, ψT and ψS represent the frequencies of different node types for faulty node and fixing ingredient respectively. K denotes set of distinct node types captured by the faulty node ψT.

Through experimentation on 224 bugs from Defects4J benchmark, the authors reported that CapGen can correctly repair 22 bugs and achieves a precision of

84%, which is the highest among existing repair techniques. Later, another study by Ye et al. marked 1 of these 22 patches as incorrect plausible using automated patch assessment approach [35]. Although the number of bugs correctly fixed by CapGen is lower compared to other techniques [4], [10], the precision indicates that the context-aware mutation operators and the context similarity models are good at differentiating between correct and incorrect plausible patches. Nevertheless, CapGen assumes that the faulty package is known, which is another reason behind its high precision. This assumption helps to constrain the search space and minimize the generation of incorrect plausible patches by preventing modification of incorrect faulty locations. For example, when the faulty package is not given, CapGen generates 14 plausible patches for Closure project from Defects4J [34]. Besides, Ghanbari et al. executed CapGen on 192 additional bugs from 8 open-source projects such as Apache Commons CLI, jsoup, etc and found that the mutation operators used by CapGen are ineffective for these bugs [34]. Furthermore, CapGen relies on program dependency graph to calculate dependency context which does not scale to moderate-size programs [25].

3.2.2.3 LSRepair

Liu et al. proposed LSRepair, a pilot technique to perform both syntactic and semantic code search to collect fixing ingredients [13]. For searching, 11,043,044 methods from 10,449 Java GitHub projects are used. As considering both similarities enlarges the search space, LSRepair works at method level to limit the space. It identifies the faulty method using spectrum-based fault localization technique [40]. Next, it generates patches by sequentially considering the following three types of similarities:

1. **Signature-similar methods:** To find methods having a similar signature like the buggy method, LSRepair matches the return types, names and argument types of the methods. After identifying similar methods, those are

sorted in ascending order using levenshtein distance. Levenshtein distance measures the minimum number of edits required for transforming the similar signature method into the buggy method [94].

2. **Syntactically similar code:** To identify syntactically similar code, LSRepair converts method bodies into feature vectors using Convolutional Neural Networks (CNNs). Next, methods are ranked in descending order using cosine similarity [90] between their and buggy method's feature vector.

3. **Semantically similar code:** For retrieving semantically similar code, LSRepair relies on FaCoY search engine [95]. FaCoY takes the buggy method as input and generates a query summarizing the structural code elements of the method. Next, it searches on StackOverflow to find methods having a similar descriptions and outputs a ranked list accordingly.

At first, LSRepair searches for methods with signature similarity. Next, patches are generated by replacing the buggy method with transformed versions of the similar methods. For example, replacing variables of the similar method with those of the buggy method. If this strategy fails to generate a patch passing all the test cases, LSRepair looks for methods with syntactic similarity. It focuses on 4 types of buggy statements, called the *pivot statement*, to generate patches using syntactically similar methods. These are *if*, *return*, *variable declaration* and *expression* statement. When the pivot statement is an *If* statement, LSRepair examines the differences between the buggy and syntactically similar method in terms of conditional expression and operator. For example, if the buggy and syntactically similar method contain the same type expression $(a > b)$ and $(c < d)$ respectively, the patch would be replacing $>$ with $<$. For other types of *pivot statements*, LSRepair looks for an associated *if* statement around it. If the buggy method contains an *if* statement but the similar method does not, patch is generated by deleting the statement. If similar method contains an *if* statement but the buggy method does

not, patch is generated by inserting an if statement. In this case, the variables of similar method are replaced with those of the buggy method. If syntactically similar methods also fail to generate a patch passing all the test cases, methods with semantic similarity are tried using the same process.

LSRepair can correctly fix 19 out of 395 bugs from Defects4J benchmark [26]. Among these bugs, 10 were not fixed by any automated program repair technique before LSRepair. It indicates that LSRepair is complementary to other approaches. However, due to working at method level, it cannot fix bugs occurring outside method body such as *field declaration* related bugs. In addition, it achieves a precision of 51.35%, which is lower than ELIXIR [10], CapGen [5] and SimFix [4]. Therefore, the similar code searching techniques need to be improved further for identifying correct patch over plausible incorrect patches.

3.3 Summary

This chapter discusses the existing approaches of automated program repair. These approaches can be broadly classified into two categories based on the patch selection mechanism. The first category is stochastic patch selection based approaches. GenProg [17], PAR [16], RSRepair [18], HDRepair [29] are the representatives of this category. These techniques generate and select patches for validation using a randomized algorithm [83]. For example, GenProg [84] and PAR [16] use genetic programming to find the correct patch. GenProg randomly modifies the faulty code using three mutation operators (insert, replace, delete) [84]. PAR uses ten pre-defined templates such as null pointer checker, to generate patches [16]. RSRepair randomly searches among the candidate patches to find the correct one [18]. HDRepair uses frequently occurring bug-fix patterns for patch generation [29]. Nevertheless, most of the patches generated by these techniques are incorrect plausible ones [3], [74].

The second category is patch prioritization based approaches. To validate potentially correct patches earlier, these techniques rank and select patches based on their probability of correctness [11]. All of these techniques use either syntactic or semantic similarity as a part of the repairing process. ELIXIR [10], ssFix [12], SimFix [4] use syntactic similarity between faulty code and fixing ingredient. CapGen [5] uses semantic similarity between faulty code and fixing ingredient. LSRepair [13] separately considers both syntactic and semantic similarity to perform code search. However, techniques using syntactic similarity such as ELIXIR [10] and ssFix [12] yield low precision. On the other hand, some semantic similarity metrics such as semantic dependency, suffer from scalability problem [25]. Besides, none of these approaches analyze the impact of integrating the strengths of both similarities to prioritize patches. The following chapter presents an empirical study on patch prioritization to analyze the impact of combining syntactic and semantic similarities.

Chapter 4

Impact of Combining Syntactic and Semantic Similarities on Patch Prioritization

Patch prioritization means sorting candidate patches based on the probability of correctness [11]. It helps to minimize the bug fixing time and maximize the precision of an automated program repair technique by ranking the correct solution before incorrect one. Recent program repair approaches [5], [10], [13] have used either syntactic or semantic similarity between faulty code and fixing ingredient to prioritize patches, as discussed in the previous chapter. However, combining both similarities may improve patch prioritization by ranking the correct solution higher. For this purpose, two patch prioritization approaches that integrate syntactic and semantic similarity, are proposed in this chapter. To understand the combined impact of similarities, the proposed approaches are compared with patch prioritization techniques that use either semantic or syntactic similarity. Results show that combined methods outperform semantic and syntactic similarity based approaches, in terms of median rank of the correct patch and average search space reduction.

4.1 Introduction

Recent automated program repair techniques have used patch prioritization to find the correct patch over incorrect plausible ones and fix bugs within the allocated time limit [23]. These approaches can be divided into two categories based on the information used for prioritization. The first category [4], [12] uses patch related attributes, for example, the number of modifications. One of such approaches ssFix performs syntactic code search from a codebase containing the faulty program and other projects [12]. To generate patches, it ranks fixing ingredients based on its syntax-relatedness to the faulty code, calculated using TF-IDF. Another approach, SimFix considers three metrics - structure, variable name and method name similarity to collect syntactically similar fixing ingredients [4]. To further limit the search space, only fixing ingredients found frequently in existing human-written patches are considered for generating patches. Both ssFix and SimFix prioritize patches based on their characteristics. For example, the fewer modifications a patch contains, the higher it is ranked. However, these approaches yield low precision, which is 33.33% and 60.70% respectively.

The second category uses similarity between faulty code and fixing ingredient to prioritize patches. ELIXIR [10], CapGen [5] and LSRepair [13] fall into this category. ELIXIR uses eight templates for generating patches such as checking array range [10]. It uses four features including contextual and bug report similarities to prioritize patches. Contextual and bug report similarity compare fixing ingredients with the faulty location's context and bug report respectively to measure syntactic similarity. Although ELIXIR can repair more bugs compared to contemporary techniques [12], [78], it obtains 63.41% precision. It indicates that ELIXIR is not good at differentiating between correct and incorrect patches. CapGen defines 30 mutation operators such as replacing conditional expression to generate patches [5]. It uses three models based on genealogical structures, ac-

cessed variables and semantic dependencies to capture context similarities at AST node level. These models mainly focus on semantic similarities between faulty code and fixing element to prioritize patches. The precision of this approach is higher (84.00%), however, it relies on the program dependency graph to calculate semantic dependency which does not scale to even moderate-size programs [25]. Another technique, LSRepair performs both syntactic and semantic code search to collect fixing ingredients [13]. To limit the search space, LSRepair works at the method level. At first, it looks for methods that are syntactically similar to the faulty method. If this strategy fails, LSRepair searches for semantically similar methods. Nevertheless, this approach also obtains low precision which is 51.35%.

The above discussion indicates that although syntactic or semantic similarity has limitations, these are effective in patch prioritization. All of these approaches obtain better results than prior stochastic search based techniques [16], [17], [18]. Nevertheless, none of these techniques examine whether patch prioritization can be further improved by integrating the strengths of both similarities. Therefore, this study aims at analyzing the impact of incorporating both syntactic and semantic similarity on patch prioritization. The details of the study is presented in the following sections.

4.2 Methodology of the Empirical Study

This study considers finding the correct patch in automated program repair as a ranking problem [96]. The goal is to rank the correct patch higher among all the generated patches. The overview of the methodology is shown in Figure 4.1. At first, relevant bugs are identified through preprocessing. Next, for each bug, the proposed approaches take source code and faulty line as input and output a ranked list of patches. For generating the ranked list, patches are sorted by integrating syntactic and semantic similarity between faulty code and fixing ingredient.

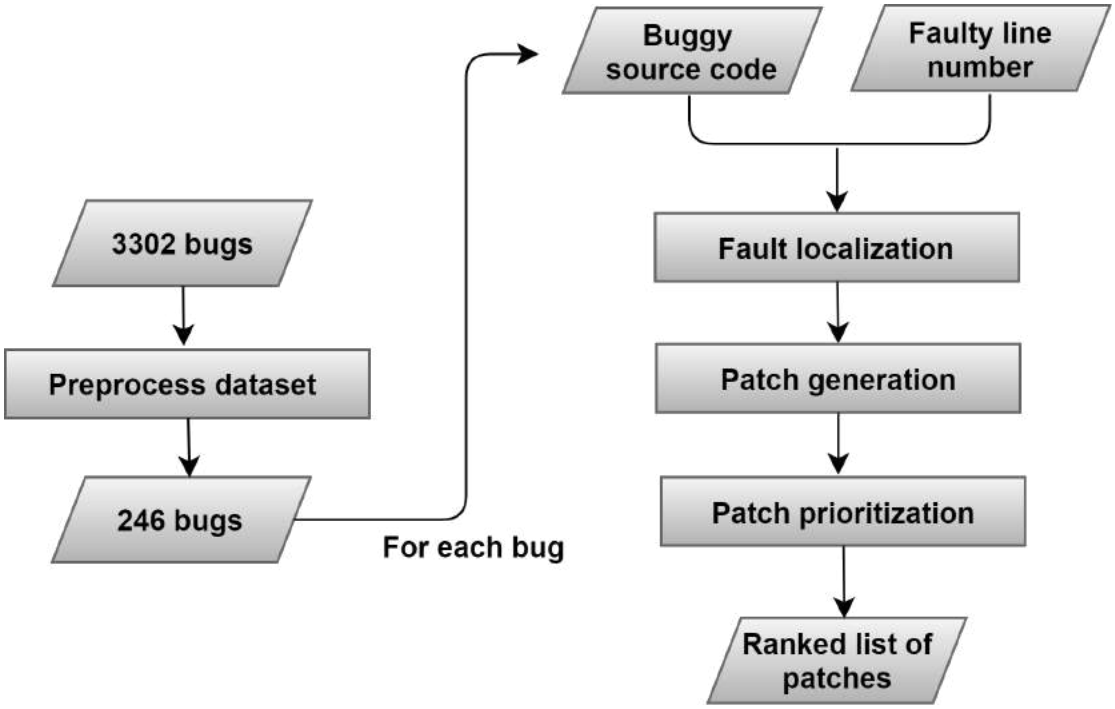

Figure 4.1: Overview of the Methodology

4.2.1 Dataset Preprocessing

To identify bugs relevant to the study, dataset is preprocessed. This study analyzes how combining syntactic and semantic similarity impacts patch prioritization. Similar to state of the art approaches [5], [16], [29], [84], it focuses on redundancy based program repair. It particularly studies replacement mutation bugs, as followed in [6]. Therefore, bugs fulfilling these requirements are selected from the dataset through preprocessing.

In this study, historical bug fixes dataset is used which comprises 3302 real bug fixes from over 700 large, popular, open-source Java projects such as Apache Commons Lang, Eclipse JDT Core etc [29]. This dataset has been adopted by previous studies as well [5], [97]. Each bug in this dataset is associated with a buggy and a fixed version file, corresponding commit hashes and a project URL. From this dataset, bugs that fulfill the following criteria are chosen, as shown in Figure 4.2.

- **Unique:** Duplicate bugs will bias the result. Two bugs are considered as

duplicate if their corresponding buggy and fixed version files are same. The dataset contained some duplicate bugs which are filtered out. For example, bug id *Lollipop_platform_frameworks_base_18* and *AICP_frameworks_base_24*, *Heliosearch_heliosearch_14* and *apache_lucene-solr_18* are duplicates.

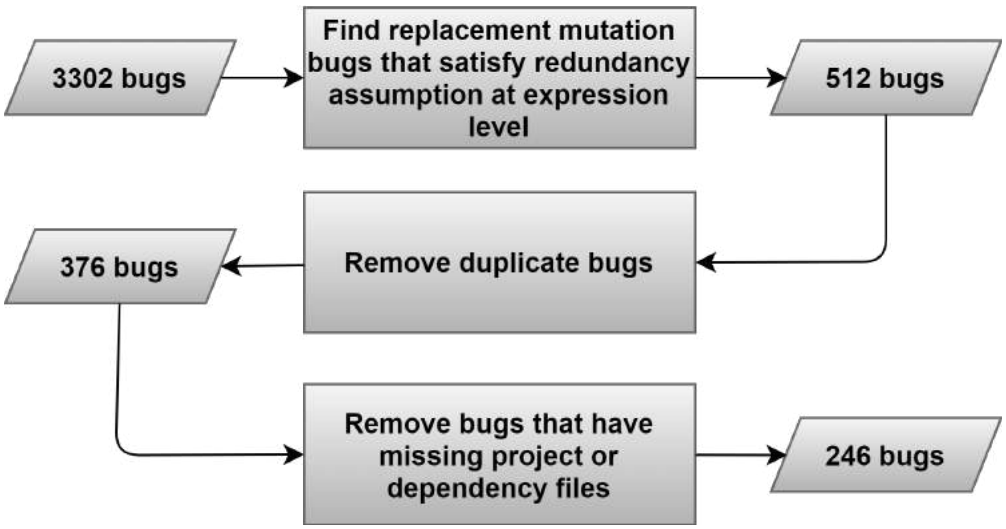

Figure 4.2: Steps of Dataset Preprocessing

- **Satisfy redundancy assumption at file level:** Similar to [5], [16], [19], this study focuses on the file level redundancy assumption (patches of bugs are found in the corresponding buggy files). Only bugs satisfying this requirement are chosen.

- **Fixed by applying replacement mutation:** The faulty code is more likely to be syntactically and semantically similar to the fixing ingredient for replacement mutation bugs [6]. Hence, only bugs that can be solved by applying replacement mutation are selected.

- **Require fixing at expression level:** A study on 16,450 bug fixing commits found that around 82.40% repair actions are associated with *expressions* [14]. Another study revealed that redundancy is higher at finer granularity and therefore, increases the probability of including the correct patch in the search space [5]. Only bugs that require fix at expression level are selected.

- **Having available project and dependency files:** To measure similarity, variables within a file need to be identified. For this purpose, the corresponding project and dependency files of a bug are needed. To obtain a project, corresponding GitHub URL is inspected. For maven projects, jar files specified in the *pom.xml* file are downloaded [98]. Next, sourcepath (a list of directories containing source code files) and classpath (a list of directories containing class or jar files) are set to include the project's source code and jar files respectively [99]. To check if all the dependencies are available, the import statements of a buggy file are resolved. If all the imports cannot be resolved, the bug has missing dependency files. It is found that some project URLs such as, *apache_james*, *bborbe_java*, *JetBrains_jetpad-projectional*, do not exist anymore. Additionally, some bugs have missing dependency files, for example, bug id *baasbox_baasbox_6*, *geotools_geotools_18*, *JetBrains_ideavim_2*, which are removed.

After filtering, it results in 246 bugs (shown in Figure 4.2), which is similar to other studies [6], [97]. Chen and Monperrus used 214 bugs for their empirical study [6], whereas Le et al. considered 100 bugs for evaluating their proposed approach [97].

4.2.2 Proposed Approach

This study examines the impact of combining syntactic and semantic similarities on patch prioritization. Figure 4.3 shows overview of the proposed technique. It works in three steps namely fault localization, patch generation and patch prioritization. For a given buggy line, fault localization extracts corresponding *Expression* nodes. Patch generation produces patches by replacing the faulty node by fixing ingredients. For each patch, a score is calculated using syntactic and semantic similarity between faulty code and fixing ingredient. Based on this score, patches are prioritized. The details of these steps are given below:

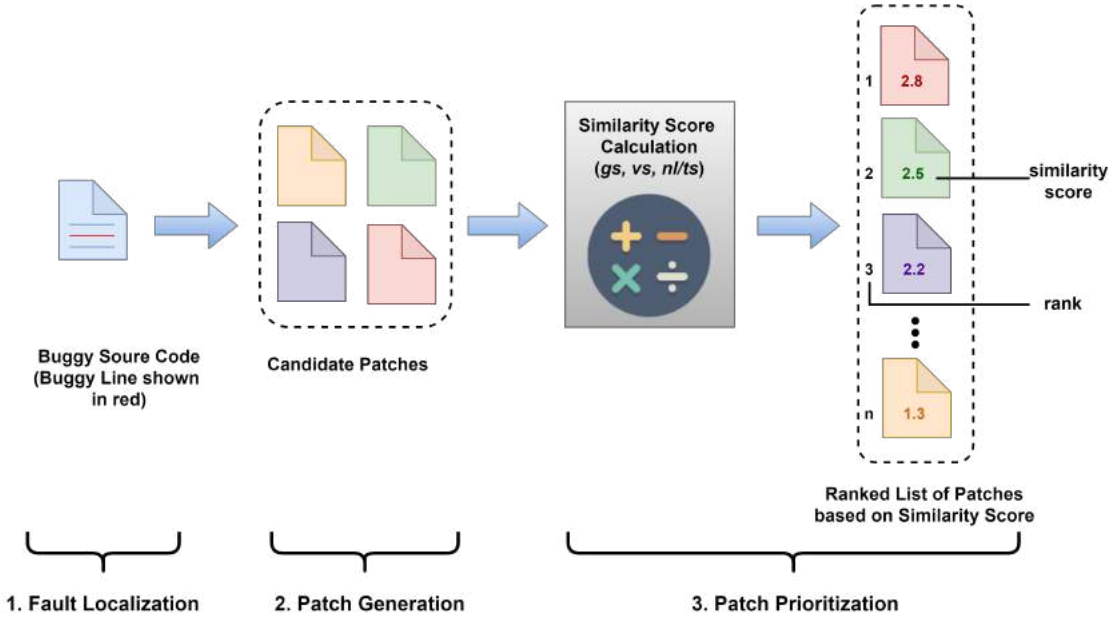

Figure 4.3: Overview of the Proposed Technique

1. **Fault Localization:** It identifies the faulty AST node of type *Expression*. Similar to [6], this study assumes that fault localization outputs the correct faulty line since the main focus is on patch prioritization. For each bug, the faulty line is identified from the difference between the buggy and fixed version files. Next, *Expression* type nodes (both buggy and non-buggy) residing in that line are extracted from AST. Listing 4.1 shows a sample bug with id = *fasseg_exp4j_4* from project exp4j. Here, line 68 is faulty. All the expressions from this line such as *Character*, *Character.isDigit(next)*, *next* etc, are extracted.

2. **Patch Generation:** This step modifies the source code to generate patches. After identifying the faulty nodes, fixing ingredients are collected. Existing studies found that considering fixing ingredients from only the buggy source file decreases the bug fixing time without substantially impacting the probability of successful repair [8], [52]. Hence, this study uses nodes from the buggy source file that have a category of *Expression* as fixing ingredients, as followed in [5]. Next, faulty nodes are replaced with fixing ingredients

for patch generation. In Listing 4.1, a sample patch is replacing *Character.isDigit(next)* with *Character.isDigit(next)* || *next* == *'.'* from line 93.

Listing 4.1: Buggy Statement, Fixed Statement and Fixing Ingredient of Bug *fasseg_exp4j_4*

```
// buggy statement
68: - if (Character.isDigit(next)) {
// fixed statement
68: + if (Character.isDigit(next) || next == '.') {
69:         complex.append(next);
70:         pos++;
71: }
// fixing ingredient
93: if (Character.isDigit(next) || next == '.') {
94:         numberString.append((char) next);
95: }
```

3. **Patch Prioritization:** The patch generation step produces numerous patches due to having large solution space. To find potentially correct patches earlier, the generated patches are prioritized. Both syntactic and semantic similarities between faulty code and fixing ingredient are used to prioritize patches (shown in Figure 4.3). To measure semantic similarity, genealogical and variable similarity are used since these are effective in differentiating between correct and incorrect patches [5]. For capturing syntactic similarity, two widely-used metrics namely normalized longest common subsequence and token similarity are considered individually [30]. Thus, two patch prioritization approaches namely *Com-L* and *Com-T* are proposed. *Com-L* combines genealogical and variable similarity with normalized longest common subsequence to rank patches. *Com-T* uses a combination of genealogical, variable

and token similarity for patch prioritization.

- **Genealogical Similarity:** Genealogical structure indicates the types of code elements, with which a node is often used collaboratively [5]. For example, node *Character.isDigit(next)* is used inside *if* statement. To extract the genealogy contexts of a node residing in a method body, its ancestor, as well as sibling nodes are inspected. The ancestors of a node are traversed until a method declaration is found. For sibling nodes, nodes having a type *Expressions* or *Statements* within the same block of the specified node are extracted. Next, the type of each node is checked and the frequency of different types of nodes (for example, number of *for* statements) are stored. Nodes of type *Block* are not considered since these provide insignificant context information [5]. On the other hand, for nodes appearing outside method body, only its respective type is stored. The same process is repeated for the faulty node (fn) and the fixing ingredient (fi). Lastly, the genealogical similarity (gs) is measured using equation (4.1).

$$gs(fn, fi) = \frac{\sum_{t \in K} min(\phi_{fn}(t), \phi_{fi}(t))}{\sum_{t \in K} \phi_{fn}(t)} \tag{4.1}$$

  where, ϕ_{fn} and ϕ_{fi} denote the frequencies of different node types for faulty node and fixing ingredient respectively. K represents a set of all distinct AST node types captured by ϕ_{fn}.

  Example: Table 4.1 shows the genealogy contexts of the faulty node *Character.isDigit(next)* and the fixing ingredient *Character.isDigit(next)* $\|$ *next* $== '.'$. For example, both the faulty node and the fixing ingredient reside in *if* statement. Based on Table 4.1, the calculation of genealogical similarity is presented in Table 4.2. Here, column one shows all the distinct node types occurred in the genealogy context of

Table 4.1: Genealogy Contexts of Faulty Node *Character.isDigit(next)* and Fixing Ingredient *Character.isDigit(next) || next == '.'*

Genealogy Context	**Node**	**Type**
faulty node *Character.isDigit(next)*	if (Character.isDigit(next)) { complex.append(next); pos++; }	if statement
	Character.isDigit(next)	method invocation
	Character	simple name
	isDigit	simple name
	next	simple name
	complex.append(next);	expression statement
	complex.append(next)	method invocation
	complex	simple name
	append	simple name
	next	simple name
	pos++;	expression statement
	pos++	postfix expression
	pos	simple name
fixing ingredient *Character.isDigit(next) || next == '.'*	if (Character.isDigit(next) || next == '.') { numberString.append((char) next); }	if statement
	Character.isDigit(next) || next == '.'	infix expression
	Character.isDigit(next)	method invocation
	Character	simple name
	isDigit	simple name
	next	simple name
	next == '.'	infix expression
	next	simple name
	'.'	character literal
	numberString.append((char)next);	expression statement
	numberString.append((char)next)	method invocation
	numberString	simple name
	append	simple name
	(char)next	cast expression
	next	simple name

Table 4.2: Genealogical Similarity between Faulty Node *Character.isDigit(next)* and Fixing Ingredient *Character.isDigit(next)* ‖ *next == '.'*

Node Type	**Frequency in Faulty Node**	**Frequency in Fixing Ingredient**	**Minimum Frequency**
if statement	1	1	1
method invocation	2	2	2
simple name	7	7	7
expression statement	2	1	1
postfix expression	1	0	0
	$\sum_{t \in K} \phi_{fn}(t) = 13$		$\sum_{t \in K} min(\phi_{fn}(t), \phi_{fi}(t)) = 11$

the faulty node (K). Column two and three display frequency of a particular node type in the faulty node (ϕ_{fn}) and the fixing ingredient (ϕ_{fi}) respectively. For example, two *method invocations* exist in the genealogy context of the faulty node and the fixing ingredient. Column four shows the minimum frequency of a node type in these genealogy contexts. As *expression statement* appears twice in ϕ_{fn} and once in ϕ_{fi}, minimum frequency is 1 here. Finally, the genealogical similarity between faulty node and fixing ingredient = 11/13 = 0.85.

- **Variable Similarity:** Variables (local variables, method parameters and class attributes) accessed by a node provide useful information as these are the primary components of a code element [5]. To measure variable similarity, two lists containing names and types of variables used by the faulty node (θ_{fn}) and the fixing ingredient (θ_{fi}) are generated. Next, variable similarity (vs) is calculated using equation (4.2).

$$vs(fn, fi) = \frac{|\theta_{fn} \cap \theta_{fi}|}{|\theta_{fn} \cup \theta_{fi}|} \tag{4.2}$$

Two variables are considered same if their names and types are exact match. To measure variable similarity of nodes that do not contain any variable, for example, *Boolean Literal* type nodes, only their respective data types are matched [5].

Example: In Listing 4.1, both the faulty node *Character.isDigit(next)* and the fixing ingredient *Character.isDigit(next)* || *next == '.'* access the same variable *next*. Here, variable similarity is 1.

- **Normalized Longest Common Subsequence:** Longest Common Subsequence (LCS) finds the common subsequence of maximum length by working at character-level [6]. This study computes normalized longest common subsequence (nl) between faulty node (fn) and fixing ingredient (fi) at AST node level using equation (4.3).

$$nl(fn, fi) = \frac{LCS(fn, fi)}{max(|fn|, |fi|)} \tag{4.3}$$

where, $max(|fn|, |fi|)$ indicates the maximum length between fn and fi.

Example: In Listing 4.1, the longest common subsequence between the faulty node *Character.isDigit(next)* and the fixing ingredient *Character.isDigit(next)* || *next == '.'* is 23. The lengths of the faulty node (fn) and fixing ingredient (fi) are 23 and 38 resectively. Hence, normalized LCS = 23/max(23,38) = 23/38 = 0.61.

- **Token Similarity:** Unlike normalized LCS, token similarity ignores the order of text [6]. It only checks whether a token (for example, *identifiers*) exists regardless of its position. For example, both faulty node *Character.isDigit(next)* and fixing ingredient *Character.isDigit(next)* || *next == '.'* have *isDigit* token in common. To calculate token similarity, at first, the faulty node and fixing ingredient are tokenized. Similar

to [10], camel case identifiers are further split and converted into lower-case format. For example, *isDigit* is converted into *is* and *digit*. Next, token similarity (ts) is computed using equation (4.4).

$$ts(fn, fi) = \frac{|\theta_{fn} \cap \theta_{fi}|}{|\theta_{fn} \cup \theta_{fi}|} \tag{4.4}$$

where, θ_{fn} and θ_{fi} represent the token set of faulty node and fixing ingredient respectively.

Example: Table 4.3 displays the token set of the faulty node *Character.isDigit(next)* and the fixing ingredient *Character.isDigit(next)* || *next == '.'*. Both of these have 7 tokens in common (no 1-7) and the total number of tokens is 10. In this case, token similarity = 7/10 = 0.70.

Table 4.3: Token Set of Faulty Node *Character.isDigit(next)* and Fixing Ingredient *Character.isDigit(next)* || *next == '.'*

No	Tokens of Faulty Node	Tokens of Fixing Ingredient
1	character	character
2	.	.
3	is	is
4	digit	digit
5	(	(
6	next	next
7	)	)
8				
9		==
10		'.'

Each of the above mentioned metrics outputs a score between 0 and 1. The final similarity score is calculated by adding these scores (sum of *gs*, *vs* and *nl* or *ts*), as followed in [4]. For example, if *gs*, *vs* and *ts* are 0.92, 0.66 and 0.57 respectively, the final score will be 2.15. Next, all the patches are sorted in descending order based on this similarity score (shown in Figure 4.3).

4.3 Experiment and Result Analysis

This section presents the implementation details, evaluation criteria and result analysis of the study. At first, the language and tools used for implementing the proposed approaches are discussed. Next, the evaluation metrics are described. Finally, results of the proposed approaches based on the evaluation metrics are reported.

4.3.1 Implementation

This study proposes two approaches to examine the impact of combining syntactic and semantic similarities on patch prioritization. The devised approaches are implemented in Java since it is one of the most popular programming languages [28], [97], [100]. It uses Eclipse JDT parser for manipulating AST [101]. It uses Javalang tool for tokenizing code [102]. Javalang takes Java source code as input and provides a list of tokens as output.

To understand combined impact of similarities, the proposed approaches need to be compared with patch prioritization techniques that use semantic and syntactic similarity individually. Therefore, this study further implements semantic or syntactic similarity based patch prioritization approaches using metrics discussed in Section 4.1.2 (genealogical similarity, variable similarity, normalized LCS and token similarity). The techniques are described below:

1. **Semantic Similarity Based Approach (*SSBA*):** It uses only semantic similarity metrics namely genealogical and variable similarity to prioritize patches.

2. **LCS Based Approach (*LBA*):** It is a syntactic similarity based approach that prioritizes patches using only normalized LCS score.

3. **Token Based Approach (*TBA*):** It is another syntactic similarity based approach that uses only token similarity to prioritize patches.

All of these three patch prioritization approaches follow the same steps as the combined ones. It ensures that the observed effects occurred due to varied similarities used in patch prioritization. The implementations of these approaches are publicly available at GitHub[1].

4.3.2 Evaluation

In this study, the following evaluation metrics are inspected:

1. **Median rank of the correct patch:** The lower the median rank, the better the approach is [6], [29].

2. **Average space reduction:** It indicates how much search space can be avoided for finding the correct patch [6]. It is calculated using equation (4.5).

$$average\ space\ reduction = (1 - \frac{mean\ correct}{mean\ total}) * 100 \tag{4.5}$$

   where, *mean correct* and *mean total* denote mean of the correct patch rank and total patches generated respectively.

3. **Perfect repair:** It denotes the percentage of bug fixes for which the correct solution is ranked at first position [6].

Similar to [6], this study considers a patch identical to human patch as correct, which is provided with the dataset.

Figure 4.4 shows the rank distributions of correct patches for *SSBA*, *Com-L*, *Com-T*, *LBA* and *TBA*. Since the data range is high (1-15005), log transformation is used in this figure. It can be seen that *Com-L* and *Com-T* outperform *SSBA*,

[1] `https://github.com/mou23/Impact-of-Combining-Sytactic-and-Semantic-Similarity-on-Patch-Prioritization`

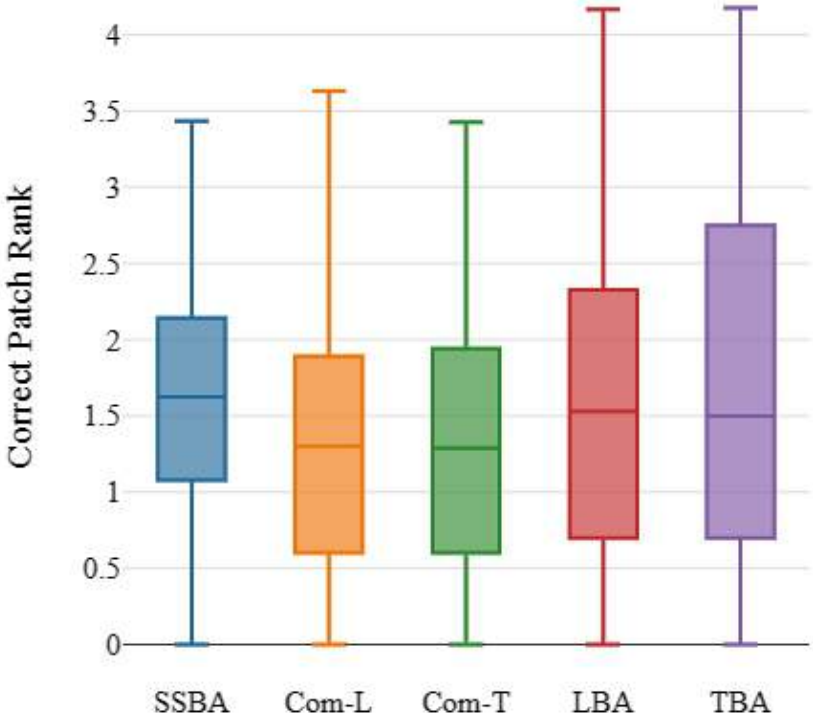

Figure 4.4: Comparison of Correct Patch Rank among *SSBA*, *Com-L*, *Com-T*, *LBA* and *TBA*

LBA and *TBA* in terms of median rank of the correct patch. The ranks are 20, 19.5, 42, 34 and 31.5 respectively. The reason is *Com-L* and *Com-T* incorporate information from multiple domains (both textual similarity and code meaning). A sample patch is shown in Listing 4.2. Here, the lines of code started with "+" and "-" indicate the added and deleted lines respectively. For this bug, *SSBA*, *LBA* and *TBA* rank the correct solution at 3, 4 and 9 respectively. On the other hand, both *Com-L* and *Com-T* rank the correct solution at 1.

Listing 4.2: A Correct Sample Patch for Bug *hornetq_hornetq_70*

```
protected void encodeBody(ConsumerFlowTokenMessage message,
        RemotingBuffer out) throws Exception {
-    out.putFloat(message.getTokens());
+    out.putInt(message.getTokens());
}
```

Figure 4.5 demonstrates that *Com-L* and *Com-T* are effective in reducing the search space compared to *SSBA*, *LBA* and *TBA*. When random search is used, on average 50% of the search space needs to be covered before finding the correct solution [6]. Using *Com-L* and *Com-T*, 96.52% and 96.62% of the total search space can be ignored to find the correct patch, whereas it is 95.38%, 84.07% and 79.49% for *SSBA*, *LBA* and *TBA* correspondingly. By using *Com-L* or *Com-T*, future automated program repair tools can consider a larger search space to fix more bugs [5].

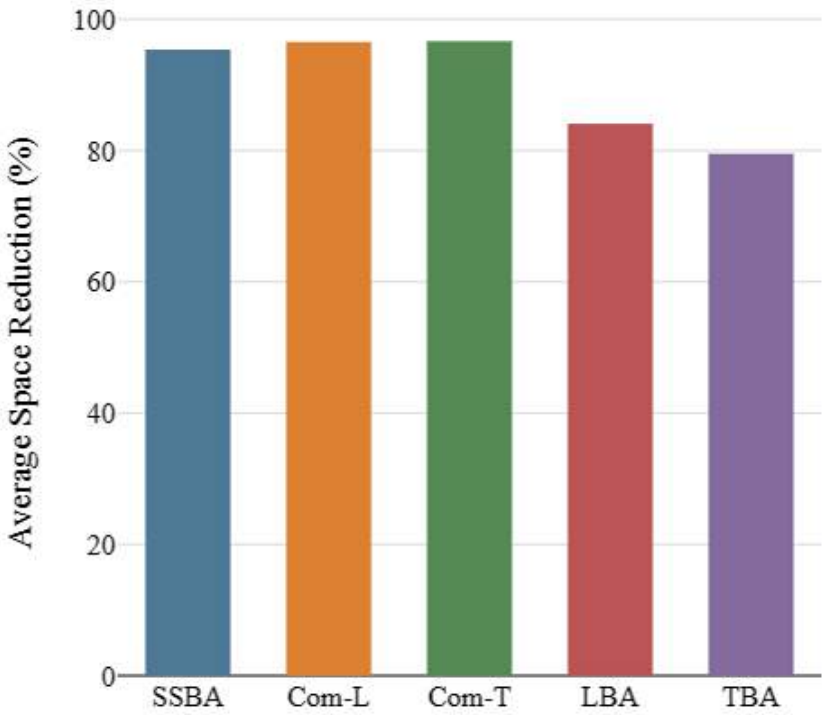

Figure 4.5: Comparison of Average Space Reduction among *SSBA*, *Com-L*, *Com-T*, *LBA* and *TBA*

In terms of perfect repair, *Com-L* and *Com-T* outperform *SSBA*, as shown in Figure 4.6. The values are 11.79%, 10.16% and 2.44% respectively. Regarding syntactic similarity based approach, *Com-L* performs better than *TBA* and as good as *LBA*. However, *Com-T* obtains lower result than *Com-L* and *LBA*. For example, for the bug in Listing 4.3, *Com-T* ranks the correct solution at 8. On

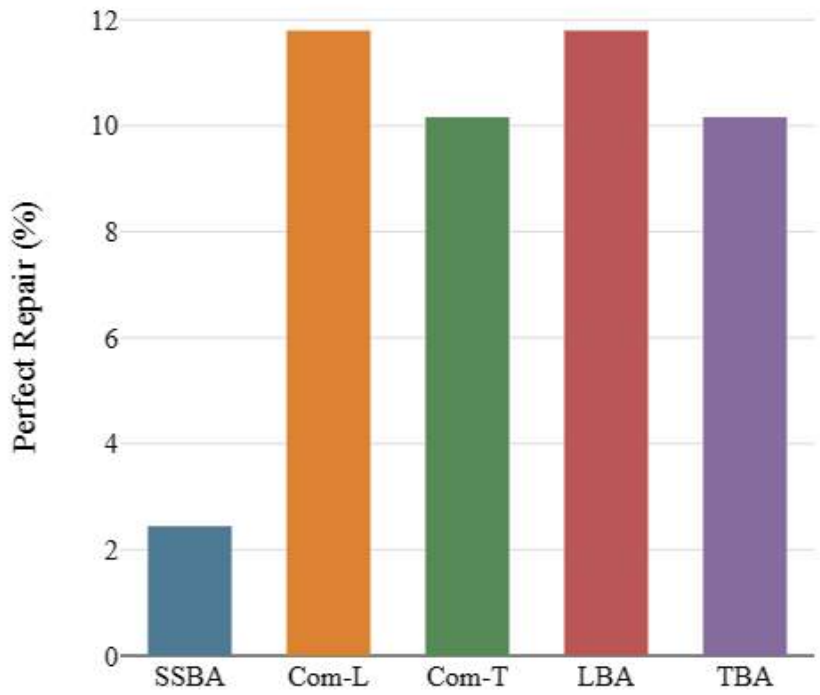

Figure 4.6: Comparison of Perfect Repair among *SSBA*, *Com-L*, *Com-T*, *LBA* and *TBA*

the other hand, both *Com-L* and *LBA* rank the correct patch at 1 due to high character level similarity. Nevertheless, the values obtained by *Com-L* and *Com-T* indicate that these approaches contribute to solving the overfitting problem. Since the first patch is the correct one in 11.79% and 10.16% cases, there is no chance of generating plausible patch before the correct one.

Listing 4.3: A Correct Sample Patch for Bug *Cervator_Terasology_2*

```
public Path getChunkPath(Vector3i chunkPos) {
-    return worldsPath.resolve(getChunkFilename(chunkPos));
+    return worldPath.resolve(getChunkFilename(chunkPos));
}
```

Table 4.4 reports the statistical significance of the obtained result using significance level = 0.05. Wilcoxon Signed-Rank test is used for this purpose since no assumption regarding the distribution of samples has been made [103], [104].

Table 4.4: Differences between Mean Ranking of Correct Patch

Compared Groups	Mean		P-value	Decision
Com-L and *SSBA*	*Com-L*	*SSBA*	0.00	Significant
	125.98	166.99		
Com-T and *SSBA*	*Com-T*	*SSBA*	0.00	Significant
	122.31	166.99		
Com-L and *LBA*	*Com-L*	*LBA*	0.00	Significant
	125.98	576.46		
Com-T and *LBA*	*Com-T*	*LBA*	0.00	Significant
	122.31	576.46		
Com-L and *TBA*	*Com-L*	*TBA*	0.00	Significant
	125.98	742.17		
Com-T and *TBA*	*Com-T*	*TBA*	0.00	Significant
	122.31	742.17		
Com-L and *Com-T*	*Com-L*	*Com-T*	0.00	Significant
	125.98	122.31		
SSBA and *LBA*	*SSBA*	*LBA*	0.13	Insignificant
	166.99	576.46		
SSBA and *TBA*	*SSBA*	*TBA*	0.00	Significant
	166.99	742.17		
LBA and *TBA*	*LBA*	*TBA*	0.29	Insignificant
	576.46	742.17		

Results show that the mean rank of *Com-L* and *Com-T* are significantly better than *SSBA*, *LBA* and *TBA*. The p-value is 0.00 in all of these cases. Although the mean rank of *SSBA* is significantly better than *TBA*, it is not significantly different from *LBA*. For some bugs such as Listing 4.4, the fixing ingredients are very different from the faulty code. To fix the bug, *null* is replaced with *paramType*. In this case, syntactic similarity based approaches *LBA* and *TBA* cannot rank the correct patch higher since there is no textual similarity between faulty code and fixing ingredient.

Listing 4.4: A Correct Sample Patch for Bug *spring-projects_spring-roo_10*

```
public int complete(String buffer, int cursor, List<String> candidates) {
-    candidate.convertFromText("*", null, include.optionContext());
+    candidate.convertFromText("*", paramtype, include.optionContext());
}
```

Results further reveal that the mean rank of *Com-T* is significantly better than *Com-L* (p-value = 0.00). The reason is when the character level difference between faulty code and fixing ingredient is high, *Com-L* can not perform well. However, *Com-T* has no such drawback. An example is shown in Listing 4.5. Here, a larger expression is replaced with a smaller one that has low character level similarity. Therefore, *Com-L* ranks the correct patch at 256, whereas *Com-T* ranks it at 87.

Listing 4.5: A Correct Sample Patch for Bug *broadgsa_gatk_2*

```
public static void scatterContigIntervals(SAMFileHeader fileHeader,
                ListGenomeLoc locs, ListFile scatterParts) {
-    totalBases += loc.getStop() - loc.getStart();
+    totalBases += loc.size();
}
```

4.4 Threats to Validity

This section presents potential aspects which may threat the validity of the study:

- **Threats to external validity:** The external threat of this study lies in the generalizability of the obtained result [29]. The analysis is conducted on 246 out of 3302 bugs from historical bug fix dataset [29], which are selected through preprocessing (details are mentioned in Section 4.2.1). To mitigate the threat of generalizability, bugs belonging to popular, large and diverse projects are used. This dataset has been adopted by existing approaches [5], [97] as well. For example, Le et al. used 100 bugs from this dataset for evaluating their proposed approach [97].

- **Threats to internal validity:** Threats to internal validity include errors in the implementation and experimentation [10], [29]. This study assumes

that fault localization outputs the correct faulty line, as followed in [6]. This assumption may not be always true [3], [66]. However, the focus of this work is patch prioritization and thereby studying fault localization is out of the scope. Similarly, analyzing the patch correctness is itself a research, which is explored by [11], [105], [106]. Following the research presented in [6], this study considers a patch identical to the patch developed by human as correct.

To generate patches, fixing ingredients are collected from the corresponding buggy file. Changing the fixing ingredient collection scope, for example, collecting fixing ingredients from the whole buggy project, may impact the obtained result. However, this process is widely followed by existing approaches [5], [16], [107]. Another threat to internal validity is errors in the tools used for implementation. For manipulating AST and tokenizing code, this study relies on Eclipse JDT parser [101] and javalang tool [102] respectively. These tools are not checked for errors. Nevertheless, these tools are widely used in automated program repair [4], [6], [108], [109].

4.5 Summary

This study proposes two patch prioritization algorithms combining syntactic and semantic similarity metrics. Genealogical and variable similarity are used to measure semantic similarity. For capturing syntactic similarity, normalized longest common subsequence and token similarity are used individually. The approaches take source code and faulty line as input and outputs a sorted list of patches. The patches are sorted using similarity score, obtained by integrating genealogical, variable similarity with normalized longest common subsequence or token similarity.

To understand the combined impact of similarities, proposed approaches are

compared with techniques that use either semantic or syntactic similarity. For comparison, 246 replacement mutation bugs out of 3302 bugs from historical bug fixes dataset are used [29]. The median ranks of the correct patch are 20 and 19.5 for these approaches, which outperform both semantic or syntactic similarity based techniques. Using combined methods, 96.52% and 96.62% of the total search space can be eliminated to find the correct patch. Results further show that these approaches are significantly better in ranking the correct patch earlier than semantic or syntactic based approaches. Moreover, these two techniques rank the correct solution at the top in 11.79% and 10.16% cases. It indicates that combined approaches have the potential to rank correct patch before incorrect plausible ones. Therefore, two automated program repair techniques are proposed in the next chapter by integrating these combined similarity based patch prioritization approaches.

Chapter 5

Combined Similarity Based Automated Program Repair Approaches

Automated program repair refers to the technique of finding the correct solution of a bug using a specification such as test cases. Existing automated program repair approaches [5], [10], [17] either avoid or restrictively work on expression level to prevent search space explosion and incorrect plausible patch generation. However, a study on 16,450 bug fix commits from 6 open-source projects found that almost 82.40% repair actions are related to expressions [14]. Consequently, although these techniques obtain good result for Defects4J benchmark projects, their performance is poor on other projects [33], [34]. Due to the importance of expression, this chapter proposes two automated program repair approaches that work at the expression level. To handle the enlarged search space and identify the correct patch over incorrect plausible ones, the proposed techniques incorporate syntactic and semantic similarity between faulty code and fixing ingredient. Experimentation result shows that these techniques can correctly fix expression level bugs from both large and small projects belonging to Defects4J and QuixBugs benchmark.

5.1 Introduction

Recently, automated program repair has become a popular research topic due to its potentiality of minimizing software debugging and maintenance effort [62], [80], [81]. Existing automated program repair approaches attempt to find the correct patch of a bug from the infinite search space and identify it before incorrect plausible ones. However, these techniques either avoid or limitedly work on expression level to prevent search space explosion and incorrect plausible patch generation. For example, GenProg [17] and RSRepair [18] only work at the statement level, whereas LSRepair [13] works at the method level. On the other hand, ELIXIR [10] and CapGen [5] cannot handle even the most frequently occurring expression type bugs such as *class instance creation expression* [14], as shown in Listing 5.1.

Listing 5.1: A Sample Patch from CodRep Dataset

```
public NamedListInteger getFacetTermEnumCounts(SolrIndexSearcher searcher,
    DocSet docs, String field, int offset, int limit, int mincount,
    boolean missing, String sort, String prefix) throws IOException {
-    BytesRef termCopy = new BytesRef(term);
+    BytesRef termCopy = BytesRef.deepCopyOf(term);
}
```

Nevertheless, Liu et al. conducted a study on 16,450 bug fix commits from 6 open-source projects such as Apache Mahout, Solr and reported that almost 82.40% repair actions are related to expressions [14]. Similarly, this study examines the CodRep dataset [6] to identify the differences between the buggy and the fixed version files at the expression level. CodRep dataset contains 58,069 one-line replacement bugs and corresponding fixes from 29 projects, for example, Apache Log4j, Spring Framework, etc. It is found that 42,856 out of 58,069 bugs (73.80%) are fixed by replacing an expression with another. For example,

Listing 5.1 presents a sample patch from CodRep dataset, where a *class instance creation expression* is replaced by a *method invocation expression*. Considering the importance of expression, this study proposes two automated program repair approaches that focus on the expression level. The methodology and evaluation of these proposed techniques are described in the following sections.

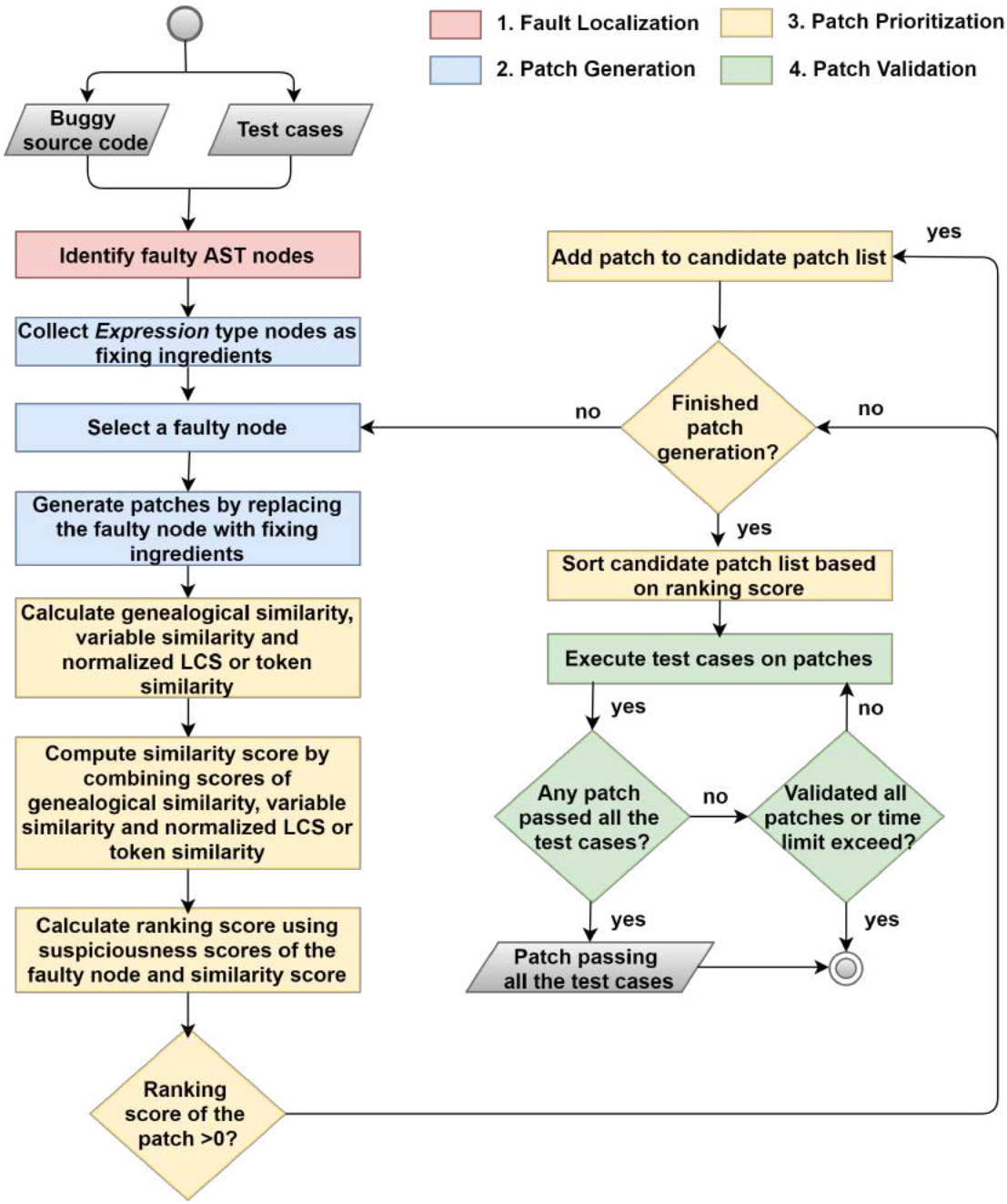

Figure 5.1: Overview of the Proposed Automated Program Repair Approaches

5.2 Methodology of the Proposed Approaches

This study proposes two combined similarity based automated bug fixing approaches namely *ComFix-L* and *ComFix-T* that work at the expression level. The empirical study in the previous chapter found that when the faulty location is known, combining syntactic and semantic similarities helps to rank the developer-written patch higher. Therefore, *ComFix-L* and *ComFix-T* integrate syntactic and semantic similarity to handle the enlarged search space, which occurred due to working at the expression level as well as identify the correct patch over incorrect plausible ones within the allocated time budget. These techniques take a buggy program and test cases with at least one failing test as input and output a program passing all the test cases, as shown in Figure 5.1. These approaches work in four steps namely fault localization, patch generation, patch prioritization and patch validation. The details of these steps are given below:

1. **Fault Localization:** This step calculates suspiciousness scores of *Expression* type AST nodes. This score indicates an expression's probability of being faulty (0-1). At first, the line-number wise suspiciousness scores of a program are computed using spectrum-based fault localization technique [31], [32]. For this purpose, Ochiai metric is used, as followed in [4], [78], [97]. Given a line of code (l), Ochiai metric uses equation (5.1) to calculate the suspiciousness score based on the number of passing and failing tests that execute it.

$$suspiciousness\ score(l) = \frac{failed(l)}{\sqrt{total\ failed * (failed(l) + passed(l))}} \quad (5.1)$$

Where, $total\ failed$ indicates the total number of failing test cases. $failed(l)$ and $passed(l)$ denote the number of failing and passing test cases that execute the line l. For the buggy source code in Listing 5.2 and the corresponding test suite in Table 5.1, the line number wise suspiciousness scores

are displayed in Table 5.2. For example, the suspiciousness score of line number 4 is 0.91. Since the proposed approaches work at the expression level, these line number wise scores are mapped to *Expression* nodes. For example, all the expressions at line 4 *source.isEmpty() ? target.length() : source.length()*, *source.isEmpty()*, *source*, *isEmpty*, *target.length()*, *target*, *length*, *source.length()* are assigned the suspiciousness score 0.91.

Listing 5.2: Buggy *levenshtein()* Method

```
1:public class LEVENSHTEIN {
2:  public static int levenshtein(String source, String target) {
3:      if (source.isEmpty() || target.isEmpty()) {
4:          return source.isEmpty() ? target.length() : source.length();
5:    } else if (source.charAt(0) == target.charAt(0)) {
6:       return 1 + levenshtein(source.substring(1), target.substring(1));
7:    } else {
8:       return 1 + Math.min(Math.min(
9:          levenshtein(source,           target.substring(1)),
10:            levenshtein(source.substring(1), target.substring(1))),
11:            levenshtein(source.substring(1), target));
12:      }
13: }
14:}
```

2. **Patch Generation:** In this step, faulty nodes are modified to generate patches. *Expression* nodes whose suspiciousness score are above 0, are considered as faulty [5]. These faulty nodes are replaced with fixing ingredients for patch generation. Similar to [5], [16], [107], this study collects fixing ingredients from the corresponding buggy source file. *Expression* nodes from

Table 5.1: Test Suite for Buggy *levenshtein()* Method

Test Case No	**Type**	**Input**		**Expected Output**
		String source	**String target**	
1	passing test case	“”	“”	0
2	failing test case	electron	neutron	3
3	failing test case	kitten	sitting	3
4	failing test case	rosettacode	raisethysword	8
5	failing test case	abcdefg	gabcdef	2
6	failing test case	hello	olleh	4

the buggy source file are considered as fixing ingredients. In Listing 5.2, the expression *target* at line 4 has a suspiciousness score of 0.91. A sample patch can be replacing this expression with *target.substring(1)* at line 6.

Table 5.2: Line Number wise Suspiciousness Score

Line Number	**Suspiciousness Score**
3	0.91
4	0.91
5	1.00
6	1.00
8	1.00
9	1.00
10	1.00
11	1.00

3. **Patch Prioritization:** After generating patches, those are prioritized using both suspiciousness score and similarity score to find potentially correct patches earlier (shown in Figure 5.1). The calculation of similarity score is the same as Chapter 4. Genealogical similarity (gs) and variable similarity (vs) are used for measuring semantic similarity. To compute syntactic similarity, normalized longest common subsequence (nl) and token similarity (ts) are considered individually. The similarity score is calculated by combining gs, vs and nl or ts (presented in equation (5.2) and (5.3)).

$$\text{For } \textit{ComFix-L}, \textit{similarity score}(fn, fi) = gs + vs + nl \tag{5.2}$$

$$\text{For } \textit{ComFix-T}, \textit{similarity score}(fn, fi) = gs + vs + ts \tag{5.3}$$

where, $\textit{similarity score}(fn, fi)$ is the similarity score between faulty node (fn) and fixing ingredient (fi). This similarity score is multiplied by the faulty node's suspiciousness value to compute the patch ranking score, as shown in equation (5.4).

$$\textit{ranking score} = \textit{suspiciousness score}(fn) * \textit{similarity score}(fn, fi) \tag{5.4}$$

where, $\textit{suspiciousness score}(fn)$ is the suspiciousness value of the faulty node fn. To constrain the search space, only patches with a ranking score greater than 0, are considered as candidate patches. To remove duplicate candidate patches, only patch with the highest ranking score is kept. These candidate patches are next sorted in descending order based on the ranking score. Figure 5.2 shows the ranked list of three patches generated by *ComFix-L*. For example, the patch replacing *source.isEmpty()* at line 4 with *source.length()* at line 4 has the suspiciousness score 0.91. Here, the values of genealogical similarity, variable similarity and normalized LCS are 1, 1 and 0.625 respectively. Therefore, the ranking score is 0.91*(1+1+0.625) = 2.389. This patch is ordered at first position since it has the highest ranking score.

4. **Patch Validation:** In this step, the correctness of the candidate patches are examined by executing test cases. To validate a patch, at first, it is applied to the buggy source code and the modified source code is compiled [10]. If the source code compiles, test cases are executed to evaluate the patch. At first, the failing test cases are executed, as followed in [4], [5]. If it passes those test cases, the passing test cases are executed. Patch validation continues until a patch passing all the test cases is found or the predefined

1. replacing source.isEmpty() at line 4 with source.length() at line 4

Suspiciousness Score: 0.91

Genealogical Similarity: 1.0

Variable Similarity: 1.0

Normalized LCS: 0.625

Ranking Score: 2.389

2. replacing target at line 4 with target.substring(1) at line 6

Suspiciousness Score: 0.91

Genealogical Similarity: 0.875

Variable Similarity: 1.0

Normalized LCS: 0.32

Ranking Score: 1.997

3. replacing 0 at line 5 with 1 at line 6

Suspiciousness Score: 1.0

Genealogical Similarity: 0.525

Variable Similarity: 1.0

Normalized LCS: 0

Ranking Score: 1.525

Figure 5.2: Ranked List of Sample Patches for the Buggy *levenshtein()* Method

time-limit exceeds. If a patch that passes all the test cases is found, this step outputs that patch.

5.3 Experiment

This section describes the implementation and experimentation of the study. At first, the language and tools used for implementing the proposed approaches are presented. Next, the experimentation dataset and evaluation metrics are discussed.

5.3.1 Implementation

Similar to [5], [10], [109], the proposed approaches are implemented in Java. The following tools are used for implementation:

- **Eclipse JDT Parser** It is used for parsing and manipulating AST [101].
- **GZoltar:** To localize fault, GZoltar tool (version 0.1.1) is used [93]. It is widely used by existing automated program repair techniques [5], [12], [52]. It takes the class files of the buggy source code and the test cases as input and outputs line-number wise suspiciousness scores of the program.
- **Javalang:** It helps to tokenize code which is used for calculating token similarity [102]. It takes Java source code as input and provides a list of tokens as output.
- **Javax.tools** It is used for compiling the source code [10].

To understand the impact of combining similarities, the proposed approaches are compared with baseline program repair techniques that use either semantic or syntactic similarity. The is because the proposed approaches target a different defect class (expression level bugs) from the existing program repair approaches [87].

Consequently, this study further implements the following semantic or syntactic similarity based bug fixing approaches:

1. **Semantic Similarity Based Approach (*SSBA*):** It uses genealogical similarity (gs) and variable similarity (vs) along with faulty node's suspiciousness value to calculate patch ranking score, as presented in equation (5.5).

$$ranking\ score = suspiciousness\ score(fn) * (gs + vs) \tag{5.5}$$

2. **LCS Based Approach (*LBA*):** It multiplies the value of normalized longest common subsequence (nl) with faulty node's suspiciousness score to measure patch ranking score, as shown in equation (5.6).

$$ranking\ score = suspiciousness\ score(fn) * nl \tag{5.6}$$

3. **Token Based Approach (*TBA*):** It considers token similarity (ts) and faulty node's suspiciousness value for computing patch ranking score, as displayed in equation (5.7).

$$ranking\ score = suspiciousness\ score(fn) * ts \tag{5.7}$$

Except ranking score calculation, all of these approaches follow the same repairing process as the combined ones. The implementations of these techniques are publicly available at GitHub[1].

5.3.2 Dataset

To evaluate the proposed approaches, those are executed on Defects4J [26] and QuixBugs [27] benchmarks. Defects4J is the most widely used benchmark in

[1] `https://github.com/mou23/Combined-Similarity-Based-Automated-Program-Repair-Approaches-for-Expression-Level-Bugs`

automated program repair [33], [82]. It contains 395 bugs from six large, open-source Java projects, for example, JFreeChart, Apache Commons Lang, as listed in Table 5.3. From this benchmark, bugs that fulfill the following criteria are selected, as shown in Figure 5.3:

- **Require fixing at a single line:** Similar to most of the automated program repair techniques [5], [10], [88], this study focuses on repairing single line bugs. Therefore, only the single line bugs are chosen from the benchmark, as followed in [10]. The list of single line bugs is obtained from the study of Sobreira et al. [110].

- **Unique:** To avoid bias, only unique bugs are chosen. If both the buggy and fixed version files of two bugs are the same, those are called duplicates. In Defects4J, *Closure_63* and *Closure_93* are duplicates of *Closure_62* and *Closure_92* respectively, which are removed.

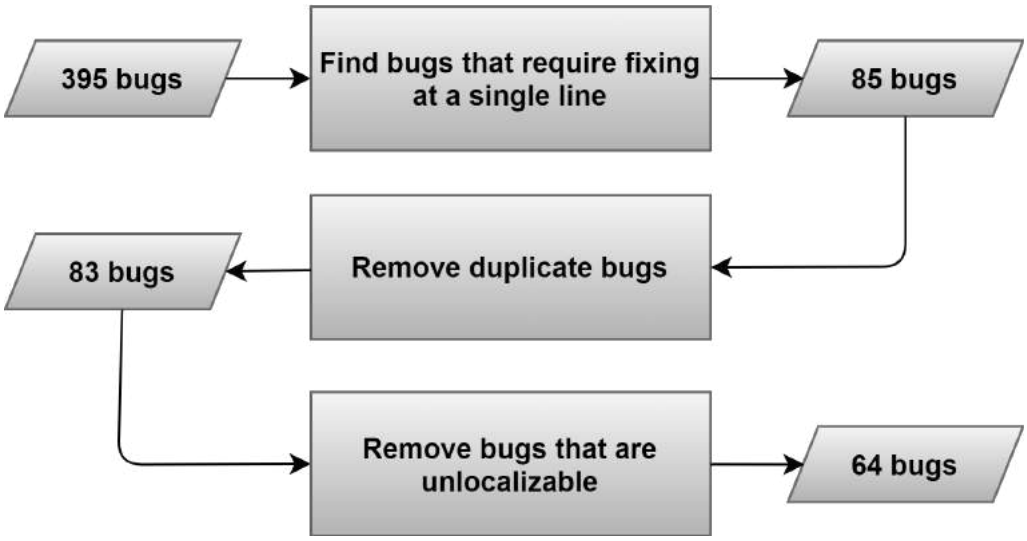

Figure 5.3: Steps for Selecting Bugs from Defects4J Benchmark

- **Localizable at line level:** A bug is localizable at line level if the output list of the fault localization technique contains the actual buggy line [3]. From the dataset, unlocalizable bugs such as *Chart_8* and *Mockito_5* are eliminated since those are impossible to fix correctly.

After filtering, it results in 64 bugs (shown in Figure 5.3). The list of these bugs is given in Appendix A.

Table 5.3: Projects of Defects4J Benchmark

Project	**Number of Bugs**	**Lines of Code**	**Number of Test Cases**
JFreeChart (Chart)	26	96000	2205
Closure Compiler (Closure)	133	90000	7927
Apache Commons Math (Math)	106	85000	3602
Apache Commons Lang (Lang)	65	22000	2245
Joda-Time (Time)	27	28000	4130
Mockito (Mockito)	38	45000	1457

To ensure generalizability, automated program repair techniques should be effective for both large and small projects. However, existing studies [33], [34], [92], [111] found that most of the program repair approaches are evaluated using only Defects4J and biased towards this benchmark. These techniques perform better for the projects of Defects4J compared to other projects. To ensure generalizability, this study uses QuixBugs benchmark [27] apart from Defects4J. This benchmark comprises single line bugs belonging to 40 small programs whose average lines of code is 190 [33]. These bugs are of diverse types such as incorrect method call, missing arithmetic expression, etc [77]. Furthermore, space and time complexity of these bugs are significant. For example, 14 programs contain recursion. From this dataset, GZoltar fails to produce output for 3 programs - *bitcount*, *find_first_in_sorted* and *sqrt* since these bugs generate infinite loop. Hence, these 3 bugs are excluded from the experiment. The experiment is run on an Ubuntu server with Intel Xeon E5-2690 v2 @3.0GHz and 64GB physical memory. For each bug of these two benchmarks, the time budget is set to 90 minutes, as followed in [5], [10], [29].

5.3.3 Evaluation Metrics

Similar to [5], [12], the proposed techniques are evaluated based on the following metrics:

- **Number of bugs correctly fixed:** If an approach fixes more bugs correctly, it is considered more effective [29].

- **Precision:** It indicates the percentage of correct patches out of the plausible ones [82]. It is calculated using equation (5.8).

$$precision = (\frac{total\ correct\ patches}{total\ plausible\ patches}) * 100 \tag{5.8}$$

  where, *total correct patches* and *total plausible patches* denote the total number of the correct and plausible patches generated respectively. If precision is high, developers do not have to manually analyze the solutions generated by the technique [5], [11].

- **Elapsed time to generate correct patches:** The less time a technique takes to generate correct patches, the better it is [29].

To assess the correctness of a plausible patch, both manual and automated analysis are performed, as suggested by [35]. For automated analysis, EvoSuite tool generated test cases are used since it is the most effective tool for identifying overfitting patches [77]. The test cases for Defects4J and QuixBugs benchmark are obtained from the study [35] and [77] respectively. These tests are generated using 30 trials of EvoSuite with 30 different seeds (1,2,...,30). From these test cases, flaky tests whose outcome (pass/fail) are depended on the environment, are removed [112]. To detect flaky tests, all the tests are executed 3 times on the fixed versions of the buggy projects, as followed in [35]. Next, tests that failed at least one time on the fixed version projects, are marked as flaky. After removing the

flaky tests, the remaining tests are executed on the plausible patches generated by *ComFix-L*, *ComFix-T*, *SSBA*, *LBA* and *TBA*.

Table 5.4: Results of Automated Patch Correctness Assessment for 5 Buggy Programs of QuixBugs

Program	**Approach**	**Remarks**	**First Failing Test**
get_factors	*LBA*	incorrect	seed 1 test3
	ComFix-L	incorrect	seed 1 test3
	SSBA	incorrect	seed 1 test3
	ComFix-T	incorrect	seed 1 test3
	TBA	incorrect	seed 1 test3
levenshtein	*LBA*	correct	
	ComFix-L	correct	
	SSBA	correct	
	ComFix-T	correct	
	TBA	correct	
shortest_path_lengths	*LBA*	incorrect	seed 3 test2
	ComFix-L	incorrect	seed 3 test2
	SSBA	incorrect	seed 3 test2
	ComFix-T	incorrect	seed 3 test2
	TBA	unsolved	
next_palindrome	*LBA*	correct	
	ComFix-L	correct	
	SSBA	incorrect	seed 1 test1
	ComFix-T	correct	
	TBA	correct	
pascal	*LBA*	correct	
	ComFix-L	correct	
	SSBA	correct	
	ComFix-T	correct	
	TBA	correct	

Table 5.4 shows the patch correctness assessment result for 5 buggy programs of QuixBugs. For example, all the patches for *get_factors* program fail in test case *test3* generated from seed 1. Consequently, those are marked as incorrect plausible patches. On the contrary, the patches for *levenshtein* program pass all the EvoSuite test cases. Therefore, those are considered as correct patches. Next, patches that passed all the test cases are manually examined to check whether those are semantically equivalent to the developer-written patches (provided with the benchmarks). For example, the correct patch for bug *Math_63* is replacing

expression *(Double.isNaN(x) && Double.isNaN(y))* $\|$ $x == y$ with expression *equals(x, y, 1)*, as shown in Listing 5.3. Another plausible patch, presented in Listing 5.4, passes all the Evosuite test cases and thereby it is labeled as correct by automated analysis. Through manual inspection, the patch is found to be incorrect. When x is set to any valid double value and y is *NAN*, this patch produces a different result from the correct one. However, such project specific input can not always be covered by EvoSuite [77].

Listing 5.3: A Correct Sample Patch for Bug *Math_63*

```
public static boolean equals(double x, double y) {
-    return (Double.isNaN(x) && Double.isNaN(y)) || x == y;
+    return equals(x, y, 1);
}
```

Listing 5.4: An Incorrect Plausible Patch for Bug *Math_63*

```
public static boolean equals(double x, double y) {
-    return (Double.isNaN(x) && Double.isNaN(y)) || x == y;
+    return (!Double.isNaN(x) && Double.isNaN(y)) || x == y;
}
```

5.4 Result Analysis

Table 5.5 and Table 5.6 present the results of *LBA*, *ComFix-L*, *SSBA*, *ComFix-T* and *TBA* on Defects4J and QuixBugs benchmarks respectively. Results demonstrate that both *ComFix-L* and *ComFix-T* can correctly fix more bugs of Defects4J than syntactic similatiry based approaches *LBA* and *TBA*. For some bugs, there exists no textual similarity between the faulty code and the correct fixing ingredient. Hence, *LBA* or *TBA* can not find the correct fixing ingredients that are

Table 5.5: Results of Different Approaches on Defects4J Benchmark

Approach	**Number of bugs correctly fixed**	**Number of bugs incorrectly fixed**	**Precision (%)**
LBA	10	12	45.45
ComFix-L	11	7	61.11
SSBA	11	7	61.11
ComFix-T	11	7	61.11
TBA	8	10	44.44

Table 5.6: Results of Different Approaches on QuixBugs Benchmark

Approach	**Number of bugs correctly fixed**	**Number of bugs incorrectly fixed**	**Precision (%)**
LBA	10	4	71.43
ComFix-L	11	5	68.75
SSBA	9	7	56.25
ComFix-T	10	6	62.50
TBA	6	4	60.00

Listing 5.5: A Correct Sample Patch for Bug *Math_59*

```
public static float max(final float a, final float b) {
-    return (a <= b) ?  b :  (Float.isNaN(a + b) ?  Float.NaN : b);
+    return (a <= b) ?  b :  (Float.isNaN(a + b) ?  Float.NaN : a);
}
```

required for generating the correct patches. As a result, these approaches fail to produce the correct patches and fix the bugs. However, *ComFix-L* and *ComFix-T* can overcome this shortcoming by considering semantic information, for example, data type of the code elements [24]. In Listing 5.5, the bug *Math_59* is fixed by replacing variable b with variable a. Neither *LBA* nor *TBA* can repair this bug due to lack of the correct fixing ingredient. Both normalized LCS and token similarity between the faulty code and the correct fixing ingredient are 0 in this case. On the contrary, combined similarity based techniques *ComFix-L* and *ComFix-T* can find the correct fixing ingredient and repair this bug due to considering semantic information such as similarity between the data type of these variables. Both

variable a and b are of type *float*.

For the same reason, *ComFix-L* and *ComFix-T* obtain higher precision than *LBA* and *TBA* in Defects4J. The values are 61.11%, 61.11%, 45.45% and 44.44% correspondingly. It indicates that combined similarity based program repair approaches are better in differentiating between correct and incorrect patches than syntactic similarity based techniques. This is because correct patches tend to have higher similarities with the faulty code than incorrect plausible patches regarding both their syntax and semantics. This finding is consistent with the study presented in [113]. Consequently, *LBA* and *TBA* are prone to generating more incorrect plausible patches due to examining only textual resemblance. For bug *Lang_59*, the actual faulty line is 884 and the correct patch is replacing variable *strLen* at line 884 with variable *width*, as shown in Listing 5.6. However, fault localization technique assigns both line 880 and 884 a suspiciousness score of 0.58.

Listing 5.6: A Correct Sample Patch for Bug *Lang_59*

```
878: public StrBuilder appendFixedWidthPadRight(Object obj, int width,
                                                char padChar) {
884: -    str.getChars(0, strLen, buffer, size);
884: +    str.getChars(0, width, buffer, size);
895: }
```

Listing 5.7: An Incorrect Plausible Patch Generated by *LBA* and *TBA* for Bug *Lang_59*

```
878: public StrBuilder appendFixedWidthPadRight(Object obj, int width,
                                                char padChar) {
880: -    ensureCapacity(size + width);
880: +    ensureCapacity(size + 4);
895: }
```

Both *LBA* and *TBA* generate an incorrect plausible patch before the correct one by replacing the method call *ensureCapacity(size + width)* at line 880 with *ensureCapacity(size + 4)*, as shown in Listing 5.7. This is because textual similarity between the faulty code and the fixing ingredient is higher for the incorrect plausible patch than the correct one. For the correct patch, normalized LCS and token similarity are 0.17 and 0 respectively, whereas those are 0.82 and 0.75 for the incorrect plausible patch. On the other hand, both *ComFix-L* and *ComFix-T* can rank the correct patch over the incorrect plausible one due to incorporating genealogical and variable similarity. Genealogical and variable similarity are 1 for the correct patch while those are respectively 0.53 and 0.50 for the incorrect plausible patch.

Listing 5.8: An Incorrect Plausible Patch Generated by *ComFix-T* for Bug *flatten*

```
13: public static Object flatten(Object arr) {
21: -    result.add(flatten(x));
21: +    result.add(x);
29: }
```

Listing 5.9: A Correct Sample Patch for Bug *flatten*

```
13: public static Object flatten(Object arr) {
26: -    result.add(flatten(arr));
26: +    return arr;
29: }
```

In QuixBugs benchmark, *ComFix-L* outperforms *LBA* and *TBA* in terms of correctly fixed bugs. *ComFix-T* performs better than *TBA* and as good as *LBA*. For *ComFix-T*, there is a tie between incorrect plausible (shown in Listing 5.8) and correct patch (shown in Listing 5.9) for the bug *flatten*. Although *ComFix-L* and *ComFix-T* achieve higher precision than *TBA*, *LBA* performs better than these

two techniques. For the bug *quicksort*, both *ComFix-L* and *ComFix-T* generate an incorrect plausible patch by replacing a variable *pivot* with a number literal *0*, as illustrated in Listing 5.10. Nevertheless, this type of incorrect plausible patches can be eliminated by using historical bug fix patterns while generating patches [5], which is out of scope of this study.

Listing 5.10: An Incorrect Plausible Patch Generated by *ComFix-L* and *ComFix-T* for Bug *quicksort*

```
public static ArrayListInteger quicksort(ArrayListInteger arr) {
-    else if (x > pivot) {
+    else if (x > 0) {
        greater.add(x);
    }
}
```

Listing 5.11: A Correct Sample Patch for Bug *next_palindrome*

```
public static String next_palindrome(int[] digit_list) {
-    otherwise.addAll(Collections.nCopies(digit_list.length, 0));
+    otherwise.addAll(Collections.nCopies(digit_list.length - 1, 0));
}
```

Listing 5.12: A Correct Sample Patch for Bug *mergesort*

```
public static ArrayListInteger mergesort(ArrayListInteger arr) {
-    if (arr.size() == 0) {
+    if (arr.size()/2 == 0) {
        return arr;
    }
}
```

Table 5.5 further reveals that semantic similarity based technique *SSBA* performs as good as *ComFix-L* and *ComFix-T* for Defects4j bugs. Similar to *ComFix-L* and *ComFix-T*, *SSBA* correctly repairs 11 bugs and obtains a precision of 61.11%. However, both *ComFix-L* and *ComFix-T* outperform *SSBA* in QuixBugs benchmark, as listed in Table 5.6. For some bugs, tie occurs among multiple patches when those are ranked using only semantic similarity. For example, in case of *SSBA*, there exists a tie between correct and incorrect plausible patch for the bugs *flatten* and *next_palindrome*. Nevertheless, *ComFix-L* and *ComFix-T* can break these ties through incorporating syntactic similarity between faulty code and fixing ingredients. For the bug *next_palindrome*, *ComFix-L* and *ComFix-T* can rank the correct patch at first position, as displayed in Listing 5.11. Similarly, for the bug *mergesort*, *ComFix-L* and *ComFix-T* rank the correct solution (shown in Listing 5.12) higher than *SSBA* due to considering textual similarity. *SSBA* ranks the correct patch at 463, whereas *ComFix-L* and *ComFix-T* rank it at 17 and 14 respectively.

To compare the repairing time of correctly fixed bugs, Wilcoxon Signed-Rank test is used since no assumption regarding the distribution of samples has been made [103], [104]. Table 5.7 reports the statistical significance of the result using significance level = 0.05. For comparing two approaches, their commonly fixed bugs are used. For example, *ComFix-L* and *SSBA* have 20 correct fixes in common. The mean repairing time of these bugs are 8.59 and 7.71 minutes for *ComFix-L* and *SSBA* respectively. Result shows that these mean times are not significantly different since the p-value is 0.18. Similar result is obtained for all the other approaches. It indicates that although *ComFix-L* and *ComFix-T* considers more similarity metrics than other techniques, it does not significantly increase the bug fixing time. *ComFix-T* calculates 3 similarity metrics (genealogical, variable and token similarity) for each patch, whereas TBA computes only 1 similarity metric (token similarity). However, result implies that calculation of those additional

Table 5.7: Differences between Mean Repairing Time (In Minutes) of Different Approaches

Compared Groups	**Mean**		**P-value**	**Decision**
ComFix-L and *SSBA*	*ComFix-L*	*SSBA*	0.18	Insignificant
	8.59	7.71		
ComFix-T and *SSBA*	*ComFix-T*	*SSBA*	0.55	Insignificant
	8.35	7.71		
ComFix-L and *LBA*	*ComFix-L*	*LBA*	0.27	Insignificant
	2.74	3.87		
ComFix-T and *LBA*	*ComFix-T*	*LBA*	0.98	Insignificant
	3.56	4.11		
ComFix-L and *TBA*	*ComFix-L*	*TBA*	0.92	Insignificant
	2.68	2.79		
ComFix-T and *TBA*	*ComFix-T*	*TBA*	0.77	Insignificant
	2.76	2.79		
ComFix-L and *ComFix-T*	*ComFix-L*	*ComFix-T*	0.24	Insignificant
	8.19	7.96		
SSBA and *LBA*	*SSBA*	*LBA*	0.39	Insignificant
	4.92	4.39		
SSBA and *TBA*	*SSBA*	*TBA*	0.16	Insignificant
	5.25	3.09		
LBA and *TBA*	*LBA*	*TBA*	0.15	Insignificant
	9.99	8.52		

metrics do not take significantly extra time.

In summary, *ComFix-L* and *ComFix-T* can correctly fix more bugs compared to syntactic or semantic similarity based techniques and achieve higher precision, as displayed in Table 5.8. This is because *ComFix-L* and *ComFix-T* consider information from multiple domains (both textual similarity and code meaning). These information work as complementary and thereby accelerates the detection of the correct solution from the search space. In the previous chapter, it is found that when the faulty location is known, the developer-written patch can be ranked higher by combining syntactic and semantic similarities. Through the current chapter, it is evident that the combination of similarities is effective even when the faulty code is unknown. Existing studies found that enhancing the search space increases the probability of generating the incorrect plausible patches [5], [23]. Nevertheless, the current study founds that the integration of syntactic and

semantic similarity can contribute to find the correct patch from the enlarged search space caused by expression level bugs. Besides, it can help to differentiate between correct and incorrect plausible patches. Therefore, by incorporating semantic and syntactic similarities, future automated program repair tools can take the advantage of each side and consider a larger search space for fixing more bugs.

Table 5.8: Overall Result of Different Approaches

Approach	**Total number of bugs correctly fixed**	**Total number of bugs incorrectly fixed**	**Precision (%)**
LBA	20	16	55.56
ComFix-L	22	12	64.71
SSBA	20	14	58.82
ComFix-T	21	13	61.76
TBA	14	14	50.00

5.5 Threats to Validity

This section discusses the threats which can affect the validity of the proposed approaches. The threats are identified from two perspectives namely threats to external and internal validity.

- **Threats to external validity:** The external threat of this study is the generalizability of the obtained result [29]. To minimize the threat of generalizability, bugs belonging to both large and small projects are used for experimentation. As the representative of large projects, Defects4J is chosen since it is the most widely used benchmark in automated program repair [33], [82]. On the other hand, QuixBugs is selected as the representative of small projects because it contains diverse types of bugs such as incorrect method call, missing condition, etc [77]. In addition, space and time complexity of these bugs are significant [77].

- **Threats to internal validity:** The first threat to internal validity is error in the implementation of the study [10], [29]. This study uses GZoltar tool

(version 0.1.1) for fault localization [93]. The results of the GZoltar tool are directly incorporated in this study without checking whether there is any defect in the tool. However, GZoltar tool is widely used by existing automated program repair approaches [5], [12], [52], [109]. The second threat to internal validity comes from an error in assessing the patch correctness. Assessing the patch correctness is itself a research, which is explored by [11], [105], [106]. However, this study performs both manual and automated analysis to evaluate the correctness of a patch, as suggested by [35]. For automated analysis, test cases generated from 30 trials of the EvoSuite tool are used. Existing studies [35], [77] also use these test cases for assessing patch correctness. On the other hand, determining patch correctness based on manual inspection is a common practice in automated program repair [4], [5], [10], [29]. [114]

5.6 Summary

This study proposes two automated program repair approaches that work at the expression level. However, considering expression level enlarges the search space and thereby decreases the probability of finding the correct patch [23]. To address this problem, the proposed approaches combine syntactic and semantic similarity to constrain the search space as well as prioritize the generated patches. To calculate semantic similarity, genealogical and variable similarity are used. To capture syntactic similarity, normalized longest common subsequence and token similarity are used individually. These techniques take a buggy program, a set of test cases as input and generate a program passing all the test cases as output. At first, the suspiciousness score of *Expression* type nodes are calculated using spectrum-based fault localization [31], [32]. Next, patches are generated by replacing the faulty nodes with the fixing ingredients. To validate potentially correct patch

earlier, patches are prioritized based on suspiciousness score and similarity score. The similarity score is measured by integrating genealogical, variable similarity with normalized longest common subsequence or token similarity. Finally, the correctness of a patch is validated by executing test cases.

To understand the impact of combining similarities, the proposed approaches are compared with techniques that use either semantic or syntactic similarity. For comparison, 64 and 37 out of 395 and 40 bugs from Defects4J and QuixBugs benchmark are selected through preprocessing. Results show that combined similarity based techniques correctly fix more bugs than approaches using either semantic or syntactic similarity. In addition, combined similarity based approaches outperform syntactic and semantic similarity based approaches in terms of precision. The result further reveals that although combined similarity based techniques consider more similarity metrics than other approaches, it does not significantly increase the bug fixing time. In the next chapter, the whole thesis will be concluded with possible future remarks.

Chapter 6

Conclusion

Automated program repair has recently drawn the attention of researchers due to its potentiality of minimizing the debugging effort [62], [80], [81]. Existing automated program repair approaches [5], [10], [16], [17] either avoid or inadequately work on expression level although these types of bug are prevailing [14]. This is because working at the expression level granularity drastically enhances the search space as well as the probability of generating incorrect plausible patches [5]. Furthermore, these techniques use either syntactic or semantic similarity to guide the repairing process. Nevertheless, none of these approaches investigate whether patch prioritization can be further improved by incorporating the strengths of both similarities. In this context, the current thesis examines the impact of combining syntactic and semantic similarities on patch prioritization. Next, it proposes two automated program repair approaches integrating this impact to work at the expression level. This chapter presents a summary of the thesis with respect to the contribution and achievements. Lastly, it is concluded with possible future research directions.

6.1 Impact of Combining Syntactic and Semantic Similarities on Patch Prioritization

At first, this thesis presents an empirical study on patch prioritization to analyze the impact of combining syntactic and semantic similarities. For measuring semantic similarity, genealogical and variable similarity between faulty node and fixing ingredient are used [5]. To calculate syntactic similarity, normalized longest common subsequence and token similarity are considered separately. Thus, two patch prioritization approaches are proposed respectively. Genealogical similarity checks whether faulty node and fixing ingredient are frequently used with the same type of code elements, for example, inside loop statement [5]. Variable similarity inspects the name and type of variables accessed by the faulty node and the fixing ingredient. Normalized longest common subsequence calculates maximum similarity at character level. Token similarity measures to what extent same tokens such as identifiers and literals, exist in the faulty node and the fixing ingredient, regardless of its position.

For analysis, 246 out of 3302 bugs from the historical bug fixes dataset are selected through preprocessing such as removing duplicate bugs [29]. These bugs are collected from over 700 large, open-source, popular Java projects such as Apache Commons Math, Eclipse JDT Core. Each bug is associated with a buggy and a fixed version file, corresponding commit hashes and project URL. From the difference between the buggy and fixed version files, the faulty line is identified. Next, AST nodes of type *Expression* are extracted from that line. This work focuses on expression level since it increases the probability of including the correct patch in the search space [5]. To generate patches, the faulty nodes are replaced with fixing ingredients. Following other program repair techniques [5], [16], [19], the fixing ingredients are collected from the source file where the bug resides. Lastly, patches are prioritized based on the proposed techniques. Similar to [6], ASTs

of each patch and the correct solution are matched to assess its correctness. The correct patches of the bugs are provided with the dataset.

For evaluation, the combined methods are compared to patch prioritization techniques using only syntactic or semantic similarity. For comparison, three metrics namely median rank of the correct patch, average search space reduction and perfect repair [6] are inspected as well as Wilcoxon Signed-Rank test is conducted. Results show that the proposed approaches outperform syntactic or semantic similarity based patch prioritization techniques in terms of median rank of the correct patch and average search space reduction. Using the combination of similarities, 96.52% and 96.62% of the total search space can be avoided to find the correct patch. These methods can rank the correct patch significantly higher than semantic or syntactic similarity based approaches. Furthermore, the proposed techniques rank the correct patch at the first position in 11.79% and 10.16% cases respectively. It indicates that these methods have the potential to rank the correct patch prior to incorrect plausible ones.

6.2 Combined Similarity Based Automated Program Repair Approaches

Based on the result of the empirical study, this thesis proposes two automated program repair approaches that work at the expression level. These techniques integrate syntactic and semantic similarity to constrain the enlarged search space caused by expression and identify the correct patch over incorrect plausible ones within the allocated time budget. These approaches take a buggy program and a set of test cases with at least one failing test as input and output a program passing all the test cases. At first, the line-wise suspiciousness scores of the buggy program are calculated using spectrum-based fault localization [31], [32]. Next, these scores are mapped to *Expression* type AST nodes and those with a suspi-

ciousness score above 0, are considered as faulty. These faulty nodes are replaced with fixing ingredients to generate patches. The fixing ingredients are collected from the corresponding buggy source files, as followed in [5], [16], [19]. To validate potentially correct patch earlier, patches are ordered based on a ranking score. To calculate the ranking score, the suspiciousness score is multiplied by the similarity score, which is obtained by incorporating genealogical, variable similarity with normalized longest common subsequence or token similarity. For limiting the search space, only patches with a ranking score greater than 0 are considered as candidate patches. Finally, the correctness of these patches are validated by executing test cases. Patch validation continues until a plausible patch is found or the predefined time-limit exceeds.

To evaluate the proposed approaches, these are compared with baseline program repair techniques that use either semantic or syntactic similarity. For comparison, 64 and 37 out of 395 and 40 bugs from Defects4J [26] and QuixBugs [27] benchmark are chosen as a representative of large and small buggy projects respectively. All of the approaches are executed on these bugs using a time budget of 90 minutes, as followed in [5], [10], [29]. Next, the results are examined based on three metrics namely the number of bugs correctly fixed, precision and repairing time [5], [12], [29]. To assess the correctness of the plausible patches, both manual and automated analysis are performed, as suggested by [35].

The result demonstrates that the proposed techniques correctly repair 22 and 21 bugs from these benchmarks, which are higher compared to approaches using either semantic or syntactic similarity. Furthermore, the proposed approaches obtain a precision of 64.71% and 61.76% respectively and outperform syntactic or semantic similarity based program repair approaches. It indicates that the combination of syntactic and semantic similarities contributes to detect the correct patch over incorrect plausible ones. Through Wilcoxon Signed-Rank test, it is found that although the proposed techniques consider more similarity metrics

than other approaches, it does not significantly increase the bug fixing time. For the devised approaches, the average repairing time is 8.19 and 7.96 minutes for the commonly fixed bugs.

6.3 Future Work

In this thesis, two automated program repair approaches are proposed that work at the expression level by combining syntactic and semantic similarities. This research can be further explored based on the following directions:

- **Experimenting with other benchmarks:** Although automated program repair experiments are time-consuming [10], [33], the proposed approaches are evaluated using both small and large projects from Defects4J [26] and QuixBugs [27] benchmark to ensure generalizability. In future, these approaches can be further explored using other benchmarks such as Bugs.jar [28] or IntroClassJava [115], to gain insight about specific types of project.

- **Identifying common fix patterns for expression type bugs:** This thesis aims at analyzing the impact of combining syntactic and semantic similarities to limit the search space and identify correct patches over incorrect plausible ones. In future, common fix patterns for expression type bugs such as replacing method or variable name, can be identified. Next, these patterns can be integrated with the proposed approaches to check whether it can further constrain the search space and eliminate incorrect plausible patches.

- **Assigning weights to the similarity metrics:** All the similarity metrics used in this work - genealogical similarity, variable similarity, normalized longest common subsequence and token similarity - may not have an equal impact in identifying the correct patch. Hence, a machine learning model

such as linear regression, can be used to assign different weights to these similarity metrics based on their importance. However, before building such a model, a large diverse dataset of bugs, their corresponding patches and projects needs to be prepared, which is out of scope of this thesis. This will be addressed as a separate work in the future.

- **Incorporating the proposed approaches with other automated program repair techniques:** The goal of this thesis is to devise program repair techniques that handle expression level bugs. In future, existing program repair approaches that work on statement level [18], [84] or method level [13] can be modified to incorporate the proposed approaches for complementing those techniques.

Currently, automated program repair is at a primary stage of research [87]. To find the correct patch, existing approaches as well as the proposed ones defined static search space. For example, GenProg [17] and LSRepair [13] work at statement and method level respectively, whereas the proposed techniques operate on the expression level. However, since these strategies are problem dependent, till now no technique has been found that performs well in all cases [111]. In future, a novel program repair technique can be devised that will dynamically adjust its search space based on the characteristics of the buggy program such as average method complexity. In addition, since program repair techniques largely depend on test suite, further research can be carried out to identify how test cases can be designed for improving automated program repair.

Appendix A

Table A.1: Selected Bugs of Defects4J Benchmark

Bug ID	**Bug ID**	**Bug ID**	**Bug ID**	**Bug ID**
Chart_1	Closure_65	Lang_26	Math_30	Math_82
Chart_9	Closure_71	Lang_33	Math_32	Math_85
Chart_10	Closure_73	Lang_38	Math_33	Math_94
Chart_11	Closure_77	Lang_43	Math_34	Math_96
Chart_13	Closure_107	Lang_51	Math_41	Math_105
Chart_24	Closure_113	Lang_57	Math_57	Mockito_7
Closure_10	Closure_123	Lang_59	Math_58	Mockito_24
Closure_14	Closure_125	Lang_61	Math_59	Mockito_29
Closure_18	Closure_130	Math_2	Math_63	Mockito_34
Closure_38	Closure_133	Math_5	Math_69	Mockito_38
Closure_51	Lang_6	Math_10	Math_70	Time_4
Closure_52	Lang_16	Math_11	Math_75	Time_19
Closure_57	Lang_24	Math_27	Math_80	

Bibliography

[1] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys (CSUR)*, vol. 51, no. 1, p. 17, 2018.

[2] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)-Volume 1*, pp. 471–482, IEEE, 2015.

[3] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, pp. 102–113, IEEE, 2019.

[4] 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)*, pp. 298–309, ACM, 2018.

[5] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pp. 1–11, ACM, 2018.

[6] Z. Chen and M. Monperrus, "The remarkable role of similarity in redundancy-based program repair," *Computing Research Repository (CoRR)*, vol. abs/1811.05703, 2018.

[7] M. White, M. Tufano, M. Martinez, M. Monperrus, and D. Poshyvanyk, "Sorting and transforming program repair ingredients via deep learning code similarities," in *Proceedings of the IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 479–490, IEEE, 2019.

[8] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches," in *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pp. 492–495, ACM, 2014.

[9] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 306–317, ACM, 2014.

[10] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *Proceedings of the 32nd International Conference on Automated Software Engineering (ASE)*, pp. 648–659, IEEE, 2017.

[11] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pp. 789–799, ACM, 2018.

[12] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *Proceedings of the 32nd International Conference on Automated Software Engineering (ASE)*, pp. 660–670, IEEE, 2017.

[13] K. Liu, A. Koyuncu, K. Kim, D. Kim, and T. F. D. A. Bissyande, "Lsrepair: Live search of fix ingredients for automated program repair," in *Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 658—662, 2018.

[14] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandé, and Y. Le Traon, "A closer look at real-world patches," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME)*, pp. 275–286, IEEE, 2018.

[15] R. Gallardo, S. Hommel, S. Kannan, J. Gordon, and S. B. Zakhour, *The Java tutorial: a short course on the basics.* Addison-Wesley Professional, 2014.

[16] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 2013 International Conference on Software Engineering (ICSE)*, pp. 802–811, IEEE, 2013.

[17] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering (TSE)*, vol. 38, no. 1, pp. 54–72, 2011.

[18] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pp. 254–265, ACM, 2014.

[19] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 31–42, 2019.

[20] S. Wang, M. Wen, X. Mao, and D. Yang, "Attention please: Consider mockito when evaluating newly proposed automated program repair techniques," in *Proceedings of the Evaluation and Assessment on Software Engineering (EASE)*, pp. 260–266, ACM, 2019.

[21] S. Wang, X. Mao, N. Niu, X. Yi, and A. Guo, "Multi-location program repair strategies learned from successful experience (S)," in *Proceedings of the 31st International Conference on Software Engineering and Knowledge Engineering (SEKE)* (A. Perkusich, ed.), pp. 713–777, KSI Research Inc. and Knowledge Systems Institute Graduate School, 2019.

[22] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[23] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, pp. 702–713, IEEE, 2016.

[24] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (FSE)*, pp. 532–542, ACM, 2013.

[25] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, pp. 321–330, ACM, 2008.

[26] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*, pp. 437–440, ACM, 2014.

[27] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH)*, pp. 55–56, ACM, 2017.

[28] R. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. Prasad, "Bugs. jar: A large-scale, diverse dataset of real-world java bugs," in *Proceedings of the 15th International Conference on Mining Software Repositories (MSR)*, pp. 10–13, IEEE, 2018.

[29] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *Proceedings of the 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, pp. 213–224, IEEE, 2016.

[30] C. Ragkhitwetsagul, J. Krinke, and D. Clark, "A comparison of code similarity analysers," *Empirical Software Engineering (EMSE)*, vol. 23, no. 4, pp. 2464–2519, 2018.

[31] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION)*, pp. 89–98, IEEE, 2007.

[32] T.-D. B. Le, F. Thung, and D. Lo, "Theory and practice, do they match? a case with spectrum-based fault localization," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, pp. 380–383, IEEE, 2013.

[33] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 302–313, 2019.

[34] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 19–30, 2019.

[35] H. Ye, M. Martinez, and M. Monperrus, "Automated patch assessment for program repair at scale," *arXiv preprint arXiv:1909.13694*, 2019.

[36] J. S. Collofello and S. N. Woodfield, "Evaluating the effectiveness of reliability-assurance techniques," *Journal of systems and software*, vol. 9, no. 3, pp. 191–195, 1989.

[37] J. A. Jones, "Fault localization using visualization of test information," in *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, pp. 54–56, IEEE, 2004.

[38] H. Zhong and Z. Su, "An empirical study on real bug fixes," in *Proceedings of the 37th International Conference on Software Engineering (ICSE)*, vol. 1, pp. 913–923, IEEE, 2015.

[39] C. Padmini, "Beginners guide to software testing," 2004.

[40] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering (TSE)*, vol. 42, no. 8, pp. 707–740, 2016.

[41] M. Zhivich and R. K. Cunningham, "The real cost of software errors," *IEEE Security & Privacy*, vol. 7, no. 2, pp. 87–90, 2009.

[42] "Iso/iec/ieee international standard - systems and software engineering – vocabulary," *ISO/IEC/IEEE 24765:2010(E)*, pp. 1–418, 2010.

[43] N. Chauhan, *Software Testing: Principles and Practices*. Oxford university press, 2010.

[44] R. Almaghairbe and M. Roper, "Separating passing and failing test executions by clustering anomalies," *Software Quality Journal*, vol. 25, no. 3, pp. 803–840, 2017.

[45] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search," in *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)*, pp. 295–306, IEEE, 2015.

[46] D. W. Binkley and K. B. Gallagher, "Program slicing," in *Advances in Computers*, vol. 43, pp. 1–50, Elsevier, 1996.

[47] W. Masri, R. Abou-Assi, M. El-Ghali, and N. Al-Fatairi, "An empirical study of the factors that reduce the effectiveness of coverage-based fault localization," in *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 1–5, 2009.

[48] S. Sun, J. Guo, R. Zhao, and Z. Li, "Search-based efficient automated program repair using mutation and fault localization," in *Proceedings of the IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 174–183, IEEE, 2018.

[49] C. Liu, L. Fei, X. Yan, J. Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Transactions on Software Engineering (TSE)*, vol. 32, no. 10, pp. 831–848, 2006.

[50] T. Ackling, B. Alexander, and I. Grunert, "Evolving patches for software repair," in *Proceedings of the 13th Annual Genetic and Evolutionary Computation Conference (GECCO)*, pp. 1427–1434, 2011.

[51] T. Ji, L. Chen, X. Mao, and X. Yi, "Automated program repair by using similar code containing fix ingredients," in *Proceedings of the 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 1, pp. 197–202, IEEE, 2016.

[52] M. Martinez and M. Monperrus, "Astor: Exploring the design space of generate-and-validate program repair beyond genprog," *Journal of Systems and Software*, vol. 151, pp. 65–80, 2019.

[53] D. Fu, Y. Xu, H. Yu, and B. Yang, "Wastk: a weighted abstract syntax tree kernel method for source code plagiarism detection," *Scientific Programming*, vol. 2017, pp. 7809047:1–7809047:8, 2017.

[54] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, pp. 783–794, IEEE, 2019.

[55] E. Fast, C. Le Goues, S. Forrest, and W. Weimer, "Designing better fitness functions for automated program repair," in *Proceedings of the 12th Annual Genetic and Evolutionary Computation Conference (GECCO)*, pp. 965–972, 2010.

[56] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. Le Traon, "Fixminer: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering (EMSE)*, pp. 1–45, 2020.

[57] X.-B. D. Le, T.-D. B. Le, and D. Lo, "Should fixing these failures be delegated to automated program repair?," in *Proceedings of the 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 427–437, IEEE, 2015.

[58] W. Weimer, S. Forrest, C. Le Goues, and T. Nguyen, "Automatic program repair with evolutionary computation," *Communications of the ACM*, vol. 53, no. 5, pp. 109–116, 2010.

[59] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pp. 364–374, IEEE, 2009.

[60] G. J. Myers, C. Sandler, and T. Badgett, *The art of software testing.* John Wiley & Sons, 2011.

[61] T. Britton, L. Jeng, G. Carver, P. Cheak, and T. Katzenellenbogen, "Reversible debugging software," *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 2013.

[62] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pp. 12–23, 2018.

[63] S. Kim and E. J. Whitehead Jr, "How long did it take to fix bugs?," in *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR)*, pp. 173–174, 2006.

[64] B. Cornu, *Automatic Analysis and Repair of Exception Bugs for Java Programs.* PhD thesis, 2015.

[65] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?," in *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pp. 361–370, 2006.

[66] Y. Qi, X. Mao, Y. Lei, and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*, pp. 191–201, 2013.

[67] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske, "A learning-to-rank based fault localization approach using likely invariants," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, pp. 177–188, 2016.

[68] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. Van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780–1792, 2009.

[69] A. Bandyopadhyay and S. Ghosh, "Proximity based weighting of test cases to improve spectrum based fault localization," in *Proceedings of the 26th International Conference on Automated Software Engineering (ASE)*, pp. 420–423, IEEE, 2011.

[70] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA)*, pp. 49–60, 2010.

[71] Y. Qi, X. Mao, and Y. Lei, "Efficient automated program repair through fault-recorded testing prioritization," in *Proceedings of the International Conference on Software Maintenance (ICSM)*, pp. 180–189, IEEE, 2013.

[72] R. K. Saha, H. Yoshida, M. R. Prasad, S. Tokumoto, K. Takayama, and I. Nanba, "Elixir: an automated repair tool for java programs," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings (ICSE)*, pp. 77–80, 2018.

[73] M. Sergey, *Semantic Program Repair*. PhD thesis, National University of Singapore (Singapore), 2018.

[74] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, pp. 24–36, 2015.

[75] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 298–312, 2016.

[76] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pp. 727–738, 2016.

[77] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the quixbugs benchmark," in *Proceedings of the 2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*, pp. 1–10, IEEE, 2019.

[78] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, pp. 416–426, IEEE, 2017.

[79] L. Zemín, S. G. Brida, A. Godio, C. Cornejo, R. Degiovanni, G. Regis, N. Aguirre, and M. Frias, "An analysis of the suitability of test-based patch acceptance criteria," in *Proceedings of the 10th International Workshop on Search-Based Software Testing (SBST)*, pp. 14–20, IEEE, 2017.

[80] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering (TSE)*, 2017.

[81] X.-B. D. Le, "Towards efficient and effective automatic program repair," in *Proceedings of the 31st International Conference on Automated Software Engineering (ASE)*, pp. 876–879, ACM, 2016.

[82] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. D. A. Bissyande, D. Kim, P. Wu, J. Klein, X. Mao, and Y. Le Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs," in *Proceedings of the 42nd International Conference on Software Engineering (ICSE)*, 2020.

[83] Z. B. Zabinsky, "Random search algorithms," *Wiley encyclopedia of operations research and management science*, 2010.

[84] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pp. 3–13, IEEE, 2012.

[85] R. Al-Ekram, A. Adma, and O. Baysal, "diffx: an algorithm to detect changes in multi-version xml documents," in *Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, pp. 1–11, Citeseer, 2005.

[86] A. Zeller, "Yesterday, my program worked. today, it does not. why?," *ACM SIGSOFT Software engineering notes*, vol. 24, no. 6, pp. 253–267, 1999.

[87] M. Monperrus, "A critical review of" automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 234–242, 2014.

[88] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis," in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pp. 772–781, IEEE, 2013.

[89] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 43–54, 2015.

[90] V. Thada and V. Jaglan, "Comparison of jaccard, dice, cosine similarity coefficient to find best fitness value for web retrieved documents using genetic algorithm," *International Journal of Innovations in Engineering and Technology*, vol. 2, no. 4, pp. 202–205, 2013.

[91] J. Gosling, B. Joy, G. Steele, G. Bracha, and A. Buckley, "The java language specification (java se 8 edition)," *California: Oracle*, 2015.

[92] K. Noda, Y. Nemoto, K. Hotta, H. Tanida, and S. Kikuchi, "Experience report: How effective is automated program repair for industrial software?," in *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 612–616, IEEE, 2020.

[93] J. Campos, A. Riboira, A. Perez, and R. Abreu, "Gzoltar: an eclipse plug-in for testing and debugging," in *Proceedings of the 27th International Conference on Automated Software Engineering (ASE)*, pp. 378–381, ACM, 2012.

[94] B. Cao, Y. Li, and J. Yin, "Measuring similarity between graphs based on the levenshtein distance," *Appl. Math*, vol. 7, no. 1L, pp. 169–175, 2013.

[95] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon, "Facoy: a code-to-code search engine," in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pp. 946–957, 2018.

[96] S. Clémençon, G. Lugosi, N. Vayatis, *et al.*, "Ranking and empirical minimization of u-statistics," *The Annals of Statistics*, vol. 36, no. 2, pp. 844–874, 2008.

[97] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax-and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE)*, pp. 593–604, ACM, 2017.

[98] F. P. Miller, A. F. Vandome, and J. McBrewster, *Apache Maven*. Alpha Press, 2010.

[99] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, "Symbolic pathfinder: integrating symbolic execution with model checking for java bytecode analysis," *Automated Software Engineering*, vol. 20, no. 3, pp. 391–425, 2013.

[100] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "Jfix: semantics-based repair of java programs via symbolic pathfinder," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 376–379, 2017.

[101] "Jdt core component." https://www.eclipse.org/jdt/core/index.php. Accessed: 2020-05-17.

[102] "Javalang." https://github.com/c2nes/javalang. Accessed: 2020-04-22.

[103] R. E. Walpole, R. H. Myers, S. L. Myers, and K. Ye, *Probability and statistics for engineers and scientists*, vol. 5. Macmillan New York, 1993.

[104] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in statistics*, pp. 196–202, Springer, 1992.

[105] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 226–236, ACM, 2017.

[106] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, "Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system," *Empirical Software Engineering (EMSE)*, vol. 24, no. 1, pp. 33–67, 2019.

[107] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 31–42, ACM, 2019.

[108] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *Proceedings of the 32nd International Conference on Automated Software Engineering (ASE)*, pp. 637–647, IEEE, 2017.

[109] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on Software Engineering (TSE)*, 2018.

[110] V. Sobreira, T. Durieux, F. Madeiral, M. Monperrus, and M. de Almeida Maia, "Dissection of a bug dataset: Anatomy of 395 patches from defects4j," in *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 130–140, IEEE, 2018.

[111] A. Aleti and M. Martinez, "E-APR: mapping the effectiveness of automated program repair," *Computing Research Repository (CoRR)*, vol. abs/2002.03968, 2020.

[112] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges," in *Proceedings of the 30th International Conference on Automated Software Engineering (ASE)*, pp. 201–211, IEEE, 2015.

[113] S. Wang and D. Zou, "Automated patch correctness assessment: How far are we?," in *Proceedings of the 35th International Conference on Automated Software Engineering (ASE)*, IEEE, 2020.

[114] Z. P. Fry, B. Landau, and W. Weimer, "A human study of patch maintainability," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pp. 177–187, 2012.

[115] T. Durieux and M. Monperrus, *IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs.* PhD thesis, Universite Lille 1, 2016.