DEGREE PROJECT IN THE FIELD OF TECHNOLOGY
INFORMATION AND COMMUNICATION TECHNOLOGY
AND THE MAIN FIELD OF STUDY
COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2018

The Essence of Similarity in Redundancy-based Program Repair

ZIMIN CHEN

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

The Essence of Similarity in Redundancy-based Program Repair

Zimin Chen
zimin@kth.se

Master's Thesis in Computer Science (30 ECTS credits)
at the School of Electrical Engineering and Computer Science (EECS)
Royal Institute of Technology year 2018

supervised by
Prof. Martin Monperrus

examined by
Prof. Erik Fransén

November 15, 2018

Abstract

Recently, there have been original attempts to use the concept of similarity in program repair. Given that those works report impressive results in terms of repair effectiveness, it is clear that similarity has an important role in the repair process. However, there is no dedicated work to characterize and quantify the role of similarity in redundancy-based program repair. This is where our paper makes a major contribution, we perform a deep and systematic analysis of the role of similarity during the exploration of the repair search space. We show that with similarity analysis, only 0.65% of the search space (on average over 2766 bugs) must be explored before finding the correct patch. And it is capable of ranking the correct ingredient first in the most of the time (65%).

Sammanfattning

Nyligen har nydanande försök gjorts att använda likhetsbegrepp i programreparation. Eftersom dessa experiment visade imponerade resultat när det gäller reparationseffektivitet, är det uppenbart att likhet spelar en viktig roll i reparationsprocessen. Dock finns det inget tidigare arbete som fokuserar på att karakterisera och kvantifiera rollen av likhet i redundansbaserad programreparation. Det är här som detta examensarbete ger ett stort bidrag. Vi utför en djup och systematisk analys av likhetsrollen vid en genomsökning av reparationssökutrymmet. Vi visar att med likhetsanalys räcker det att genomsöka 0.65% av sökutrymmet(medeltal 2766 buggar) för att hitta den rätta satsen. Dessutom visas att likhetsanalysen lyckas rangordna den rätta satsen som första alternativ för de flesta fallen (65%).

Acknowledgments

I would like to start by thanking my supervisor, Prof. Martin Monperrus, for his invaluable inspirations, countless discussions and continual inputs.

Thank you Prof. Matias Martinez for providing the interface needed from ASTOR.

Contents

Chapter 1

Introduction

In program repair research, one can identify at least two broad categories of papers. On the one hand, there are papers which propose new repair techniques based on a powerful innovative idea (such as [43, 15, 29, 41], to only mention recent ones). On the other hand, there are papers which pursue the endeavor of understanding the core structure and challenges of the repair search space (such as [26, 34, 24, 22]). The work presented here fits in this latter category.

Redundancy-based program repair consists of repairing programs with code taken from elsewhere, for instance from the application under repair. Redundancy-based program repair is fundamental to the repair community, early approaches such as GenProg [42] and major state-of-the-art approaches (e.g. [43, 17, 15]) both relied on redundancy. While the empirical presence of redundancy has been studied [27, 5], the search space of redundancy-based repair is little known.

Recently, there have been original attempts to use the concept of similarity in redundancy-based repair techniques. In [45, 43, 17], different kinds of similarity analysis are part of the overall repair technique. Qi et al. [45] use TFIDF to compute the similarity between the buggy statement and its context. Wen et al. [43] prioritize potential repair code snippets that have a similar context, with three similarities stacked in a clever manner. Jiang et al. [17] heavily rely on name similarity (variable name and method name) to find code snippets that are present in other similar methods.

1.1 Problem

Given that those works report impressive results in terms of repair effectiveness, it is clear that similarity has an important role in the repair process. However, there is no dedicated work to characterize and quantify the role of similarity in redundancy-based program repair. Thus in this study, we want to understand the search space of redundancy-based program repair and how similarity helps program repair to explores the search space.

```
public class Test{
    int a = 1;
-   int b = 0.1;
+   double b = 0.1;
}
```

Listing 1: A toy example of a one-line replacement patch. Removed line is prefixed by "-" and inserted line is prefixed by "+"

1.2 Objective

Our study of similarity in redundancy-based program repair aims to use a strictly principled methodology. First, we will carefully qualify the similarity relationship that we consider: in the context of replacement patch where one code snippet is replaced by another code snippet, we measure the similarity between the removed and the inserted code. We will define three similarity metrics that capture either syntactic or semantic similarity. Second, we will isolate the similarity component in ssFix [45], a state-of-the-art redundancy-based repair algorithm. This isolation means that we simplify the repair algorithm as much as possible (no randomness, no combined techniques) so that similarity analysis becomes the main component of the repair process. Thus, we ensure that the observed effects on the search space are those of similarity analysis.

1.3 Terminology

In this section, we define the main terminology of redundancy-based repair that is used in our paper.

1.3.1 Replacement Patch

Program repair systems use several different ways to generate patches. In this paper, we focus on replacement-only patches, that are patches made by replacing a code fragment with another code fragment. Deletion of code is obviously not redundancy-based repair. Addition of new code that already exists is redundancy-based repair but is not considered in this paper.

In the rest of the paper, without loss of generality, we will focus on one-line replacement patches, which are patches that only one code line is replaced by another code line in a file. They are really well known by professional developers, who call them "one-line patches", such as patch #977532 in the Linux kernel[1] and our toy example in Listing 1.

[1] https://lore.kernel.org/patchwork/patch/977532/

1.3.2 Modification Point

The modification point is where the source code is modified to fix a bug. In our toy example from Listing 1, the modification point is *int b = 0.1;*. Modification point is also the unit of fault localization. When doing fault localization at the line level, all suspicious lines are potential modification points, which will also be called suspicious modification points.

1.3.3 Repair Ingredient

In this paper, the repair ingredient is the code that replaces the code at the modification point, i.e. the line that replaces the faulty line. In the search space of all possible ingredients, we have "correct ingredient", "plausible ingredient" and "incorrect ingredient". The "plausible ingredient" is the ingredient that passes the test suite after replacing the modification point. The "correct ingredient" is the ingredient that is plausible and actually fix the bug. The "incorrect ingredient" is the ingredient that is not plausible. In our toy example from Listing 1, the correct ingredient is *double b = 0.1;*. Per the definitions of [27], the "ingredient pool" is a shorter name for the search space of redundancy-based program repair.

1.4 Delimitation

Our study will only focus on redundancy-based program repair. We only use three similarity metrics and only measure the similarity between the modification point and the correct ingredient.

1.5 Ethics

Manual bug fixing is a costly and time consuming process. Britton et al. [7] found that 312 billion dollars are spent annually on debugging and developers spend 50% of their time on bug fixing. Automatic program repair tries to automate the process and save resources from the software development process. However, this automation can cause ethical debate.

One obvious concern for many people is that automatic program repair might end up taking their jobs. Even though automatic program repair still faces many problems and are not being used in practice. The recent progress in machine learning and the need for automation might speed up the research in APR.

Another hot topic in today's society and is relevant for APR is the concern for the "evil" AI. By developing automatic program repair, we are giving programs the ability the modify its own source code, which can be problematic in some cases. Imagine if human defined rules implemented in the source code are mistakenly considered as bugs. Then, APR repairs the "bugs", and it will most certainly also alter the rules defined by humans. It can cause AI to behave differently and have catastrophic consequences.

1.6 Structure of Report

The remainder of this report will be structured as follows. Chapter 2 gives the background knowledge to automatic program repair. Chapter 3 provides information regarding the implementation. Chapter 4 explains the experimental methodology and the used datasets. Chapter 5 presents all results obtained from experiments and discusses the implication around the results. Chapter 6 presents the conclusion.

Chapter 2

Background

In this chapter, the background knowledge will be provided. We will start by briefly explain automatic program repair (APR)[1] in general. Then, we move our focus towards redundancy-based program repair and similarity in program repair.

2.1 Automatic Program Repair

Automatic program repair, is the transformation of of an unacceptable behavior of a program execution into an acceptable one according to a specification [32]. The unacceptable behavior is often called bug. Generally, APR performs three steps to achieve this transformation: fault localization, patch generation, patch validation. Fault localization identifies suspicious source code that cause the bug. Patch generation tries to fix the bug by generate a patch where the source code is modified. Patch validation checks if the patched program execution is acceptable according to the specification.

The most dominant fault localization strategy is spectrum-based fault localization. Spectrum-based fault localization approaches execute test suite of the program and record the entities(e.g. statement, methods, lines) that were executed by each test case [2]. Then, it uses formula to calculate a suspicious score for each entity indicating how likely it caused the bug. The intuition is that entities executed by failing test cases are more likely to be faulty.

Redundancy based APR, is one of the most common patch generation technique. It assumes that the correct ingredient exists somewhere else in the program [5]. Normally, the search space for possible patches is infinite, but the redundancy based APR reduces the problem by only searching code snippet in the same program that can fix the bug. GenProg [21], CapGen [43], DeepRepair [44], ssFix [45], SCRepair [16] and SimFix [17] are all example of redundancy based APR.

Test-suite based APR, the undisputed most popular patch validation technique, uses test suite as the program specification. It assumes that the test suite

[1]In our paper, we use "automatic program repair", "APR" and "program repair" interchangeably

defines the normal behavior of the program, and any failing test cases are exposure of bugs. Therefore, test-suite based repair tries to find patch(es) that pass all test cases. Nearly all APR tools are all test-suite based repair. But there are exceptions, such as DeepFix [13] and RLAssist [12], which use compiler as patch validation.

2.2 Redundancy-based Program Repair

Redundancy-based program repair consists of repairing programs with code taken from elsewhere, for instance from the application under repair. The insight behind it is that source code is very redundant [11], which means that the same function or snippet can be implemented at multiple locations in slightly different ways. Consequently, a bug that affects a code snippet can be fixed by another code snippet.

Redundancy-based program repair is fundamental to the repair community as it originates from the inception of the field: back in 2009, GenProg [42] already leveraged redundancy. Today, major state-of-the-art approaches still heavily depend on redundancy (e.g. [43, 17, 15]).

The redundancy assumption — the assumption that code may evolve from existing code that comes from somewhere else — has been empirically verified. Martinez et al. [27] and Barr et al. [5] both verified its validity by analyzing thousands of past commits: between 3% and 17% of commits are only composed of existing code. Recently, it has been reported that a redundancy-based approach successfully scales to an industrial project [33].

Furthermore, Martinez et al. [27] have refined this idea with the concept of "redundancy scope", which defines the boundaries from which the repair ingredients are taken. For instance, the file scope means that repair ingredients are only taken in the same file from which the suspicious faulty statement comes (this is the default scope of GenProg). For another example, the application scope means that the ingredients come from the whole application. In this paper, we focus on the application scope.

2.3 Search space of Redundancy-based Repair

In theory, the search space of program repair is virtually infinite, composing of all possible edits on a program. Redundancy-based program repair is one way to dramatically reduce the search space. In essence, it reduces the search space to the number of repair ingredients that already exist, where a repair ingredient is either a token, a line or a full snippet. Consider the following example as a analogy, imagine in the math textbook, we have:

$$\text{Find } X, \text{such that } f(X) = 7$$

The search space for X is infinite, it can be any number that exists on the number line. But the redundancy-based equation solver (analogy to the redundancy-based

```
public double getPct(Object v) {
-     return getCumPct((Comparable<?>) v);
+     return getPct((Comparable<?>) v);
}
```

Listing 2: A real one-line patch, from Apache Commons Math, with high similarity between the modification point and the repair ingredient.

program repair) collects all numbers (analogy to the repair ingredients) that exists in the same textbook, and try them one by one. For example if Euler's identity, $e^{i\pi} + 1 = 0$, exists in the textbook, the redundancy-based equation solver might try $f(e), f(1)$ etc., and check if the solutions is 7.

While the search space under the redundancy assumption is effectively reduced, it may still be too large in practice. In essence, it depends on the redundancy scope. The redundancy scope defines the scope where the ingredients are taken from. For instance, at the file scope, the search space size is proportional to the file size (usually hundreds of lines, at most thousands). At the application scope, the size of the search space, with some simplification, is the number of lines in the application. Thus, for a 1 million LOC application, the search space is composed of 1 million elements (or a certain number proportional to it), which is quite large.

In practice, exploring the search space means picking an ingredient, compiling it, and executing test cases. If we assume that the whole procedure takes 1 second for each ingredient, for 1 millions ingredients it would take 12 days. Since this takes some time, the search space of redundancy-based repair is often too big to be exhaustively explored.

When it is too slow to exhaustively explore the search space of redundancy-based repair, what is the solution to only explore the relevant parts of the search space? This is the open question we systematically explore in this paper.

2.4 Similarity Analysis for Redundancy-based Repair

Major recent work in program repair has shown that similarity analysis is a useful concept for program repair, being present in ssFix [45], CapGen [43] and SimFix [17]. For instance, let us consider the human patch of bug Math-75 from Defects4J [18] in Listing 2. We can see that the inserted line and the removed line are syntactically very similar. However, ssFix, CapGen and Simfix, they all consider different forms of similarity and use different metrics for similarity analysis.

For instance, ssFix finds suspicious statements that are more likely to be faulty with fault localization. Then, for each suspicious statement, the context and the suspicious statement is extracted to identify similar code chunk in the codebase with Lucene's default TFIDF model. The ingredient is then obtained from the similar code chunk to generate a patch.

On the other hand, CapGen considers context similarity, name similarity and dependency similarity between the suspicious code fragment and the ingredients at expression level. They extend the similarity analysis with results from their empirical study which calculates the frequency between different mutation operators (replacement, insertion and deletion) and the type of involved code fragment (method invocation, if statement and etc.) to prioritize ingredients that are more likely to fix the bug.

SimFix does also consider structure similarity, variable name similarity and method name similarity between the suspicious statement and ingredient. But they remove ingredients that are less frequent in existing patches. From this point of view, SimFix and CapGen are two very similar approach.

However, their evaluation only focus on the number of bugs that their tools can fix, not how similarity analysis affect the program repair process. In this paper, we will exclusively focus on the similarity in the context of redundancy-based repair for replacement patches. We will extensively study the similarity between the code being replaced and the repair ingredient according to our definition.

2.5 Overfitting in program repair

Overfitting is a problem when the generated patches are plausible but not correct. A patch is plausible when it passes the test cases. A patch is correct when it is plausible and generalizes outside of test cases. A patch is incorrect when it failed to pass to the test cases (Similar to the ingredient definition in subsection 1.3.3). The phenomenon is usually caused by weak test suite, where incorrect or plausible patches can still pass the test suite. Which is a setback to APR, since most of the repair approaches use test suite as program specification and validate any generated patches against it. This problem is known to be important: Qi et al.[35] analyzed patches generated by GenProg, RSRepair and AE. They found that only 5 of 414 GenProg patches, 4 of 120 RSRepair patches and 3 of 54 AE patches are correct (that it actually fixed the bug).

This problem has gained a lot of attention from the community, and is the next huge obstacle for APR[37]. But CapGen [43], a recent approach that uses similarity analysis, has achieved new progress in this problem. They managed to prioritize the correct patch before 98.78% of plausible ones [43].

2.6 Similarity Metrics

Our core idea is to compute the similarity between the modification point and all possible repair ingredients. Then, the repair ingredients are ordered by decreasing similarity, and tried in this order. In other words, the most similar ingredient is compiled first and executed against the failing test case.

In this section, we present the similarity metrics that we consider and the reasoning on why we chose these metrics.

2.6.1 Three Metrics

In this study, we will study three different similarity relationship, longest common subsequence, term frequency & inverse document frequency and word embedding based on unsupervised learning.

Longest Common Subsequence

Longest common subsequence (LCS), is the problem to find the longest common subsequence in two sequences. For example the LCS for *"ABCDEF"* and *"AJKBCIF"*, is *"ABCF"* of length 4. Normally, LCS outputs the longest common sequence, or the length of it. But LCS can be normalized [4] to output a score between 0 and 1. 0 means that two sequences are dissimilar, 1 means that two sequence are similar.

$$normalized\_LCS(x,y) = \frac{LCS(x,y)}{max(|x|,|y|)}$$

$LCS(x,y)$ is the length of LCS between x and y, $max(|x|,|y|)$ is the maximum length between x and y.

Term Frequency & Inverse Document Frequency

Term frequency & inverse document frequency, TFIDF, is a numerical static that is used to describe document based on the words that occur in them. TFIDF increases proportionally to the number of times the word occurs in the document and inverse to the frequency of the document containing the same word in the corpus. By doing so, TFIDF is capable to focus on keywords instead of word like *"the"*, which appears in almost all documents.

Word Embedding

Word embedding (Embedding) is a technique that maps words to vectors of real numbers. The vector is a high dimension representation and is capable of containing the semantic information of the word. Word2vec is a model that can used to produce word embedding [31], and has been proved to be very successful. The core idea behind word2vec is that words with similar context have similar meaning. Consider the following two text, *"The quick brown fox jumps over the lazy dog"* and *"The quick white fox jumps over the lazy dog"*. The word *brown* and *white* are surrounded by the same words, therefore they should have similar meaning. Word2vec have showed to be able to produce embedding such as *vector("King") - vector("Man") + vector("Woman")* $\approx$ *vector("Queen")* [30].

2.6.2 Rationale

The rationale behind using these three strategies is that they cover three different families of similarity relationships:

- **LCS** is purely syntactic because it works at the character level. There exist very efficient implementations of it in all major languages. In the context of programming, LCS is capable of capturing similarity between words like *port1* and *port2.*

- **TFIDF** is based on words frequency and rarity, which translates to token frequency and token rarity in the context of programs. Being based on tokenization, it is much less syntactic than LCS. The strength of TFID is its ability to focus on the important tokens. If the variable name *veryRareName* only occurred once in the file and it is in the modification point. Then, if we have an ingredient that contains *veryRareName*, it is likely that ingredient is a good candidate.

- **Embedding** is a numerical vector associated to an object, meant to capture semantic relationships. For instance, in natural language processing, a 3-dimension word embedding of "foo" could be $< 1, 0.9, 42 >$. Embeddings are typically learned from a dataset. In the context of programming, embeddings can be computed for tokens, lines, functions, etc. Embeddings are useful for computing similarity: a numerical distance metric (Euclidean or Cosine) in the vector space can be used straightforwardly. Learning an embedding on program tokens is meant to be less syntactic than LCS and TFIDF. For example, *dog* and *cat* should be considered semantically similar, i.e. adjacent in the vector space, while LCS and TFIDF would both consider them to be different compared to the word *fog* in their respective syntactic spaces.

2.6.3 Computation

In this section we will describe how we calculate the similarity between the suspicious modification point and ingredients.

Cosine Similarity

Cosine similarity measure similarity between two vectors which is based on the cosine of the angle between them. It is considered as standard practice in the field for measuring vector distance (inverse similarity) [36]. Consider two vectors X and Y, the cosine similarity is defined as:

$$Sim(X,Y) = \frac{\sum_{i=1}^{n} X_i Y_i}{\sqrt{\sum_{i=1}^{n} X_i^2}\sqrt{\sum_{i=1}^{n} Y_i^2}}$$

Similarity Score

We compute the three similarity relationships are follows:

- **LCS:** The ingredients and modification point are considered as sequences of characters. For each pair $< modificationpoint, ingredient >$, the normalized LCS is computed.

- **TFIDF:** All code lines are considered as documents. We then tokenize the lines, per the tokenization rule of the considered language (Java in our experiments). The term frequency and inverse document frequency is calculated for each token. All code lines will be converted into vectors of TFIDF scores. Finally, the similarity score is computed by using cosine similarity between the vector for modification point and vector for the ingredient.

- **Embedding:** From the trained Word2vec [31], each token will be associated with corresponding vector. In order to compute the embedding of an entire line, we take the average of all token embeddings in the code line. To average all token embedding in an entire line to get line embedding has been showed to be a strong baseline or feature for many tasks [20]. Finally, the similarity score is also computed by using cosine similarity between the vector for modification point and vector for the ingredient..

Chapter 3

Implementation

We now present our main implementation choices for our program repair tool and embedding.

3.1 Repair Tool

In order to study the role of similarity in redundancy-based program repair, we have implemented a prototype tool called 3sFix, on top of the ASTOR program repair framework [25]. It does replacement-based repair, at the line level, using similarity ordering of repair ingredients as described in section 2.6. It can be considered as a simple version of ssFix [45], hence its name 3sFix, standing for **S**imple **SS**Fix.

We emphasize that the tool is not a repair system per se, its goal is rather to be a scientific instrument to observe the search space of redundancy-based program repair and the role of similarity. The tool is made publicly available for the sake of open-science and reproducible research.

Classically, the repair loop is composed of three steps:

I **Fault localization:** We run the test suite and obtain all suspicious modification points based on Ochiai [1]. All suspicious modification points are ordered by the suspiciousness value, which indicates how suspicious the line is to cause the bug. Only the n most suspicious modification point are considered for repair, in decreasing suspiciousness order.

II **Patch generation:** We extract all ingredients at the application scope in order to have the ingredient pool. For each suspicious modification point, all ingredients from the pool are sorted based on the similarity score defined in subsection 2.6.3. The number of tried ingredients per suspicious modification point is bounded by a configuration parameter m. This means that the search is bounded by $n \times m$. As such, the search of 3sFix is fully bounded and controlled, and the repair approach is fully deterministic. Consequently, we have a complete understanding and characterization of the complexity of the search, which is not common in repair approaches.

III **Patch validation:** After we have replaced the code line at the suspicious modification point, ASTOR will run the test suite again and verify if it passes the test suite.

3.2 Embedding Implementation

For computing the embedding, we train a word2vec [31] implementation from Google's TensorFlow[1] on a large corpus of Java files (described in subsection 4.3.1). The embedding size is set to 64, which means that each token of the modification point and repair ingredient is represented by 64 floating point values. The token vocabulary is set to the 100,000 most common tokens in the corpus. There is also $< UNKNOWN >$ token for those tokens that are out of our vocabulary. The model is trained in 1,000,000 iterations.

[1]`https://www.tensorflow.org/`

Chapter 4

Experimental Methodology

We present the research questions and the design of original experiments to study the role of similarity in redundancy-based program repair.

4.1 Research Questions

- **RQ1**: *What is the search space of redundancy-based repair?*
  Redundancy-based repair approaches use ingredients that can be found somewhere else. Only a few studies have systematically studied the redundancy assumption [5, 27]. The goal of this research question is twofold. First, we want to improve the external validity of those past experiments by using a dataset that is large and new (i.e. different from those used in [5, 27]). Second, this initial study also acts as a sanity check for the next research question.

- **RQ2**: *How does similarity-based repair explores the Defects4J search space?*
  Defects4J is a benchmark of bugs heavily used in program repair research. However, to the best of our knowledge, there are no studies examining the characteristics of Defects4J with respect to redundancy-based repair. The goals of RQ2 are: 1) assessing whether Defects4J is appropriate for studying the search space of redundancy-based program repair, and 2) getting a comprehensive understanding on the strengths and limitations of similarity analysis on the bugs of Defects4J.

- **RQ3**: *How effective are the different similarity metrics to handle the search space of redundancy-based repair?*
  In redundancy-based repair, the search space consists of all ingredients that can be found in a certain scope. The number of ingredients in that scope is sometimes overwhelming and may not be exhaustively explored. Therefore, most redundancy-based repair tools are to only select ingredients at random. The main purpose of RQ3 is to compare the three similarity metrics presented in section 2.6. RQ3 is based on an original experimental protocol

(see subsection 4.2.3) that allows us to compute the search space reduction at a scale that is larger than those of Defects4J.

- **RQ4**: *What is the feasibility of using similarity analysis on a bigger redundancy scope?*
  The next frontier in redundancy-based program repair would be an ingredient scope that is across multiple software projects, bigger than the application scope. For instance, the Github scope in Java consists of all unique Java lines that are present on Github. The goal of this last research question is to study the feasibility of the Github scope, and whether similarity analysis would scale to a potentially immense ingredient pool.

4.2 Protocols

In this section, we explain our protocol to answer each research questions.

4.2.1 Protocol of RQ1

RQ1 studies the search space of redundancy-based program repair based on past commits. The idea is to compute the search space of redundancy-based program repair of one-line replacement patches that can be found in open-source repositories. This is done as follows:

I **Select all one-line replacement patches from a corpus of commits.** The corpus that we use is briefly presented in subsection 4.3.2.

II **Extract the removed line (modification point) and the inserted line (the repair ingredient).** The removed line will be seen as modification point and the inserted line will be seen as the correct repair ingredient from each one-line replacement patch. We explicitly removed one-line replacement patches if the repair ingredient is not unique, since it might give higher redundancy rate.

III **Compute the search space.** For each one-line replacement patch, the search space is composed of all unique lines of the application at the point in time of the commit. The formatting of the lines, incl. the indentation, is removed in order to have a canonical version.

IV **Analyze the search space.** We measure the size of the search space, and we look at whether the correct repair ingredient (extracted in step II) already exists in the application.

4.2.2 Protocol of RQ2

RQ2 studies the redundancy-based search space in the Defects4J benchmark. The protocol consists of two parts.

First, we study the characteristics of Defects4J according to redundancy. The number of bugs in Defects4J that are one-line replacement patches are computed with the help of the study by Sobreira et al. [38]. Then, we compute the ingredient pool, which is composed of all ingredient that can be found in the application. And look at whether each Defects4J bug can be repaired by redundancy-based program repair. This would give us the theoretical upper bound of what redundancy-based program repair can achieve on Defects4J.

Second, we perform redundancy-based program repair on each appropriate bug using 3sFix. For each suspicious line, the top 100 repair ingredients are tried based on the similarity score. In addition, we use a timeout of two hours per bug. We manually analyze the test-suite adequate patches to see whether they are syntactically identical or incorrect. For each bug that could in theory be handled by redundancy-based program repair (because the correct repair ingredient exists in the search space), we analyze the reasons of success or failure to repair. This can be considered as a systematic and qualitative analysis of the search space.

4.2.3 Protocol of RQ3

RQ3 is an experiment that performs realistic program repair simulations. Our idea is to take a real one-line patch, and to see whether redundancy-based program repair would predict the inserted line as patch when we know the modification point. This is done as follows:

I **Extract repair tasks.** We extract all one-line replacement commits from a corpus from the literature, presented in subsection 4.3.2 . Since the whole experiment is too computationally expensive, we take a random sample of them. For each commit, the removed line is considered as the modification point, and the inserted line is considered as the ground-truth correct ingredient.

II **Compute the search space for each task.** For each repair task, all ingredients at application scope are collected.

III **Compute the three similarity metrics.** We compute the similarity between the modification point and all ingredients with three similarity metrics described in section 2.6.

IV **Compute the ranking of the correct repair ingredient.** We check that the correct repair ingredient (the new line) has a higher similarity value than all other ingredients. If not, we compute its rank in the search space according to similarity.

V **Assess similarity effectiveness.** The collected values enables us to see whether similarity analysis yields a “perfect repair”, where perfect repair means that the ground-truth line is ranked first in the ingredient pool. It also enables us to compute the search space reduction. The search space reduction is defined as the ratio between the number of ingredients in the search space that in average must be considered before finding the solution and the total size of the search space. Assume a search space of 100 ingredients and the search space contains the correct ingredient. If a similarity based technique puts the correct ingredient in a short-list of 5 ingredients, it means that this technique yields a search space reduction of 5/100, i.e., the search space is reduced to 5% of the original size.

We call this a repair simulation for two reasons. First, the protocol assumes that the fault localization step works perfectly and returning the actual modification point. Second, we do not run any test. Running tests is not required because we focus on perfect repairs, where our approach outputs the actual human patch as first patch. This protocol is meant to eliminate the validity threats and the uncontrolled variables in order to purely focus on the effectiveness of the similarity relationships.

4.2.4 Protocol of RQ4

RQ4 is a speculative experiment to study whether one can vision to use ultra-large ingredient pools in redundancy-based program repair. In particular, we want to make first exploration of the potential Github scope: all code on Github written in a given programming language. This experiment consists of three main phases:

I **Create the Github scope.** Collecting all Java projects on Github is impossible due to bandwidth and rate limiting constraints. We approximate the Github scope by collecting all unique of curated snapshot of Github, described in subsection 4.3.1. Because of the size of Github scope, we use a Bloom filter [6] as data structure to store each line of code from Github Java Corpus. Bloom filter is a probabilistic data structure to check whether an element is in a set, hence is perfectly appropriate to study redundancy. It is probabilistic because false positive are possible, but false negative are not. As a result, the false positive rate – the rate of repair ingredients considered as redundant while they are not – is not null, yet small because $< 1e{-9}$ by configuration.

II **Re-execute the protocol of RQ1 using the Github ingredient pool.** All one-line replacement patches obtained from protocol of RQ1 will be used again to analyze the search space. We will measure the size of the search space (at Github scope this time) and check if the correct repair ingredient exist in the bloomfilter.

III **Analyze the trade off at the Github scope.** We perform an exploratory analyze on the benefits and difficulties of using Github scope. We will compared Github scope with the result of application scope obtained from RQ1.

4.3 Data

In this section, we present the dataset that we used in order to answer all the research questions.

4.3.1 Github Java Corpus

In two experiments, RQ3 and RQ4, we use Github Java Corpus [3], in the former case as training data for the learning the embedding, in the latter case as approximation of the Github scope for redundancy-based program repair. Github Java Corpus is a collection of Java code at large scale, it contains 14807 Java projects from Github, which are selected as being above average quality. In total, Github Java Corpus contains 2,130,264 Java files and 352,312,696 LOC.

4.3.2 CodRep Corpus

In RQ1 and RQ3, we need a corpus of one-line patches. For this, we use the CodRep corpus [8], The CodRep corpus contains 171605 patches from commits in 29 distinct projects in previous studies from the literature. This corpus is appropriate for our experiments because it contains 36562 unique one-line replacement patches. This corpus enables is to study redundancy-based program repair at a large-scale.

4.3.3 Defects4J Benchmark

Identifier	**Project name**	**# bugs**
Chart	JFreeChart	26
Closure	Closure Compiler	133
Lang	Apache commons-lang	65
Math	Apache commons-math	106
Mockito	Mockito	38
Time	Joda Time	27

Table 1: List of bugs included in Defects4J benchmark

In RQ2, we evaluate 3sFix on Defects4J. Defects4J is a large collection containing reproducible Java bugs to support software engineering research [18]. It consists of 395 bugs from 6 different open source projects. Many studies have evaluated their performance on Defects4J, which facilitate the comparison between different approaches. In our paper, we will call each bug by its identifier and number, i.e. the 14th bug in Closure Compiler will be called Closure_14.

Chapter 5

Experimental Results and Discussions

We now present the results of our large scale and novel experiments on the role of similarity in redundancy-based program repair, along with implication for each result on how we can use each finding in the broader context. The results will be summarized and the ethical aspects will be discussed.

5.1 Research Question 1 (Search Space Statistics)

Research question: What is the search space of redundancy-based repair?

In our first experiment, per the protocol described in subsection 4.2.1, we have analyzed the redundancy-based search space of 36562 one-line patches coming from all projects in CodRep corpus.

Finding 1. For those 36562 one-line replacement patches, 2781 (8%) of them satisfy the redundancy assumption at the application scope.

Implication 1. This confirms previous research showing that there is $3 - 17\%$ redundancy at the application scope [27, 5]. The external validity of this empirical knowledge is improved by our usage of a completely new dataset.

Finding 2. For those 36562 one-line replacement patches, the average search space consists of 113362 ingredients.

Implication 2. Given that in practice, the trial of one repair ingredient is of the order of magnitude of 1 - 10s (as reported anecdotally in [34] and confirmed by our own experiments), it is impossible to exhaustively explore the search space, because it would take 31 hours (assuming 1s second per ingredient) to 310 hours (in the 10s worst case) per suspicious line. In the optimistic case, exhaustively analyzing the search space of 100 suspicious statements would take 3100 hours, i.e. 18 weeks. This shows the need for a principled way to systematically explore the search space of redundancy-based program repair (as opposed to randomly), which is the most important motivation of our paper.

5.2 Research Question 2 (Redundancy in Defects4J)

Research question: How does similarity-based repair explores the Defects4J search space?

We used simple and deterministic program repair tool, 3sFix, to understand the search space on Defects4J. All 395 bugs are analyzed and we focused on bugs that in theory can be repaired by our redundancy-based program repair tool.

Finding 3. Out of the 395 bugs in Defects4J, 75 bugs are fixed in the human patch by a single one-line replacement. And out of 75 Defects4J bugs that can be fixed by a one-line replacement patch, 11 of them (15%) satisfy the redundancy assumption, meaning that the correct repair ingredient – the replacement line – is present elsewhere in the application.

Implication 3. These 11 bugs are appropriate subjects to study redundancy-based one-line replacement repair. Furthermore, it has been shown in Finding 1 that 8% of one-line replacement patches are amenable to redundancy-based program repair. Although the proportion here (15%) is not exactly the same, it is not fundamentally different, showing that only a minority replacement patches are fully redundant. Consequently, in the following, we apply 3sFix on those 11 bugs: Chart_1, Chart_12, Chart_20, Closure_86, Closure_123, Math_5, Math_41, Math_57, Math_70, Math_104 and Mockito_5.

Finding 4. The correct patch that lies somewhere in the search space of those 11 bugs is actually found by our prototype in three cases: Chart_1, Math_5 and Math_70. Listing 3 , Listing 4 and Listing 5 show the generated patch.

Implication 4. What is remarkable is that the same technique captures completely different repair operators: Listing 3 changes a single literal; Listing 4 changes method call arguments and Listing 5 changes a binary operator. In all cases, we can see that the inserted and removed code line are very similar, which is the main factor that has driven the search. All those patches are achieved thanks to a purely generic approach, where similarity makes no assumption whatsoever on the repair type. In contrast to repair systems dedicated to specific operators (e.g. Nopol [48]), we believe that generic approaches have the potential to discover original patches.

In program repair, there are sometimes plausible patches, which are patches that pass all human test cases, yet which may be incorrect. We observe that, there are other plausible patches in the search space of Defects4J. Yet, in all cases, the correct patch is always ranked first, before the other plausible ones, thanks to the similarity ranking. This shows the relevance of similarity analysis when the correct patch lies somewhere in the large search space.

```
if (real == 0.0 && imaginary == 0.0) {
-    return NaN;
+    return INF;
}
```

Listing 3: Correct patch for Math_5 ranked 1st with similarity analysis

```
{
- return solve(min, max);
+ return solve(f, min, max);
}
```

Listing 4: Correct patch for Math_70 ranked 1st with similarity analysis

Finding 5. For the remaining 8/11 bugs that were not fixed while the patch is in the search space, the reasons are as follows: 1 of them was caused by fault localization ineffectiveness, 5 was caused by fault localization failure and 2 was caused by similarity ineffectiveness.

Implication 5. Fault localization ineffectiveness is defined as when the fault localization assigns low rank to the faulty line. It is the case for one bug, Closure_123, where the actual faulty line is ranked 305 among all 486 suspicious modification points. As a result, the repair attempt has reached the 2-hour timeout before trying to repair the correct modification point. Upon closer inspection of the modification point, the rank of the correct ingredient is 23 according to similarity. This means that if we extend the execution, Closure_123 would in theory be fixed (because we know for sure that patch is in the search space). To verify our reasoning, we have re-executed 3sFix on Closure_123 with a 10-fold bigger timeout. The correct patch, identical to the human patch, was actually found in the search space, before other plausible patches, which confirms our understanding of the search space and the quality of the implementation. Interestingly, it is the first time ever that Closure_123 is reported as fixed in program repair.

Fault localization failure is defined as when fault localization completely fails and the faulty line was not returned by the fault localization component. This happens for the five bugs, Chart_12, Chart_20, Closure_86, Math_104 and Mockito_5. For repair approaches based on fault localization, it is literally impossible to fix a bug if fault localization does not work properly for it. The reasons is that the repair attempts are only made on suspicious elements.

```
CategoryDataset dataset = this.plot.getDataset(index);
- if (dataset != null) {
+ if (dataset == null) {
  return result;
}
```

Listing 5: Correct patch for Chart_1 ranked 1st with similarity analysis

Similarity ineffectiveness is defined as when the correct ingredient was not highly ranked by similarity, i.e. because the correct ingredient is not among the most similar ingredients. This happens for two bugs, Math_41 and Math_57. Recall that for each suspicious statement, 3sFix is configured to try at most n repair ingredients, in order to have a fully controlled search space ($n = 100$ in this experiment, based on results discussed in section 5.3). With a bigger value of n, those bugs would be repaired.

To sum up, there is always the option to explore more the search space, either by considering more suspicious modification point, or by trying more ingredients per suspicious modification point. The art of configuring the repair system lies is the balance between the considered number of suspicious points and the number of ingredients. In this experiment, the biggest problem hampering repair (6 cases) relates to fault localization, which calls for more research or implementation work.

Finding 6. Out of curiosity, we have tried 3sFix on the remaining bugs for which we know that the correct patch is not in the search space of one-line redundancy-based repair (according to the ground-truth human patch). A plausible patch is found for 18 bugs, yet all those patches are overfitting (they pass all tests but are incorrect).

Implication 6. This confirms that the test suites of Defects4J are weak at completely specifying the patch [28]. This also shows that overfitting patches do largely exist in the ingredient pool of redundancy-based repair, which has been suggested previously [24]. This suggests that redundancy-based program repair must be coupled with a overfitting detection system (e.g. [46]).

5.3 Research Question 3 (In-Vitro Repair)

Research question: How effective are the different similarity metrics to handle the search space of redundancy-based repair?

For 2766 one-line replacement patches sampled from the CodRep corpus (see subsection 4.3.2), we have applied the repair simulation described in subsection 4.2.3. Recall that this simulation isolates the role of similarity comparison, by considering the idealized repair task where we assume that the fault localization step gives the good modification point. The main result is shown in Table 2. The first column lists all projects. The second column shows the number of one-line replacement patch we considered. The third column display the average search space size for respective projects. The fourth, fifth and sixth column indicate the average rank of the correct ingredient, and in the parentheses we have the average rank put into perspective with the average size, showing the top percentage that the correct ingredient is in. The seventh, eighth, ninth column display the percentage of cases that the correct ingredient is ranked first, which we consider to be a perfect repair.

Finding 7. Similarity analysis enables to reduce the search space very effectively. For all considered similarity metrics, the ranking of repair ingredients results in

Project	# bugs	Average search space size	Average rank (reduction). Lower, better.			Perfect repair. Higher, better.		
			LCS	TFIDF	Embedding	LCS	TFIDF	Embedding
PocketHub	100	5963	90 (1.51%)	9 (0.15%)	35 (0.59%)	63%	53%	45%
Elasticsearch	100	119217	70 (0.06%)	43 (0.04%)	2885 (2.42%)	51%	63%	36%
LibGDX	100	112945	1009 (0.89%)	876 (0.78%)	955 (0.85%)	41%	86%	16%
Eclipse	8	2528	7 (0.28%)	3 (0.12%)	9 (0.36%)	50%	88%	50%
SWT	100	186958	1161 (0.62%)	10244 (5.48%)	14589 (7.8%)	50%	65%	28%
AspectJ	100	156625	1850 (1.18%)	12250 (7.82%)	14300 (9.13%)	58%	43%	41%
ZXing	58	38291	18 (0.05%)	2387 (6.23%)	2517 (6.57%)	62%	57%	31%
Ant	100	71949	137 (0.19%)	1628 (2.26%)	2259 (3.14%)	51%	60%	38%
JMeter	100	50048	71 (0.14%)	549 (1.1%)	1239 (2.48%)	60%	72%	56%
Log4j	100	24249	24 (0.1%)	17 (0.07%)	342 (1.41%)	60%	67%	51%
Tomcat	100	134265	889 (0.66%)	2110 (1.57%)	3617 (2.69%)	58%	61%	40%
Xerces	100	64410	558 (0.87%)	619 (0.96%)	1581 (2.45%)	49%	60%	24%
ECF	100	56175	382 (0.68%)	2749 (4.89%)	3320 (5.91%)	58%	55%	49%
WTP Incubator	100	59634	153 (0.26%)	1588 (2.66%)	2244 (3.76%)	49%	55%	36%
Xpand	100	51132	302 (0.59%)	208 (0.41%)	268 (0.52%)	62%	74%	48%
Cassandra	100	41541	324 (0.78%)	112 (0.27%)	205 (0.49%)	57%	74%	53%
Lucene/Solr	100	192628	186 (0.1%)	3044 (1.58%)	6125 (3.18%)	50%	74%	52%
OpenJPA	100	141508	1373 (0.97%)	4570 (3.23%)	7142 (5.05%)	46%	56%	42%
Wicket	100	61256	187 (0.31%)	84 (0.14%)	370 (0.6%)	55%	67%	34%
Commons-codec	100	9302	70 (0.75%)	92 (0.99%)	95 (1.02%)	71%	71%	64%
Commons-collections	100	30263	10 (0.03%)	186 (0.61%)	354 (1.17%)	56%	69%	48%
Commons-compress	100	15468	71 (0.46%)	155 (1%)	282 (1.82%)	65%	72%	51%
Commons-csv	100	2272	9 (0.4%)	6 (0.26%)	36 (1.58%)	78%	74%	66%
Commons-io	100	13894	122 (0.88%)	112 (0.81%)	201 (1.45%)	60%	77%	61%
Commons-lang	100	37354	238 (0.64%)	148 (0.4%)	72 (0.19%)	65%	77%	55%
Commons-math	100	68587	121 (0.18%)	883 (1.29%)	1726 (2.52%)	53%	70%	44%
Spring Framework	100	211132	2755 (1.3%)	2076 (0.98%)	10082 (4.78%)	73%	56%	32%
Storm	100	14783	421 (2.85%)	327 (2.21%)	490 (3.31%)	56%	62%	41%
Wildfly	100	181498	1346 (0.74%)	4274 (2.35%)	7376 (4.06%)	47%	47%	23%
Summary	2766	77276	504 (0.65%)	1820 (2.36%)	3024 (3.91%)	57%	65%	43%

Table 2: Experimental results on the effectiveness of using similarity to reduce the search space of redundancy-based program repair.

having the correct ingredient in the top of the search space (e.g. in the top 0.04% for project Elasticsearch with LCS ranking, and in the top 9.13% for project AspectJ with embedding ranking).

Implication 7. In the average case, the correct ingredient is ranked among the top 0.65% with LCS ranking, it means that in average,the search space is reduced by 150. This opens up new possibilities: either one use the saved resources to explore more suspicious statements, or one could imagine a much bigger search space. For instance, one may be able to use Github as ingredient pool.

Finding 8. According to this setup, TFIDF is the most effective technique to rank the correct ingredient first. For 27/29 projects, the TFIDF median rank for the correct ingredients is 1. Both LCS and Embedding are slightly less effective, but with no dramatic difference.

Implication 8. This result is directly actionable. To use similarity analysis of repair ingredients in an industrial repair system based on redundancy, TFIDF based on tokens is the best solution to implement.

Finding 9. Considering TFIDF, out of 2766 repair tasks, we obtain 1800 perfect repairs, that is 65%. Recall that we define a perfect repair as syntactically identical to the human patch, which is considered as a ground-truth, correct patch.

Implication 9. This result suggests that similarity comparison has an impact on overfitting in redundancy-based program repair. Since the correct ingredient is the ingredient ranked first, no patch is generated before that. By construction, the correct ingredient corresponds to ground-truth human patch. Consequently, there is no possible overfitting happening in those 1800 cases. This is known purely analytically, without requiring human and generated test cases. This confirms the result on Defects4J showing that when the ingredient is in the search space, it is found first.

Finding 10. The average rank of the correct ingredient for Embedding is always worse than for LCS and TFIDF. Compared to LCS and TFIDF, similarity comparison based on embedding is less powerful in the context of program repair.

Implication 10. As stated in section 2.6, we have chosen to study the Embedding similarity comparison, because it is meant to capture more meanings than the character-based LCS and the token based TFIDF. However, this sophistication does not prove to be necessary.

Our hypothesis is that we are at the bleeding edge of embedding research here. As discussed in subsection 2.6.3, our embedding technique has two phases: a token-embedding phase (which is a direct application of word embedding) followed by a line embedding phase (which is a variant of sentence embedding). In the machine learning community, word embedding is considered mature. On the contrary, there is no consensus and dominant technique on how to compute sentence embedding [9]. Our similarity comparison of repair ingredients is purely based on sentence embedding techniques, and we assume that there is room for improvement in this part of our proposed technique.

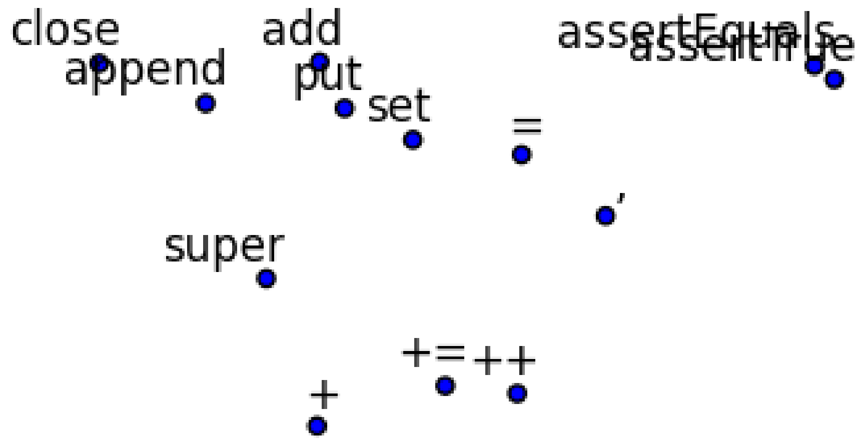

Figure 1: Excerpt of the Embedding space of Java tokens after learning on Github Java Corpus. The 64 dimensions have been project on a 2-dimension space.

Finding 11. The learned Embedding is meaningful as witnessed by a 2D projection shown in Figure 1. Tokens that are closely related at a semantic level are also close in the 2D space: `add` and `put` are close, so are the operators `+`, `+=` and `++`. The most interesting case is that `assertEquals` and `assertTrue` (top right part of the figure), with are extremely closed in this 2D projection.

Implication 11. While the previous finding is rather pessimistic (the sophisticated Embedding is not as good as the straight TFIDF), we want to present here the same result in an optimistic manner. The learned Embedding does capture meaningful similarity as shown in Figure 1. We consider this as encouraging evidence to continue researching about code analysis using embeddings (e.g. [14, 10]).

5.4 Research Question 4 (Limits of Similarity Comparison)

Research question: What is the feasibility of using similarity analysis on a bigger redundancy scope?

Recall that our technique's effectiveness is proportional to the number of ingredients: one always perform n similarity comparison per suspicious statement where n is the total number of repair ingredients. In this research we speculatively explore the maximum value of n.

Finding 12. In our implementation, the best similarity comparison based on average ranking is LCS. Within a time budget of 1 minute on a standard 2018 server (2.3GHz, 16Gb RAM), it enables us to compute 50000 similarity values between code lines.

Implication 12. In practice, it is reasonable to allocate 1 minute per suspicious statements to compute the similarity against the whole ingredient pool. With the current implementation, this fits with the application scope of most applications.

Under the assumption that a repair attempt for a single ingredient lasts 5 seconds (see Implication 2) and that the reduced search space contains 504 elements in average (per the results of Table 2). This single minute saves $5 \times (77276 - 504) = 383860$ seconds, that is 107 hours, where 77276 is the average search space size (per the results of Table 2).

Finding 13. The size of the search space at the Github scope is 58 millions, as captured in the considered corpus.

Implication 13. Without search space reduction such as the one presented in this paper based on similarity, the Github scope is impossible. Now, by using similarity analysis and based on our measurement from Finding 12, calculate the similarity score for each ingredient at the Github scope would take around 19 hours for a single suspicious modification point. This is not acceptable in the general case.

Finding 14. 27% of one-line replacement patches can be repaired with an ingredient that exists at the Github scope. This is more than 3 times as much as the application scope, as discussed in Finding 1.

Implication 14. The 27% repair redundancy at Github scope is exciting, it shows that the effectiveness of redundancy-based program repair could potentially be improved threefold. However, we have just shown that we are not able to rank such a large search space effectively in a reasonable time. But the promises of at least 3 times more interesting repair ingredients is worth optimizing the implementation (data structures, algorithms). Beyond pure engineering, it is also worth exploring in future work other similarity metrics that would be significantly faster and that would break the Github frontier.

5.5 Summary of the Experimental Results

The findings from RQ1 gave us the fundamental statistic about the search space of redundancy-based repair: the average size of the search space consists of 113362 code lines which is arguably too large to be exhaustively explored. We continued with an analysis of the search space for Defects4J, which is an heavily used research benchmark. The major finding from RQ2 shows that similarity works well if two conditions are met: if the correct ingredient is in the pool and if fault localization works properly. Our results highlights the importance of having good fault localization. Then, in RQ3, we have set up a controlled environment to isolate and demonstrate the effectiveness of using similarity. Our results indicate that similarity can reduce the search space by more than a factor of 100 (in average). Most importantly, in the majority of cases, similarity analysis is capable of ranking the correct ingredient first among hundred of thousand ingredients.

Finally, we have speculated about the Github redundancy scope in RQ4. The Github scope consists of 58 millions ingredients. This is too large to be handled out-of-the-box by any system, but it suggests that doing program repair at the Github scope with the help of similarity may be achievable in the mid-term.

5.6 Related work

In this section, we present the related work on various aspects, and the difference between our study and related work are also discussed.

5.6.1 Redundancy in Programs

After the initial successes of GenProg, studies have looked at the underlying redundancy assumption. Barr et al. [5] checked it against 12 Apache project, and found that changes are 43% redundant at line level. Martinez et al. [27] measured redundancy the at line and token level for 6 projects. They found that $3-17\%$ of commits are redundant at the line level. Sumi et al. [39] further conducted redundancy experiments using a larger dataset and obtain similar results. Lin et al. [23] examined code redundancy for 2640 Java projects with different token lengths for several types of code constructs, they studied how it affects the performance of code completion. Gabel et al. [11] investigated the opposite property, which is the uniqueness of source code. They found that software lacks uniqueness at the granularity of one to seven lines of code.

Repair approaches based on code exploits redundancy in a more functional way. Xiong et al. [47] use code search on Github to find snippets. Ke et al.'s approach [19] searches for existing code snippets (i.e. redundant ones that match a given input-output specification.

5.6.2 Analyses of the Repair Search Space

Martinez et al. [26] analyzed the repair actions over commits for 14 Java projects. They showed that certain repair actions are more common than others, *statement insertion of method invocation* is for instance the most common repair action. They analyze the search space of program repair with respect to those repair actions. Our study is different compared to theirs because we concentrate on repair ingredients while they focus on combinations of repair actions (what they call repair shapes).

Qi et al. [34] studied how random search compares to genetic programming to guide program repair through the search space. They showed that in most cases,random search outperforms GenProg. While they only focus on the first plausible patch, we compute the size of the whole search space, in order to calculate the search space reduction.

Long et al. [24] analyzed the search space for several repair systems. They found that are correct patches are sparse in the search space, while plausible patches are abundant. They also showed that increasing the size of search space decreases the ability to find the correct patch. To some extent, our study confirms the sparseness insight. It also provides new insights into the search space problem. First, we have studied how sparse the correct ingredient is in the search space, in a way that is decoupled to a specific repair approach. Second, we have measured how using similarity is effective in reducing the search space, and in finding the correct ingredients in the search space.

5.6.3 Similarity in Program Repair

Ji et al. [16] are possibly the first to have propose that the repair ingredients should be taken from similar code. Xin et al. [45] further built on this idea and proposed TFIDF to compute two similarities: the similarity between the ingredients and their respective context, and the buggy statement and its context. White et al. [44] uses deep learning to reason about similarity between the method body containing the modification point and the method body of ingredients. Tanikado et al. [40] proposed the original idea of looking at how fresh repair ingredients are, where freshness is defined by on the last updated time. Compared to our tool, the difference is that they consider a constant-sized region for each program statement while we considers code lines. Wen et al. [43] prioritized ingredients that have similar programming context based on program analysis. Jiang et al. [17] considers three different similarity levels, structure similarity, variable name similarity and method name similarity and the final similarity score is the sum of the three similarities. Those works have innovatively used similarity for program repair and we build on their initial results. Yet, they all focus on proposing a new repair technique and do not systematically analyze the search space. On the contrary, our paper is a principled study dedicated to the role of similarity in redundancy-based program repair.

5.7 Threats to Validity

The main threat to RQ1 is that the considered commits do not faithfully represent the field. To mitigate this threat, it is to be noted that the 29 different considered open-source projects were sampled independently.

There are little threats in RQ2 because there is ample knowledge about Defects4J in the literature. Our findings fit with what has been reported by previous work. Yet, an unfortunate bug in our experimental code may result in an underestimation of the number of repaired bugs.

RQ3 has shown that a semantic similarity metric based on a learned embedding is not better. One potential threat to validity is that the used corpus is not large enough to capture the intended semantics, or that a configuration parameter (the size of the embedding or of the vocabulary) could be further optimized.

Finally, RQ4 speculatively explores a new research direction, and as such is subject to many threats, such as the actual coverage of Github. Future work will mitigate those threats with specific protocols.

Chapter 6

Conclusion

We have designed and performed four large-scale experiments. The main goal of the experiments was to study the search space reduction obtained with similarity analysis. Our experimental results are clear-cut. First, the search space of redundancy-based program repair is indeed too large and it is not possible to exhaustively explore it. Second, using similarity analysis is effective in reducing the search space: our experiments show that compared to naive exhaustive exploration, only 0.65% of the search space (on average) must be explored before finding the correct patch. Third, our principled study methodology is validated, it enables us to have a full control over the search and a deep understanding of the search space.

To sum up, we make the following contributions:

- A principled conceptual framework to study three kinds of similarity in redundancy-based program repair.
- A quantitative analysis of the search space of redundancy-based repair over 171605 past commits from 29 projects, as well as the characteristics of Defects4J with respect to redundancy-based repair.
- A set of fundamental empirical findings about the remarkable performance of similarity in redundancy-based repair over 2766 bugs. Using similarity analysis enables us to explore only 0.65% of the search space, suggesting its essential role of cutting the search space by orders of magnitude (154x smaller).
- An original speculative result about the possibility of using the whole software universe (approximated by Github) for redundancy-based program repair

Based on our results, future work can explore alternative similarity metrics that would be either more effective or faster to compute. Our speculative research question RQ4 has shown that it is required to be able to perform redundancy-based program repair at really large scope such as the whole Github. Finally, we believe that similarity analysis would also be beneficial for synthesizing patches that involve multiples lines of code.

References

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. “An evaluation of similarity coefficients for software fault localization”. In: *Dependable Computing, 2006. PRDC’06. 12th Pacific Rim International Symposium on*. IEEE. 2006, pp. 39–46.

[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. “On the accuracy of spectrum-based fault localization”. In: *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007)*. IEEE. 2007, pp. 89–98.

[3] Miltiadis Allamanis and Charles Sutton. “Mining Source Code Repositories at Massive Scale using Language Modeling”. In: *The 10th Working Conference on Mining Software Repositories*. IEEE. 2013, pp. 207–216.

[4] Daniel Bakkelund. “An LCS-based string metric”. In: *Olso, Norway: University of Oslo* (2009).

[5] Earl T Barr et al. “The plastic surgery hypothesis”. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2014, pp. 306–317.

[6] Burton H. Bloom. “Space/Time Trade-offs in Hash Coding with Allowable Errors”. In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782.

[7] Tom Britton et al. “Reversible debugging software”. In: *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep* (2013).

[8] Zimin Chen and Martin Monperrus. *The CodRep Machine Learning on Source Code Competition*. Tech. rep. 1807.03200. arXiv, 2018.

[9] Ishita Dasgupta et al. “Evaluating Compositionality in Sentence Embeddings”. In: *arXiv preprint arXiv:1802.04302* (2018).

[10] Daniel DeFreez, Aditya V Thakur, and Cindy Rubio-González. “Path-Based Function Embedding and its Application to Specification Mining”. In: *Proceedings of ESEC/FSE*. 2018.

[11] Mark Gabel and Zhendong Su. “A study of the uniqueness of source code”. In: *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*. ACM. 2010, pp. 147–156.

[12] Rahul Gupta, Aditya Kanade, and Shirish Shevade. “Deep Reinforcement Learning for Programming Language Correction”. In: *arXiv preprint arXiv:1801.10467* (2018).

[13] Rahul Gupta et al. “DeepFix: Fixing Common C Language Errors by Deep Learning.” In: *AAAI*. 2017, pp. 1345–1351.

[14] Jordan Henkel et al. “Code Vectors: Understanding Programs Through Embedded Abstracted Symbolic Traces”. In: (2018).

[15] Jinru Hua et al. “Towards practical program repair with on-demand candidate generation”. In: *Proceedings of the 40th International Conference on Software Engineering*. ACM. 2018, pp. 12–23.

[16] Tao Ji et al. "Automated Program Repair by Using Similar Code Containing Fix Ingredients". In: *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)* 1 (2016), pp. 197–202.

[17] Jiajun Jiang et al. "Shaping Program Repair Space with Existing Patches and Similar Code". In: (2018).

[18] René Just, Darioush Jalali, and Michael 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 2014. San Jose, CA, USA: ACM, 2014, pp. 437–440. ISBN: 978-1-4503-2645-2.

[19] Yalin Ke et al. "Repairing Programs with Semantic Code Search". In: *Proceedings of the International Conference on Automated Software Engineering.* 2015.

[20] Tom Kenter, Alexey Borisov, and Maarten de Rijke. "Siamese cbow: Optimizing word embeddings for sentence representations". In: *arXiv preprint arXiv:1606.04640* (2016).

[21] Claire Le Goues et al. "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 '12. Zurich, Switzerland: IEEE Press, 2012, pp. 3–13. ISBN: 978-1-4673-1067-3.

[22] Xuan Bach D Le et al. "Overfitting in semantics-based automated program repair". In: *Empirical Software Engineering* (2018), pp. 1–27.

[23] Bin Lin et al. "On the uniqueness of code redundancies". In: *Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on.* IEEE. 2017, pp. 121–131.

[24] Fan Long and Martin Rinard. "An analysis of the search spaces for generate and validate patch generation systems". In: *Proceedings of the International Conference on Software Engineering.* 2016, pp. 702–713.

[25] Matias Martinez and Martin Monperrus. "ASTOR: A Program Repair Library for Java". In: *Proceedings of ISSTA.* 2016.

[26] Matias Martinez and Martin Monperrus. "Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing". In: *Empirical Software Engineering* 20.1 (2015), pp. 176–205.

[27] Matias Martinez, Westley Weimer, and Martin 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 Companion 2014. Hyderabad, India: ACM, 2014, pp. 492–495. ISBN: 978-1-4503-2768-8.

[28] Matias Martinez et al. "Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4J Dataset". In: *Empirical Software Engineering* 22.4 (2017), pp. 1936–1964.

[29] Sergey Mechtaev et al. "Semantic Program Repair Using a Reference Implementation". In: *Proceedings of ICSE.* 2018.

[30] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. "Linguistic regularities in continuous space word representations". In: *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies.* 2013, pp. 746–751.

[31] Tomas Mikolov et al. "Efficient estimation of word representations in vector space". In: *arXiv preprint arXiv:1301.3781* (2013).

[32] Martin Monperrus. "Automatic Software Repair: A Bibliography". In: *ACM Comput. Surv.* 51.1 (Jan. 2018), 17:1–17:24. ISSN: 0360-0300.

[33] Keigo Naitou et al. "Toward introducing automated program repair techniques to industrial software development". In: *Proceedings of the 26th Conference on Program Comprehension.* 2018, pp. 332–335.

[34] Yuhua Qi et al. "The strength of random search on automated program repair". In: *Proceedings of the 36th International Conference on Software Engineering.* ACM. 2014, pp. 254–265.

[35] Zichao Qi et al. "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 2015. Baltimore, MD, USA: ACM, 2015, pp. 24–36. ISBN: 978-1-4503-3620-8.

[36] Adriaan MJ Schakel and Benjamin J Wilson. "Measuring word significance using distributed representations of words". In: *arXiv preprint arXiv:1508.02297* (2015).

[37] Edward K Smith et al. "Is the cure worse than the disease? overfitting in automated program repair". In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering.* ACM. 2015, pp. 532–543.

[38] Victor Sobreira et al. "Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J". In: *Proceedings of SANER.* 2018.

[39] Soichi Sumi et al. "Toward improving graftability on automated program repair". In: *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on.* IEEE. 2015, pp. 511–515.

[40] Akito Tanikado et al. "New Strategies for Selecting Reuse Candidates on Automated Program Repair". In: *Computer Software and Applications Conference (COMPSAC), 2017 IEEE 41st Annual.* Vol. 2. IEEE. 2017, pp. 266–267.

[41] Rijnard van Tonder and Claire Le Goues. "Static Automated Program Repair for Heap Properties". In: (2018).

[42] Westley Weimer et al. "Automatically finding patches using genetic programming". In: *Proceedings of the 31st International Conference on Software Engineering.* IEEE Computer Society. 2009, pp. 364–374.

[43] Ming Wen et al. "Context-Aware Patch Generation for Better Automated Program Repair". In: ICSE. 2018.

[44] Martin White et al. "Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities". In: *arXiv preprint arXiv:1707.04742* (2017).

[45] Qi Xin and Steven P. Reiss. "Leveraging Syntax-related Code for Automated Program Repair". In: *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering.* 2017, pp. 660–670.

[46] Yingfei Xiong et al. "Identifying patch correctness in test-based program repair". In: *Proceedings of the 40th International Conference on Software Engineering.* 2018, pp. 789–799.

[47] Yingfei Xiong et al. "Precise condition synthesis for program repair". In: *Proceedings of the 39th International Conference on Software Engineering.* IEEE Press. 2017, pp. 416–426.

[48] Jifeng Xuan et al. "Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs". In: *IEEE Transactions on Software Engineering* (2016).

TRITA EECS-EX-2018:729