The Remarkable Role of Similarity in Redundancy-based Program Repair

Zimin Chen  
KTH Royal Institute of Technology

Martin Monperrus  
KTH Royal Institute of Technology

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 most of the time (65%).

I. 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 similarity techniques based on a powerful innovative idea (such as [32, 9, 22, 30], 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 [20, 25, 17, 15]). 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 [31] and major state-of-the-art approaches (e.g. [32, 11, 9]) both relied on redundancy.

While the empirical presence of redundancy has been studied [21, 4], 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 [34, 32, 11], different kinds of similarity analysis are part of the overall repair technique.

Xin & Reiss [34] use TFIDF to compute the similarity between the buggy statement and its context.

Wen, Chen, Wu, Hao & Cheung [32] prioritize potential repair code snippets that have a similar context, with three similarities stacked in a clever manner.

Jiang, Xiong, Zhang, Gao & Chen [11] heavily rely on name similarity (variable name and method name) to find code snippets that are present in other similar methods. 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 similarity analysis during the exploration of the search space.

Our study of similarity in redundancy-based program repair uses a strictly principled methodology. First, we isolate the similarity component in a generic redundancy-based repair process that is representative of the state-of-the-art [34]. 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. Second, we systematically qualify the similarity relationships 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 define and investigate three similarity metrics that capture either syntactic or semantic similarity.

Then, we design and perform four large-scale experiments. The main goal of the experiments is to study the search space reduction obtained with similarity analysis. The search space reduction is defined as the ratio between the number of points in the search space that must be considered before finding the solution and the total size of the search space. Assume a search space of 100 code snippets and the search space contains the correct patch, if a similarity based technique puts the correct patch in a short-list of 5 snippets, 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. The experiments also contribute to the understanding of other important aspects of the search space of redundancy-based program repair.

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

II. BACKGROUND

A. Overview of 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 [8], 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 [31] already leveraged redundancy. Today, major state-of-the-art approaches still heavily depend on redundancy (e.g. [32, 11, 9]).

The redundancy assumption — the assumption that code may evolve from existing code that comes from somewhere else — has been empirically verified. Martinez, Westley & Monperrus [21] and Barr, Brun, Devanbu, Harman & Sarro, 2014 [4] 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 [24].

Furthermore, Martinez et al. [21] 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 and ingredient. But they remove in-scope ingredient similarity and method name 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.

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, an AST node, a line or a full snippet.

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. 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. Since there myriads of possible ingredients, this is too long: 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.

C. 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 [34], CapGen [32] and SimFix [11]. For instance, let us consider the human patch of bug Math-75 from Defects4J [12] in Listing 1. 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.

III. RESEARCH METHODOLOGY

A. Problem

In this paper, we aim at doing two contributions:
program repair, refined a generic repair process as follows.

C. Generic Repair Process

similarity on the search space of program repair. Nobody has ever devised a protocol to study the impact of projects, we apply our repair process and measure the actual reduction. For large set of past bugs from open-source

– typically the full method – and the enclosing context of the removed code inserted code. (Note that the related work rather measures the we measure the similarity between the removed and the snippet is replaced by another code snippet. For all them,

This isolation means that we ensure that the observed effects on the search space are those of the different similarity metrics we consider.

Second, we systematically define three similarity metrics for redundancy-based program repair. All of them are qualified in the context of replacement patches where one code snippet is replaced by another code snippet. For all them, we measure the similarity between the removed and the inserted code. (Note that the related work rather measures the similarity between the enclosing context of the removed code – typically the full method – and the enclosing context of the ingredient). We define three similarity metrics that capture either syntactic or semantic similarity, they are presented in subsection III-D.

Third, we define a metric for capturing the search space reduction. For large set of past bugs from open-source projects, we apply our repair process and measure the actual search space reduction.

This research methodology is novel. To our knowledge, nobody has ever devised a protocol to study the impact of similarity on the search space of program repair.

B. Overview of the Research Methodology

Our research methodology is as follows.

First, we define a generic repair process that is amenable to comparing similarity metrics. This repair process is a normal, redundancy-based, generate-and-validate one, where similarity analysis is isolated in a single component with well defined boundaries with respect to the rest of the process. This isolation means that we ensure that the observed effects on the search space are those of the different similarity metrics we consider.

Second, we systematically define three similarity metrics for redundancy-based program repair. All of them are qualified in the context of replacement patches where one code snippet is replaced by another code snippet. For all them, we measure the similarity between the removed and the inserted code. (Note that the related work rather measures the similarity between the enclosing context of the removed code – typically the full method – and the enclosing context of the ingredient). We define three similarity metrics that capture either syntactic or semantic similarity, they are presented in subsection III-D.

Third, we define a metric for capturing the search space reduction. For large set of past bugs from open-source projects, we apply our repair process and measure the actual search space reduction.

This research methodology is novel. To our knowledge, nobody has ever devised a protocol to study the impact of similarity on the search space of program repair.

C. Generic Repair Process

In order to study the role of similarity in redundancy-based program repair, refined a generic repair process as follows.

I Fault localization We run the test suite and obtain all suspicious modification points. 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 Ingredient selection We extract all ingredients at the application scope in order to have the ingredient pool.

For each suspicious modification point, we compute a similarity metric between the modification point and the ingredient. Then, the repair ingredients are ordered by decreasing similarity, and tried in this order.

III Search space bounding The number of tried ingredients per suspicious point is bounded by a configuration parameter \( m \). This means that the search is bounded by \( n \times m \).

D. Similarity Metrics for Repair

Our core idea is to compute the similarity between the modification point and all possible repair ingredients. In this study, we define three different similarity relationship. Our goal is that they are different in nature, and capture different characteristics of similarity.

1) Rationales

In this paper, we consider longest common subsequence (LCS), term frequency inverse document frequency (TFIDF) and word embedding based on unsupervised learning (Embedding). The rationale behind using these three metrics is as follows:

LCS is purely syntactic because it works at the character level. LCS has been proposed for program repair by Yokoyama et. al [38]. 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. TFIDF has been used in the context of program repair by Xin & Reiss [34]. Being based on tokenization, it is much less syntactic than LCS. The strength of TFIDF 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. The usage of embeddings for repair is a recent topic [33, 2]. 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) Similarity Calculation

We compute the three similarity relationships are follows.

LCS: The ingredients are considered as sequences of characters, for each pair \(<\text{suspiciousline, ingredient}>\), 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, as standard practice in the field for measuring vector distance [26].

**Embedding:** We train Word2vec [23] on a corpus of Java files. Each java file is considered as a series of tokens. The target vector space is of typical size (64 dimensions). This results in that each token of the program is associated to a vector of 64 real numbers. 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 [14]. Finally, the similarity score is also computed by using cosine similarity in the embedding space.

### E. Implementation

We now present our main implementation choices.

1) **Repair Tool Implementation**

We have implemented a prototype tool called 3sFix, on top of the ASTOR program repair framework [19]. It does replacement-based repair at the line level. The fault localization is based on Ochiai.

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 at https://github.com/kth-tcs/3sFix-experiments.

2) **Embedding Implementation**

For computing the embedding, we train a word2vec [23] implementation from Google’s TensorFlow\(^1\) on a large corpus of Java files (described in subsubsection IV-C.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 \(<UN\text{UN}{\text{KNOW}}N>\) token for those tokens that are out of our vocabulary.

### IV. EXPERIMENTAL METHODOLOGY

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

#### A. Research Questions

**RQ1:** Do the findings of previous research on the redundancy of commits hold on our new dataset?

Redundancy-based repair approaches use ingredients that can be found somewhere else in the code. Only a few studies have systematically studied the redundancy assumption [4, 21].

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 [4, 21]). Second, this initial study also acts as a sanity check for the next research questions.

**RQ2:** How effective are the different similarity metrics on handling 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 RQ2 is to compare the three similarity metrics presented in III-D. RQ2 is based on an original experimental protocol (see IV-B.2) that allows us to compute the search space reduction.

**RQ3:** How does similarity-based repair explore the search space of Defects4J?

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 RQ3 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.

**RQ4:** What research opportunities are open by our results on similarity for program repair?

The next frontier in redundancy-based program repair would be an ingredient scope that is bigger than the application scope, that spans across multiple software projects. For instance, the Github scope consists of all unique lines that are present on Github for a given programming language (say Java). We perform a last experiment, of speculative nature, to see whether similarity analysis would scale to a potentially immense ingredient pool.

#### B. Protocols

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** We use a corpus, briefly presented in IV-C.2.
- **II Extract the removed line (modification point) and the inserted line (the repair ingredient)** We explicitly removed one-line replacement patches if the repair ingredient is not unique, since it might give a higher redundancy rate.
III Compute the search space The search space is composed of all unique lines of the application of the point in time of the commit. The formatting of the lines, including 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 repair ingredient already exists in the application.

2) Protocol of RQ2
RQ2 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. This is done as follows:

I Extract repair tasks We extract all one-line replacement commits from a corpus from the literature, presented in Section IV-C.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 modification point, all ingredients at application scope are collected.

III Compute the three similarity metrics We compute the similarity between the modification point between all ingredients. The training phase of Embedding is made according to the procedure presented in III-E.2.

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, which we define as the ratio between the rank of the correct ingredient according to similarity) and the size of the search space (i.e. the total number of repair ingredients).

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 the 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.

3) Protocol of RQ3
RQ3 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, Durieux, Madeiral, Monperrus & Maia [27]. Then, we compute the ingredient pool, 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, semantically equivalent or overfitting. 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) 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 curated snapshot of Github, described in IV-C.1.

II Re-execute the protocol of RQ1 using the Github ingredient pool. Because of the size of Github scope, we use a Bloom filter as data structure to store each line of code from Github Java Corpus [5]. Bloom filter is a probabilistic data structure to check whether an element is in a set, hence is perfectly appropriate to study redundancy.

In order to cope with the scale of the search space, a Bloom filter is probabilistic. 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.

III Compute the similarity metrics at the Github scope We perform an empirical worst case analysis by computing $10^k$ similarity comparisons (with LCS) with ever greater values of $k$ to identify the feasibility limit and compare it to the size of the Github scope

C. Data

1) Github Java Corpus
In two experiments, RQ2 and RQ4, we use Github Java Corpus [1], 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.

2) CodRep corpus
In RQ1 and RQ2, we need a corpus of one-line patches. For this, we use the CodRep corpus [6], 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 us to study redundancy-based program repair at a large-scale.

V. EXPERIMENTAL RESULTS
We now present the results of our large scale and novel experiments on the role of similarity in redundancy-based program repair.

1) Research Question 1 (Search space statistics)
Do the findings of previous research on the redundancy of commits hold on our new dataset?

In our first experiment, per the protocol described in IV-B.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 [21, 4]. 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 [25] 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.

2) Research Question 2 (In-vitro Repair)
How effective are the different similarity metrics on handling the search space of redundancy-based repair?

For 2766 one-line replacement patches sampled from the CodRep corpus (see IV-C.2), we have applied the repair simulation described in Section IV-B.2. 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 I. 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 3: Similarity analysis enables to reduce the search space very effectively. For all considered similarity metrics, the ranking of repair ingredients results in 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 3: In the average case, the correct ingredient is ranked among the top 0.65% with LCS ranking, it means that on 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 4: 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.

Finding 5: 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 5: 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 6: 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 6: As stated in subsection III-D, 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.

As discussed in III-D.2, our embedding technique has two phases: a token-embedding phase (which is a direct application of word embedding) followed by a code line
embedding phase (which is a variant of sentence embedding in computational linguistics). Since sentence embedding is an active research field [7], we assume that future advances in that field will enable to revisit this question later.

3) Research question 2 (Redundancy in Defects4J)

How does similarity-based repair explore the search space of Defects4J?

Finding 7: 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 7: 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 8: 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 2, Listing 3 and Listing 4 show the generated patch.

Implication 8: What is remarkable is that the same technique captures completely different repair operators: Listing 2 changes a single literal; Listing 3 changes method call arguments and Listing 4 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 [37]), 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.

Finding 9: 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 9: 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.

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 subsubsection V-.2). 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 points, or by trying more ingredients per sus-

Listing 2. Correct patch for Math_5 ranked 1st with similarity analysis

```java
if ( real == 0.0 && imaginary == 0.0) {
    return NaN;
} else {
    return INF;
}
```

Listing 3. Correct patch for Math_70 ranked 1st with similarity analysis

```java
CategoryDataset dataset = this.plot.getDataset(index);
if ( dataset != null ) {
    return solve(min, max);
} else {
    return solve(f, min, max);
}
```

Listing 4. Correct patch for Chart_1 ranked 1st with similarity analysis

```java
if ( dataset == null) {
    return solve(min, max);
} else if ( dataset == null) {
    return result;
}
```
| Project     | # bugs | Average search space size | LCS | Tfidf | Embedding | LCS | Tfidf | Embedding |
|------------|--------|--------------------------|-----|-------|-----------|-----|-------|-----------|
| PocketHub  | 100    | 5963                     | 90  | 9     | 35        | 63 %| 53 %  | 45 %      |
| Elasticsearch | 100   | 119217                   | 70  | 43    | 2885      | 51 %| 63 %  | 36 %      |
| LibGDX     | 100    | 112945                   | 1009| 876   | 955       | 41 %| 86 %  | 16 %      |
| Eclipse    | 8      | 2528                     | 7   | 3     | 9         | 50 %| 88 %  | 50 %      |
| SWT        | 100    | 186695                   | 1161| 10244 | 14589     | 50 %| 65 %  | 28 %      |
| AspectJ    | 100    | 156625                   | 1850| 12250 | 14300     | 58 %| 43 %  | 41 %      |
| ZXing      | 58     | 38291                    | 18  | 2387  | 2517      | 62 %| 57 %  | 31 %      |
| Ant        | 100    | 71949                    | 137 | 1628  | 2259      | 51 %| 60 %  | 38 %      |
| JMeter     | 100    | 50048                    | 71  | 549   | 1239      | 60 %| 72 %  | 56 %      |
| Log4j      | 100    | 112945                   | 889 | 2110  | 3617      | 58 %| 61 %  | 40 %      |
| Tomcat     | 100    | 134265                   | 558 | 619   | 1581      | 49 %| 60 %  | 24 %      |
| Xerces     | 100    | 64410                    | 382 | 2749  | 3320      | 58 %| 55 %  | 49 %      |
| ECJ        | 100    | 56175                    | 153 | 1588  | 2244      | 49 %| 55 %  | 36 %      |
| WTP Incubator | 100   | 59634                    | 302 | 208   | 268       | 62 %| 74 %  | 48 %      |
| Xpand      | 100    | 51132                    | 324 | 112   | 205       | 57 %| 74 %  | 53 %      |
| Cassandra  | 100    | 41541                    | 186 | 3044  | 6125      | 50 %| 74 %  | 52 %      |
| Lucene/Solr | 100   | 192628                   | 889 | 2110  | 3617      | 58 %| 61 %  | 40 %      |
| OpenIPA    | 100    | 141508                   | 1375| 4570  | 7142      | 46 %| 56 %  | 42 %      |
| Wicket     | 100    | 61256                    | 187 | 84    | 370       | 55 %| 67 %  | 34 %      |
| Commons-codec | 100   | 9302                     | 70  | 92    | 95        | 71 %| 71 %  | 64 %      |
| Commons-collections | 100   | 30263                    | 186 | 354   | 354       | 56 %| 69 %  | 48 %      |
| Commons-compress | 100   | 154684                   | 558 | 619   | 1581      | 49 %| 60 %  | 24 %      |
| Commons-iso | 100   | 2272                     | 9   | 6     | 36        | 78 %| 74 %  | 66 %      |
| Commons-lang | 100   | 13894                    | 122 | 112   | 201       | 60 %| 77 %  | 61 %      |
| Commons-math | 100   | 37354                    | 123 | 123   | 123       | 65 %| 77 %  | 55 %      |
| Spring Framework | 100   | 119217                   | 2755| 2076  | 10082     | 73 %| 56 %  | 32 %      |
| Storm      | 100    | 14783                    | 421 | 327   | 490       | 56 %| 62 %  | 41 %      |
| Wildfly    | 100    | 181498                   | 1346| 4274  | 7376      | 47 %| 47 %  | 23 %      |
| Summary    | 2766   | 77276                    | 504 | 1820  | 3024      | 57 %| 65 %  | 43 %      |

**TABLE 1**

**EXPERIMENTAL RESULTS ON THE EFFECTIVENESS OF USING SIMILARITY TO REDUCE THE SEARCH SPACE OF REDUNDANCY-BASED PROGRAM REPAIR.

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 10:** 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 10:** This confirms that the test suites of Defects4J are weak at completely specifying the patch [18]. This also shows that overfitting patches do largely exist in the ingredient pool of redundancy-based repair, which has been suggested previously [17]. This suggests that redundancy-based program repair must be coupled with a overfitting detection system (e.g. [35]).

4) **Research Question 4 (Call to action))**

What research opportunities are open by our results on similarity for program repair?

Our findings are actionable, they call for exploring the frontiers of redundancy-based repair.

**Finding 11:** The first action based on our results is for practitioners: if you ever plan to design a repair system based on redundancy, do use LCS. In our experiment, LCS is both the most effective but it is also very fast. Our unoptimized implementation on a standard 2018 server enables us to compute 50000 similarity values per minute.
Implication 11: In practice, it is reasonable to allocate 1 minute per suspicious statements to compute the similarity against the whole ingredient pool. With an optimized implementation, it is likely feasible to consider the application scope for most small and medium-sized libraries and applications.

Finding 12: The second action based on our results is for researchers: consider exploring ultra-large ingredient scopes. According to our data, 27% of one-line replacement patches can be repaired with an ingredient that exists at the Github scope. This is a 3x increase compared to the application scope which is the state-of-art today.

Implication 12: The 27% repair redundancy at Github scope is exciting, it shows that the effectiveness of redundancy-based program repair could potentially be improved threefold. To explore this uncharted territory of redundancy-based repair at an ultra-large scope, the research community needs to work on two major open questions: 1) ultra-fast similarity comparison 2) large scale manipulation of ingredients, possibly based on code hashes [3].

A. 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. Then, in RQ2, 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. We continued with an analysis of the search space for Defects4J, which is an heavily used research benchmark. The major finding from RQ3 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. 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.

VI. 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.

RQ2 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.

There are few threats in RQ3 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.

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.

VII. Related Work

A. Redundancy in Programs

After the initial successes of GenProg, studies have looked at the underlying redundancy assumption. Barr et al. [4] checked it against 12 Apache project, and found that changes are 43% redundant at line level. Martinez et al. [21] 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, Higo, Hotta & Kusumoto [28] further conducted redundancy experiments using a larger dataset and obtain similar results. Lin, Ponzanelli, Mocci, Bavota & Lanza [16] 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 & Su [8] 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. [36] use code search on Github to find snippets.

Ke, Stolee, Le Goes & Brun’s approach [13] searches for existing code snippets (i.e. redundant ones that match a given input-output specification.

B. Analyses of the Repair Search Space

Martinez & Monperrus [20] 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, Mao, Lei, Dai & Wang [25] 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 & Rinard [17] 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.

C. Similarity in Program Repair

Ji, Chen, Mao & Yi [10] are possibly the first to have propose that the repair ingredients should be taken from similar code.

Xin & Reiss [34] 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, Tufano, Martinez, Monperrus & Poshvanyk [33] uses deep learning to reason about similarity between the method body containing the modification point and the method body of ingredients.

Tanikado et. al [29] 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. [32] prioritized ingredients that have similar programming context based on program analysis.

Jiang et al. [11] 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.

VIII. Conclusion

We have performed an original study of the search space of redundancy-based program repair and in particular, the role of using similarity analysis to explore it. Our findings show that similarity can effectively reduce the search space in order to find the correct repair ingredient. Over 2766 tasks, similarity reduces the search space by a factor of 154 on average. Furthermore, similarity is able to rank the correct ingredient first, which means that it contributes to avoiding overfitting ingredients in the search space.

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] 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.

[2] Leonardo Afonso Amorim, Mateus F Freitas, Altino Danas, Eduard F de Souza, Celso G Camilo-Junior, and Wellington S Martins. “A new word embedding approach to evaluate potential fixes for automated program repair”. In: 2018 International Joint Conference on Neural Networks (IJCNN). IEEE, 2018, pp. 1–8.

[3] Muhammad Asaduzzaman, Chanchal K Roy, Kevin A Schneider, and Massimiliano Di Penta. “LHDiff: A language-independent hybrid approach for tracking source code lines”. In: 29th IEEE International Conference on Software Maintenance. IEEE, 2013, pp. 230–239.

[4] Earl T Barr, Yurii Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. “The plastic surgery hypothesis”. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2014, pp. 306–317.

[5] 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.

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

[7] Ishita Dasgupta, Denu Guo, Andreas Stuhlmüller, Samuel J Gershman, and Noah D Goodman. “Evaluating Compositionality in Sentence Embeddings”. In: arXiv preprint arXiv:1802.04302 (2018).

[8] 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.

[9] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. “Towards practical program repair with on-demand candidate generation”. In: Proceedings of the 40th International Conference on Software Engineering. ACM, 2018, pp. 12–23.

[10] Tao Jiang, Liqian Chen, Xiaoguang Mao, and Xin Yi. “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.

[11] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangan Chen. “Shaping Program Repair Space with Existing Patches and Similar Code”. In: (2018).

[12] 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.

[13] Yalin Ke, Kathryn T. Stolte, Claire Le Goues, and Yurii Brun. “Repairing Programs with Semantic Code Search”. In: Proceedings of the International Conference on Automated Software Engineering. 2015.

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

[15] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. “Overfitting in semantics-based automated program repair”. In: Empirical Software Engineering (2018), pp. 1–27.

[16] Bin Lin, Luca Ponzanelli, Andrea Mocci, Gabriele Bavota, and Michele Lanza. “On the uniqueness of code redundancies”. In: Program Comprehension (ICPC), 2017 IEEE/ACM 25th International Conference on. IEEE, 2017, pp. 121–131.

[17] 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.

[18] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. “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.

[19] Matias Martinez and Martin Monperrus. “ASTOR: A Program Repair Library for Java”. In: Proceedings of ISSTA, 2016.

[20] 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.
[21] Matias Martinez, Wesley 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.

[22] Sergey Metchaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. “Semantic Program Repair Using a Reference Implementation”. In: Proceedings of ICSE. 2018.

[23] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. “Efficient estimation of word representations in vector space”. In: arXiv preprint arXiv:1301.3781 (2013).

[24] Keigo Naitou, Akito Tanikado, Shinshuke Matsumoto, Yoshiki Higo, Shinji Kusumoto, Hiroyuki Kiminuki, Toshiyuki Kurabayashi, and Haruto Tanno. “Toward introducing automated program repair techniques to industrial software development”. In: Proceedings of the 26th Conference on Program Comprehension. 2018, pp. 332–335.

[25] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chensong Wang. “The strength of random search on automated program repair”. In: Proceedings of the 36th International Conference on Software Engineering. ACM, 2014, pp. 254–265.

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

[27] Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo A. Maia. “Dissection of a Bug Dataset: Anatomy of 395 Patches from Defects4J”. In: Proceedings of SANER. 2018.

[28] Soichi Sumi, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto. “Toward improving graftability on automated program repair”. In: Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on. IEEE, 2015, pp. 511–515.

[29] Akito Tanikado, Haruki Yokoyama, Masahiro Yamamoto, Soichi Sumi, Yoshiki Higo, and Shinji Kusumoto. “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.

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

[31] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. “Automatically finding patches using genetic programming”. In: Proceedings of the 31st International Conference on Software Engineering. IEEE Computer Society, 2009, pp. 364–374.

[32] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. “Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities”. In: arXiv preprint arXiv:1707.04742 (2017).

[33] Meng Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. “Context-Aware Patch Generation for Better Automated Program Repair”. In: ICSE. 2018.

[34] 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.

[35] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. “Identifying patch correctness in test-based program repair”. In: Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 790–799.

[36] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. “Precise condition synthesis for program repair”. In: Proceedings of the 39th International Conference on Software Engineering. IEEE Press. 2017, pp. 416–426.

[37] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. “Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs”. In: IEEE Transactions on Software Engineering (2016).

[38] Haruki Yokoyama, Yoshiki Higo, Keisuke Hotta, Takafumi Ohta, Kozo Okano, and Shinji Kusumoto. “Toward improving ability to repair bugs automatically: A patch candidate location mechanism using code similarity”. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing. ACM, 2016, pp. 1364–1370.