Empirical Analysis of 1-edit Degree Patches in Syntax-Based Automatic Program Repair

Piotr Dziurzanski, Simos Gerasimou, Dimitris Kolovos, Nicholas Matragkas
[1]University of York, Deramore Lane, York, YO10 5GH, UK.
{firstname.lastname}@york.ac.uk

***Abstract*—In this paper, software patches modifying a single line (aka 1-edit degree patches) of buggy Java open-source projects have been generated automatically using computational search and experimentally evaluated. We carried out the presumably largest to date experiment related to 1-edit degree patches, consisting of almost 27,000 computational jobs upper bounded with 107,000 computational hours. Our experiments show the benefits and drawbacks of such kind of patches. In particular, the search space size has been shown to be reduced by several orders of magnitude. The volume of tests that can be filtered out without any negative impact while generating 1-edit degree patches has been increased by about 97%. Finally, the effectiveness of finding 1-edit plausible patches is compared with multi-line plausible patches found with state-of-the-art syntax-based Automatic Program Repair tools. It is shown that despite patching fewer bugs in total, 1-edit degree patches have potential to patch some extra bugs.**

I. Introduction

Syntax-based automatic program repair (APR), a main branch of APR, focuses on synthesising fixes for identified software defects (bugs) through the application of syntactic source code modifications [1]. State-of-the-art approaches of this APR paradigm employ search-based techniques (e.g., genetic programming, evolutionary algorithms) to manipulate source code extracts by applying mutation and crossover operations to the original (buggy) source code. The seminal work on GenProg demonstrated the feasibility of this paradigm by synthesising patches for 55 out of 105 bugs in 8 open-source programs totalling 5.1 million lines of C++ code and involving 10,193 test cases [2]. The crux of the paradigm lies at generating candidate repairs to replace the suspicious source code extracts, termed *modification points (MPs)*, with each candidate repair being an evolved sequence of edit operations (i.e., deletion, insertion, replacement) of source code fragments from the same software project, which are termed *ingredients*. Driven by these encouraging results, recent research proposed new syntax-based APR techniques aiming at reducing the substantial manual effort required to debug the software and generate patches [3], [4].

Despite the benefits of this paradigm, further analysis of the GenProg approach pointed several drawbacks. Recent research has shown that random search is more effective that the more sophisticated genetic approach applied in GenProg both in the number of patch trials and test case executions for most of the examined cases [5]. More importantly, [6] demonstrates that more than 90% of plausible patches (i.e., test-suite adequate, though not necessarily correct) generated by GenProg are equivalent to a single modification that removes functionality. The authors also stress that plausible patches could have significant negative effects. Moreover, due to the randomness of genetic operators used in GenProg, it is likely that an automatically generated patch would contain at least an order-of-magnitude more changes than are necessary to repair the program and these extra edits incorporate insignificant or even dangerous changes which are not captured due to test suite imperfection [2]. Hence, GenProg applied repair minimisation to eliminate the modifications that do not influence the testing result, yet in [7] it was demonstrated that such patch minimisation does not reduce overfitting. Interestingly, a large percentage of bugs repaired by GenProg can be also repaired with just a single edit, i.e., with 1-edit degree patches [8]. Yet, this kind of patches have not been analysed heavily in the past, especially with regards to the benefits and drawbacks stemming from the respective significant search space pruning and test filtering. In this paper, we fill this gap by performing a large-scale experiment generating solely 1-edit degree patches and analysing its results.

As reported in [9], the majority of syntax-based APR tools use GZoltar [10] to determine suspiciousness of faulty statements (ranging from 0 to 1) to form a set of potential MPs in the source code and define the granularity of MPs at the code line level. The suspiciousness threshold is a parameter of GZoltar and no guidance for selecting values of this parameter has been published so far. In the popular ARJA framework [11], the 40 most suspicious lines are used by default, but this limit can be overridden with a command-line parameter. It is worth stressing that for several bugs in Defects4J [12], the most popular APR benchmark [13], the buggy lines can be filtered out with that default setting and hence cannot be patched regardless of the remaining settings [14]. Similarly, the Cardumen mode of ASTOR described in [15] uses suspiciousness threshold γ for values computed by GZoltar. It is then worth analysing the correlation between the suspiciousness value assigned to an MP by GZoltar and the likelihood of that MP to be plausibly patched. Furthermore, the total number of suspicious lines as identified by GZoltar should be evaluated. Then, the search space size could be determined in popular benchmark projects by analysing the number of MPs and their compatible ingredients.

Benefiting from the independence of 1-edit degree patch generation, we performed a large-scale parallel execution of an APR tool in a high performance computing cluster on popular open-source Java projects. Eventually, we make the following contributions:

- We have evaluated experimentally the 1-edit degree patch search space size and found that, on average, it is smaller by a few hundreds orders of magnitude in comparison with the search space sizes of the corresponding multi-edit degree patches,
- We have assessed the reduction of the test number needed to evaluate 1-edit degree patches and demonstrated its significant decrease by 97% in comparison with the figures for the corresponding multi-edit degree patches,
- We have determined that the number of MPs (i.e., suspicious lines) with insufficient test coverage is significant and in all considered projects there are MPs with no positive tests,
- We have determined the impact of the usual MP trimming on the number of plausible tests generated and found that several plausibly patchable MPs are likely to be filtered out by typical APR tools due to their low suspiciousness,
- We have compared the number of 1-edit degree plausible patches with the number of plausible patches of a higher edit degree generated for the same software projects and found that although this number is lower, new bugs can be plausibly patched due to the lack of trimming the MPs with lower suspiciousness.

The rest of this paper is organised as follows. In the next section, five important research questions related to 1-edit degree patches are formulated. Section III describes the experiment conducted to answer the research questions, followed by the presentation of the experimental results in Section IV. Section V briefly describes related work and Section VI concludes the paper.

II. Research questions

Our systematic experimental evaluation involves carrying out a large-scale experiment trying to mutate all suspicious MPs of open-source Java projects included in the popular Java benchmark for APR, Defects4J [12]. The goal of this experiment is to answer a number of fundamental questions regarding the MPs, tests and plausible 1-edit degree patches, as enumerated below together with the rationale behind choosing these questions.

RQ1. What is the typical number of suspicious lines in open source Java projects and what is the number of the ingredients these lines can be replaced with?

The answer to this question can help to select a technique for performing APR. In case of a low number of suspicious lines, each of them can be attempted to be modified in order to find a patch. In such situation, the line suspiciousness can be used not as a base for trimming the number of MPs under analysis, but to define an order in which MPs are analysed in sequence, to maximise the chances of finding a plausible patch earlier. Assuming 1-edit degree only (i.e., single MP), modifications of various MPs can be performed independently in parallel. Moreover, if the number of their ingredients (i.e., the compatible code extracts that can be used to replace these MPs) is low, the MP replacing can be performed exhaustively (as in [8]). Otherwise, the ingredients can be selected randomly. By knowing the search-space size, it is possible to assess the percentage of the tried patches and draw some statistical conclusions regarding the probability of the plausible patch finding.

```
 1  if (fa * fb >= 0.0 ) {
 2      throw new ConvergenceException(
 3          ”number of iterations={0},
 4          maximum iterations={1}, ” +
 5          ”initial={2}, lower bound={3},
 6          upper bound={4}, final a value={5}, ” +
 7          ”final b value={6}, f(a)={7}, f(b)={8}”,
 8          numIterations, maximumIterations, initial,
 9          lowerBound, upperBound, a, b, fa, fb);
10          }
```

Listing 1. Math-85 bug example from Defects4J benchmark

RQ2. What is the number of tests in test suites of typical open source Java projects that cover the suspicious lines?

The importance of this RQ2 stems from the possibility of reduction of the number of tests needed to be executed in order to determine the 1-edit degree patch quality, as the tests not executing the code line with the MP under analysis can be filtered out. As the tests are usually executed sequentially and their execution can constitute even more than 60 per cent of the entire APR process [4], decreasing the number of executed tests can reduce the computation time to a large extent. However, too low number of tests for an MP is also undesired, as it can lead to a mutation removing a vital functionality or having an adverse side-effect. This problem can be exemplified with the Math-85 bug from Defects4J, which is shown in Listing 1. This bug is detected by one unit test. The human-written patch changed the relational operator in line 1 from greater-than-or-equal-to to greater-than. However, this fix is relatively difficult to be found by an automatic tool as there is no positive test for MP in line 2 (in contrast to line 1, which has 16 positive tests). Hence, any side-effect-free modification of line 2 would make the negative test pass as no exception will be thrown. For example, one of the fixes found by jGenProg replaces the throw statement with rather useless substitution `double ret=Double.NaN;`. What is even worse, the *throw* statement is more likely to be replaced than the conditional expression (i.e., the real bug) as its suspiciousness determined by GZoltar is much lower. Actually, based on the suspiciousness value, this modification point would be removed by default e.g. by ARJA as it considers only the 40 the most suspicious MPs (this value can be modified with parameter `maxNumberOfModificationPoints`), as written earlier in this paper.

The analysis of the number of tests of each MP can indicate the under-tested code lines and encourage the developers to add a number of missing unit tests. Also, when a number of alternative plausible patches needs to be analysed by developers, it may be beneficial to start with the patches modifying

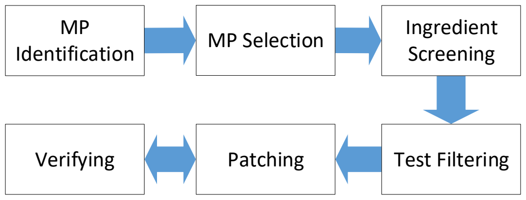

Fig. 1. A general scheme of the conducted experiment

MPs tested with the largest numbers of positive tests as they may be considered as being more likely to be correct.

RQ3. What is the correlation between the line suspiciousness and their modification in plausible patches?

As discussed earlier, only a subset of suspicious lines is usually considered by APR tools. The strategy of omitting the less suspicious lines during the APR process, although intuitively promising, could possibly lead to decreasing the number of plausibly patched bugs. This consideration is in line with the conclusions from [9], where it was shown that increasing the number of MPs by using a newer version of GZoltar can localise 58 more bugs in the Defects4J benchmark set.

RQ4. How common is that the set of plausible 1-edit degree patches of the same bug alter more than one MP?

Ideally, a bug is located in one line and hence all corresponding plausible 1-edit degree patches would alter the same MP that includes the bug. However, due to possibility of multi-line bug existence or under-tested MPs (as exemplified earlier with the Math-85 issue), it is possible that the patches generated by an APR tool change more than one line. This situation would harden the fault localisation process. The planned large-scale 1-edit degree patch generation can help in determining the likelihood of such an event.

RQ5. Is the number of bugs plausibly patched by 1-edit degree patches equal to the number of bugs that have been reported to date to have been patched with multi-edit degree patches by syntax-based APR tools?

Prior research such as [8] states that the majority of plausible patches found during an evolutionary search process modify just a single statement or are equivalent to 1-edit degree patches considering the applied test suite. However, several patches found by evolutionary search process in e.g. [11] are reported to have a larger edit degree than one. Since we intend to execute a large-scale experiment that not only modifies all suspicious lines, but also uses a quite substantial number of (random) ingredient substitution, the number of the bugs plausibly patched should be similar to that reported in [11], [3] provided that 1-edit degree patches are equivalent (considering the applied test suite) to multi-line patches found by the state-of-the-art-tools. Notice, by performing so many random ingredient substitutions, we aim to bridge the gap between the genetic and random search process which was reported in [11] (in contradiction to [5] which has shown the lack of it).

III. Experimental design

To answer the research questions listed in the previous section, a large-scale generation of 1-edit degree patches needs to be performed. The patch generation has been carried out using the ARJA framework [11] by focusing on a single MP during the APR process. The repair algorithm evaluated in this paper has been added as a new class extending the AbstractRepairAlgorithm class in the ARJA framework. Most of the original ARJA modules, especially related to the fault localisation, coverage analysis, test filtering, ingredient screening and fitness evaluation, have not been modified by us.

The experiment has been conducted using a local high performance computing cluster which includes 173 nodes with a total 42TB of memory, connected by a high-speed 100Gb Infiniband network, 7024 Intel Xeon Gold/Platinum cores and 2.5PB of high performance storage with 12GB/s data transfer rate whose estimated performance is equal to 435.2 TFLOPS. The cluster is governed by the Slurm Workload Manager.

Similarly to other papers related to APR of programs written in Java, the Defects4J benchmark [12] has been used for experiments. This set consists of 357 real bugs extracted from five popular open source projects. The Closure Compiler project from Defects4J does not use typical JUnit test cases and hence it is usually excluded from experiments [11]. The names of the remaining projects are: `JFreeChart`, `JodaTime`, `Commons Lang` and `Commons Math`, abbreviated in this paper to Chart, Time, Lang and Math. A typical naming convention in the APR domain is to follow the project name with a dash and a number of the bug index from a benchmark set.

The four projects considered include 224 bugs in total. However, two bugs (Chart-8 and Lang-26) have not been detected by the ARJA framework in the version we used (i.e., ARJA found no failed tests). Similarly, the ARJA framework found no modification points for 2 other bugs (Math-35 and Math-45), despite the results related to these bugs have been reported in the original ARJA paper. For the majority of bugs in the Time project, the default version of GZoltar used in ARJA (0.1.1) has identified significantly more failed tests than reported in Defects4j documentation. Consequently, we used a newer version instead (1.7.3) for that project. This decision is similar to the one made in [11], where the authors used GZoltar 1.6.2 to localise faults in several bugs (not providing the detailed list).

The experiment has been performed in the way schematically depicted in Fig. 1. Firstly, the suspicious lines (MPs) for a considered bug were identified. Since we aimed to generate 1-edit degree patches, each from the identified MPs was selected and attempted to be patched independently from each other. Consequently, a separate APR job was executed for each MP of each considered bug. For the selected MP, its ingredients were found by screening all statements in the scope of a given MP. Then, the tests unrelated to the considered MP were filtered out. Finally, the bug was attempted to be

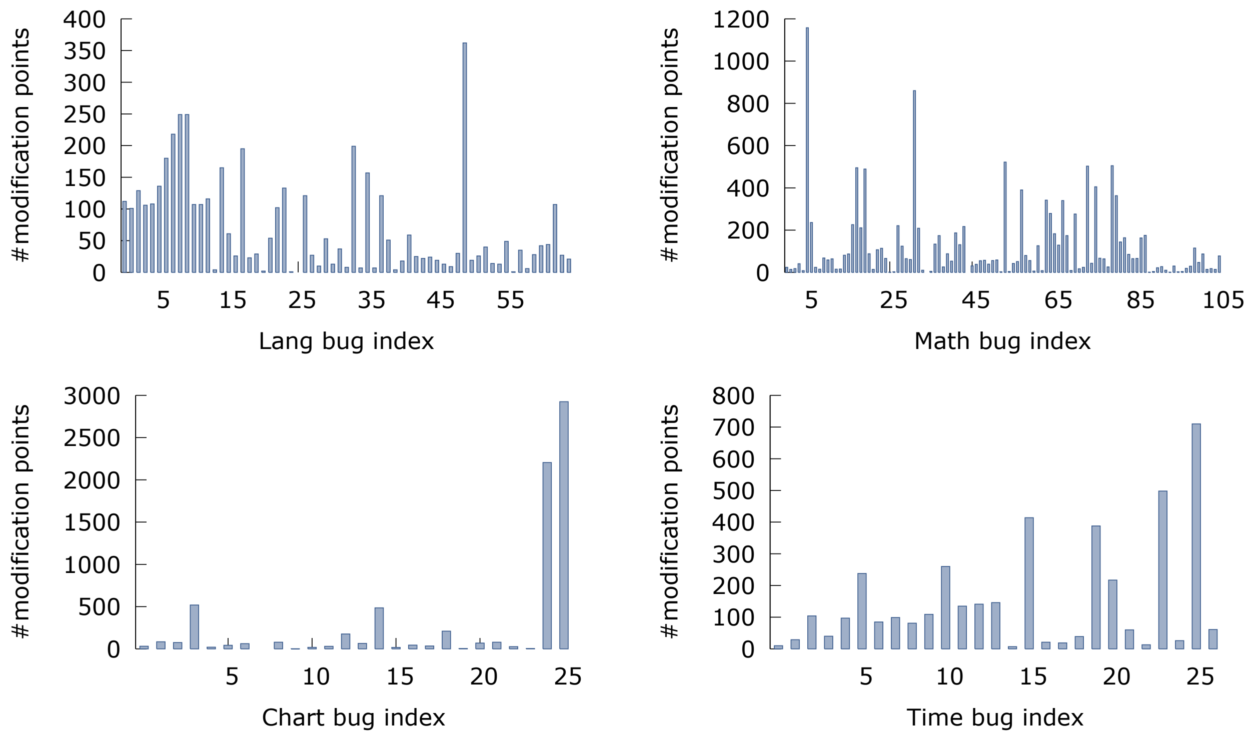

Fig. 2. Number of modification points in Defects4J projects

fixed. During each patching process, up to 10,000 random mutants of a selected MP were created, compiled and tested against the filtered test suite using a delete, insert before or replace operator, also selected randomly. These mutations have been performed at the code line granularity level. The delete operator removes a selected MP, whereas insert before and replace use ingredients from the MP's Java package, following strategy used in GenProg, to replace a selected MP or to insert them just before a selected MP, respectively.

During the experiment, 26,933 jobs have been executed in total. Each of them has been configured to use only one core and up to 4.8GB of memory. The timeout of each job has been set to 4 hours, so the upper bound of the computational time has been equal to 107,732 hours. Only 10.4 per cent of the jobs have finished before the timeout; for the remaining jobs, the number of generated patch candidates has equalled 99,305,101. From the generated patch candidates, 66% did not compile successfully whereas testing of 11% of patch candidates exceeded the timeout of test execution that was set to 6s (ARJA default). Only 0.08% of patch candidates (not necessarily different from each other) were plausible patches.

The experimental results are presented and discussed in the following section.

IV. Experimental results

In this section, the answers to the research questions formulated in Section II are given, based on the results of the experiment created in the way described in the previous section.

*1) **RQ1**:* The number of MPs in the considered projects is shown in Fig. 2. It is rather diverse, ranging from 1 (Math-26, Math-89, Math-94, Chart-10, Lang-25, Lang-27) to 2925 (Chart-26). The standard deviation of the number of MPs for the bugs in each considered problem is also quite diverse and equal to 24.5, 692.6, 178.6 and 172.8 for projects Lang, Chart, Math and Time, respectively. It may be then concluded that focusing on a constant number of MPs for each bug, which is a usual procedure in APR, results in very different treatment of each bug and in the extreme case of Chart-26, the default settings of the state-of-the-art APR tools such as ARJA ignore about 75% of MPs. In general, as many as 58% of bugs would be trimmed with the default ARJA settings. It may be then recommended to replace the constant number of analysed MPs to a value depending on the total number of MPs for a certain bug. But even for bugs with relatively low number of MPs omitting the MPs with lower suspiciousness values is risky. In the Math-85 case shown earlier in this paper, the total number of MPs is equal to 65. Yet, the line with the real bug is omitted by default as its suspiciousness value is not in the top 40. In this case, the low position in the suspiciousness ranking is caused by the lack of positive tests of this MP in the test suite.

Knowing the number of ingredients that can be substituted to each MP, shown in Fig. 3, it is possible to assess the search space size for each bug in terms of the possible replacements. These values for each bug are presented in Fig. 4. Again, the diversity of the obtained values is significant for each analysed project. For example, the number of possible mutations ranges

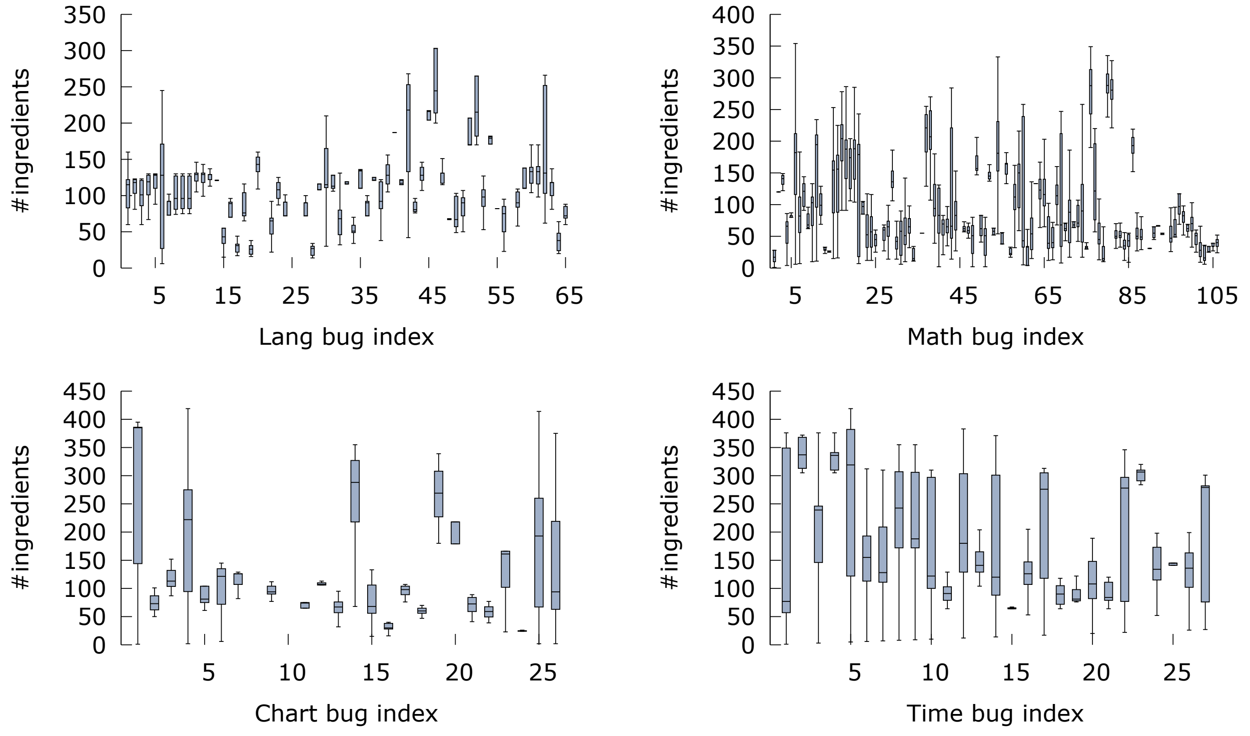

Fig. 3. Number of ingredients for Defects4J bugs

from 1 to 408,741 for the bugs in the Chart project. Assuming that 10,000 mutations can be performed and tested in a reasonable time (as shown in this experiment), as many as 54%, 55%, 53% and 26% of bugs can be attempted to be repaired exhaustively for the Chart, Lang, Math and Time project, respectively. However, the search space sizes provided in this section have been determined for the replacement operation only, i.e., when one MP is substituted with a compatible ingredient. If we allow the insert-before mutation operation, where a compatible ingredient is inserted before an MP, the search space size will double. The third popular mutation operation, removing an MP, would result in the search size increase by the number of MPs shown in Fig. 2.

The relatively large number of random mutations performed in the experiment (10,000 for each MP if the 4h timeout has not been reached) is likely to cover the vast portion of the search space for the majority of considered MPs. On average, the search space size decreased at least by 132, 73, 137 and 185 orders of magnitude for the Chart, Lang, Math and Time projects, respectively, in comparison with the search spaces for all MPs for a given bug (these figures are underestimated as for several bugs the search space size for multi-edit degree patches exceeded the range of the Double data type and hence the maximum Double value has been used instead). The difference between search space sizes exceeded 300 orders of magnitude for 6, 25 and 10 bugs from projects Charts, Math and Time, respectively. The largest search-space size reduction for the Lang project was equal 135 orders of magnitude.

*2) **RQ2**:* The total number of tests for bugs varies significantly in each considered project. In the extreme case of the Math-1 bug, there are 5219 positive and 3 negative tests. By allowing 1-degree edit patches only, significantly fewer tests can be considered since the positive tests unrelated to the considered MP can be safely filtered out [11]. For the Math-1 example, the average number of positive tests for all 24 MPs is equal to 10.8, i.e., is reduced by more than 99%, and 18 MPs have fewer positive tests than 10. The number of positive tests for all considered bugs is shown in Fig. 5. In this graph, other bugs with a significant reduction of positive tests for certain MPs can be identified in all the considered projects. On average, the number of positive tests after filtering for single MPs has been reduced by 97% (98%, 98%, 99% and 87% for Chart, Lang, Math and Time projects, respectively) in comparison with the number of positive tests being filtered for all MPs for a given bug. Similarly, it can be concluded that a number of tests has no sufficient test coverage. The case of zero positive tests has been noticed in almost 4% of MPs in all the considered projects, as presented in Figure 6. However, replacement of 1.6% of MPs with no positive tests have resulted in obtaining plausible patches. The correctness of these patches can be viewed as rather dubious.

*3) **RQ3**:* Overall, during the entire experiment, 47 bugs in Defects4J have been plausibly patched with 1-edit degree patches. Notice, in the current RQ only MPs belonging to these 47 buggy code versions are considered. All of them are listed in Fig. 7, where the left and right bar for each bug shows the average value of suspiciousness of the MPs that

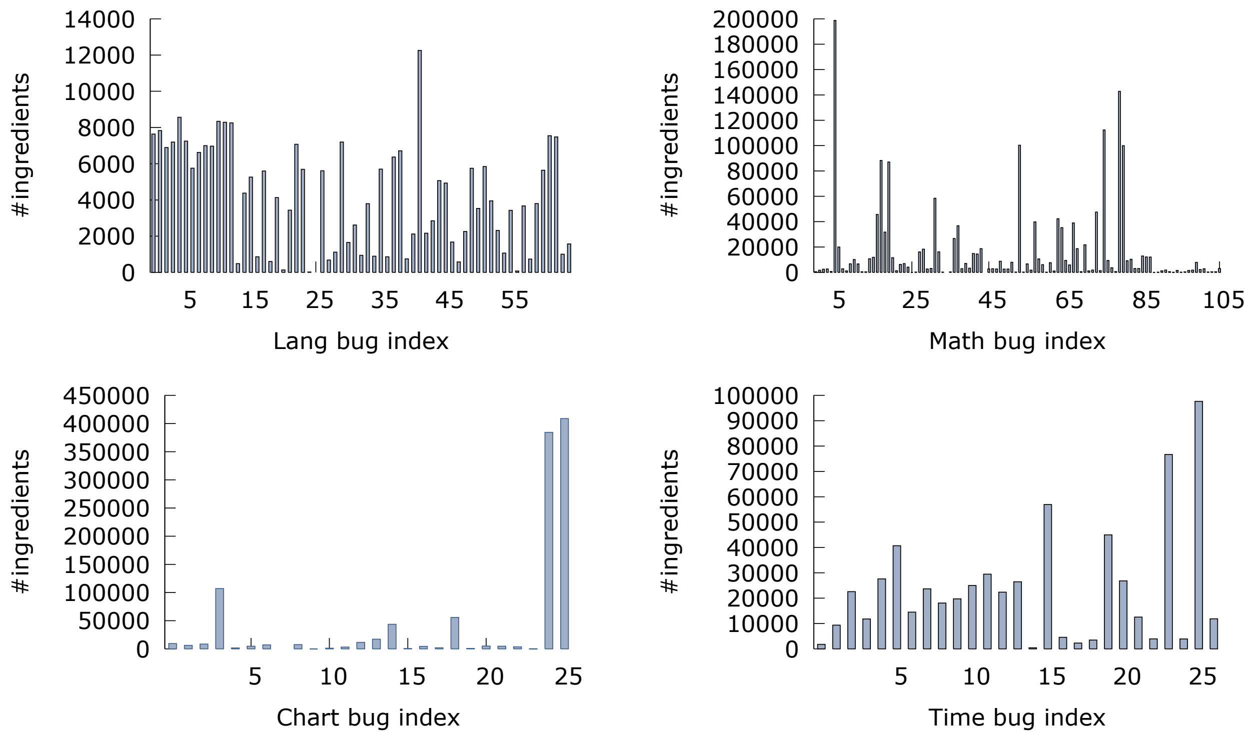

Fig. 4. The total number of ingredients (i.e., the search space size) for Defects4J bugs

TABLE I
SUSPICIOUSNESS VALUES OF PLAUSIBLE PATCHED AND NON PATCHED MPS

Suspiciousness Range	Percentage of MPs non-patched	Percentage of MPs plausibly patched
$[0.1, 0.2)$	32.90%	23.26%
$[0.2, 0.3)$	25.04%	11.78%
$[0.3, 0.4)$	15.64%	16.62%
$[0.4, 0.5)$	7.03%	7.85%
$[0.5, 0.6)$	5.16%	18.73%
$[0.6, 0.7)$	8.12%	1.51%
$[0.7, 0.8)$	2.81%	7.25%
$[0.8, 0.9)$	0.45%	0.00%
$[0.9, 1)$	1.14%	0.00%
1	1.71%	12.99%

have not been and have been, respectively, plausibly patched. As it is shown in the figure, for 32 bugs, plausibly patched MPs have higher values of suspiciousness, but in 15 cases, the opposite is true. In 16 cases, the most suspicious MP has been plausibly patched (including the 11 cases when the highest suspiciousness has been assigned ex aequo to MPs that have not been plausibly patched). The suspiciousness median of plausibly patched MPs equals 0.378, whereas the suspiciousness median for the remaining considered MPs is equal to 0.267. The distribution of the suspiciousness is shown in Table I for both the plausible patched MPs and other MPs considered in this RQ. In this table, it is clearly visible that plausible patched MPs have typically higher suspiciousness value range, yet more than 23% of such MPs have suspiciousness value below 0.2. Conversely, almost 3% of non-patched MPs have supsiciousness value higher or equal than 0.9. It can be than concluded that by focusing on the most suspicious MPs, a number of plausibly patchable MPs can be omitted. Finally, the suspiciousness values of both kinds of MPs (either plausibly patched or not) have been statistically compared using Mann–Whitney U test. U test is nonparametric and its null hypothesis assumes that it is equally likely that a randomly selected value from one population is lower or greater than a randomly selected value from a second population. The obtained p-value is lower than 0.00001, so the null hypothesis can be rejected under significance level $\alpha = 0.05$. Hence, plausible patched MPs are likely to have higher suspiciousness than other MPs for the same bugs.

4) ***RQ4****:* Most of the plausibly patched bugs have been patched by several 1-edit degrees patches which modify different MPs. The extreme situation has occurred for the Chart-25 bug that has been plausibly patched by modifying as many as 48 MPs in 3 different classes in total by 509 1-edit degree patches. These MPs constitute 2% of all suspicious MPs for this bug (this bug has one of the largest number of suspicious MPs, as shown in Fig. 2). As the human-written patch for this bug adds 14 lines in a single class [16], it is rather unlikely that any of the 1-edit degree patches is correct. Similarly, the plausible patches found by APR tools (and, as shown in Table II, this bug is plausibly patched by all considered tools) should be treated with caution. The set of 1-edit degree patches modifying more than one MP for the same bug are rather rule than exception, as for all 47 plausibly patched bugs, only in 14 cases all plausible patches found modify the same

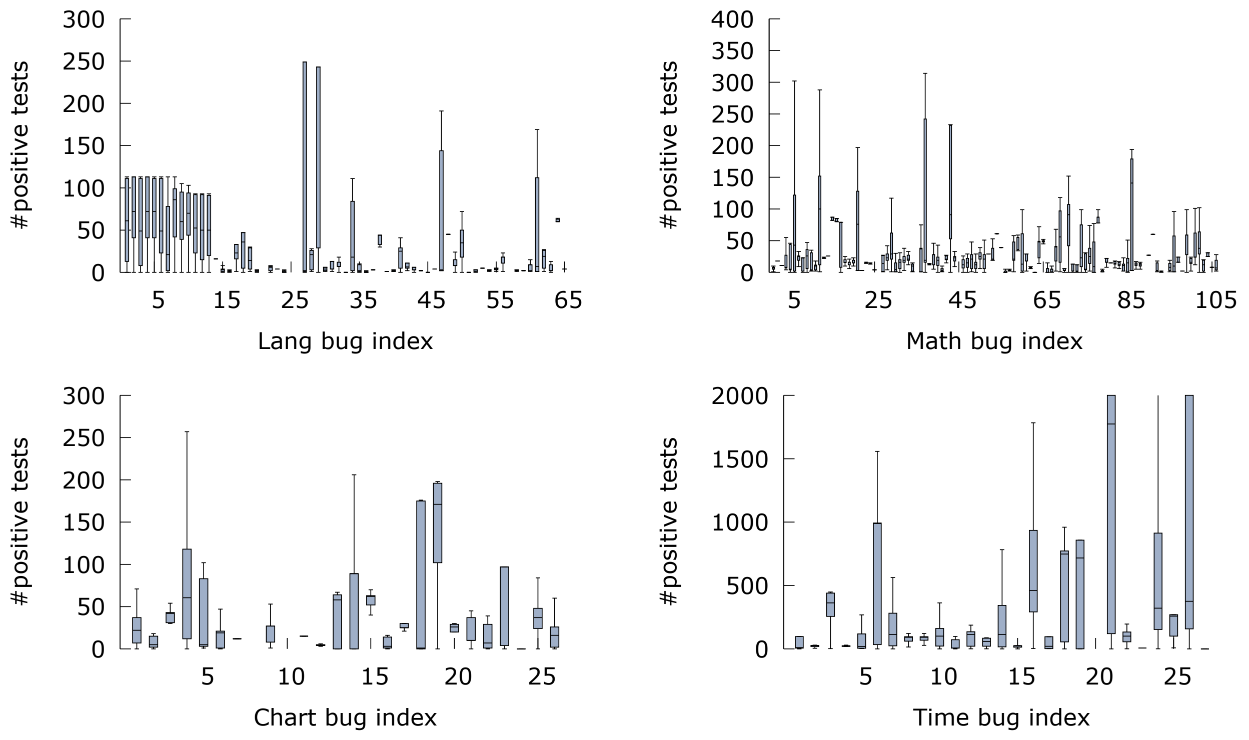

Fig. 5. Number of positive tests for Defects4J bugs

line. On average, the generated sets of 1-edit degree patches modify more than 7 MPs for a single bug, which points out a significant flaw of the considered benchmark at least in terms of the test coverage.

*5) **RQ5**:* The list of plausible patches generated by various state-of-the-art APR tools has been provided in Table II for the bugs considered in this paper. Nopol [17] is the only considered semantic-based repair method, the remaining approaches are syntax-based. The results obtained during the performed experiment are given in column `1-edit degree`. Regardless of the large number of mutations that is likely to balance the inferiority of random search in comparison with evolutionary-based search, reported in [11], 1-edit degree patches have not been found for several bugs plausibly patched by other syntax-based APR tools. The difference is particularly visible in the case of Math projects. It may be then concluded that some multi-edit degree patches generated by syntax-based APR tools are not equivalent to 1-edit degree patches, contrary to the claims in [6] or [8]. It is worth noting, however, that due to the large-scale search process, a few plausibly patched bugs have not been patched by the considered tools according to [9], [11]: Time-2, Time-9, Time-27, Lang-24, Lang-38, Math-64. Four bugs plausibly patched with a 1-edit degree patches have been patched only by Kali or Nopol (Chart-26, Lang-27, Lang-44, Lang-58, Math-88) (Kali does not use ingredients and hence by default 500 MPs with the highest suspiciousness are considered; Nopol is a semanti-based tool that fixes conditional statement bugs only). All these bugs are written in bold in the table. Since 1-edit degree patches can be generated quicker than longer patches (at least due to the lower number of positive tests to be executed, as shown earlier in this section), a larger number of MPs can be analysed and hence some bugs can be patched that would be difficult to be patched with other syntax-based APR approaches.

V. Related work

Since the earliest research in the APR domain, not much attention has been paid to 1-edit degree patches. Hence, their potential has not been well explored despite the radically smaller search space that for numerous cases could even be searched exhaustively (which has been shown in this paper) or a lower number of tests required to evaluate a patch. Kali [6] has modified only one functionality and it has been demonstrated that the number plausible patches generated in this way is quite similar to the number of patches found by the tools generating multi-edit degree patches. AE [8] has modified just a single MP in a patch and, similarly, generated a comparable number of plausible patches. Those observations have been made for C++ codes but have not been replicated for Java. For example, jKali, a Kali variant for Java from [18], has been reported significantly inferior with respect to the number of plausibly fixed bugs in [13]. In [19], 6 real-world Java programs have been transformed randomly by applying a single mutation to identify the code regions which do not alter functionality according to their test suites. The presence of such *plastic code regions* is likely to increase the number of the plausible patches found and this influence needs to be determined in future. Several papers related to search space

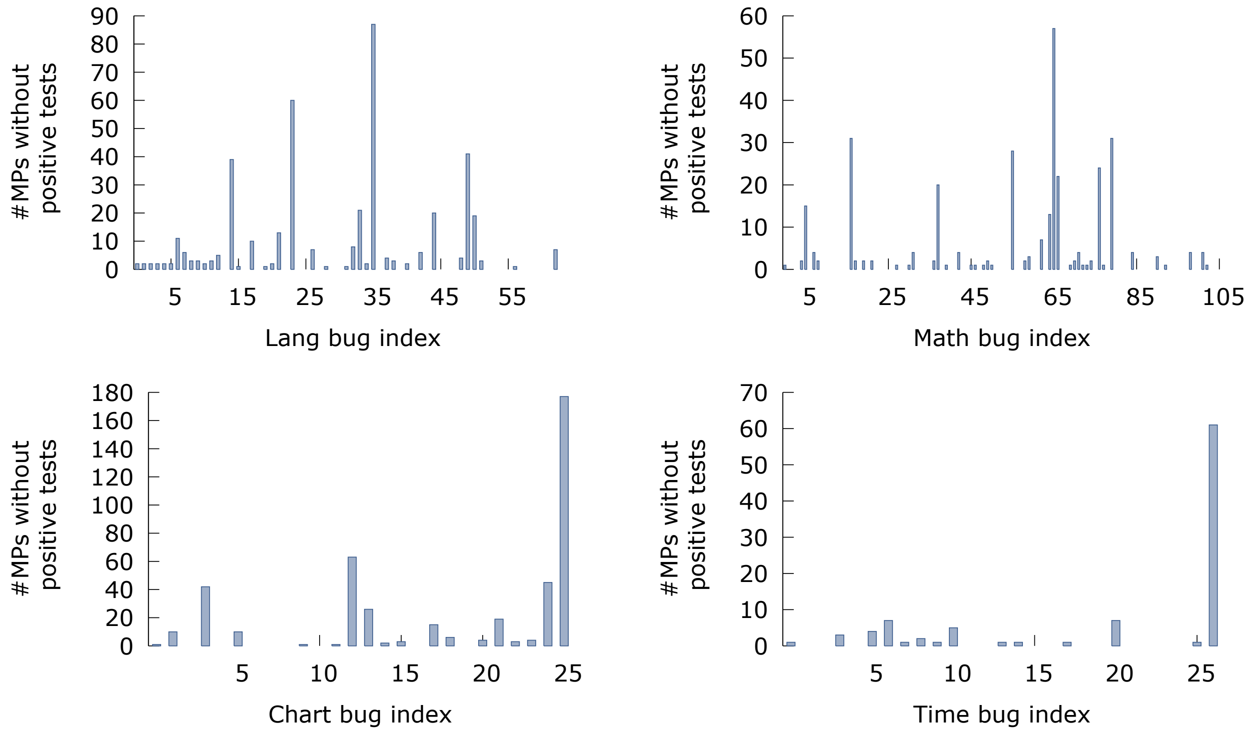

Fig. 6. Number of modification points with no positive test in Defects4J projects

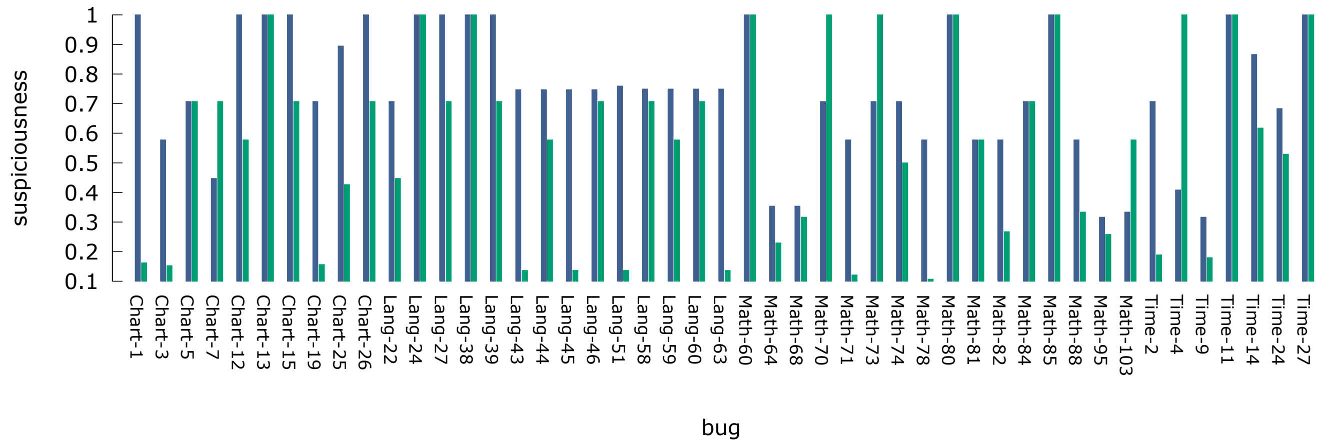

Fig. 7. The average suspiciousness for MPs with (right) and without (left) plausible patches for analysed bugs

analysis for functional improvement have been surveyed in [20].

Although most of the syntax-based APR tools have the capability of generating multi-line patches, most of such patches have been reported to be semantically equivalent to 1-edit degree patches [6]. For example, GenProg [2] keeps adding new modifications every generation during the genetic evolution process and hence its patches are likely to modify several lines even if they are correct but the modification is not caught by any test in a test suite [11] (our experimental analysis of jGenProg found that after 2000 generations of patching bug Math-85, the number of edits in a certain patch reached 1210, among which only ten modified different MPs; for example as many as 668 edits modified line 201 in `UnivariateRealSolverUtil.java` file, i.e., overridden the previous modifications of the same patch). ARJA [11] has also allowed multi-line patches but used the minimal size of a patch as one of two criteria for optimisation (the second one is, traditionally, minimisation of the number of failed tests in a suite). Also, the chromosome encoding used in ARJA guarantees that no two edits in a single patch modify the same MP. One of the research questions considered in [11] was related to multi-line patches explicitly. It has been shown that fixing at least 11 bugs from Defects4J needed more than 2

TABLE II
PLAUSIBLE PATCHES FOUND FOR DEFECTS4J PROJECTS WITH VARIOUS APPROACHES

Project	ARJA [11]	GenProg [11]	Kali [11]	jGenProg [3]	jKali [3]	Nopol [3]	1-edit degree
Chart	1, 3, 5, 7, 12, 13, 15, 19, 25	1, 3, 7, 12, 13, 18, 25	1, 5, 12, 13, 15, 25, 26	1, 3, 5, 7, 13, 15, 25	1, 5, 13, 15, 25, 26	3, 5, 13, 21, 25, 26	1, 3, 5, 7, 12, 13, 15, 19, 25, **26**
Time	4, 11, 14, 15	4, 11, 24	4, 11, 24	4, 11	4, 11	11	**2**, 4, **9**, 11, 14, 24, **27**
Lang	7, 16, 20, 22, 35, 39, 41, 43, 45, 46, 50, 51, 55, 59, 60, 61, 63	7, 22, 35, 39, 43, 51, 55, 59, 63	7, 22, 27, 39, 44, 51, 55, 58, 63			39, 44, 46, 51, 53, 55, 58	22, **24**, **27**, **38**, 39, 43, **44**, 45, 46, 51, **58**, 59, 60, 63
Math	2, 5, 6, 8, 20, 22, 28, 31, 39, 49, 50, 53, 56, 58, 60, 68, 70, 71, 73, 74, 80, 81, 82, 84, 85, 86, 95, 98, 103	2, 6, 8, 20, 28, 31, 39, 40, 49, 50, 71, 73, 80, 81, 82, 85, 95	2, 8, 20, 28, 31, 32, 49, 50, 80, 81, 82, 84, 85, 95	2, 5, 8, 28, 40, 49, 50, 53, 70, 71, 73, 78, 80, 81, 82, 84, 85, 95	2, 8, 28, 32, 40, 49, 50, 78, 80, 81, 82, 84, 85, 95	32, 33, 40, 42, 49, 50, 57, 58, 69, 71, 73, 78, 80, 81, 82, 85, 87, 88, 97, 104, 105	60, **64**, 68, 70, 71, 73, 74, 78, 80, 81, 82, 84, 85, **88**, 95, 103
Total	**59**	**36**	**33**	**27**	**22**	**35**	**47**

modifications. Yet, it has not been excluded then that there are no equivalent 1-edit degree plausible patches as a small part of the search space has been considered, as discussed earlier in this paper.

The patch search space size is hardly analysed in the literature and hence it does not influence the APR tool tuning. Typically, the computation is stopped after a predefined timeout, selected arbitrarily, e.g. three hours per repair attempt in [11]. As shown in this paper, the size of search space can differ significantly and it may be valuable to tune an APR tool respectively. Weimer et al. in [8] have presented the first formal cost model for syntax-based APR. Their model has accounted for the size of both the fault and the patch spaces. Following that model, the authors of that paper have been able to assess the search space reduction due to the applied approximate program equivalence. Wen et al. in [14] have performed correlation analysis between APR efficiency and such factors as fault space accuracy or test coverage. Liu et al. in [9] have also focused on the impact of various fault localisation techniques on APR efficiency when applied to the Defects4J benchmark. They have found that about one third of Defects4J bugs cannot be localised by the commonly used automated fault localisation techniques, such as GZoltar employed in this paper. Haraldsson et al. performed empirical analyses of the search landscape for mutated Python programs in [21]. According to their findings, the fitness often does not change after a single mutations of a considered software, which may be caused by the insufficient test coverage, which has been also demonstrated in this paper.

VI. CONCLUSION

In this paper, we conducted an experimental evaluation of the 1-edit degree patches generated automatically using a typical syntax-based APR technique. We investigated the differences between these patches and the corresponding patches generated by state-of-the-art syntax-based multi-edit APR tools and concluded that 1-edit degree patches have relatively small search space that in the majority of cases can be browsed in an exhaustive manner. Similarly, the number of tests needed to be executed for 1-edit patches is significantly reduced. In many cases, enumerated in this paper, there are no positive tests for certain MPs and hence the corresponding plausible patches are unlikely to correct the bug and should be treated with special care. We have also found that for many bugs, the 1-edit degree patches plausibly patch different lines in a project, sometimes even in different files. Again, this behaviour signals that such patches are likely to be incorrect. Although the number of bugs plausibly patched is lower for some projects, 1-edit degree patches can be generated for a larger number of MPs and hence it is possible to plausibly patch some bugs usually omitted by other syntax-based APR techniques (we have found 11 such cases).

From the results, it can be concluded that a certain number of 2-edit degree patches can have a relatively small search space that can be also browsed efficiently. Since, according to [8], most plausible patches are either 1-edit or 2-edit degree, this analysis would consist a practically-viable complement of the results presented in this paper and hence we plan to conduct it in future. Also, we plan to evaluate patches generated for other benchmarks to avoid the overfitting to Defects4J benchmark, as identified in [13].

ACKNOWLEDGMENT

The authors acknowledge the support of the EPSRC-funded MANATEE project.

References

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

[2] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, Jan 2012.

[3] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Softw. Engg.*, vol. 22, no. 4, p. 1936–1964, Aug. 2017. [Online]. Available: https://doi.org/10.1007/s10664-016-9470-4

[4] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation*, ser. GECCO '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 947–954. [Online]. Available: https://doi.org/10.1145/1569901.1570031

[5] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014. New York, NY, USA: ACM, 2014, pp. 254–265. [Online]. Available: http://doi.acm.org/10.1145/2568225.2568254

[6] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ser. ISSTA 2015. New York, NY, USA: ACM, 2015, pp. 24–36. [Online]. Available: http://doi.acm.org/10.1145/2771783.2771791

[7] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun, "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*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 532–543. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786825

[8] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE'13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 356–366. [Online]. Available: https://doi.org/10.1109/ASE.2013.6693094

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

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

[15] M. Martinez and M. Monperrus, "Ultra-large repair search space with automatically mined templates: The cardumen mode of astor," in *Search-Based Software Engineering*, T. E. Colanzi and P. McMinn, Eds. Cham: Springer International Publishing, 2018, pp. 65–86.

[11] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[12] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440. [Online]. Available: http://doi.acm.org/10.1145/2610384.2628055

[13] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: ACM, 2019, pp. 302–313. [Online]. Available: http://doi.acm.org/10.1145/3338906.3338911

[14] M. Wen, J. Chen, R. Wu, D. Hao, and S. Cheung, "An empirical analysis of the influence of fault space on search-based automated program repair," *CoRR*, vol. abs/1707.05172, 2017. [Online]. Available: http://arxiv.org/abs/1707.05172

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

[17] J. Xuan, M. Martinez, F. DeMarco, M. Clément, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.

[18] M. Martinez and M. Monperrus, "Astor: Exploring the design space of generate-and-validate program repair beyond genprog," *Journal of Systems and Software*, vol. 151, pp. 65 – 80, 2019. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121219300159

[19] N. Harrand, S. Allier, M. Rodriguez-Cancio, M. Monperrus, and B. Baudry, "A journey among java neutral program variants," *Genetic Programming and Evolvable Machines*, pp. 531–580, 06 2019.

[20] J. Petke, B. Alexander, E. T. Barr, A. E. I. Brownlee, M. Wagner, and D. R. White, "A survey of genetic improvement search spaces," in *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, ser. GECCO '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1715–1721. [Online]. Available: https://doi.org/10.1145/3319619.3326870

[21] S. O. Haraldsson, J. R. Woodward, A. E. I. Brownlee, and D. Cairns, "Exploring fitness and edit distance of mutated python programs," in *Genetic Programming*, J. McDermott, M. Castelli, L. Sekanina, E. Haasdijk, and P. García-Sánchez, Eds. Cham: Springer International Publishing, 2017, pp. 19–34.