VarFix: Balancing Edit Expressiveness and Search Effectiveness in Automated Program Repair

Chu-Pan Wong
Carnegie Mellon University
Pittsburgh, PA, USA

Priscila Santiesteban
Coe College
Cedar Rapids, IA, USA

Christian Kästner
Carnegie Mellon University
Pittsburgh, PA, USA

Claire Le Goues
Carnegie Mellon University
Pittsburgh, PA, USA

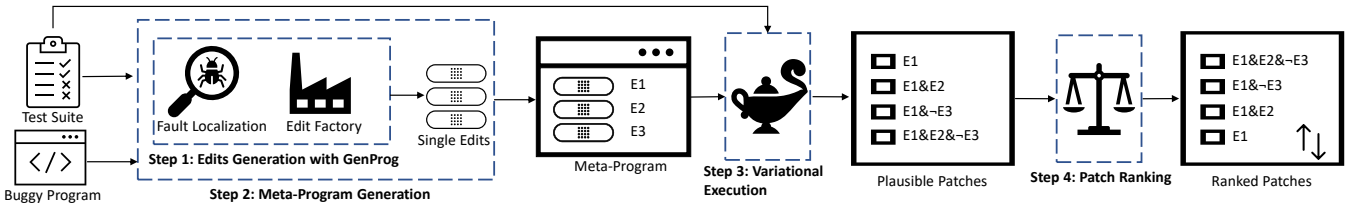

Figure 1: Overview of our program repair approach.

ABSTRACT

Automatically repairing a buggy program is essentially a search problem, searching for code transformations that pass a set of tests. Various search strategies have been explored, but they either navigate the search space in an *ad hoc* way using heuristics, or systemically but at the cost of *limited edit expressiveness* in the kinds of supported program edits. In this work, we explore the possibility of *systematically* navigating the search space without sacrificing *edit expressiveness*. The key enabler of this exploration is variational execution, a dynamic analysis technique that has been shown to be effective at exploring many similar executions in large search spaces. We evaluate our approach on IntroClassJava and Defects4J, showing that a systematic search is effective at leveraging and combining fixing ingredients to find patches, including many high-quality patches and multi-edit patches.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**; **Error handling and recovery**.

KEYWORDS

automatic program repair, variational execution

ACM Reference Format:
Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 2021. VarFix: Balancing Edit Expressiveness and Search Effectiveness in Automated Program Repair. In *Proceedings of the 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '21), August 23–28, 2021, Athens, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3468264.3468600

This work is licensed under a Creative Commons Attribution International 4.0 License.

ESEC/FSE '21, August 23–28, 2021, Athens, Greece
© 2021 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-8562-6/21/08.
https://doi.org/10.1145/3468264.3468600

1 INTRODUCTION

We propose a novel search strategy for automated program repair to systematically explore possible repairs in rich search spaces with diverse fixing ingredients, including multi-edit patches. Bugs are pervasive and costly [41] and traditional manual repair is expensive and time-consuming [33]. Automated program repair is a promising idea, aimed at automatically finding patches to a program, such that the program passes all provided tests. Automated program repair has seen significant research attention [27] as well as initial industrial adoption [3, 21].

Automated program repair techniques generate *patches* for a buggy program to satisfy a given (partial) *specification* of program behavior, usually provided as a set of tests [18, 20, 25, 27, 44, 45]. A *patch* typically contains one or more code *edits* that are generated by instantiating *edit templates* for different fixing ingredients, similar to how mutation operators are used to generate mutations in mutation testing research. Given a program and its test suite with at least one failing test, repair approaches search for patches such that the test suite passes.

Existing approaches to automatic program repair essentially solve a *search problem*. The *search space* is defined by the set of possible edits, e.g., copying code [44, 45] or modifying expressions [8, 18, 25]; it is typically large and grows *exponentially* if combinations of multiple edits are considered. To navigate such a huge search space, different search strategies have been explored, such as genetic programming [44], guided search using statistical models [20, 45], and program synthesis [8, 15, 25]. There are two keys to solving the search problem, *edit expressiveness* and *search effectiveness*. These concerns are in tension, and existing techniques usually prioritize one of them:

- *Heuristics-based approaches* like GenProg [44], PAR [14], and CapGen [45] excel at edit expressiveness. They explore a large space of potential edits by executing the test suite for one patch candidate at a time. This process is well-suited for trying many different kinds of edits, including adding and removing *statements* [44], mutating expressions [15, 22, 25, 45], and edits following specific patterns such as inserting null checks [14]. But, as search spaces grow, search effectiveness typically declines [19].
- *Semantics-based approaches* like Angelix [25] and S3 [15] excel at search effectiveness by encoding the search as a synthesis problem [8, 15, 15, 25, 37, 51] (using classic AI search techniques focused on pruning infeasible parts quickly, as in SAT and SMT [34]) to find expression-level changes that meet the constraints collected with program analysis (usually *symbolic execution*) from tests. The synthesis technique limits expressiveness and scalability, resulting typically in a narrow focus on few kinds of edits, e.g., edits of *expressions* of boolean and integer types in conditions or assignments [8, 15, 15, 25, 37, 51].

In this work, we pursue higher *search effectiveness* without sacrificing *edit expressiveness*. Our goal is to perform an efficient systematic search in a search space of allowing many different kinds of edits as in heuristics-based approaches. Our key insight is that repeated test executions for many potential patches are very similar (as edits tend to be focused in a relatively small part of the trace as narrowed down by fault localization techniques) and exploiting those similarities can speed up the search. To exploit test execution similarity, we use *variational execution* [1, 26, 29, 47].

Variational execution is a dynamic analysis that executes a program for multiple inputs or multiple variants once, sharing the execution whenever possible, and systematically exploring all alternatives and interactions. Reminiscent of many model checking strategies, when an edit is encountered, variational execution *splits* the execution to compute the program states with and without the edit. Also, importantly, variational execution then *merges* executions again to execute the rest of the program only once. Conceptually, given a finite search space of edits as fixing ingredients, a single run of variational execution is equivalent to running all edits and their combinations in isolation. With sufficient sharing, variational execution often can systematically explore large search spaces efficiently, as shown in recent work on tracking sensitive information flow [2, 52], testing highly configurable systems [26, 29], and finding higher-order mutants [46].

As we will explain in more detail, our approach consists of four steps, also illustrated in Figure 1: **(1)** We collect a set of possible edits, similar to traditional heuristics-based techniques. **(2)** We merge all collected edits into a meta-program, where all edits are guarded by if-conditions that control whether to include the edit at runtime. **(3)** We execute the meta-program with variational execution to observe which combinations of edits (in terms of the Boolean variables that guard the edits) pass each test, returned as propositional formulas, from which we enumerate all patches that pass all tests within the search space. **(4)** To identify likely high-quality patches among all plausible patches, we further filter and rank patches, using heuristics based on their influence on program state and control flow.

```
1  if (a < b && a < c && a < d) {
2      smallest = a;
3  } else if (b  1  < ⇒ <=  a && b < c && b < d) {
4      smallest = b;
5  } else if (c < a && c < b && c < d) {
6      smallest = c;
7  } else {
8      if (d < a && d < b  2  && ⇒ ||  d < c) {
9        smallest = d;
10     }
11     3  smallest = c;
12 }
```

Figure 2: Incorrect program with suggested 3-edit patch, simplified from `smallest-1b31fa5c-003`. Intuitively, Edit 3 makes `c` the default output, Edit 1 handles cases where `a` and `b` are equal and smallest, and finally Edit 2 checks for cases where `d` is the smallest.

We evaluate our approach both with the 297 bugs from IntroClassJava and with 282 bugs from Defects4J. We show that our approach can fix 107 IntroClassJava bugs and 35 Defects4J bugs, including many that require edits in multiple locations, can scale to fairly large programs and search spaces, can generate very large sets of patches for bugs (up to 32841 correct patches for a single bug), and can effectively rank correct patches highly.

We make the following contributions in this work:

- We prototype a repair tool named VarFix built on top of GenProg and VarexC, the state-of-the-art implementation of variational execution for Java.
- We demonstrate that it is promising to systematically explore large search spaces for program repair using variational execution (e.g., fixing 142 out of 579 evaluated bugs).
- We demonstrate the ability to find high-quality patches, made possible by a systematic exploration of the search space (e.g., generating thousands of correct patches).
- We demonstrate the ability to better find multi-edit patches, as a key benefit of the more efficient search strategy (e.g., finding hundreds of patches that require 3 edits).
- We demonstrate the benefit of enumerating and ranking many plausible patches over just reporting the first one found in terms of improved patch quality (e.g., ranking the correct patch in top 10 in 23 out of 24 cases).

2 REPAIR CHALLENGES

We use an example to discuss limitations of existing work and benefits of our approach. In Figure 2, we show a buggy program taken from IntroClassJava and a patch found by our approach. The program should output the smallest of four inputs. Due to the use of incorrect relational operators, it fails if the smallest number appears more than once in the inputs. Our patch consists of three edits: two modify operators in Boolean expressions, while the third inserts a statement taken from the existing code. For the same program, other patches can be found, all requiring multiple edits.

2.1 Edit Expressiveness vs. Search Effectiveness

The success of existing program repair techniques largely depends on the balance between edit expressiveness and search effectiveness, which affects their ability to fix programs like our example.

Heuristics-Based Repair. Heuristics-based approaches usually can experiment with many different kinds of edits, including the statement and expression changes in Figure 2. For example, GenProg [17, 44] can add/replace/delete statements in the original program, such as Edit 3 in Figure 2; JMutRepair [22] uses classic mutation operators to tweak existing expressions, such as Edits 1–2 in our example; others specialize edits for certain classes of faults, such as patterns mined from human patches and patterns for common mistakes used in PAR [14], SPR [18] and Prophet [20]. Overall, the community is moving toward increasingly larger search spaces of edits.

The larger the space of possible fixing ingredients, the more challenging the search for a patch [19]. Even just executing tests for single-edit patches can take a long time and a systematic exploration of multi-edit patches is usually not considered feasible. Instead, existing approaches largely rely on different heuristic search, such as random search [32], genetic programming [17, 44], deterministic heuristic traversal of single-edits [43], and probabilistic exploration favoring certain kinds of edits, possibly informed by distributions of human patches[18, 20, 45]. Mechtaev et al. [23] divide individual fixing ingredients into test-equivalence classes to reduce test execution cost and thus improve search efficiency of existing repair techniques. None of the current approaches can systematically explore larger search spaces, and current search heuristics often do not work well for multi-edit patches (e.g., fitness functions in genetic programming cannot well recognize partial correctness in multi-edit patches); it is possible but unlikely that they find the right combinations of the three edits in our motivating example. Indeed, empirical evidence suggests that existing search strategies perform worse in larger search spaces, generating fewer correct patches due to timeout or stopping with plausible but ultimately incorrect patches [19].

Semantics-Based Repair. Semantics-based approaches can effectively explore search spaces by delegating the search to program synthesis techniques, but they usually restrict themselves to small edits *at the expression level* (e.g., tweaking Boolean expressions in `if` conditions) to make program synthesis tractable. Edit expressiveness is mainly restricted by the scalability of constraint solvers and the types of theory they can reason about [15]. Thus, existing approaches are unlikely to find patches for our motivating example, because (1) statement level changes are not supported and (2) synthesizing multiple expression-level changes in different locations poses a scalability challenge for synthesis, especially for large programs in practice. For example, DirectFix needs to symbolically execute the whole program under repair to synthesize edits at multiple locations [24]; Angelix mitigates this issue by applying symbolic execution exclusively to a few suspicious expressions, but requires buggy locations to be physically close [25]. S3 essentially repairs each buggy location separately, putting more pressure on the underlying program synthesizer [15].

2.2 Open Challenges

Several open challenges of automatic program repair boil down to maintaining a good balance between edit expressiveness and search effectiveness. Our approach strives for a balance by pursuing a more systematic search within a search space of more expressive edits. In the following, we discuss some open challenges our approach can potentially shed light on:

General-Purpose Repair. Usually, the more expressive edits are supported, the larger the search space becomes, and the more difficult it becomes to explore that search space systematically. Approaches with more expressive edits have the theoretical ability to repair more bugs, but may find it harder to find actual repairs in practice. Our approach can easily integrate different kinds of edits, just like heuristics-based approaches, promising high edit expressiveness. It then enables an efficient search among a *finite* number of edits and their interactions, more similar to test execution in heuristics-based approaches, without being limited by the capabilities of synthesis approaches.

Multi-Edit Repair. Recent empirical studies suggest that making multiple edits is common when developers fix bugs [12, 54], but automatically generating multi-edit patches remains challenging. In theory, many repair approaches can find multi-edit patches. In practice, they struggle with finding them because techniques do not explore edit interactions systematically and do not have a good way of judging partial correctness that may help to guide the search for multiple fixing ingredients. Our approach can explore interactions among multiple edits systematically (up to a configurable bound), ensuring that, if our search terminates, we can identify all available multi-edit patches within a given search space.

Patch Quality. The research community has identified the tendency of program repair to produce low-quality patches that pass all provided tests, but do not generalize beyond them, as a key challenge [27, 38]. We follow the typical distinction between *plausible patches* as those that pass all tests and *correct patches* that meet the intended specification (typically not easily available, and may require human judgement). Long and Rinard [19] found that plausible patches often starkly outnumber correct patches, making automated repair possibly less useful, especially those techniques that simply report the first identified plausible patch. Recent work primarily focuses on anti-patterns to skip low quality patches [40] or on tweaking search strategies to prioritize edits that are less likely to overfit [49]. In contrast, we use a different strategy of embracing a rich and diverse edit space, then enumerating all plausible patches within the search space, but adding a final filtering and ranking step to suggest those patches that are likely of higher quality. In line with prior suggestions [19, 49], our ranking step uses information beyond just the test suite to evaluate patch quality, and additionally compares the severity of data and control-flow changes for the enumerated plausible patches. Ranking patches from a large pool of plausible patches provides a more promising way of identifying high-quality patches than simply returning the first plausible patch (as in heuristics-based approaches) or the smallest plausible patch (as in semantics-based approaches).

3 META-PROGRAM GENERATION

The first part of our approach (cf. Figure 1, Steps 1 and 2) is to generate a large set of edits and combine them into a single meta-program, where each edit is guarded by a control-flow condition. With this encoding, we can later use variational execution to efficiently run the tests across all combinations of edits. The edits in the meta-program define the (finite) search space of patches.

```
if (a < b && a < c && a < d) {
    smallest = a;
} else if (b [1 (e1 ? b <= a : b < a)] a && b < c && b < d) {
    smallest = b;
} else if (c < a && c < b && c < d) {
    smallest = c;
} else if ([2 e2?(d<a && d<b || d<c):(d<a && d<b && d<c)]) {
    smallest = d;
} else {
  [3 if (e3)]
  [3   smallest = c;]
  [3 else]
  [3   smallest = Integer.MAX_VALUE;]
}
```

Test	Original	Passing Condition
`assert smallest(1, 2, 3, 4) == 1`	✓	`true`
`assert smallest(1, 1, 1, 1) == 1`	✗	`e3`
`assert smallest(1, 1, 2, 3) == 1`	✗	`e1`
`assert smallest(1, 2, 3, 1) == 1`	✗	`e2`
Whole test suite	✗	e1 ∧ e2 ∧ e3

Figure 3: The upper part is an example of meta-program that encodes 3 edits for `smallest-90834803-005`. The lower part is a manually constructed test suite for demonstrating how variational execution is used. The Original column reports test outcomes of the original buggy program.

For generating edits, we build our approach on top of a GenProg implementation for Java [39] because of its conceptual simplicity and extensibility. We first use GenProg's fault localization technique to narrow down where to generate edits. In those likely-faulty locations, we use GenProg's edit templates to generate edits, one at a time. We inherit the three classic edit templates `APPEND`, `REPLACE`, and `DELETE` that modify statements, as well as five additional classic mutation operators: Arithmetic Operator Replacement (`AOR`), Relational Operator Replacement (`ROR`), Logical Connector Replacement (`LCR`), Absolute Value Insertion (`ABS`), and Unary Operator Insertion (`UOI`). We selected these five operators for several reasons. First, they mutate expressions, allowing our search space to include finer-grained edits than GenProg 's original statement-level changes. Second, research has shown the usefulness of these operators in mutating programs [30, 31]. Finally, they can be applied to a wide variety of programs, including simple programs in IntroClassJava. Other edit templates (e.g., those that target specific fault classes [19, 35, 45]) can be easily integrated as extensions.

We modified GenProg to repetitively mutate the given buggy program until a specified number of different edits are generated or all possible edits from the templates are exhausted. We then discard edits that are not compilable (a standard GenProg step). Finally, we merge those edits into one meta-program.

Similar to prior work that uses meta-programs [6, 13, 43, 46], we encode all edits as optional code paths (guarded by `if-then-else` statements/expressions) into a single meta-program. For each edit, we introduce a Boolean option (e.g., global static field in Java) that decides whether the original or the edited code is executed in runtime, as illustrated in Figure 3.

While the approach is simple and flexible, supporting many different edits, there are two technical challenges in the implementation: First, putting too many edits into a single Java method can exceed size limits of Java bytecode. We refactored a few gigantic Java methods (e.g., more than 800 lines of code) in some of the Closure programs into smaller methods because of size limits. This is unique to the specific bytecode transformation implemented in VarexC, the tool we use for variational execution, rather than to variational execution in general. Whenever possible, we push edits into small methods after performing semantics-preserving refactoring. We run all existing tests to validate the refactoring changes. Second, our approach inherits nondeterminism from GenProg when generating edits, thus while we will systematically explore the search space, the generated edits in the search space may differ between runs.

4 SYSTEMATIC SEARCH

After creating the meta-program, we execute the test suite using variational execution to determine what combinations of edits can pass all tests (cf. Figure 1). Adopting variational execution raises search effectiveness because, with it, we can explore the search space in an *efficient* and *comprehensive* way, rather than checking one patch at a time.

4.1 Background on Variational Execution

Variational execution can be seen as a specialized form of symbolic execution or model checking that aggressively joins state and that limits symbolic inputs to those with finite domains (usually Boolean values). For the purpose of this paper, a mental model of symbolic execution will provide a close-enough approximation of how the approach works. Given that we largely reuse variational execution as an off-the-shelf black-box technique, we will only provide a quick overview, and refer interested readers to existing literature for a more in-depth explanation and formalization [1, 26, 29, 47].

Using variational execution, we execute the test suite of a meta-program with the test suite's *concrete inputs*, but with *symbolic values* for all Boolean variables introduced in the meta-program to guard the edits to be explored (e.g., `e1`, `e2` in Fig. 3). Variational execution will execute a test normally with concrete values until it hits a decision that depends on an a symbolic value (representing an edit), where it explores both branches. After exploring both branches under corresponding symbolic path conditions, variational execution joins the state again to execute subsequent statements only once, while representing state differences as if-then-else expressions. For example, `x = e1 ? (e2 ? 2 : 9) : 1` represents concrete value 1, 2 or 9 depending on whether edits `e1` and `e2` were selected. When the execution hits an assertion, we can derive a propositional formula describing the set of assignments to symbolic values (i.e., the set of edit combinations) that violate the assertion; for example, `assert x>5` fails under condition ¬`e1` ∨ `e2`. Similarly, we can record the path condition when an exception is thrown. That is, for each test execution, we can collect the set of edit combinations for which the test passes as a propositional formula, without having to execute the test for every edit combination separately.

If edits interact too much, variational execution can run into the same state explosion problem as other analysis techniques;

the key question is whether the program repair problem contains sufficient sharing to make variational execution feasible. The key insight in using variational execution is that most test executions are similar or even identical when edits modify the buggy program. Many edits have only minor and local effects on control flow and program state of some test executions, and most computations are the same or similar independent of whether an edit is applied. Variational execution exploits those sharing opportunities among many similar executions with minor differences, regarding both data flow and control flow. But, it also systematically explores all interactions among options if they lead to state differences. Note that variational execution is unusually aggressive in how it always merges all program state at the level of variables and fields after each statement, helped by its restriction to finite search spaces. Similar state merging techniques have also been discussed and adopted for model checking and symbolic execution (e.g., [36, 42]), achieving significant improvement over conventional state merging.

4.2 Using Variational Execution to Find Plausible Patches

To identify which combinations of edits can pass all tests and fix the buggy program, we execute each test for the meta-program (with symbolic values representing all edits) and capture for each test the *passing condition*, a propositional formula compactly describing the combinations of edits for which the test passes. Any combination of edits that meets the passing conditions of *all* tests represents a plausible patch. To enumerate all plausible patches, we use a SAT solver or BDD to enumerate all solutions that satisfy the conjunction of all passing conditions.

For example, in Figure 3, `assert smallest(1, 2, 3, 4) == 1` passes under condition `true`, meaning that the test passes without any edits and no edit breaks the test either. However, the second test `assert smallest(1, 1, 1, 1) == 1` passes under condition `e3`, meaning that it can pass only when Edit 3 is applied, independent of the other edits. Finally, the conjoint passing condition for the whole test suite is `e1` ∧ `e2` ∧ `e3`, suggesting that we need all three edits to fix the program.

If variational execution and the SAT solver terminate, we have explored the entire given search space of patches. We can be sure that we will have found all edit combinations that pass the test suite and no additional ones. If the joint passing condition is unsatisfiable, we know that there is no edit combination in the search space that passes all tests—that is, we simply do not have the right fixing ingredients to patch the program.

4.3 Implementation and Limitations

We use VarexC, a state-of-the-art variational execution engine for Java developed in prior work [47]. In practice, especially for large search spaces that contain thousands of edits, we still often experience slow executions and executions that do not terminate within time budget due to state space explosion. To address these, we made several adjustments to make it better fit the characteristics of program repair problems, usually by focusing the search on smaller spaces. We name our overall repair tool VarFix.

Early termination. We use variational execution to execute test cases one at a time (with all edit combinations), but prioritizing test cases that the original buggy program fails. After each test execution, we check if there exist any combinations that can pass the tests executed so far, and continue to the next test only if there remain solutions in the search space. This way, we can often terminate the search early, knowing that no combination of the edits in the meta program will pass all tests.

Bounded search. While variational execution can conceptually explore all possible interactions within the search space, doing so may be expensive when many edits interact. In addition, patches that combine more than a few edits may be too complex to be useful. We extended VarexC to (optionally) bound the search space to limit the number of individual edits in a patch. Technically, we simply prune states and execution paths that are only feasible with more than n activated edits. Note that we still systematically explore all edit combinations *within the bound*.

Partitioning edits. Orthogonal to bounding the degree of interactions, we can also explore interactions among fewer edits. We implement an option to partition the edits, considering at most n edits and their interactions at a time. The partition strategy can be customized, including random (default) or partition by location or fault-localization suspiciousness. Within each partition, we still explore interactions among edits, but we would not detect patches that require combinations of edits from different partitions, thus may miss patches that would be feasible in the larger search space.

Fast mode. We noticed that variational execution spends a lot of time exploring exception handling paths (e.g., division by zero), which is expensive in VarexC [47]. While successful patches might conceivably involve throwing and catching exceptions, we (optionally) prune those execution paths where exceptions are thrown due to one or more edits. Note that we may miss patches due to this optimization, unless we explore the pruned paths later. This optimization is similar to how S3 uses symbolic execution to run *failure-free* execution paths [15].

Limitations. Similar to symbolic execution, variational execution needs to handle the environment barrier when interacting with an external runtime environment, for example, via I/O and native method calls. This issue can typically be mitigated with engineering effort, such as implementing model classes to mimic behavior of the environment [47]. As a research prototype, VarexC provides model classes for many common APIs, but not for all; hence we performed minor refactoring in some of the Closure programs to use functionally equivalent model classes that VarexC already supports, such as replacing `StringWriter` with `StringBuilder`.

In some systems we still observed a small number of tests that we could not execute without investing significant engineering effort. Here we adopted a hybrid strategy: First, we collect passing conditions from all tests we can execute with variational execution, skipping the problematic ones. Second, we execute the skipped tests normally without variational execution for all edit combinations that pass the joint pass condition to see whether they fail any of the skipped tests. This way, variational execution significantly prunes the search space, but some final exploration remains limited to executing the remaining tests one patch candidate at a time.

Finally, edits often introduce infinite loops in some execution paths. Since we do not want to terminate the test case for all executions, we prune execution paths that exceed an upper limit for the number of executed basic blocks in a method or a maximal stack height.

5 PATCH FILTERING AND RANKING

With variational execution, we can enumerate all plausible patches in a search space, often finding large numbers of plausible patches that can pass *all* tests. However, not all patches generalize well beyond the test suite. Our approach opens new opportunities to try to identify higher-quality patches among a pool of plausible patches, rather than simply returning the first patch found. We demonstrate the benefits of this strategy with a simple filtering and ranking strategy, that can easily be extended in future work.

Filtering by patch minimization. For every plausible patch, we often find additional patches with a superset of edits. This often happens when some edits are needed to pass the tests, and other edits can be added that do not influence the test outcome or are not even executed by the tests. The extra changes may fix issues for inputs not tested by the test suite, may break behavior for inputs not tested, or may simply be behavior-preserving for all possible inputs—we have no way of telling. Patch quality is often measured with held-out tests in academic evaluations [15, 28, 38] or by comparing against another oracle, but neither of those is available in practical settings.

In a *patch minimization* step, we filter all patches for which we found another patch with a strict subset of edits. That is, we remove all edits that are not necessary for passing all tests. This way, we avoid showing many similar patches with the same core edits to developers and we generally favor shorter patches. This is conceptually equivalent to patch minimization using delta debugging in the original GenProg implementation [44].

Patch ranking. Since all plausible patches are indistinguishable from the test suite's perspective (i.e., all tests pass for all plausible patches), we need to judge patch quality using other information [19]. Our solution is inspired by insights from Xiong et al. [49], who observed that a correct patch should have little effect on passing tests (in terms of similarity between execution traces), but should cause failing tests to behave differently. A side benefit from variational execution is that we can easily track rich information about differences between executions, by tracking path conditions and conditions that describe state differences. Note that variational execution is not essential here, just convenient; such information could conceptually also be collected with other tools as in the work of Xiong et al. [49].

In a nutshell, we compute a distance between the unpatched and the patched program. We experimented with five different distance measures. In addition to two static *edit-distance measures*, (1) number of AST transformations measured with GumTree [10] and (2) number of characters changed measured with Levenshtein distance, we measure the difference between a test's *execution* on the original unpatched program and the test's execution on the patched program: (3) frequency with which we assign different values in statements that change program state, (4) frequency of different branches taken at control-flow statements, and (5) differences in terms of executed lines. For the last three measures, we aggregate distances across tests similar to Xiong et al. [49], adding the maximum distance among all previously passing test and the average distance of all previously failing tests, with the intuition that large changes even to a single passing test are undesirable, whereas larger changes to failing tests are sometimes expected. We rank all patches by distance, reporting those patches with the smallest distance between patched and unpatched programs first. A detailed description of the distance computations can be found in the supplementary material.

6 EVALUATION

We evaluate our approach along multiple dimensions. First, central to any automatic program repair tool is its ability to identify patches, so we measure repair effectiveness in terms of the number and the quality of repaired bugs:

RQ1: How effective our approach is in finding patches within a large search space of fixing ingredients?

RQ2: To what extent do our generated patches overfit to the provided tests?

Our approach should benefit from increased edit expressiveness without suffering much in terms of search effectiveness and easily incorporate more fixing ingredients, hence:

RQ3: To what extent can our approach make use of different kinds of fixing ingredients?

Another key benefit of a systematic search in a rich space of fixing ingredients is the ability to find multi-edit patches. We expect to find more multi-edit patches than prior approaches:

RQ4: How effective is our work in generating multi-edit patches?

Finally, enumerating all plausible patches opens new opportunities for selecting high-quality patches from a pool of many plausible patches, hence:

RQ5: How effective is our patch ranking?

Where available, we compare our results with the state-of-the-art approaches by taking numbers verbatim from prior work, because reproducing previous results can be expensive and time-consuming [7]. Experimental conditions of prior work (e.g., fault localization and edit templates used) are different so numbers should be compared with care. We report numbers from existing work nonetheless to put our approach into context. Our experiment setup and evaluation data are publicly available in the supplementary material.

6.1 Experimental Setup

While the specific experimental design differs between research questions, all research questions are evaluated on the same set of 579 bugs and with the same generated meta-programs.

Bugs analyzed. To enable comparison with other approaches, we build our evaluation on 579 bugs in eight subject systems from two commonly used benchmarks, characterized in Table 1. Each bug consists of an implementation and a test suite in which at least one test fails. The two benchmarks have very different characteristics, allowing us to triangulate the evaluation results.

Table 1: Evaluation subjects

Subject	Description	LOC	Test LOC	#Bugs	#Tests
`median`	median of 3 integers	78	130	57	13
`smallest`	smallest of 4 integers	80	158	52	16
`digits`	digits of an integer	83	153	75	16
`grade`	compute letter grade	82	174	89	18
`checksum`	checksum of a string	89	172	11	16
`syllables`	count syllables	88	152	13	16
`Math`	math library	85k	19k	106	3,602
`Closure`	JavaScript compiler	90k	83k	176	7,927

LOC for IntroClassJava are counted using SLOCCount and averaged over all bugs. LOC for Defects4J are taken from the original work [12]. #Bugs denote the total number of bugs available in the datasets. #Tests denotes the number of test cases.

The first set of subject systems, drawn from the IntroClassJava benchmark [9], is derived from student solutions to assignments in an introductory programming course [38]. The small size of these programs minimizes the impact of fault localization and allows us to comprehensively evaluate our approach under different experimental conditions; the simplicity of these programs also allows us to often evaluate correctness of patches formally.

To evaluate our approach at a more realistic scale, we also analyze 282 bugs from the Math and Closure open-source projects in the Defects4J benchmark [12], broadly used in repair research to evaluate effectiveness and scalability on real bugs [4, 11, 35, 48]. We limit our evaluation to the two largest subject systems in the dataset, to balance between ability to compare to other reported results and required engineering effort (wrt. environment barrier, see Section 4.3) and evaluation cost.

Meta-Program Generation. To reduce nondeterminism and computational effort, we generate one meta-program for each buggy program and use it consistently in all experiments. We use all eight edit templates (Section 3) and repeatedly apply edit templates at all locations identified by the fault localization technique. In programs with very large numbers of possible edits, we generate edits randomly until we reach 500 edits for IntroClassJava programs and 5000 for the larger Defects4J programs.[1] Generated edits are then compiled individually and only compilable ones are merged into the meta-program.

Threats to validity. We encountered technical issues with 98 out of 282 Defects4J bugs, mostly due to unsupported API calls—a challenge shared by semantics-based tools and research prototypes of analysis tools in general, that can be addressed with more engineering effort (cf. Sec. 4.3). Although VarFix did not produce any results for these bugs, we do not exclude these bugs when we report numbers from prior work. We conjecture that a more mature variational execution engine would allow us to repair more bugs.

Randomness might affect several components of our experiment, such as generating meta-programs, sampling edits for variational execution, and genetic search of GenProg. To limit the impact of randomness, we intentionally generate a large pool of applicable edits when we generate meta-programs. We also take different samples or seeds when applicable.

[1]We assign a 10 times lower chance to `UOI` and `ABS` edits because they can easily create lots of edits around constants and numeric variables that are often of little value.

As with all existing approaches, our approach is evaluated on a limited number of bugs and thus results must be generalized with care. We expect that overfitting to the benchmark [7] is less of an issue as we only use generic edit templates and focus on search effectiveness.

6.2 RQ1 (Effectiveness) & RQ2 (Patch Quality)

The first obvious questions are how many bugs we can fix and how good those patches are.

Experiment Setup: Finding Patches (RQ1). We run our approach on each subject's meta-program for up to 3 hours for IntroClassJava and 6 hours for Defects4J, in line with recent prior work [7, 11, 35]. With this time budget, variational execution would typically not be able to explore all combinations of edits, hence we carefully set a number of options to limit the search discussed in Section 4.3. We run multiple experiments with increasingly larger bounds until we reach the time limit. For IntroClassJava, we bound the search to combinations of at most 3 edits (cf. Sec. 4.3) for programs without loops; in programs with loops (which are often copied by edits), so we start with a bound of 2, and continue with 3 if there is time left. For Math and Closure, where loops and recursions are common, we start with bound 1 and all edits, then explore combinations of 500 randomly selected edits at a time with bound 2, and finally combinations of 300 edits with bound 3. We continue sampling in restricted edit spaces until the time limit is reached.

We experimentally compare our approach directly to GenProg's genetic-programming-based search strategy (which still provides a competitive baseline despite 10 years of program-repair research [4, 11, 15, 35, 45, 48, 51, 53]), executing tests against one patch at a time. For fairness, we exclude edit generation and compilation and run tests against the same meta-program. We configure GenProg to run up to 20 and 40 seeds for IntroClassJava and Defects4J subjects respectively with the same 3h and 6h time limits and record all patches found within the time limit rather than stopping after the first plausible patch.

To complement the effectiveness comparison, we additional compare *efficiency* by measuring the time it takes to generate the *first* plausible patch. We configured GenProg to search more efficiently by sampling 10% of the provided tests when validating patch candidates. Only when all the tests in the sample pass will GenProg continue validation with the rest of the tests.

We set up the experiments in Docker containers and ran all performance measurement on Amazon Fargate. Each Fargate instance has 2 vCPU and 16 GB of RAM. Altogether, the experiments for RQ1 took more than 5000 hours of CPU time.

For other repair approaches, we report previous findings from the literature gathered on the same subjects with similar setups [4, 11, 35, 45, 48]. While not directly comparable, this provides a reasonable context, without having to invest enormous additional computational resources.

Experiment Setup: Evaluating Patch Quality (RQ2). To measure patch quality, we aim to distinguish **plausible patches** that pass the test suite (i.e., all results found by repair approaches) from *correct* patches that actually meet the program specification (if available) and would pass *all possible* tests. Since correctness is often difficult

to establish, we adopt the common notion of *generalizable* patches that additionally pass a high-quality held-out test suite which was not available to the repair approach—a pragmatic approach to evaluate patch quality common in repair research [15, 28, 38]. We use a best effort approach to identify **high-quality patches** among all plausible patches, judging correctness where feasible and generalizability otherwise.

For IntroClassJava, the specifications are simple enough that we can verify patch *correctness* formally via symbolic execution for most of the bugs. We implemented a small symbolic-execution engine to verify equivalence between a patched program and a reference solution, considering only correct patches as high-quality patches. For the *digits* subject, the frequent use of loops prevents full verification, so we consider only symbolic inputs in the $[-100, 100]$ range and classify programs that are behaviorally equivalent to a reference solution for those inputs as high-quality patches. For three *syllables* and *checksum* bugs where advanced language features prevented verification, we do not attempt to identify high-quality patches.

For Defects4J, verification is infeasible, so we follow the standard practice to use an additional held-out test suite [15, 28, 38]. We reuse existing high-quality held-out tests that were previously generated for evaluating patch quality [28] and classify a patch as a *high-quality patch* if it passes all held-out tests. For 5 patched bugs, we were not able to acquire held-out tests, but instead manually examined each generated patch and classify it as *high-quality* only if it is syntactically or semantically close to the developer patch. We use the term semantically close to conservatively refer to cases where programs are behaviorally similar but not strictly semantically equivalent. One example is Math-50—while the developer patch removes an if statement, our patch replaces the same if statement with a code snippet that is unlikely to affect program state (see supplementary material for code examples).

Results. VarFix is effective at finding high-quality patches in all subject systems. For IntroClassJava, it significantly outperforms all other approaches on the number of bugs for which it finds plausible and high-quality patches, see Table 2 for details. VarFix did not attempt repairing 38 out of 297 bugs that do not pass any provided test, because fault localization (inherited from GenProg) fails. As expected, bugs repaired by VarFix are a strict superset of those repaired by GenProg in the same search space. Importantly, VarFix produces high-quality patches for 40 bugs that no other approach fixed.

For the much larger subjects in Defects4J, we observe similar patterns though fewer patched bugs overall, shown in Table 3. VarFix again strictly outperforms GenProg for searching in the same search space in terms of bugs fixed with plausible and high-quality patches; the effect is larger for *Closure* because the larger test suite slows down GenProg's search more. Other approaches often repair a similar number or slightly more bugs, likely because recent work uses more diverse or more tailored fixing ingredients. For example, Hercules can fix `Math-24` by inserting a method call that is absent from the buggy source file, a fixing ingredient not available in our evaluated implementation. In all cases where VarFix fails to produce any patches, it is because VarFix exceeds the time budget at higher search bounds (148 bugs) or because of technical limitations of the variational execution engine (98 bugs, cf. Sec. 4.3). Importantly, VarFix can uniquely fix 7 bugs with high-quality patches (4 for Math and 3 for Closure) not fixed by any other approaches. We observed that VarFix can repair bugs that existing approaches did not fix because our search space is more expressive (i.e., we do not restrict the types of edits) and our search is systematic with regard to the given search space (i.e., we do not use heuristics to traverse the space). Interestingly, manual verification of the 35 fixed bugs showed that VarFix found the actual developer

Table 2: Number of repaired bugs for IntroClass (high-quality/plausible)

Subject	VarFix	GenProg	CapGen	S3	Nopol	jMutRepair
`median`	23/32	13/18	8/-	-	16	7
`smallest`	15/38	2/18	11/-	22/-	12	9
`digits`	22/30	16/26	3/-	-	2	4
`grade`	4/4	4/4	3/-	-	2	4
`checksum`	0/2	0/1	0/-	-	0	0
`syllables`	0/1	0/1	0/-	-	0	0
Total	64/107	35/68	25/-	22/-	32	24

A hyphen (-) denotes missing data. For VarFix, GenProg, CapGen, and S3, each cell shows the number of bugs patched by **high-quality/plausible** patches. For Nopol and jMutRepair, each cell shows the number of bugs patched by **plausible patches** since patch quality is not evaluated [7].

Table 3: Number of repaired bugs for Defects4J (high-quality/plausible)

Subject	VarFix	GenProg	Hercules	SimFix	ssFix	CapGen
Math	11/24	7/16	20/29	14/26	10/26	13/-
Closure†	6/11	0/1	8/13	5/7	2/11	-/-
Total	17/35	7/17	28/42	19/33	12/37	13/-

Each cell shows the number of bugs patched by **high-quality/plausible** patches. Numbers for existing approaches are taken from the corresponding papers[4, 11, 35, 45, 48]. CapGen was not evaluated on Closure and the paper only reports high-quality patches.
†: bugs Closure-62 and Closure-63 are the same. We count only one of them and manually adjust numbers of other approaches for consistency.

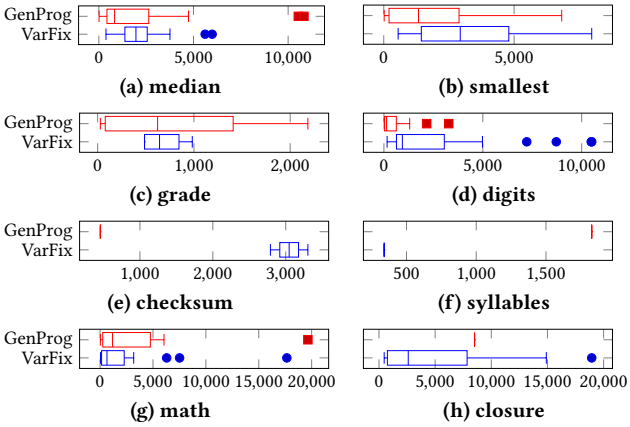

Figure 4: Efficiency comparison between VarFix and GenProg. In each box plot, we show the time taken to find the first patch for all the bugs of the given subject. The horizontal axis displays time in seconds.

patch among all the patches in all the 9 cases where the needed fixing ingredients were in the search space (in the other 26 cases, the exact ingredients were not in the search space because of inaccurate fault localization and limited edit templates).

In addition to patching more bugs than GenProg, VarFix also finds vastly more plausible and high-quality patches per bug, often more than 1000 patches per program and still dozens of patches after minimization, as summarized in Tables 4 and 5 (for details on individual bugs see supplementary material). This shows not only plausible, but also high-quality patches are abundant, both for tiny programs like the ones in IntroClassJava and larger ones in Defects4J (in contrast to prior observations that found many plausible but only few high-quality patches [19], possibly caused by a less efficient search strategy).

In terms of search efficiency, VarFix tends to take slightly more time than GenProg to find the first patch for IntroClassJava, as shown in Figure 4. However, VarFix can often find many more plausible patches at the same time as opposed to one at a time for GenProg. For Defects4J, VarFix can search more efficiently than GenProg because the overhead of variational execution is offset by larger search spaces and more expensive test executions when fixing bugs in large programs.

Overall, VarFix's more effective search strictly outperforms GenProg in the same search space; it significantly outperforms prior work on IntroClassJava and can uniquely fix 7 Defects4J bugs. It is effective not only at finding plausible patches, but also at identifying high-quality patches. Search efficiency of VarFix is on par with GenProg despite a more heavyweight search, the cost of which can be offset in large repair problems.

6.3 RQ3 (Fixing Ingredients)

To demonstrate VarFix's ability to incorporate and take advantage of different fixing ingredients, we explore it's ability to fix bugs with different ingredients.

Experiment Setup. We conduct two separate experiments: (1) For IntroClassJava bugs, we compare VarFix's ability to find patches with different fixing ingredients. Specifically, we compare the results from RQ1 which used *eight* edit templates, with a separate trial where only a subset of the edit templates was available, namely the *three* original edit templates of GenProg that append, replace, and delete statements. (2) For Defects4J bugs, we do not repeat the entire experiment due to high computational costs, but instead investigate whether failure to fix bugs can be explained by missing fixing ingredients. To this end, we randomly selected 10 multi-edit bugs that have not been fixed by any prior work (characteristics shown in Tab. 6) and manually enhanced the meta-programs with fixing ingredients of the developer's patch. We specifically select multi-edit bugs that need two or three edits to demonstrate the ability to fix even challenging bugs when the fixing ingredients are available. In both experiments, we run VarFix and GenProg otherwise with the same settings as in RQ1 (same timeouts, maximum edits considered, search bounds, etc).

Results. As expected, our results in Table 4 (columns $\text{Bug}_{pl/hq}$) confirm that VarFix can fix more bugs when more fixing ingredients are available, whereas GenProg benefits from additional fixing ingredients to a much lesser degree. For example, VarFix can fix 23 instead of just 7 bugs when given more ingredients, whereas GenProg improves only from 7 to 13; moreover, GenProg actually fixed fewer `smallest` bugs when more ingredients were available, indicating that search in a larger space is not necessarily more effective. If comparing the number of patches found, rather than the number of bugs fixed, we see more variance and fewer consistent trends. We conjecture that, using more edit templates can increase the likelihood of a bug being fixed, but can also decrease the number of patches if certain important fixing ingredients are squeezed out of the search space because of bounds on edit combinations and the maximum number of edits considered, where some edits may squeeze out others.

Our experiments with manually injected fixing ingredients in 10 challenging multi-edit Defects4J bugs show that VarFix can find the developer patch within the search space for all 10 bugs even when the needed edits span multiple methods and classes, whereas GenProg can only fix 5 of those bugs within the same time limit, as shown in Table 6. Interestingly, VarFix found patches that contain fewer edits than the developer patch for 4 bugs and patches that only use some of the developer ingredients for 2 bugs, suggesting that developer patches sometimes contain edits that are not necessary to pass all tests and that parts of developer patches can be sometimes swapped out for other ingredients, i.e., one does not need to have the exact ingredients in the search space.

In summary, we find that VarFix can effectively make use of extra fixing ingredients in the search space, whereas the traditional search strategy in GenProg more easily misses patches when searching in larger search spaces.

6.4 RQ4 (Multi-Edit)

To understand the importance of VarFix's ability to find multi-edit patches, we break down found patches by the number of edits.

Experiment Setup. For each bug that VarFix can fix with high-quality patches in RQ1, we recorded the minimum number of edits needed for each found high-quality patch. We also count the total number of patches grouped by their number of edits after filtering down to minimized patches (cf. Sec. 5).

Results. While many bugs can be fixed with a single edit, the number of bugs that can be fixed with *high-quality* patches increases significantly if we search for combinations of up to three edits, from 28 to 64 in IntroClassJava and from 13 to 17 in Defects4J, as shown in Tables 5 and 7. Many of the IntroClassJava bugs that *only* VarFix can fix require multiple edits, which explains why VarFix can fix significantly more bugs than existing approaches (cf. Tab. 2). For many IntroClassJava bugs, we actually find a much larger number of (minimized) multi-edit patches than single-edit patches. Code examples of generated multi-edit patches are available in the supplementary material.

Overall, we find that VarFix's search is effective at finding patches with multiple edits and that this ability helps to fix more bugs.

6.5 RQ5 (Patch Ranking)

Finally, we explore VarFix's ability to filter and rank patches to suggest high-quality patches in a large pool of plausible patches.

Table 4: Number of patches generated for IntroClass using eight edit templates versus three edit templates (eight templates / three templates)

	VarFix						GenProg					
Subject	Bug_{pl}	Bug_{hq}	$\overline{Pl}$	$\overline{Mini\text{-}Pl}$	$\overline{HQ}$	$\overline{Mini\text{-}HQ}$	Bug_{pl}	Bug_{hq}	$\overline{Pl}$	$\overline{Mini\text{-}Pl}$	$\overline{HQ}$	$\overline{Mini\text{-}HQ}$
median	32 / 18	23 / 7	3261 / 1944	42 / 34	3222 /4377	23 / 28	18 / 14	13 / 7	77 / 158	4 / 4	79 / 274	3 / 5
smallest	38 / 30	15 / 4	2423 / 3908	102 / 310	219 / 1908	7 / 44	18 / 20	2 / 3	42 / 74	3 / 3	3 / 23	1 / 1
digits	30 / 12	22 / 8	280 / 135	9 / 10	238 / 164	6 / 9	26 / 12	16 / 7	229 / 255	4 / 6	140 / 144	3 / 3
grade	4 / 0	4 / 0	102 / 0	4 / 0	99 / 0	4 / 0	4 / 0	4 / 0	44 / 0	2 / 0	41 / 0	2 / 0
checksum	2 / 2	- / -	4 / 4	4 / 4	- / -	- / -	1 / 1	- / -	22 / 64	2 / 2	- / -	- / -
syllables	1 / 1	- / -	105 / 12	7 / 12	- / -	- / -	1 / 1	- / -	18 / 93	2 / 4	- / -	- / -

Each cell contains two numbers: the left number uses all eight edit templates and the right number uses only three edit templates. Numbers on the right will be discussed in RQ3 (Fixing Ingredients). **Bug_{pl}** reports the number of bugs fixed by *plausible* patches. **Bug_{hq}** reports the number of bugs fixed by *high-quality* patches. **$\overline{Pl}$** reports the average number of *plausible* patches. **$\overline{Mini\text{-}Pl}$** reports the average number of *minimized plausible* patches. **$\overline{HQ}$** reports the average number of *high-quality* patches. **$\overline{Mini\text{-}HQ}$** reports the average number of *minimized high-quality* patches. A hyphen (-) indicates we cannot use our symbolic execution engine to evaluate patch quality.

Table 5: Number of patches generated for Math and Closure

		Plausible Patches		High-Quality Patches		
Bug	Dev	GenProg	VarFix	GenProg	VarFix	#Edits
Math-5	✓	1	1	1	1	1/0/0
Math-8		3	14	0	0	
Math-22	✓	0	2	0	1	0/1/0
Math-24		0	1	0	0	
Math-28		32	69	0	0	
Math-29		1	4	0	0	
Math-35		2	4	1	2	0/1/1
Math-40		1	9	0	0	
Math-49		5	6	0	0	
Math-50	✗	11	30	7	16	16/0/0
Math-53	✓	2	2	2	2	2/0/0
Math-56		0	3	-	-	
Math-62		0	3	0	3	0/3/0
Math-65		1	1	-	-	
Math-70	✓	3	3	2	2	2/0/0
Math-73		0	2	0	0	
Math-80		0	6	0	0	
Math-81		0	19	0	1	1/0/0
Math-82	✓	1	6	1	6	4/2/0
Math-84		4	5	0	0	
Math-85	✓	11	28	2	4	2/2/0
Math-88		0	1	0	1	0/0/1
Math-95		10	13	0	0	
Math-96		1	1	-	-	
Closure-11	✗	0	1	0	1	1/0/0
Closure-13	✗	0	28	0	27	27/0/0
Closure-19		0	5	0	0	
Closure-21		0	78	0	0	
Closure-22		0	97	0	0	
Closure-62*	✓	0	2	0	2	2/0/0
Closure-63*	✓	0	2	0	2	2/0/0
Closure-66		0	8	0	0	
Closure-73	✓	0	1	0	1	1/0/0
Closure-86	✓	0	1	0	1	1/0/0
Closure-126	✗	4	9	0	0	
Closure-161	✗	0	1	0	1	1/0/0

Dev denotes developer patch. ✓ denotes that VarFix found the developer patch. ✗ means the developer patch can, in theory, be generated by our edit templates, but the exact fixing ingredients are missing in the meta-programs due to inaccurate fault localization. An empty cell means the developer patch can not be generated by our edit templates.
#Edits column breaks down VarFix's high-quality patches (second to last column) by the number of composing edits, from 1 edit to 3 edits.
Closure-62 and Closure-63 are duplicate. We show them in this table for consistency with prior work, but only count one of them when we report repaired bug counts [35].

Experiment Setup. We experiment with IntroClassJava patches found in RQ1, skipping Defects4J patches, because in most cases there are too few patch candidates to make ranking interesting or necessary.

Table 6: Analyzed bugs with manually encoded fixing ingredients

Bug	Edits	Methods	Classes	VarFix	GenProg
Math-1	2	2	2	✓	✓
Math-26	2	1	1	✓	✓
Math-52	3	1	1	✓	✓
Math-83	3	2	1	✓	✗
Math-86	2	1	1	✓	✓
Closure-7	2	1	1	✓	✗
Closure-9	2	2	2	✓	✓
Closure-27	3	2	1	✓	✗
Closure-72	2	2	2	✓	✗
Closure-75	2	2	1	✓	✗

Edits denotes the number edits in the developer patch, *Methods* the number of methods modified, and *Classes* the number of classes modified in the developer patch.

Table 7: VarFix patches for IntroClassJava by number of edits

		Patched By			Minim. High-Qual. Patches			
Subject	Bug_{hq}	1E	2E	3E	Total	1E	2E	3E
median	23	7	14	18	537	18	87	432
smallest	15	1	3	14	116	1	9	106
digits	22	16	22	2	135	29	89	17
grade	4	4	2	0	18	10	8	0

Bug_{hq} denotes the number of bugs that can be fixed by *high-quality* patches. 1E, 2E, 3E represents one-edit, two-edit, three-edit patch, respectively. We omit checksum and syllables as we cannot verify them with symbolic execution due to advanced language features used, as discussed in the experiment setup of RQ2.

For all IntroClassJava patches found in RQ1, we compare the number of patches before and after minimization. Subsequently, we rank the minimized patches to measure where *high-quality* patches are ranked. Among the 64 bugs with at least one high-quality patch, we analyze the 24 bugs that have at least one plausible but incorrect patch, because ranking a pool of all high-quality patches is not interesting. We report the position of the highest-ranked *high-quality* patch. We report ranking results separately for the 5 ranking criteria introduced in Section 5.

Results. As expected the number of minimized patches is *much* smaller than the number of all patches found, often by two orders of magnitude, as shown in Table 4. Interestingly, patch minimization reduced patch quality for a few patches, where the test suite still passes after removing an edit from a high-quality patch, but that

Table 8: Patch ranking results (excerpt)

Bug	Mini. Pl. Patches	Mini. HQ. Patches	AST	Leven.	State	CF	Line
m-1bf7-03	111	3	48	54	3	8	2
m-317a-02	154	150	1	1	1	1	1
m-6e46-03	52	4	27	21	3	6	2
s-3b23-08	92	2	4	4	17	35	9
...(20 more in the supplementary material)							
Top 1			4	4	6	5	6
Top 5			10	10	14	7	16
Top 10			13	13	16	14	23

Numbers in the last 5 columns denote the rank of the first correct patch in the ranking: *AST* denotes ranking based on edit distance, *Leven.* based on Levenshtein distance, *State* based on state differences in assignments, *CF* based on branch differences in control-flow statements, and *Line* based on differences in executed lines.

reduced patch no longer generalizes. That is, an edit is needed to repair the bug that is indistinguishable from the test suite's perspective. Given that these cases are very rare, we expect that the benefits of having orders of magnitude fewer patch candidates outweigh the rare drawback of losing a correct patch through minimization.

While none of our ranking heuristics can reliably rank high-quality patches always in the first position, some heuristics can rank high-quality patches in the top 10 patches for most bugs and even within the top 5 patches for many. Ranking based on comparing executed statements, similar to prior work [49], seems to be the most promising ranking strategy. While more research on patch ranking is needed (potentially exploring other measures and combinations of measures), we argue that having to inspect 5 to 10 patches among hundreds (or thousands if not minimized) is a promising starting point for a practical setting.

In summary, filtering by minimization reduces the number of plausible patches by orders of magnitude without much loss in quality and patch ranking can usually report correct patches among the 10 highest-ranked results.

7 RELATED WORK

We already discussed general trends and tradeoffs in program repair in Section 2, focusing on pros and cons of different *search strategies*. A recent comprehensive survey [27] provides an excellent overview of the field. Here, we focus on details related to technical considerations of our approach.

Patch quality. The tendency of automated repair approaches to overfit patches to the test suite has received extensive attention [15, 27, 38]. There is little agreement on how to best measure patch quality or correctness. Commonly patches are compared against developer patches [11, 18, 25, 35, 45], but our results show that there are often many different patches for a bug that are all semantically equivalent but often syntactically different from the developer patch. Another common strategy is to evaluate patches with held-out tests [15, 28, 38], which we adopt in this work for patches we cannot formally verify against a reference implementation.

Multi-edit patches. Empirical studies of developer patches have shown that multi-edit patches are common [12, 54]. As discussed, while most repair approaches can theoretically find multi-edit patches, they often struggle to find them in large search spaces. We are aware of only two approaches that specifically target this problem: Angelix can target multiple suspicious expressions at the same time by using symbolic execution to capture inter-location dependencies as constraints, but often require multiple suspicious locations to be close for symbolic execution to be effective [25]. Hercules exploits the fact that similar code changes are often made to similar locations (termed sibling relationship) to pro-actively derive similar edits at multiple locations at the same time [35]. In contrast, when variational execution succeeds, our approach can efficiently search for multi-edit changes even if the locations are scattered and ingredients are different.

Patch ranking. Many existing program repair approaches guide the search to favor certain kinds of patches, prioritizing those that are more likely to pass all tests, or have higher quality [5, 15, 17, 18, 20, 24, 25, 44, 45]. To rank *patch candidates*, existing approaches exploit different information, such as the number of passing and failing tests [17, 44], syntactic and semantic distance to the original program [5, 15, 24, 25], and probabilistic models learnt from existing human patches [16, 20, 45, 50]. In contrast, we rank *plausible patches* after the search instead of ranking *patch candidates* during the search. We additionally use runtime information from test executions for ranking and show that it outperforms edit-distance based measures.

Closest to our ranking approach, JAID also finds multiple plausible patches and rank them, but based on fault localization suspiciousness [4]. Our ranking strategy is inspired by Xiong et al. [49] who attempt to *classify* whether found patches are correct by comparing their execution traces.

8 CONCLUSIONS

Existing approaches to automatic program repair essentially solve a search problem in which a trade-off is made between edit expressiveness and search effectiveness. While most existing work prioritizes one or the other, our work uses variational execution to effectively navigate a large search space of diverse fixing ingredients. We evaluate our work on IntroClassJava and Defects4J, showing that systematically exploring the search space of program repair has numerous benefits comparing to prior work, including effectively leveraging fixing ingredients to fix more bugs, systematically finding more high-quality patches and multi-edit patches, and comprehensively revealing dynamic information that is useful for distinguishing high-quality patches from a larger pool of plausible patches. We hope that our work can inspire future research on effective search strategies, patch ranking, patch quality and multi-edit patch generation.

ACKNOWLEDGMENTS

This work has been supported by NSF awards 1552944 and 1750116. Priscila participated in this work via the NSF-funded Research Experiences for Undergraduates in Software Engineering (REU-SE) program. We are grateful to all who provided feedback on this work, including Abhik Roychoudhury, Heather Miller, the anonymous reviewers, and the audiences of early presentations of this work.

SUPPLEMENTARY MATERIAL

Supplementary material of this work is available online at https://chupanw.github.io/varfix-supplement/.

REFERENCES

[1] Thomas H. Austin and Cormac Flanagan. 2012. Multiple Facets for Dynamic Information Flow. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, NY, USA, 165–178. https://doi.org/10.1145/2103656.2103677

[2] Thomas H. Austin, Jean Yang, Cormac Flanagan, and Armando Solar-Lezama. 2013. Faceted Execution of Policy-Agnostic Programs. In *Proceedings of the Eighth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS '13)*. ACM, New York, NY, USA, 15–26. https://doi.org/10.1145/2465106.2465121

[3] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (Oct. 2019), 159:1–159:27. https://doi.org/10.1145/3360585

[4] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-Based Program Repair without the Contracts. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Urbana-Champaign, IL, USA, 637–647.

[5] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program Repair with Quantitative Objectives. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Vol. 9780. Springer International Publishing, Cham, 383–401. https://doi.org/10.1007/978-3-319-41540-6_21

[6] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. 2016. Featured Model-Based Mutation Analysis. In *Proceedings of the 38th International Conference on Software Engineering (ICSE '16)*. ACM, New York, NY, USA, 655–666. https://doi.org/10.1145/2884781.2884821

[7] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. 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 - ESEC/FSE 2019*. ACM Press, Tallinn, Estonia, 302–313. https://doi.org/10.1145/3338906.3338911

[8] Thomas Durieux and Martin Monperrus. 2016. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. In *Proceedings of the 11th International Workshop on Automation of Software Test - AST '16*. ACM Press, Austin, Texas, 85–91. https://doi.org/10.1145/2896921.2896931

[9] Thomas Durieux and Martin Monperrus. 2016. *IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs*. Technical Report. University of Lille.

[10] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-Grained and Accurate Source Code Differencing. In *Proceedings of the International Conference on Automated Software Engineering*. Association for Computing Machinery, Västeras, Sweden, 313–324. https://doi.org/10.1145/2642937.2642982

[11] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 298–309. https://doi.org/10.1145/3213846.3213871

[12] René Just, Darioush Jalali, and Michael D. Ernst. 2014. 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)*. ACM, New York, NY, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[13] Christian Kern and Javier Esparza. 2010. Automatic Error Correction of Java Programs. In *Proceedings of the 15th International Conference on Formal Methods for Industrial Critical Systems (FMICS'10)*. Springer-Verlag, Berlin, Heidelberg, 67–81.

[14] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-Written Patches. In *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, San Francisco, CA, USA, 802–811. https://doi.org/10.1109/ICSE.2013.6606626

[15] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and Semantic-Guided Repair Synthesis via Programming by Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*. ACM Press, Paderborn, Germany, 593–604. https://doi.org/10.1145/3106237.3106309

[16] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, Suita, 213–224. https://doi.org/10.1109/SANER.2016.76

[17] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (Jan. 2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[18] Fan Long and Martin Rinard. 2015. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. ACM Press, Bergamo, Italy, 166–178. https://doi.org/10.1145/2786805.2786811

[19] Fan Long and Martin Rinard. 2016. An Analysis of the Search Spaces for Generate and Validate Patch Generation Systems. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. ACM Press, Austin, Texas, 702–713. https://doi.org/10.1145/2884781.2884872

[20] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages - POPL 2016*. ACM Press, St. Petersburg, FL, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[21] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott. 2019. SapFix: Automated End-to-End Repair at Scale. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP '19)*. IEEE Press, Montreal, Quebec, Canada, 269–278. https://doi.org/10.1109/ICSE-SEIP.2019.00039

[22] Matias Martinez and Martin Monperrus. 2019. Astor: Exploring the Design Space of Generate-and-Validate Program Repair beyond GenProg. *Journal of Systems and Software* 151 (May 2019), 65–80. https://doi.org/10.1016/j.jss.2019.01.069 arXiv:1802.03365

[23] Sergey Mechtaev, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-Equivalence Analysis for Automatic Patch Generation. *ACM Trans. Softw. Eng. Methodol.* 27, 4, Article 15 (Oct. 2018), 37 pages. https://doi.org/10.1145/3241980

[24] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE Press, Florence, Italy, 448–458.

[25] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. ACM Press, Austin, Texas, 691–701. https://doi.org/10.1145/2884781.2884807

[26] Jens Meinicke, Chu-Pan Wong, Christian Kästner, Thomas Thüm, and Gunter Saake. 2016. On Essential Configuration Complexity: Measuring Interactions in Highly-Configurable Systems. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. ACM, New York, NY, USA, 483–494. https://doi.org/10.1145/2970276.2970322

[27] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *Comput. Surveys* 51, 1 (Jan. 2018), 1–24. https://doi.org/10.1145/3105906

[28] Manish Motwani, Mauricio Soto, Yuriy Brun, Rene Just, and Claire Le Goues. 2020. Quality of Automated Program Repair on Real-World Defects. *IEEE Transactions on Software Engineering* (2020), 1–1. https://doi.org/10.1109/TSE.2020.2998785

[29] Hung Viet Nguyen, Christian Kästner, and Tien N. Nguyen. 2014. Exploring Variability-Aware Execution for Testing Plugin-Based Web Applications. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 907–918. https://doi.org/10.1145/2568225.2568300

[30] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. 1996. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering and Methodology* 5, 2 (April 1996), 99–118. https://doi.org/10.1145/227607.227610

[31] A. Jefferson Offutt, Gregg Rothermel, and Christian Zapf. 1993. An Experimental Evaluation of Selective Mutation. In *Proceedings of 1993 15th International Conference on Software Engineering*. IEEE, Baltimore, MD, USA, 100–107. https://doi.org/10.1109/ICSE.1993.346062

[32] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 254–265. https://doi.org/10.1145/2568225.2568254

[33] RTI. 2002. *The Economic Impacts of Inadequate Infrastructure for Software Testing*. Technical Report. National Institute of Standards and Technology.

[34] Stuart Russell and Peter Norvig. 2002. *Artificial Intelligence: A Modern Approach*.

[35] Seemanta Saha, Ripon k. Saha, and Mukul r. Prasad. 2019. Harnessing Evolution for Multi-Hunk Program Repair. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 13–24. https://doi.org/10.1109/ICSE.2019.00020

[36] Koushik Sen, George Necula, Liang Gong, and Wontae Choi. 2015. MultiSE: Multi-Path Symbolic Execution Using Value Summaries. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*. ACM Press, Bergamo, Italy, 842–853. https://doi.org/10.1145/2786805.2786830

[37] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. 2013. Automated Feedback Generation for Introductory Programming Assignments. *ACM SIGPLAN Notices* 48, 6 (June 2013), 15–26. https://doi.org/10.1145/2499370.2462195

[38] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. 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 - ESEC/FSE 2015*. ACM Press, Bergamo, Italy, 532–543. https://doi.org/10.1145/2786805.2786825

[39] squaresLab. 2021. genprog4java. https://github.com/squaresLab/genprog4java.

[40] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-Patterns in Search-Based Program Repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*

(FSE 2016). Association for Computing Machinery, New York, NY, USA, 727–738. https://doi.org/10.1145/2950290.2950295

[41] Tricentis. 2018. Tricentis Software Fail Watch Finds 3.6 Billion People Affected and $1.7 Trillion Revenue Lost by Software Failures Last Year. Available at https://www.globenewswire.com/news-release/2018/01/24/1304535/0/en/Tricentis-Software-Fail-Watch-Finds-3-6-Billion-People-Affected-and-1-7-Trillion-Revenue-Lost-by-Software-Failures-Last-Year.html (2021/06/18).

[42] Alexander von Rhein, Sven Apel, and Franco Raimondi. 2011. Introducing Binary Decision Diagrams in the Explicit-State Verification of Java Code. In *Proc. Java Pathfinder Workshop*. 82.

[43] Westley Weimer, Zachary P. Fry, and Stephanie Forrest. 2013. Leveraging Program Equivalence for Adaptive Program Repair: Models and First Results. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*. IEEE Press, Silicon Valley, CA, USA, 356–366. https://doi.org/10.1109/ASE.2013.6693094

[44] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[45] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering - ICSE '18*. ACM Press, Gothenburg, Sweden, 1–11. https://doi.org/10.1145/3180155.3180233

[46] Chu-Pan Wong, Jens Meinicke, Leo Chen, João P. Diniz, Christian Kästner, and Eduardo Figueiredo. 2020. Efficiently Finding Higher-Order Mutants. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, Virtual Event USA, 1165–1177. https://doi.org/10.1145/3368089.3409713

[47] Chu-Pan Wong, Jens Meinicke, Lukas Lazarek, and Christian Kästner. 2018. Faster Variational Execution with Transparent Bytecode Transformation. *Proc. ACM Program. Lang.* 2, OOPSLA (Oct. 2018), 117:1–117:30. https://doi.org/10.1145/3276487

[48] Qi Xin and Steven P. Reiss. 2017. Leveraging Syntax-Related Code for Automated Program Repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Urbana-Champaign, IL, USA, 660–670.

[49] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying Patch Correctness in Test-Based Program Repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg Sweden, 789–799. https://doi.org/10.1145/3180155.3180182

[50] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th International Conference on Software Engineering (ICSE '17)*. IEEE Press, Buenos Aires, Argentina, 416–426. https://doi.org/10.1109/ICSE.2017.45

[51] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* 43, 1 (Jan. 2017), 34–55. https://doi.org/10.1109/TSE.2016.2560811

[52] Jean Yang, Travis Hance, Thomas H. Austin, Armando Solar-Lezama, Cormac Flanagan, and Stephen Chong. 2016. Precise, Dynamic Information Flow for Database-Backed Applications. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 631–647. https://doi.org/10.1145/2908080.2908098

[53] Yuan Yuan and Wolfgang Banzhaf. 2020. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. *IEEE Transactions on Software Engineering* 46, 10 (Oct. 2020), 1040–1067. https://doi.org/10.1109/TSE.2018.2874648

[54] Hao Zhong and Zhendong Su. 2015. An Empirical Study on Real Bug Fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE '15)*. IEEE, Piscataway, NJ, USA, 913–923. https://doi.org/10.1109/ICSE.2015.101