EMSE manuscript No.
(will be inserted by the editor)

Program Transformation Landscapes for Automated Program Modification Using Gin

Justyna Petke · Brad Alexander · Earl T. Barr · Alexander E.I. Brownlee · Markus Wagner · David R. White

Received: date / Accepted: date

Abstract Automated program modification underlies two successful research areas — genetic improvement and program repair. Under the generate-and-validate strategy, automated program modification transforms a program, then validates the result against a test suite. Much work has focused on the search space of application of single fine-grained operators — COPY, DELETE, REPLACE, and SWAP at both line and statement granularity. This work explores the limits of this strategy. We scale up existing findings an order of magnitude from small corpora to 10 real-world Java programs comprising up to 500k LoC.

We decisively show that the grammar-specificity of statement granular edits pays off: its pass rate triples that of line edits and uses 10% less computational resources. We confirm previous findings that DELETE is the most effective operator for creating test-suite equivalent program variants. We go farther than prior work by exploring the limits of DELETE's effectiveness by exhaustively applying it. We show this strategy is too costly in practice to be used to search for improved software variants.

We further find that pass rates drop from 12–34% for single statement edits to 2–6% for 5-edit sequences, which implies that further progress will need human-inspired operators that target specific faults or improvements.

A program is *amenable to automated modification* to the extent to which automatically editing it is likely to produce test-suite passing variants. We are the first to systematically search for a code measure that correlates with a program's amenability to automated modification. We found no strong correlations, leaving the question open.

Keywords Automated Program Modification · Genetic Improvement · Automated Program Repair · Search-Based Software Engineering

J. Petke[1], B. Alexander[2], E.T. Barr[1], A.E.I. Brownlee[3], M. Wagner[4], D.R. White[5]
[1]University College London, UK, [2]University of Adelaide, Australia, [3]University of Stirling, UK, [4]Monash University, Australia, [5]University of Sheffield, UK
E-mail: j.petke@ucl.ac.uk, bradley.alexander@adelaide.edu.au, e.barr@ucl.ac.uk, alexander.brownlee@stir.ac.uk, markus.wagner@monash.edu

1 Introduction

Automating routine program improvements and fixes promises to free software developers to tackle more challenging tasks. With progress in automated program repair (APR) and genetic improvement (GI), this future might be closer than previously thought. APR's focus is on automatically fixing software bugs (Gazzola et al. (2019)), while GI uses automated search to find improved software variants (Petke et al. (2018)). Both fields have vibrant research communities and enjoy impact both in academia beyond computer science and in industry. For example, Langdon et al. (2015) and Langdon and Lam (2017) used GI to improve runtime of large open-source bio-informatics software, with patches being incorporated into development; while Haraldsson et al. (2017); Marginean et al. (2019) and Kirbas et al. (2021) incorporated APR techniques into their companies' software development processes.

Both APR and GI share a key commonality: they automatically create program variants from existing software. We refer to this shared feature as *automated program modification* (APM). Software variants created via APM are represented as a sequence of software edits applied to an initial program, *i.e.*, patches to existing code.

Notwithstanding APM's successes, a key challenge remains: how to effectively and efficiently navigate the APM search space (Petke et al. (2019)) and scale up the size of individual edit sequences to match the size of patches routinely applied by software developers, *e.g.*, to fix bugs (Zhong and Su (2015)).

A seminal goal of APM is to find operators and search strategies that bring multi-edit improvements into reach. Further progress hinges on a better understanding of the landscape of APM program transformation spaces. Landscape studies (Reeves (1999)) seek to characterise a search space, in its entirety, unfiltered. The goal is to look for topological features that may be exploitable to efficiently and effectively navigate the search space. In our case, this means that we consider all patches to a working program, including the majority, which break the program. Our experimental procedure covers an unprecedented scale for an APM landscape study.

Current practice has focused on the DELETE, SWAP, COPY, and REPLACE operators (CDRS) at line and statement granularity, because these operators at these granularities are *universal*: they can be combined to produce any valid program, including desirable improved variants. This universality comes at a cost, however, as they produce a huge number of variants, and that number grows exponentially in the number of times the operators are applied. Previous studies (Harrand et al. (2019)) have thus focused on single edits to use testing resources efficiently under the assumption that the results from considering a single edit would generalise.

We drop this assumption and advance the state of the art by systematically studying multi-edit patches up to five edits. We deploy two tactics to scale to five edits. First, we employ the power of uniform sampling. Second, we restrict edits to *hot* methods, those which profiling identifies as frequently used. We turn to the goal of APM to justify this bias. APM seeks to improve a program;

in general, improving a hot method will more substantially improve a program than improving a cold method. This justification rests on the fidelity of the probability distribution over a program's inputs used to identify hot methods. We approximate this distribution with a program's test suite. We acknowledge the attendant validity threat of making this assumption. We are not alone, however, in resorting to it: much of the related work resorts to it to make progress. Technically, we leverage the genetic improvement tool, Gin[1], to focus on hot methods.

APM research has focused on finding successful variants, not the cost of doing so. To achieve industrial uptake, APM should now more carefully model and report its costs. With this in mind, we formalise the cost of APM into its three stages — mutating, compiling, and testing. Of these, mutation requires choosing an operation, finding where to apply it, and then applying it. Surprisingly, mutation's three-part cost has often been neglected. We hope that our formalisation encourages standardisation in how APM researchers report the costs of their techniques.

As usual, we require a test suite and quantify the *effectiveness* of a patch as the proportion of tests the program variant produced by its application passes. We systematically study the APM landscape, asking five research questions: (1) What is the relative effectiveness of the conventional edit operators: COPY, DELETE, REPLACE and SWAP? (2) How effective is DELETE when used alone? (3) Which is more effective: line or statement granular CDRS edits? (4) How much does effectiveness drop with the number of edits in a patch? and (5) What is the correlation between subject's features and its *plasticity* — the likelihood that applying APM to it will be effective?

For the first question, we find that the median single edit pass rates for statement deletions are highest, at 30.2%, while statement SWAPs are second highest, with 23.6% pass rates (Section 5.2). This finding ranks DELETE first, in line with previous findings Le Goues et al. (2012); Harrand et al. (2019). The effectiveness of DELETE could be due to test suite inadequacy, in which case, if its application is not too expensive, DELETE might serve as a measure of (in)adequacy. Prior work has also suggested that DELETE produces variants that are themselves more amenable to APM, *i.e.* it increases plasticity Harrand et al. (2019)[2]. In either case, we wondered if DELETE's search space contained rare, jagged regions. With these ideas in mind, we asked the second question above: "How effective is DELETE when used alone?". To answer it, we are the first to exhaustively explore DELETE's space of single applications on *hot* methods (those which profiling identifies as frequently used) at both statement and line granularity. For 10 projects, we found that DELETE's search space is smooth. Further, single statement deletion pass rates reached 82%. More study is needed, but this finding suggests that uniformly sampling

[1] https://github.com/gintool/gin

[2] Harrand et al. (2019) define plasticity as the "intrinsic capability at being changed to another code, while keeping functional correctness, with respect to a given test suite"

DELETE applications may effectively measure test suite adequacy or increase plasticity (Section 5.3).

A seminal question in APM is to find, for an improvement goal, an edit granularity that produces the richest search space. To reduce the cost of finding edit locations, most research has centered on two granularities — line and statement. Among these two, we definitively answer the third question. Line edits produce $\frac{1}{3}$ test passing variants of statement edits, while costing twice as much computational resources on average (Section 5.4). Therefore, statement granular edits should be preferred.

Most developer patches make multiple edits (Zhong and Su (2015); Callan et al. (2022)). Although genetic improvement has successfully improved software using CDRS, it typically uses genetic programming and/or local search Blot and Petke (2021) to navigate the space of CDRS edits. The fourth question essentially asks how sparse are effective modifications in a multi-edit space? Using systematic random sampling, we conclude that compilation rates alone reduce by roughly 50% with each additional line edit and by 25% with each additional statement edit (Section 5.5). This result implies that clever search heuristics are needed to efficiently and effectively navigate the multi-edit search space. Based on this finding, we argue that, to better approximate human patches, we either need to devise equivalence classes over variants to reduce the search space or to sacrifice universal operators that can produce any patch for less widely applicable operators that make more extensive changes. The resulting landscapes will necessarily be less general, built for particular problems or improvements, similar to defect classes Monperrus (2014).

Finally, we want to know which programs are amenable to APM. This would give developers guidance about where CDRS-based APM would be most effective or how to adapt a method to facilitate APM. Unfortunately, none of the code measures we tested show a strong correlation with test pass rate. Only cyclomatic complexity and normalised def-use show any correlation with pass rate, and it is weak. Our study, therefore, leaves open the question of where to best apply APM. To facilitate future work, we provide a list of methods that we found to be particularly amenable to APM (and those that are not) for future research (Section 5.6).

To summarise, our key contributions are:

- We formalise the cost of automated program modification (Section 2.2);
- We show that exhaustively applying DELETE generates a smooth search space, suggesting that uniformly sampling DELETE applications may measure test suite adequacy and increase plasticity (Section 5.3).
- We provide conclusive evidence that statement granular edits are more effective than line (Section 5.4);
- To spur future work, we propose plasticity, the problem of identifying code amenable to APM, conduct preliminary experiments that show how hard it is, and provide two lists of methods — those particularly amenable and those particularly resistant to APM (Section 5.6).

All our scripts and data are available in the following repository: `https://github.com/automatedprogrammodification/automatedprogrammodification/`.

2 Bedrock Program Modification Landscapes

The success of automated program modification (APM) rests on finding ways to effectively search a space of useful program variants. For example, hill-climbing efficiently searches convex spaces. Navigating this space requires understanding its landscape (Reidys and Stadler (2002)), which can be defined as follows:

Definition 2.1 (Fitness Landscape (Ochoa et al. (2008))) A *landscape* is a 3-tuple (S, V, f) where

1. S is a set of potential solutions or search space,
2. $V : S \to 2^S$ assigns a set of neighbours to every solution to form a neighbourhood structure, and
3. $f : S \to \mathbb{R}$ is a fitness function.

This definition assigns a fitness value to each solution. Usually, the higher the fitness value, the closer the quality of that solution to optimal. For genetic improvement, solutions are programs and fitness is a program's value to the users (which could be expressed as the number of test cases passed, runtime performance, *etc.*). Applied to program modification, the challenge is to define the neighbourhood structure via single-step program transformations that *approximate* human modifications in order to devise a universal, domain-independent fitness function.

When modifying a program, the neighbourhood function V in Definition 2.1 is M, a set of mutation operators on a program, or program transformations[3]. An operator $m \in M$ has the form $l \to r$: it matches the pattern l in the source and replaces it with r. A pattern that l matches is a redex. We permit l to mix variables and terminal symbols; this permits a pattern to use terminals as anchors and its variables to span the symbols between two anchors. This permits a redex to span arbitrary subsequences of P. An operator's right hand side r can be shorter than its l and remove characters or longer and add characters. For us, two solutions (*a.k.a.* programs) share a neighbourhood (*i.e.*, an edge connects them) if a single application of an operator transforms one into the other.

M's operators can be universal or language-specific and, if language-specific, they can be only *syntax-aware* or, additionally, *semantics-aware*. A syntax-aware deletion operator can avoid deleting definitions or the header of a loop. A semantics-aware insertion operator can rename the variables in

[3] In Martinez and Monperrus (2015)'s terminology, M is a repair model and $m \in M$ is a repair action.

the text it inserts, either to correctly bind them to in-scope variables or to avoid name capture. For almost all programming languages, the number of well-formed programs is vast, but very few of them are valuable to users, *e.g.*, meet a specification. Thus, the challenge is to define M and an accompanying search procedure that efficiently finds valuable programs.

Universal operators require no parsing or analysis to find their redexes, and they can, in principle, produce any interesting improvement, because they define languages that contain all improved versions of P. Their drawback is that they generate vast, sparse search spaces that include many variants that fail to improve upon P, let alone those that do not even compile. Language-specific operators are restricted to the language they target and require parsing, for syntax-awareness, or static analysis, when semantics-aware. They usually produce variants that compile and they can greatly reduce search space, because of the constraints on their redexes. Semantics-aware operators cannot, however, realise all improvements. Due to computational complexity, they must target specific functional or non-functional improvements and, even for a particular improvement type, they can only fix some subset.

Given these trade-offs, most automated program improvement approaches, have targeted universal operators — character-granular to produce Σ^* or line-granular generating $L(\text{lines})$ — or weakly syntax-aware operators, notably statement-granular producing $L(\text{stmt})$ (Petke et al. (2018)). Two key reasons underlie this preference: 1) the search spaces of universal and weakly syntactic operators contain all functional and non-functional improvements and 2) semantics-aware operators must be tailored to specific improvements. Two subproblems lurk in the second reason: it is not clear which improvements to target and, having chosen specific improvements, it is often not clear how to define or tailor effective operators for them. In the limit, when the improvement is sufficiently unique, no effective operator can be extracted from the edits that realise the improvement. In short, research to date has preferred the challenge of trying to efficiently traverse a vast search space to the challenge of designing bespoke operators. Although some progress has been made in the field of automated program repair, where template-based approaches have been tried (*e.g.*, Liu et al. (2019); Martinez and Monperrus (2018)). Much less progress has been made in the field of non-functional genetic improvement (Petke (2017)).

2.1 Landscape Formation: The Operators Studied

This paper's focus is on the limits of universal or weakly syntactic operators, so we consider four: COPY, DELETE, REPLACE and SWAP, at line and statement granularity. To identify statements, we generate an AST and consider only statement nodes. For a program, let L be the set of its lines, S be the set of its statements and its source text be the sequence $p = \langle u_1, u_2, \ldots, u_n \rangle$, for

$u_i \in L \oplus u_i \in S$. For $i, j \in [1..n]$, we have

$$\begin{aligned}
\text{COPY}(p, i, j) &= \langle \ldots, u_i, u_j, u_{i+1}, \ldots, u_{n+1} \rangle \\
\text{DELETE}(p, i, j) &= \langle \ldots, u_{i-1}, u_{i+1}, \ldots, u_{n-1} \rangle \\
\text{REPLACE}(p, i, j) &= \langle \ldots, u_{i-1}, u_j, u_{i+1}, \ldots, u_n \rangle \\
\text{SWAP}(p, i, j) &= \langle \ldots, u_{i-1}, u_j, u_{i+1}, \ldots, u_{j-1}, u_i, u_{j+1}, \ldots, u_n \rangle, i < j.
\end{aligned}$$

where DELETE ignores its third parameter.

REPLACE replaces a line or a statement with another one, copied from somewhere else in the program. SWAP, as its name implies, swaps the locations of its operands: it can model micro refactorings and/or optimisations. While GenProg used SWAP, most recent work in non-functional optimisation has not. Indeed, Le Goues et al. (2012) observed that REPLACE is often preferred to SWAP, because SWAP has been found to be an order of magnitude less effective than other operators.

DELETE and INSERT are universal primitive operators, able to construct Σ^* for any finite alphabet Σ. Thus, we would like to include both. DELETE actually shrinks the variant search space, but raw, unconstrained INSERT grows it, making landscape analysis infeasible. Dropping INSERT altogether produces a stale, artificial landscape: INSERT is overwhelmingly the most common operator used to construct code.

To conduct a landscape analysis, such as ours, we therefore need to square the circle and find an INSERT operator that defines a feasibly sized yet still realistic search space. One solution is the bounded insert over the terminals of the grammar of the programming language of the subject programs, in our case Java. The trouble is the mismatch between these terminals and the length of tokens: a large enough bound to produce tokens, like keywords and identifiers, would be prohibitive. To solve the problem of a feasible insert that is realistic in the sense that it can, in some cases, match insertions developers make, we use COPY. This operator is a form of INSERT with its input domain restricted to inserting token sequences (in this study, either lines or statements) that already occur in the subject program. In using COPY, we are in good company; many papers in the APM space use it (Petke et al. (2018)) and it is an application of the plastic surgery hypothesis (Barr et al. (2014)).

2.2 Fitness for Automated Program Modification

Effective automated program modification requires knowing whether a change indeed improves a program, *i.e.*, whether it fixes a bug, or preserves functional semantics while improving a non-functional property (latency, memory consumption, energy efficiency, *etc.*). Definitively answering this question requires an oracle that checks the program's behaviour, pre and post modification, against its specification. Unfortunately, we usually lack the specification, so APM techniques often resort to the program's test suite, which underapproximates its specification (Petke et al. (2018)).

Variants that pass all of the tests in a test suite are equivalent under that test suite. The literature calls such variants *neutral* (Schulte et al. (2014)). These variants are candidates for non-functional improvements, unless one is willing to sacrifice functional correctness for improvements; they form a neutral landscape whose neighborhood may contain functional fixes not immediately reachable from the subject program (Schulte et al. (2014)). Thus, these test-passing variants are key to understanding the APM search space. Two measures of this set are Schulte et al. (2014)'s *software mutational robustness* and Harrand et al. (2019)'s *neutral variant rate*:

Definition 2.2 (Mutational Robustness, Neutral Variant Rate) Let $M(P) = V$ be the set of variants generated by applying sequences of mutations from M to the program P. Let $V_c \subseteq V$ denote all program variants that compile and $V_T \subseteq V_c$ denote all program variants that pass all the tests in P's test suite T. Then, software mutational robustness (SMR) and neutral variant rate (NVR) are calculated as follows:

$$SMR = \frac{|V_T|}{|V|} \qquad (2.1) \qquad\qquad NVR = \frac{|V_T|}{|V_c|} \qquad (2.2)$$

SMR shows how large the program search space is for APM under the test suite T. To test a program, we must successfully compile it: both compilation and testing can be expensive. NVR shows the proportion of the test passing variants over those that successfully compile; it is the proportion of variants on which we spent the resources to both compile and test them. It measures how well mere compilation indicates successfully passing T's tests.

For generate and validate approaches, the goal is to maximise $|V_T|$ while using as few resources as possible, and each such resource contributes an objective to our multi-objective optimisation problem. We will now formalise this. Without loss of generality, we assume a single resource that is additive in nature, such as time taken or energy consumed, and our objective is to minimise it. First, given a program P and a set of APM operators M, let $\delta = (m_1, \cdots, m_k)$ be a sequence of k operators from M. Then

$$v = \delta(P) = m_k(m_{k-1}(\cdots(m_1(P))),\ m_i \in M \in 2^O \qquad (2.3)$$

is one variant generated by applying a sequence δ of k operators to P. Next, given a set of variants V, which can be obtained by edits from M, and a test suite T, we define the total APM cost (Equation 2.4) as the sum of: (1) finding and applying those edits to produce the variants (Equation 2.5), (2) the compilation cost (Equation 2.6), (3) and the testing cost (Equation 2.7)

of the successfully compiled variants V_c:

$$\$(V) = \$_M(V) + \$_c(V) + \$_T(V_c) \tag{2.4}$$

$$\$_M(V) = \sum_{v \in V} \big(cost(find(\delta)) + cost(\delta(P)) \big) \text{ s.t. } \delta(P) = v \tag{2.5}$$

$$\$_c(V) = \sum_{v \in V} cost(compile(v)) \tag{2.6}$$

$$\$_T(V_c) = \sum_{v_c \in V_c} \sum_{t \in T} cost(test(v_c, t)) \tag{2.7}$$

where *cost* measures a property of interest (*e.g.*, runtime or energy), *find* denotes a search procedure, $compile(v)$ denotes compilation of the given variant v, and $test(v_c, t)$ denotes running test t on a successfully compiled variant v_c. $|V \setminus V_T|$ is the set of generated variants that do not pass the test suite, and hence its generation is wasted. Given these costs, the overarching problem that we are trying to solve in APM is inherently multi-objective:

$$\underset{M_i \in 2^O}{\operatorname{argmin}} \$(V) \tag{2.8}$$

$$\underset{M_j \in 2^O}{\operatorname{argmax}} |V_T| \tag{2.9}$$

$$\text{subject to } M_i = M_j.$$

Here, we aim to find the set of operators that minimise the overall cost (Equation (2.8)) while maximising the number of test-suite passing variants, V_T (Equation (2.9)). To demonstrate that Equation (2.4) is practical, we instantiate it later in Section 5.4 by empirically solving it when 2^O contains only two sets — one containing line and the other statement granular COPY, DELETE, REPLACE, and SWAP operators.

We highlight two points. First, the set of operators M is bounded only by human ingenuity, so researchers will need to propose different M to explore this problem's search space. Second, because the components in $\$(V)$ interact, we cannot decompose the problem. For example, cost-efficient variant generation (in Equation (2.5)) may be detrimental if few of its variants pass the tests (in Equation (2.7)). Hence, the problem needs a holistic approach.

Lastly, our definition in Equation (2.4) can be further adjusted according to the situation at hand. For example, firstly, if we move from sets to multi-sets in our definitions, then we can incorporate the cost incurred by finding both duplicates and equivalents, which can happen in the stochastic search for patches. Secondly, because many *costs* are stochastic, we could wrap the right-hand side of all defining equations with $\mathbb{E}$, the expectation operator. In our formulation above, we follow the common practice in noisy optimisation: this expectation optimisation is not spelled out, and the true mean is approximated through sampling.

3 Research Questions

In this section we present our research questions. Firstly, we consider the sparsity of bedrock landscapes induced by applications of our operators at two granularities. APM is expensive. To reduce that cost, we ask, in closing, whether we can identify lightweight program features that indicate how amenable a subject program is to APM.

To answer all of the research questions below, except RQ4, we produce a variant as follows: for each method in each program in our corpus, for each operator, we uniformly choose a location within the target method at the operator's granularity, then apply the operator once. We do this repeatedly to sample the variant space and explore the landscape. Under this construction, all variants are a single step from the starting program. In RQ4, we apply operators up to k times. Sections 4.2 and 4.4 detail these procedures.

Given the lack of specifications, comparing operators in terms of effectiveness is not easy (Section 2.2). Variants that improve non-functional properties or define a neutral landscape (Schulte et al. (2014)) that can serve as the staging ground for a repair (Renzullo et al. (2018)) are a subset of test-passing variants, so, to the extent to which the test suites adequately capture the subject programs' semantics, test-passing variants establish an upper bound on the number of improving program variants. Thus, we answer RQ1–4 in terms of the number of test-passing variants.

Operator Effectiveness We aim to systematically explore and characterise the APM search space. We would like to know: which edit operators are most likely to produce test-passing variants? Universal, unrestricted edit operators produce an infeasible search space, especially when those operators can lengthen the subject program. So we instead consider our COPY, DELETE, REPLACE, and SWAP operators (Section 2.1) and ask the more concrete question:

> **RQ1** How often do single applications of the COPY, DELETE, REPLACE, and SWAP (CDRS) operators produce test-passing variants?

The compilation and testing that testing-based APM requires is expensive. A strong separation in operator performance will point to operators to be dropped or replaced. While no consensus has emerged about the other three, DELETE has been repeatedly found to be the most effective operator at both bug fixing (Le Goues et al. (2012)) and generating test-passing program variants[4] (Harrand et al. (2019)), often leading to efficiency improvements. Our high-level finding confirms previous results: DELETE generates the most testing-passing variants (Section 5.2).

The high pass rate of DELETE's variants is puzzling, because human patches tend to add more than they remove (Zhong and Su 2015, Finding 9). Psychology research has shown that we humans are biased toward additive over subtractive solutions (Meyvis and Yoon (2021)). The reason for DELETE's pass

[4] In Harrand et al.'s nomenclature, test-passing variants are called "test-suite neutral", because test-based fitness does not differentiate them.

rate may be that there is more scope for subtractive solutions in code than we find intuitive; and it may be that software often contains over-engineered code or broken future-proofing against a future that will never come.

Alternately, DELETE's pass rate may just reflect the degree to which the test suite used during APM underapproximates the specification. When this underapproximation is intentional, DELETE could be an effective way to automatically specialise a program, reducing its CPU and memory consumption, for that portion of its behaviour that the test suite exercises. When this underapproximation is unintentional, DELETE's test-passing, but incorrect, variants may help localise and understand a defect. Indeed, Qi et al. (2015) stated

> "The Kali [Qi et al.'s deletion-focused APR tool] patches often pinpoint precisely the exact line or lines of code to change. And they almost always provide insight into the defective functionality, the cause of the defect, and how to correct the defect."

Ginelli et al. (2022) did not confirm this claim, but found that DELETE variants revealed problems with the test suite. This suggests that DELETE's test-passing variants can guide the search for new tests to add to improve the given test suite. Finally, DELETE's pass rate could be used as a new measure of test suite quality: if DELETE succeeds too often, the test suite is inadequate.

Exhaustive Deletion Since it shrinks the search space, DELETE is well-suited to landscape analysis. This fact and the pass rate of its variants raises the next research question, which we are the first to consider exhaustively:

> **RQ2**: What percentage of all variants produced by *exhaustively* applying DELETE to each line or statement in a method passes all tests?

If the cost of computing this percentage is affordable, it can be used to assess test suite quality. If this percentage is also substantial, then researchers may want to consider applying DELETE first to produce its set of test-passing variants, then using that set as the launchpad for subsequent modifications. Exploiting this set may increase both the effectiveness and efficiency of APM (Harrand et al. (2019)). For instance, Callan et al. (2022) have recently shown that the most frequently used strategy by Android developers for performance improvements is redundancy removal. Section 5.3 presents our answer to RQ2.

Operator Granularity The vast majority of APM work has improved programs at the granularity of either lines or statements. An et al. (2018) were the first to compare these granularities in the context of automated program repair. They found that statement-level changes tend to be more effective at bug-fixing, but less efficient in terms of overall runtime. Their study rests on the results of a single tool on a small Python benchmark.

Here, we extensively compare the two granularities over our CDRS landscape on a substantial corpus of large, popular Java programs (Section 4.1). A trickiness in comparing the two granularities is that they are not isomorphic. Section 4.5 discusses the specifics of how our experimental harness handles this

issue. Line granular operators generate a set of strings that subsume any programming language and can even improve non-code properties, like comments or configuration settings. Statement granular operators are language-specific, since they still require parsing. Because they are weakly syntactically aware, statement-granular modifications are more likely to compile than line-granular modifications. Because some statements contain others, modifying them affects more lines than line granular operators. To discover which is better, we ask:

RQ3: At which granularity — line or statement — do single applications of the CDRS operators produce more test-passing variants?

Improving APM's overall effectiveness depends on making it more efficient. Answering this research question will shed light on how best to improve its efficiency. Given that localising where to fix or improve code will always be imprecise, we deem it beneficial to continue to focus on generic operators, like line-granular ones, and look to either speed or constrain search or is it better to instead consider operators that can only make some improvements but create a smaller search space? Section 5.4 reports our findings.

Multiple Edits Most developer-authored patches contain many edits. APM must find a way to generate such patches if it is to substantially increase the productivity of industrial developers. Some edit sequences first break, then restore syntactic validity. How often does this occur? To find out just how sparse the multi-edit search space is, we ask:

RQ4: How often does a sequence of CDRS edits create a test-passing variant?

This question will have different answers, depending on the techniques used to choose the edit sequence. We expect this question to be seminal, to open a new line of research into techniques aimed at making multi-edit patches feasible for APM. In this setting, it stands to reason that grammar-aware operators will be worth their parsing cost and further separate themselves from universal operators. Indeed, improvement specific semantic operators may overcome their cost and limited applicability for generating multi-edit patches. It may also be worthwhile to consider operators that take the sequence so far, including errors, into account. However, if grammar-oblivious edits recover often enough, it may be more efficient to continue to use them over devising property-specific operators. Section 5.5 answers this question and presents our findings.

Suitability for Automated Program Modification Conducting the search to find test-passing variants is expensive: computing fitness involves running a program's test suite. We would use testing resources more effectively if we could efficiently identify those programs, or parts of programs, that are more likely to be suitable for APM. Therefore, we ask:

RQ5: Which features predict code's plasticity, its amenability to APM modulo a test suite?

To answer this question, there is a plethora of features to consider, ranging from syntactical through structural to application domains. We focused

on the following measures: cyclomatic complexity, the count of source statements (excluding comments), the number of possible execution paths (Nejmeh (1988)), the median and average number of lines between variable definitions and their subsequent use in the target methods. For a selection of methods, we correlate these measures and test pass rates to see if any of these measures indicate a program's suitability for APM. The consequences of any successful finding would be immense: it would lower one of the most significant barriers to APM's industrial adoption: its poor yield on the computation resources it requires. A negative result, on the other hand, would suggest that other or new measures are needed. Worst case, it would lend evidence to the hypothesis that suitablity to APM may fall into the AI-complete category of "I know it when I see it" Gewirtz (1996) properties. Section 5.6 details our findings.

4 Experimental Setup

To answer our research questions we sample parts of the patch space, as defined by the line and statement CDRS operator sets. We measure the impact of the code changes for a given patch by running unit tests that execute methods containing the patch, recording the effect on compilation and unit test behaviour. We ran two experiments: Random Sampling and Delete Enumeration. The random sampling experiment is designed to generate data to answer RQ1, RQ3, RQ4, by uniformly sampling the space of CDRS operators, with both line and statement granularity, with patches of 1–5 edits. The delete enumeration experiment is designed to generate data to answer RQ2, and provide additional data to answer RQ3. Post-hoc analysis of the results from the random sampling experiment was carried out to answer RQ5. Full details of both experiments follow over the remainder of this section.

For all our experiments we use Gin, a dedicated tool for experimentation with genetic improvement (Brownlee et al. (2019)). Gin is open-source, available on GitHub[5]. To the best of our knowledge Gin is the only GI tool that contains an in-built profiler that automatically determines which test cases cover which methods. Additionally, we wrote a PatchSampler class to calculate the single-edit search space for our 10 subject programs, and the PatchTester class that we use to calculate self-repairs via recoveries to previous software version (Section 5.5).

4.1 Corpus

To empirically evaluate the impact of different transformation operators, we apply them to a corpus of large, widely-used, and actively maintained open-source Java projects hosted on GitHub. Our core goal is APM landscape analysis. We want to know how amenable is a program to mutation under the

[5] Gin is available at `https://github.com/gintool/gin`. We used Gin's version at commit e897ad3487eaf21511e740a6828c6c20b168a278.

Table 4.1: CS_A: Curated projects selected for experimentation.

Project	Description	Licence	LoC
jcodec	audio and video codec	FreeBSD	136k
mybatis-3	SQL mapper framework	Apache 2.0	115k
spatial4j	geospatial library	Apache 2.0	14k

Table 4.2: CS_B: Projects obtained from systematic search.

Project	Description	Licence	LoC
arthas	diagnostic tool	GNU GPL 3.0	30k
disruptor	inter-thread messaging library	Apache 2.0	20k
druid	database connection pool	Apache 2.0	497k
gson	(de-)serialization library	Apache 2.0	58k
junit4	testing framework	Eclipse PL 1.0	50k
mybatis-3	SQL mapper framework	Apache 2.0	115k
spark	web framework	Apache 2.0	15k

CDRS operators. We do not focus on any particular improvement objective. The ability to change code automatically brings into reach non-functional improvements, and also can allow fixing bugs not revealed by (existing) tests. Therefore, we start with test-passing software. We selected our corpus systematically. We consider two sets: (a) three hand-curated projects chosen as particularly appropriate for this experimentation, denoted CS_A and (b) a larger set of projects systematically generated by querying GitHub, denoted CS_B. In both cases we had a number of criteria in mind, which were formalised when generating the second corpus:

- Java as the main language.
- Open-source with a permissive licence.
- Test suites with only passing tests.
- Widely used (over 7000 stars on GitHub).

In selecting CS_A we also tried to select projects from a variety of application domains; we manually searched lists of popular Java projects on GitHub, examining individual projects and selecting those primarily or completely using pure Java, and possessing test suites that would allow us to extensively evaluate the impact of operator application. The projects chosen for CS_A are given in Table 4.1.

For CS_B 55 projects[6] met our criterion, of which we systematically eliminated those projects that were not primarily code-focused or did not compile and test cleanly on our experimental platform. A summary of the chosen projects is given in Table 4.2.

[6] From `https://tinyurl.com/github10kstarsjava`, accessed on 17 December 2018.

Table 4.3: Mean μ and median $\tilde{x}$ for various measures over the *hot* methods identified by Gin's profiler in each project, and all methods in the classes containing the *hot* methods. 'Tests' refers to unit tests. 'Cyc cmp' is cyclomatic complexity. Instruction and branch coverage (inst cov & br cov) refer to the % of instructions / branches covered by the test suite for all / *hot* methods. The aggregate figures over all projects are taken over the separate figures for each method in each project.

Project	All methods in classes containing *hot* methods										
	Method count	Method count	Lines/ method		Statements/ method		Cyc cmp/ method		Inst cov	Br cov	
			μ	$\tilde{x}$	μ	$\tilde{x}$	μ	$\tilde{x}$	%	%	
arthas	1638	8	57	12.5	47.13	11.5	15	5	5	7	
disruptor	375	159	7.87	5	4.23	2	1.44	1	82	78	
druid	18440	8297	10.32	5	8.61	2	2.96	1	75	65	
gson	899	504	10.15	6	8.33	4.5	3.45	2	83	79	
jcodec	6591	1879	10.97	7	9.25	5	3.29	2	46	34	
junit4	1617	1216	6.24	4	4.65	2	1.72	1	86	84	
mybatis-3	2501	1255	8.13	4	6.66	3	2.31	1	83	80	
opennlp	3694	829	12.41	6	10.62	4	3.59	2	35	37	
spark	853	452	6.32	4	4.65	2	1.79	1	71	60	
spatial4j	723	435	9	5	7.31	3	2.76	1	79	72	
all	37331	15034	9.84	5	8.13	3	2.84	1	77	68.5	

	hot methods											
	Method count	Method count	Lines/ method		Statements/ method		Cyc cmp/ method		Inst cov	Br cov	Tests/ method	
			μ	$\tilde{x}$	μ	$\tilde{x}$	μ	$\tilde{x}$	%	%	μ	$\tilde{x}$
arthas	1638	3	135.67	16	112	15	34	7	79	75	1.00	1
disruptor	375	12	11.33	7.5	7	3.5	2.5	1	100	100	1.16	1
druid	18440	534	19.87	9	18.54	8	5.32	2	90	82	2.55	1
gson	899	68	18.41	13	14.76	9	4.65	2	93	90	3.58	2
jcodec	6591	477	16.77	13	14.35	10	4.66	3	94	85	2.51	1
junit4	1617	403	6.67	4	5.1	3	1.82	1	88	90	5.26	3
mybatis-3	2501	323	12.53	9	10.82	8	3.35	2	91	88	5.26	2
opennlp	3694	204	22.26	14	20.15	12	5.74	3	93	86	3.21	1
spark	853	56	14.82	10	12.36	8	4.05	2.5	90	81	4.39	1
spatial4j	723	77	12.3	6	10.26	4	3.61	2	93	91	10.05	2
all	37331	2157	15.51	9	13.59	7	4.17	2	92	85	3.9	1

Our experiments target *hot* methods identified by Gin's profiler (Section 4.3). These methods are those that occur most frequently on the stack trace during test runtime, thus we ensure these are covered by existing test suites. Consequently, these methods are also a good target for improvement of runtime, although in this work we do not consider any particular improvement objective. To understand the generality of our results, we now consider whether *hot* methods are fundamentally different to general methods in our corpus. Table 4.3 gives method-level statistics for the *hot* methods identified by Gin's profiler, compared to all methods in the classes containing the *hot*

methods, broken down by project, with figures for all projects in the last line. We report the number of methods in each category, number of lines and statements in each method, cyclomatic complexity of each method[7], and test coverage[8]. Test counts refer to those unit tests found to have called a method by the profiler, so are only given for the *hot* methods.

In terms of these measures, *hot* methods are broadly representative of all methods. Wilcoxon signed rank tests comparing the values for all methods with those for *hot* methods, over all projects, found $p < 0.001$ for all measures (i.e., line and statement counts, cyclomatic complexity, and coverage). However, this is largely due to the large sample sizes: the effect size for all these tests was < 0.2, usually regarded as small (Cohen (1969)), and equivalent to more than 85% overlap in the distributions. So, *hot* methods do show a statistically significant difference to methods in general, but the differences in median values for each measure are still small. Lines and statements per method define the search space at each granularity: *hot* methods tend to be fairly small, with medians of 9 lines and 7 statements, and means of 15.5 and 13.6, per *hot* method across the whole corpus. However, methods in general are slightly smaller (medians being 2 less and means being 5 less for both lines and statements). Cyclomatic complexity for *hot* methods (median 2) is higher than for methods in general (median 1). Instruction and branch coverage are higher for the *hot* methods, confirming that our profiling identified the portions of the code that are most exercised by the test suite (although interestingly, this high coverage is achieved even though *hot* methods have a median of 1 unit test each (Section 6)). For instance, a mean of only 5% of instructions for `arthas` are covered by the test suite, while a mean of 79% of instructions in the *hot* methods are covered. Although `arthas` is an outlier in terms of proportion of *hot* methods in classes containing hot methods, it was not an outlier in our results (Section 5). Overall, among our corpus *hot* methods are slightly smaller, and slightly more complex, than the average method, with the only substantial difference being that *hot* methods have much greater test coverage.

4.2 Overall Procedure

Our experiments targeted ten open source applications $\mathbb{P}$; here, we consider each application as a target program P. Since we study large real-world programs, it would be infeasible to analyse the whole CDRS edit space. We sample the space as follows:

For each program $\mathbb{P}$ we identified *hot* methods, H_P, which are those that use most of the computation time when the tests are executed. We also identified the set of unit tests T_{H_P} that result in a call to each *hot* method $h \in H_P$.

[7] Reported by the checkstyle tool, Version 8.36.2 — `https://checkstyle.sourceforge.io/config_metrics.html`

[8] Reported by the Jacoco tool, Version 0.7.9.201702052155 — `https://www.jacoco.org/jacoco/`

Algorithm 1: Experimental Procedure: *hotMethods*(P) uses profiling to identify target methods H_P and tests that call these methods T_{H_P}; £ denotes cost, *e.g.*, time and memory usage; $\mathbf{e}_c$ is a compiler error; T_δ is the set of tests that call methods changed by δ; and r_t is the result of running a test t on a compiled program variant v_c — it can pass, fail or return an error.

```
   Input: ℙ                                                          // Corpus
   Input: 𝕄 = {CDRS_l, CDRS_s}                                       // Operator sets
   Input: k = 1..5                                                   // Number of edits per patch
   Input: count = 10000                                              // Patches per number of edits
   Input: sample                  // Sampling procedure, returns a set of patches
   Output: R                                                         // Set of result tuples
 1 R ← ∅
 2 forall P ∈ ℙ do
 3     ⟨H_P, T_{H_P}⟩ ← hotMethods(P)
 4     forall M ∈ 𝕄 do
 5         R ← R ∪ runTests(sample, P, H_P, T_{H_P}, M, k, count)
 6 return R

 7 def runTests(sample, P, H_P, T_{H_P}, M, k, count):
 8     R ← ∅
 9     forall δ ∈ sample(P, H_P, M, k, count) do
10         ⟨v, £_δ⟩ ← δ(P)
11         ⟨v_c, £_c⟩ ← compile(v)
12         forall t ∈ T_δ = {t ∈ T_{H_P} | δ(h) ≠ h ∈ H_P} do
13             R_t ← {⟨t, e_c, 0⟩}
14             if e_c = ∅ then
15                 ⟨r_t, £_t⟩ ← test(v_c, t)
16                 R_t ← {⟨t, r_t, £_t⟩}
17             R ← R ∪ {⟨P, δ, v, £_δ, v_c, £_c, R_t⟩}
18     return R
```

This process is detailed in Section 4.3. We then ran two experiments applying large numbers of edits to the *hot* methods.

The **Random Sampling** experiment generated 10 000 patches for each edit granularity type (line and statement), for 1–5 edits, for each of the 10 projects, making a total of 1 000 000 patches. The **Delete Enumeration** experiment generated every possible line and statement delete edit over all the *hot* methods. Section 4.4 details each experiment's edit sampling procedure.

Algorithm 1 presents the overall, shared experimental framework. For each program $P \in \mathbb{P}$, after we identify *hot* methods and their associated tests (line 3, Section 4.3), for each of our two operator sets $\mathbb{M}$, containing CDRS operators either at the line or statement level (line 4), we sample *count* patches with k edits each (line 9), applying each patch δ to P (line 10), compiling it (line 11), and, upon successful compilation, running tests T_δ that call the methods modified by δ (line 15) on the compiled program variant v_c. We record all

compilation errors, as well as all test results (including failure types) for each test t that was run[9].

4.3 Profiling

Any practical APM process will first profile the code to identify the most promising parts of software for improvement. In the context of automated program repair, for instance, fault localisation techniques are used for this purpose. In the context of runtime improvement, one might target code that uses the most time to execute. The code targetted for improvement will be deemed *hot* in this work.

For each program P, we use Gin's profiler (as implemented in its `Profiler` class) to determine the set of *hot* methods H_P. In Gin these are the methods that are seen most often at the top of the stack trace, when the tests are executed. The assumption is that these are the methods that consume most of the given program's computational time. Note that, however, the *hot*-ness measure can easily be replaced with other measures; thus, one can redirect the focus to other non-functional properties or even functional ones.

The same *hot* methods were then targeted in both Random Sampling and Delete Enumeration experiments. Summary statistics for the *hot* methods identified in each project can be found in Section 4.1.

4.4 Edit Sampling

Our two experiments use two samplers, which we now detail. Our experiments define *locs*, in Equation (4.1), to limit the source and target of an operator $m \in \{\text{COPY, DELETE, REPLACE, SWAP}\}$. This definition limits COPY's and REPLACE's source operand to the class containing a hot method h; it limits SWAP's source to only to the code of a hot method itself and limits all operators' targets to the code of a *hot* method. Note that DELETE does not have a source, only a target. The *locs* returns the set of possible combinations at the correct granularity (line or statement).

$$locs(m, h) = \begin{cases} source(class(h)) \times target(h) & \text{if } m = \text{COPY} \vee m = \text{REPLACE} \\ target(h) & \text{if } m = \text{DELETE} \\ source(h) \times target(h) & \text{if } m = \text{SWAP} \end{cases} \tag{4.1}$$

Algorithm 2 shows the sampler used in our **Random Sampling** experiment. For a given budget of edit applications given by k, it generates *count* patches, where it generates each patch by uniformly choosing a hot method,

[9] We record patch application and patch compilation costs for a sample of 1000 patches in Table 5.2. All other parameters are reported for all patches from the two experiments.

Algorithm 2: Multi-edit patch sampler. We **choose()** uniformly at random.

```
   Input: P                                              // Target class
   Input: H_P                                            // Hot methods in P
   Input: M                                              // Operations
   Input: k = 5                                          // Number of edits
   Input: count = 10000                                  // Patches per edit length
 1 V ← ∅
 2 for i in [1..k] do
 3     for count times do
 4         h ← choose(H_P)                               // Choose hot method
 5         δ^0 ← P                                       // Initial, identity patch
 6         for j in [1..i] do
 7             m ← choose(M)                             // Choose operator
 8             (l_0, l_1) ← choose(locs(m, h))           // Choose locations
 9             δ^j ← m(δ^{j−1}, l_0, l_1)
10         V ← V ∪ {δ^i}
11 return V
```

then the edit operator m, before calling *locs* and applying the operator. All edits in a single patch target the same *hot* method. Gin's `RandomSampler` class implements this sampler. Note that the first loop iteration initialises δ^0 with the original program in Line 5, and all subsequent patches build upon it, *i.e.* each δ^j represents a patch with j edit operations.

Algorithm 2 picks each of the DELETE, COPY, REPLACE and SWAP operators with equal probability. This samples more of DELETE's space relative to the other operators, because the space for DELETE is only the number of editable locations, rather than the product of two editable locations as for all other edits. As far as we know, all heuristic-based improvement frameworks to-date more intensively sample DELETE's search space. We follow the same strategy and leave considerations of equally sampling each operator's space to future work. Smigielska et al. (2021) showed that sampling strategy that searches an equal proportion of each operator's search space could indeed help find useful patches for the purpose of bug fixing. This might be due to DELETE not being as effective at bug fixing yet sampled frequently using standard sampling strategy. Results might look different for runtime improvement, for instance. Here we are concerned with the more general idea of how *plastic* is code under APM, leaving considerations of effectiveness of particular operator types for specific improvement objectives for future work.

The sampler for the **Delete Enumeration** experiment builds and samples $V = \{\delta^1 \mid h \in H_P \wedge (-, l) \in locs(d, h) \wedge \delta^1 = d(P, l)\}$, where $d(P, l)$ denotes deletion of location l in program P. For each granularity (line or statement), this procedure visits each location in H_P and creates a patch comprising a single DELETE edit for that location. Gin's `DeleteEnumerator` class implements this sampler.

```
1 private void skippedQuietly(org.junit.internal.AssumptionViolatedException
      e, Description description, List<Throwable> errors) {
2     try {
3         if (e instanceof AssumptionViolatedException) {
4             skipped((AssumptionViolatedException) e, description);
5         } else {
6             skipped(e, description);
7         }
8     } catch (Throwable e1) {
9         errors.add(e1);
10    }
11 }
```

```
1 private void skippedQuietly(org.junit.internal.AssumptionViolatedException
      e, Description description, List<Throwable> errors) {
2             skipped(e, description);
3 }
```

Fig. 4.1: Example of a SWAP edit. The red parts in the original listing (top) are effectively deleted by the edit that swaps the `try..catch` block (lines 2-10) with the `skipped()` method invocation (line 6), producing the patched listing (bottom).

4.5 Operator Implementation

The implementation details of edit operations affect the likelihood of producing test-passing variants, so we summarise them, for both granularities, and note the implications.

Within a given patch it is possible to have the same target location addressed by more than one operation. Moreover, it is possible to have the target location of an earlier operation addressed as the source for a later operation. In Gin, consistent substitutions within a patch are maintained by the following rules: (1) all operations affect only their target location. The IDs of other code locations are unaffected (so, *e.g.*, deleting line n does not cause line $n+1$ to be relabelled as n); (2) to prevent a later operator wasting effort by nullifying an earlier one, once a location is changed by a DELETE, SWAP or REPLACE operation it cannot serve as a source for a subsequent operation in the patch, so operations using it as a source become noOps; (3) COPY inserts to a given target location are chained after the target location. The inserted code cannot be addressed as a source for subsequent operations and cannot be deleted by subsequent operations; (4) once a location is deleted, subsequent DELETE, SWAP and REPLACE operations to that location in the patch have no effect. However, COPY operations can still target an insertion at this location.

A line can contain multiple statements or none, as when a single statement is formatted across several lines. More frequently, however, statements contain multiple lines, because control statements tend to contain statement blocks that, in turn, contain statement lists. Consider the main event loop of

any reactive system or the instruction loop of an interpreter when these are implemented using a while statement. Denominated in lines, then, statements tend make more changes than line operators do. For this reason, we consider edits of both statement and line granularity in our experiments. It is important to note that our definition of *statement* is that of JavaParser's, and thus includes block statement (*i.e.*, lists of smaller statements wrapped in braces, such as the body of a loop or `if` structure). This makes it possible for larger sections of code to be deleted or moved in a single operation, and both block statements and the statements contained within them can be manipulated by our edits.

One important implication of this is connected to our implementation for SWAP, as this is relevant to our explanation of the results later. We implement SWAP as a pair of delete-insert operations. In the case where one of the pair of statements to be swapped is contained within the other, effectively the swap becomes a delete operation. An illustrative example is in Figure 4.1. Here, the swap targeted two statements in JUnit's `TestWatcher` class. The first statement was the whole `try...catch` block (lines 2–10), and the second was the `skipped()` method invocation (line 6). The result was the deletion of most of the method, which still compiles correctly.

5 Results

In this section, we detail our computational resources we used, the scale of our experiment, and the patch filtering we used for RQ1, RQ2, RQ4, and R5, then answer each research question.

We provide full experimental results, and the scripts required to reproduce those results using Gin: `https://github.com/automatedprogrammodification/automatedprogrammodification/blob/main/replication_package/`.

Execution Environment We conduct all experiments on a compute server with four AMD Opteron 6348 (2.8 GHz) processors (total: 48 physical cores), 128 GB RAM, using Java 1.8.0_192 on CentOS 6.9. Both during profiling and each experimental run, each test case is run once with a timeout of 10s (Gin's default). On this machine, our experimental runs took a little under two weeks of single-core wallclock time. Gin's Profiler run for all 10 projects took 48.5 hours. The Random Sampling experiment took 38.2 hours for line and 150.4 hours for statement edits. The Delete Enumeration experiment took 50.6 hours for all 10 projects.

5.1 Experimental Scale

Our experimental procedure covers an unprecedented scale for an APM landscape study. We have covered 10 real-world open source projects, and our sampling procedure was designed to test enough applications of edits across

Table 5.1: Search space sizes for statement and line edits in Gin.

Project	Single line edits	Single statement edits
`arthas`	526 538	343 642
`disruptor`	57 272	16 693
`druid`	38 907 104	46 219 904
`gson`	1 327 524	903 400
`jcodec`	6 386 328	4 203 594
`junit4`	734 594	405 950
`mybatis-3`	2 761 544	1 738 527
`opennlp`	2 095 158	1 438 388
`spark`	226 328	116 515
`spatial4j`	445 162	273 041

to start making more general conclusions. In particular, we applied every single possible line and statement delete edit to the *hot* methods identified in these projects.

Overall we analyse 1 000 000 randomly sampled sequences of edits, and 33 458 single-line and 24 396 single-statement edits. The largest corpus for an APM study was used by Harrand et al. (2019), where they analysed 180 207 single-statement edits. Note we are analysing the search space of test-suite adequate program variants. Arguably, we could have started with the fixed versions of the famous Defects4J[10]. However, we thought of it as a dataset designed specifically for the APR community. Instead, we systematically sampled open source projects to avoid such a bias. Furthermore, the most widely studied, original version of Defects4J contains programs whose size totals 321kLoc, while here `druid` alone contains 497kLoc.

Table 4.3 contains a summary of the data extracted. The numbers of all possible single line and single statement edits are presented in Table 5.1. The data shows that the search space size for line edits is roughly twice as big as for statement edits, with the exception of `druid`, having more possible statement than line edits. Moreover, `druid`'s search space is much larger than that of other projects, even though the number of *hot* methods, although largest, is not that much bigger than that of `jcodec`, as seen in Table 4.3.

Random Sampling The random sampling experiment generated a total of 1 000 000 patches; 10 000 per patch size, sizes 1–5, 2 granularity levels, and 10 projects). For line edits, after removal of duplicates and identity patches, we were left with 73 301, 97 349, 99 585, 99 847, and 99 916 patches of 1–5 edits, respectively. Over all line patches, 7731 were identity patches, and 23 892 were duplicates (1621 were both identity and duplicates). Duplicates appeared at different rates. There are far fewer possible DELETE edits, because DELETE only has one associated location rather than two. There are also fewer SWAP edits, because the two locations are limited to the same hot method rather than one location being drawn from the whole class. Over all single line edit

[10] https://github.com/rjust/defects4j

patches, the number of duplicates removed were 926 COPY, 14 031 DELETE, 956 REPLACE, 6577 SWAP. For statement edits, after removal of duplicates and identity patches, we were left with 61 066, 91 959, 97 871, 98 948, and 99 223 patches of 1–5 edits, respectively. In total, we report results for 469 998 (line) and 449 067 (statement) unique random patches, with 1 to 5 edits. Over all line patches, 7731 were identity patches, and 23 892 were duplicates (1621 were both identity and duplicates). Duplicates appeared at different rates. There are far fewer possible DELETE edits, because DELETE only has one associated location rather than two. There are also fewer SWAP edits, because the two locations are limited to the same hot method rather than one location being drawn from the whole class. We can only easily analyse single edit patches for this because multi-edit patches can contain a mixture of edit types, but over all single line edit patches, the number of duplicates removed were 926 COPY, 14 031 DELETE, 6577 REPLACE, 956 SWAP. For statement edits, after removal of duplicates and identity patches, we were left with 61 066, 91 959, 97 871, 98 948, and 99 223 patches of 1–5 edits, respectively. Over all statement patches, 27 502 were identity patches, and 32 028 were duplicates (8597 were both identity and duplicates). Duplicates also appeared at different rates for statements. Over all single statement edit patches, the number of duplicates removed were 2118 COPY, 15 236 DELETE, 2001 REPLACE, 8755 SWAP. In total, we report results for 469 998 (line) and 449 067 (statement) unique random patches, with 1 to 5 edits.

Delete Enumeration In total there were 33 458 and 24 396 single-delete edits for line and statement types respectively. Interestingly, single delete edits make up less than 1% of the overall transformation space in our experiments.

Patch Filtering In answering all research questions other than RQ3 and RQ5, we exclude identity patches, those that do not change the code's syntax, *e.g.* replacing line 10 with line 10. Formally, an identity patch is one with an edit sequence δ where $P = \delta(P)$, for all programs P. Such "do nothing" patches are rare. We removed them because they simply waste resources. Section 5.1 reports how many identity function we filter. The goal of this work is to explore the search space of CDRS edit operations. Duplicated patches simply repeatedly visit the same location in this space, adding no new information, so we report results on unique patches only.

5.2 Which Edit Operators Maximise Test-Passing Variants?

To determine which edit operators, among COPY, DELETE, REPLACE, SWAP (CDRS), are most effective when applied once we analysed the single 73 301 line and 61 066 single statement edits from the Random Sampling experiment. We calculated the frequency with which each single edit type still passed all the tests. These rates per operator type vary per project, and so are presented per project in Figure 5.1. DELETE is by far the most effective single edit operator, with SWAP being second-best. This echoes the work by Le Goues et al.

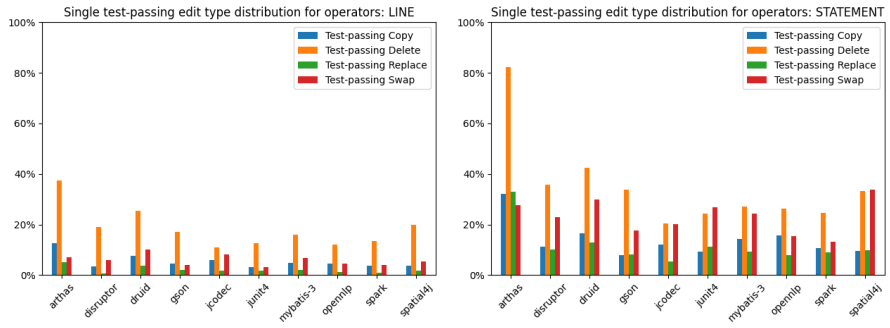

Fig. 5.1: The percentage of variants produced by a single application of each edit operation that pass the tests.

(2012) on C programs, where the DELETE was found most effective, followed by REPLACE and SWAP (equally effective), with COPY being least effective. In our case, however, SWAP was found to be significantly better than REPLACE. The test pass rates of COPY and REPLACE never exceed that of DELETE and are generally less than half that of DELETE at either line or statement granularity.

Now, we are in a position to answer RQ1: "How often do single applications of the COPY, DELETE, REPLACE, and SWAP (CDRS) operators produce test-passing variants?". Under our sampling scheme (see Section 4.4), we find that:

Answer to RQ1

The median single edit test pass rates for our edit operators are: DELETE (line: 16.5% / statement: 30.2%) > SWAP (5.6% / 23.6%) > COPY (4.6% / 11.6%) > REPLACE (1.7% / 9.6%), with a substantial gap (10.9 points / 6.6 points) between DELETE and SWAP.

Interestingly, for statement edits, the SWAP operator turned out to be the next most effective after DELETE in our experiments. Figure 5.1 shows SWAP having around twice the pass rate of COPY and REPLACE on `disruptor`, `druid`, `gson`, `jcodec`, and `mybatis-3`, and exceeds even DELETE on `junit4` and `spatial4j` for statement-level edits. Some of this might be explained by the situation described in Section 4.5, whereby swapping nested statements effectively deletes the parent statement. In our experiments we found that, among the single-edit patches sampled, this occurs for around 13% of all SWAP edits; though the rate varies per project from 8.9% for `jcodec` to 18.3% for `junit4`. Assuming the test suite adequately tests the effect of a SWAP, SWAP-produced variants pass when the order of the swapped snippets does not matter functionally. A deep dive on the target methods for the four projects having the highest effective statement edit rates for swaps according to Figure 5.1 (`jcodec`, `junit4`, `mybatis-3` and `spatial4j`) revealed that they also appeared to have a large number of independent statements. Examples include assignments to local variables, or

swapping the content of an `else` block to after the closing brace, where the corresponding `if` finished with a `return`. Swapping may, however, provide a non-functional benefit, when the new order better fills the CPU's instruction pipeline. For improvement of non-functional properties then, the decision to drop SWAP should be revisited.

One big picture goal of this study is to point to ways to make APM more effective. We asked RQ1 to find which primitive operators are most effective. We pose that constraining operators or defining new ones might be still more effective. Thus, we investigated the reasons for test failures for single-edit patches[11] to look for any patterns that might suggest such constraints or new operators. Overall, the most common reasons for test failures for compilable edits were due to either a `java.lang.NullPointerException` (5.3% of cases) or `java.lang.AssertionError` (5.9% of cases). This finding suggests that introducing a new operator that makes a simple, conservative null pointer check after applying a CDRS operator might pay off. Line granular edits can break a language's syntax in many more ways than can statement granular edits. Indeed, the compilation rate of statement granular edits was 26.1%, 16.7 points greater than the 9.4% of line granular ones. It is not surprising then that statement granular changes exhibited more test failures for compilable variants (14 889 *vs* 5955) and test failure types (72 *vs* 62) than line granular ones, because the statement granular sample size is larger.

5.3 DELETE's Landscape

DELETE was the most effective. This is well-known, but still surprising because most developer commits add more code than they delete. DELETE also has the smallest search space. Thus, we can, and are the first to, exhaustively determine the upper bound on its effectiveness in a nontrivial search space, so we now turn to answering RQ2.

In total, there were 33 458 and 24 396 single-delete edits for line and statement types respectively. As an exhaustive search, these already comprised only unique edits. This experiment took just under 53 hours (single-core wallclock time) for all 10 projects: 17 hours and 54 minutes for the line deletions and 35 hours and 1 minute for the statement deletions. While there were one-fifteenth as many patches as in the random sampling experiment, this time it was not 15x shorter because many patches failed to compile, which means we can skip to the next patch without also running the unit tests. The differing compile rates also explain the shorter run times for line edits (see Figure 5.3).

We report compilation rates between 15% and 41% for line edits, 31% and 90% for statement edits, pass rates (SMRs, see Equation 2.1) between

[11] Data for all patches from the Random Sampling experiment is visualised here: `https://github.com/automatedprogrammodification/automatedprogrammodification/blob/main/replication_package/results/graphs/sample/TestEdits_STATEMENT.png` and `https://github.com/automatedprogrammodification/automatedprogrammodification/blob/main/replication_package/results/graphs/sample/TestEdits_LINE.png`

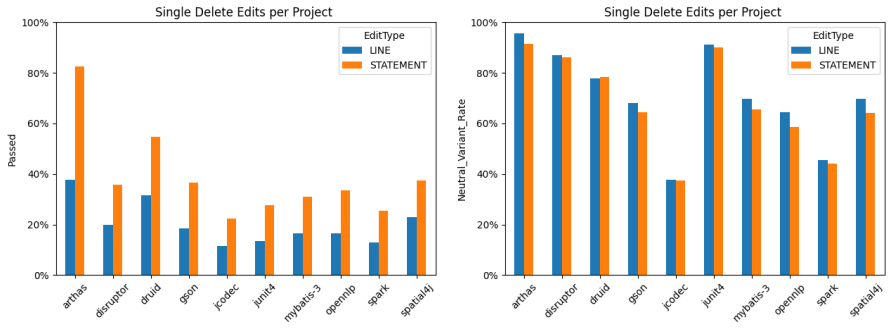

Fig. 5.2: Pass rates and neutral variant rates, per project, for single deletes.

12% and 38% for line edits, 22% and 82% for statement edits, and neutral variant rates (NVRs, see Equation 2.2) between 38% and 96% for line edits, and between 37% and 92% for statement edits. Pass and neutral variant rates for all projects are shown in Figure 5.2.

We also gathered data for the most common reasons for test failures of single deletes. In 61% of cases the patch did not compile and in 28% instances the test passed. For the compiling patches the most common reasons for test failures encountered have been: `java.lang.AssertionError` and `java.lang.NullPointerException`[12]. Interestingly, null-pointer exceptions are also among the most common types of error that programmers encounter Coelho et al. (2017); Hassan et al. (2020).

We are now able to answer RQ2. "What percentage of all variants produced by exhaustively applying DELETE to each line or statement in a method passes all tests?":

Answer to RQ2

Pass rates for DELETE reach 38% and 82% for line and statement granularity, respectively. DELETE mostly fails at compile time, with 60% of compiling variants passing all tests for both line and statement deletes.

This finding means that DELETE is not only successful in bug fixing Le Goues et al. (2012), but also has most potential for improvement of non-functional properties. However, exhaustive enumeration of single-deletes is unlikely to be cost-effective (in terms of wall-clock time). Perhaps, if the most frequently used methods were only investigated, this could be used as a first step, before GI with other edit operators is run on such an abridged software.

[12] `https://github.com/automatedprogrammodification/automatedprogrammodification/blob/main/replication_package/results/graphs/delete/TestSingledeletes_LINE.png` and `https://github.com/automatedprogrammodification/automatedprogrammodification/blob/main/replication_package/results/graphs/delete/TestSingledeletes_STATEMENT.png`

Interestingly, the median pass rates we report here, over exhaustive enumeration, are close to the pass rate of variants produced by delete during the edit sampling experiment (Section 5.2). Over all projects, delete's median pass rate was 17.5% (line) and 34.5% (statement) for enumeration *vs* 16.5% and 30.2% for sampling. Section 6 discusses the construct validity threat that test suite inadequacy poses to the test-passing results reported here.

5.4 Operator Granularity: Line *vs* Statement

The vast majority of APM work has improved programs at the granularity of either lines or statements. Here, we compare the two and report our findings over our CDRS landscape. The previous two experiments — edit sampling and exhaustive DELETE enumeration — generated the data we need to answer RQ3. First, we compare the granularities in terms of pass rate, then their costs. For this experiment, we do not, however, delete duplicate or identity patches, since we must account for their cost.

For the edit sampling experiment, Figure 5.1 shows that pass rates for line edits are lower than for statement edits: 0.03–10.97% *vs* 2.32–33.88%, with medians, across all projects, being 5.07% *vs* 15.05%. Looking at specific examples for illustration, DELETE's pass rate on `arthas` is 38% for line and 83% for statement; on `disruptor`, COPY's is 4% for line and 11% for statement. Across all edit operators and all projects in our corpus, the rates for line are around a third that for statement. Compilation rates explain much of this difference: the median compilation rate for single edits across all projects is 9% for line and 26% for statement edits.

The DELETE enumeration experiment tells a similar story. Figure 5.2 reports compilation rates of 15%–41% for line and 31%–90% for statement edits. The figure clearly shows that the rates for line, on any one project, are around half that of statement. Unsurprisingly, line edits are typically more likely to break syntax and produce uncompilable code than statement edits. In contrast, the fractions of compiling code that then passes the tests are very close: 38%–96% for line edits and 37%–92% for statement edits, leading to overall pass rates of 12%–38% for line and 15%–41% for statement. The medians over all projects are 5% and 15%, respectively. This suggests that the major difference in the effectiveness of line and statement granularities is in getting past compilation: if the edits produce code that compiles, the chances that it will then pass the tests are much the same at either granularity.

Unlike line edits, statement edits are grammatically aware: they cannot introduce imbalanced parentheses, for instance. The only syntax error they can introduce is creating empty statement lists where the grammar requires a nonempty list. Thus, our compilation finding is perhaps surprising only in that statement granular edits do not outperform line granular ones by a still greater factor. Both granularities waste resources on failed compilations, although statement generates fewer. The grammar-awareness of statement granular edits has a price — parsing. To quantify the cost of constructing V_T (*i.e.*,

Table 5.2: Median times (ms) and total times (s) for each stage of applying and testing single edits across all projects (rounded to 3 sig. figs.), for a subsample of 100 edits for each project drawn uniformly from the Random Sampling results (*i.e.*, 5000 edits in total for each granularity).

	Count		Median time (ms)		Total time (s)	
Measure	Line	Statement	Line	Statement	Line	Statement
Apply Edit	5000	5000	0.135	7.937	1.077	98.461
Apply Edit (V_T)	71	334	0.152	6.423	0.012	5.074
Apply Edit ($V \setminus V_T$)	4929	4666	0.135	8.049	1.065	93.386
Compile (all)	5000	5000	9.073	31.089	111.116	250.238
Compile (fail)	4844	4134	8.890	29.929	94.261	195.261
Compile (success)	156	866	62.428	37.269	16.854	54.977
Test (all)	156	866	2.889	2.425	448.356	254.058
Test (fail tests)	85	532	3.136	2.650	441.242	229.108
Test (pass tests)	71	334	2.368	1.819	7.114	24.949

Table 5.3: The cardinalities of the variants produced, those that compile, and those that pass the test suite T, over all 10 000 edits and the 10 projects.

	Line	Statement
$\lvert V \rvert$	73301	61066
$\lvert V_c \rvert$	6855	15946
$\lvert V_T \rvert$	4537	10312

Table 5.4: Total costs in seconds over entire sample set; columns 2–5 are Equation 2.4.

M	$\$_M(V)$	$+\ \$_c(V)$	$+\ \$_T(V)$	$=\ \$(V);$	$\$(V \setminus V_T)$
CDRS_l	1.08	111	448	560	536
CDRS_s	98.5	250	254	603	517

program variants that pass all tests) under both granularities, we repeated the random sampling experiment with additional time measurements for 100 edits sampled uniformly from the 10 000 for each project and granularity. This experiment was much smaller in scale than those described in Section 4.2: run time measurement is notoriously noisy, so to mitigate this as far as possible the timing runs were performed one at a time on a standalone workstation (Debian OS, two 16-core Intel Xeon E5-2620 v4 CPUs @2.1 GHz, with 32GB DDR4 RAM), with no graphical desktop environment and no other CPU-intensive tasks running. The results for this are in Table 5.2.

Applying line edits requires only dividing source into a list of strings. For statement edits we must build a parse tree. Over all projects in our random sampling experiment, the median cost of applying a single line edit was 0.135ms, considerably faster than the 7.94ms to apply a single statement edit.

Compilation time varies with success. Syntax errors, for example, terminate compilation early. In Table 5.2, the greater gap between successful compilations *vs* failed compilations for line *vs* statement reflects this fact. Indeed, this gap implies that the majority of failures for line edits are syntactic, whereas grammar-aware statement edits tend to produce errors that are detected later, such as variables out-of-scope.

Only variants that compile (V_c) can be tested. Ignoring test suite adequacy, only those that pass the tests (V_T) do not waste effort. The cardinalities of these two sets are compared in Table 5.3: it is clear that large numbers of edits are excluded from V_c, then V_T in turn, for both granularities. Across all the edits, including both passing and failing variants, the median cost of testing was similar for line and statement edits: 2.89ms and 2.43ms. This difference is much smaller than those for applying edits or compiling the resulting variants. The testing timings in Table 5.2 do not necessarily mean that the statement edits produced faster variants than line edits: while the set of all line edits was applied to the same target methods as the set of all statement edits, the methods for which these edits compile were different, so different unit tests would have been run for each.

Table 5.4 summarises and compares the cost, in seconds, of the two granularities in terms of Equation 2.4, drawn from the 100 edit subsample results in Table 5.2. $\frac{\$(V \setminus V_T)}{\$(V)}$ is wasted computation, up to test suite adequacy: Line edits wastes $\frac{536}{560} = 95.7\%$ and statement wastes $\frac{517}{603} = 85.7\%$. Where the two really separate is the cost paid for each test passing variant, *i.e.* $|V_T|$, we obtain by normalising against the cardinalities in Table 5.3. $\frac{560}{4537} = 123$ms for line *vs* $\frac{603}{10312} = 58$ms for statement: line edits cost over twice as much for each test passing variant. This factor grows with multiple edits. As Section 5.5 shows, the pass rate for line edits drops off much faster than that for statement.

These results enable us to answer RQ3: "At which granularity — line or statement — do single applications of the CDRS operators produce more test-passing variants?". Across all projects, we find that:

Answer to RQ3

Line edits generate $\frac{1}{3}$ as many test passing variants as statement edits, with median test pass rates over all projects of 5% *vs* 15%. They waste 10% more computational resources, and cost twice as much on average.

This finding means that there is little motivation for further focus on edits that ignore language grammar as line edits do. Rather, it provides strong justification for further research into more strongly grammar-aware edits. Once the cost of parsing has been paid, more sophisticated statement edits than CDRS can be applied with little extra computational effort (Brownlee et al. (2020)). It also justifies investigation into even more expensive, *property-aware*, edits that rely on program analysis to select where they should be applied and produce variants with known properties, like correctness modulo an oracle, thereby obviating testing.

A caveat on this conclusion concerns non-code modifications, notably in comments. Unlike line granular edits, statement granular edits cannot make them. Because we lack tests, let alone test oracles, for non-code modifications, our results here do not account for this advantage that line granularity has over statement granularity (Zhong and Su 2015, Finding 1).

Statements form a tree, which can contain nested statements. Thus, a single statement edit can span multiple lines and make a larger change than a single line edit. Nesting may also explain the performance of SWAP discussed in Section 5.2. This fact poses a construct validity threat to these results, which compare the two granularities in terms of edits. To mitigate this threat, we now show below that the average conversion of statement edits into line edits is three and observe that mulitiple line edits perform even worse than a single edit, thereby establishing that our central finding understates the disadvantage line granular operations have relative to statement.

To determine the extent of statement nesting, for every statement in a *hot* method, we computed its height and fanout. Over all statements the median height is 1 (IQR 2), and median fanout is 1 (IQR 1). Lest this result be dismissed as uninteresting because it is dominated by atomic statements, we also report these measures for the block statements of method bodies, whose median height is 4 (IQR 4), and median fanout is 2 (IQR 3). Because a method body always contains at least one statement, we found these results for method bodies to be surprisingly low. Despite both findings, over all statements in our corpus, we found that a substantial proportion (45.2%) contain at least one nested statement.

To approximate the granularity of statement edits when converted into lines, we reviewed all statement edits generated in our random sampling experiment. Across all edits applied in our experiments, the median number of lines covered by a statement edit was 3. One might expect that the figure would be lower than this; the average is pulled up by JavaParser's definition of statement, which includes blocks (*i.e.*, anything in braces). So, to make the same magnitude line change as that of a single statement edit using line edits we need three line edits. The compilation rate for patches of one statement edit is 25% (the first STATEMENT bar in Figure 5.3), while, for patches of three line edits, it is 1.3% (the third LINE bar in Figure 5.3). The point here is that even controlling for the increased number of lines that statement edits cover compared to line, line still performs worse. Section 5.5 shows that this difference only increases with the number of edits applied.

Our definition of SWAP means a child node can replace its parent. This coupled with the prevalence of statement nesting partially explains why SWAP has performed so well in our study: it is acting more like a DELETE. Over all 449 966 unique non-identity statement patches, there were 1 434 100 individual edits (recall that we generated patches with 1–5 edits), of which 377 460 (or 26%) were SWAP. Of these, 49 415 (13% of all statement SWAP edits) were nested child statements replacing an ancestor, thereby implicitly causing many deletions. Different projects had different rates of such nesting: the lowest was `jcodec` with 8.9%, and highest was `junit4` with 18.3%.

5.5 Pass Rate of Multi-Edit Patches

We now present the results of applying patches containing 1–5 edits sampled at random from the search space. Figure 5.3 gives a head-to-head comparison of compilation rates over all operators for the two granularities. For each operator type and each edit sequence size, this is the percentage of patches that resulted in code that compiled successfully. We also considered the same results broken down by project. There was little variation between projects. The trend in compilation rates for each granularity and across the number of edits in each patch per project appears to be consistent with the overall trend in Figure 5.3[13].

There are three major observations from these results. Firstly, similarly to the results of Langdon and Petke (2017) on non-Java software, we see a remarkable robustness for single edits: compilation rates of 4%–15% for line and 16%–45% for statement edits per project. Secondly, there is a power law drop-off in compilation rates per project and per edit type. Thirdly, as for single edits, compilation rates for line are worse than for statement edits.

With regards to pass rates, aggregated across all projects, the trends are similar to those for compilation rates in Figure 5.3. As with compilation rates, the breakdown of pass rates for each application does not vary substantially from the overall trend across all projects. However, more variability per project is seen in neutral variant rates, which are shown in Figure 5.4. For the majority of patches that compile, the modified code runs without error. The dropoff in NVR varies per project, with `disruptor` actually showing an increase from 1 to 2-edit patches. NVRs vary from around 40% to 90% for 1-edit patches, and 20% to 90% for 5-edit patches. Harrand et al. (2019) report 16% to 30% neutral variant rates for single edits. This is contradictory to our results, which show much higher neutral variant rates. Interestingly, Langdon and Petke (2017) report high neutral variant rates (up to 89%) for three non-Java programs. One explanation is that non-object-oriented programs, such as C, are more amenable to APM mutations. With regards to the difference with Harrand et al.'s results, they focus on whole Java classes, while we modify methods.

We find that, for up to five edits, the pass rate is quite low (median 0.14% for line and 6.2% for statement granular edits across all projects), confirming the conventional wisdom about the sparseness of this search space. This suggests that grammar-aware operators are indeed needed. Interestingly, however, many of the test-passing multi-edit patches contain evidence of interaction between edits in the form of self-repairs.

We can now answer RQ4, "How often does a sequence of CDRS edits create a test-passing variant?":

[13] Per project data visualisation is available here: `https://github.com/automatedprogrammodification/automatedprogrammodification/blob/main/replication_package/results/graphs/sample/Compiled_LINE.png` and `https://github.com/automatedprogrammodification/automatedprogrammodification/blob/main/replication_package/results/graphs/sample/Compiled_STATEMENT.png`

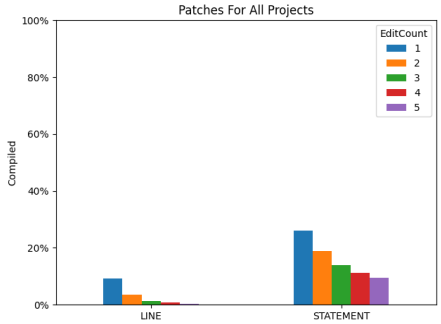

Fig. 5.3: % of patches for which the code successfully compiled, for sequences of 1–5 edits.

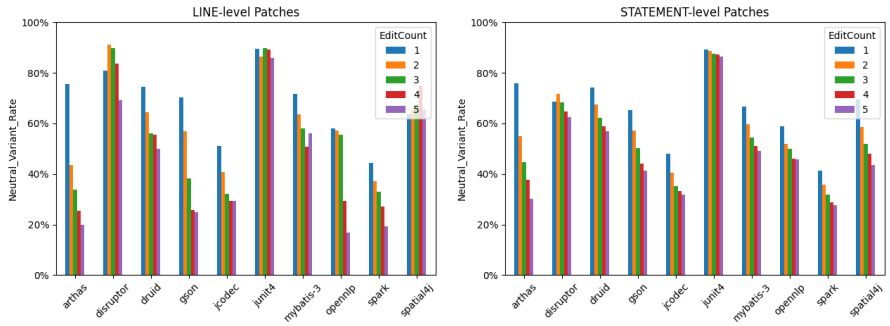

Fig. 5.4: % of patches for which the code successfully passed the test cases, normalised to only those for which the code also compiled, for sequences of 1–5 edits (the Neutral Variant Rate).

Answer to RQ4

Compilation rates are reduced by roughly 50% with each additional line edit; and by 25% for statement edits. Pass rates for multi-edit patches are low and drop with the number of edits.

In practice, real-world patches usually change multiple lines or statements. Yet we observe a consistent drop in the number of program variants that pass tests as the number of edits in a patch increases, using our universal operators at either granularity. We also find that statement edits outperform line edits, which we attribute to their grammar-awareness. These results confirm the conventional wisdom at unprecedented scale: the APM search space is *sparse*. We believe that future work needs to tackle sparsity straight-on. Two promising paths are exploring operators that are not universal, but instead tailored to specific defect classes or improvements, and definition of equivalence

classes that permit cheaply, but accurately, discarding variants without expensive analysis or testing. Bespoke operators can make more extensive changes to a smaller, more focused set of program locations, similar to the templates already finding success in the APR community(Liu et al. (2019)). Varfix (Wong et al. (2021)) exemplifies leveraging equivalence classes. Other example approaches leverage information from existing bug fixes for more effective patch selection (Soto and Goues (2018); Koyuncu et al. (2020)).

Some edits in a patch interact. For instance, two edits may both be needed for a patch to fail or pass. An edit interaction is a *self-repair* when first edit takes a test-passing variant and produces a test-failing variant and the second edit fixes the first edit's breakage and produces a test-passing variant. Recall that, for an edit sequence δ, we write $\delta(P)$ to apply δ to the program P and produce the variant of P under the edits δ (Section 2). For the program P, test suite T, and the patch with edit sequence $\delta \in CDRS^*$, δ contains a *self-repair* if there exists a decomposition of δ, $\alpha m_b \beta m_r \gamma = \delta$, such that $m_b, m_r \in CDRS$ and $\alpha(P)$ passes T, $\alpha m_b(P)$ fails T, while $\alpha m_b \beta m_r(P)$ passes T.

From our data of 919 065 unique patches, we gathered those whose edit sequences had length two or greater and produce test-passing variants. We focus only on those for which we have data for intermediate patches (*i.e.*, for patches with n-edits we have data for patches with $(n-1)-\ldots 1-$edits). There were 11 259 such patches. In the majority of cases, each subsequence of edits was a passing one, leaving 2344 cases (775 for line, and 1569 for statement), where although the patch ultimately passed the test suite, it contained edit sequences that either caused a compilation error or a test failure. 92% of these self-repairs contain recoveries from a compilation failure. Many of these recoveries are simply due to edits that undo the change that caused compilation failure leaving the code unchanged: *e.g.*, a DELETE followed by a COPY that places an identical piece of code back in the same place.

To determine the nature of self-repairs that are not effectively identity patches, we now present a deep dive on `opennlp`. This subject had the highest percentage of self-repairs that did not simply revert to a previous software version, 98%. Manual investigation of `opennlp`'s 212 (57 for line, 155 for statement) self-repairing patches revealed that, typically, those line edits that broke compilation and yet could be repaired were the deletion or insertion of braces, due to the later addition or removal of another brace. Statement edits that broke compilation typically copied a variable out of scope, duplicated variable declarations, copied a `return` statement creating dead code, or, in one case, copied a `case:` outside of a `switch` block. In the majority of cases, a later edit 'repaired' these by deleting the enclosing statement. In one case, a variable declaration was deleted and a later edit deleted the only reference to that variable. In another, a long `if. . . else` block that set the value of a variable had the content of its `else` block deleted, breaking compilation because the variable may have been uninitialised in later use. A later edit 'repaired' this by copying a value assignment to immediately after the variable's declaration.

Across all 5-edit patches, self-repairs are infrequent: 1.9% for line and 2.1% for statement. Considering only test-passing patches, however, self repair av-

erage 21% over all patches, pulled up by the maximum of 31.6% for 5 edit patches. In short, eventually successful patches are *much* more likely to self-repair, *i.e.* to be plastic. As we have shown, computing which patches are eventually successful is expensive! This self-repair finding, therefore, brings to the fore the question of whether we can identify plastic code snippets more cheaply than via testing. Any efficient means of identification would leverage code features, leading us to our final research question, which we answer next.

5.6 Which Methods are Plastic, or Amenable to APM?

We conducted an initial exploration of the most plastic methods, and which software metrics might influence code plasticity. The idea is that these metrics might serve as a proxy so we might better target edits to specific locations, or avoid costly evaluations of the patched code.

Finding a definitive set of interpretable measures remains an open question. Here we select a classic well-known set of measures. Additionally, we added def-use distance because we hypothesised that a large def-use distance between variables gave more space for edits to move or duplicate code between variables' definition and use.

We applied the checkstyle tool[14] to the *hot* methods in each project. For each method, the cyclomatic complexity, the count of statements excluding comments (referred to by checkstyle as non-commenting source statements — JavaNCSS), and the number of possible execution paths (NPATH) were calculated. NPATH (Nejmeh 1988) is an extension of cyclomatic complexity that seeks to address issues like nesting level within a function and lack of distinction between different kinds of control flow structures. In addition, we used the Soot static analysis tool[15] to collect data on the median and average number of lines between variable definitions and their subsequent use in the target methods. We also calculated normalised values of these metrics adjusted for method length in lines.

The full set of random sampling results targeted 2157 methods across the ten projects. These were filtered to include results for only those methods with:

- a length of 10 lines or more (leaving 741 methods), and
- those whose code was sampled relatively frequently relative to the number of lines (*i.e.*, those with a number of patches equal to or greater than the method length in lines – leaving 467 methods), and
- those with 100% test coverage of their statements (leaving 272 methods).

We applied the length filter above to ensure that the normalised def-use distances and other complexity measurements were sampled from large enough tracts of code to be meaningful. The other two filters above were applied to avoid potential bias arising from sparse sampling and coverage. In addition

[14] Version 8.36.2 — `https://checkstyle.sourceforge.io/config_metrics.html`
[15] Version 3.0.3 at: `https://github.com/soot-oss/soot`

Table 5.5: Correlations between complexity metrics and test pass rates for each edit type. Bold figures indicate where the p-value associated with the correlation is < 0.05.

Metric	Line edits	Statement edits
Cyclomatic	**-0.164**	-0.028
Cyclomatic normalised	**-0.132**	0.074
JavaNCSS	-0.029	-0.078
JavaNCSS normalised	0.041	0.036
NPATH	-0.067	-0.039
NPATH normalised	-0.081	-0.021
Average def-use distance	0.063	0.030
Average def-use distance normalised	**0.253**	0.204
Median def-use distance	0.108	0.070
Median def-use distance normalised	**0.224**	**0.190**
Method Length	-0.074	-0.095

there are also three percent of the corpus of methods where the Java processing toolchain was unable to return an accurate linecount and, on the same method, Soot was unable to extract def-use dependencies. The distributions of pass-rates for line and statement edits for the 229 excluded methods are not significantly different (according to the Mann-Whitney two-tailed U-test) from those for the 1929 methods in the rest of the corpus.

Over the filtered results, we calculated the Pearson's correlation between test pass rate and the measures above. We also calculated correlations for average def-use distance, and for method length in lines. The results are given in Table 5.5. Normalised average and median def-use distance show a weak positive correlation with pass rates for both edit types, and cyclomatic complexity shows a weak negative correlation with test pass rates for line edits. However, both correlations are very weak, and none of the other measures show much of a relationship at all, so it would seem that none of these metrics reliably predict plasticity. It is possible that stronger correlations to code metrics are present at the level of individual projects. To test for this possibility we ran the same correlation analysis for each project with enough individual samples. The results of this analysis mirrored the findings for the entire corpus. A small proportion of project/metrics produced moderate to week correlations with a significant p-value but no features or patterns in the results provided paths to follow up.

Case Study: Looking for Rogues: The foregoing analysis shows only weak correlations between the above features and methods' plasticity. In this section, we briefly survey the characteristics of methods whose plasticity is more easily predicted (good-citizens) and those whose plasticity defies easy prediction (rogues). In particular, by identifying distinctive features of rogues we can identify promising ways to boost the skill of future predictive models of plasticity.

To split rogues from good citizens we built four linear predictive models to be able to identify methods that are:

Table 5.6: The four best predictive models derived from grid search using the variables for: normalized average def-use distance (*normAveDU*); normalized median def-use distance (*normMedDU*); and cyclomatic complexity. The best models depended only on *normAveDU* and *normMedDU*.

EditType	Predict Plastic	Predict Non-plastic
line	(1) $normMedDU >= 0.51$	(3) $normMedDU <= 0.2 \wedge normAveDU <= 0.3$
statement	(2) $normMedDU >= 0.53$	(4) $normMedDU <= 0.2 \wedge normAveDU <= 0.31$

1. highly plastic with respect to line edits,
2. highly plastic with respect to statement edits,
3. non-plastic with respect to line edits, and
4. non-plastic with respect to statement edits.

To build these models, we first ranked all methods according to their plasticity w.r.t single line and w.r.t single statement edits. Then we searched for the best versions of each model above scored according to the ratio:

$$model\_score = \frac{correct\_prediction\_count}{incorrect\_prediction\_count + 1}$$

where *correct_prediction_count* is the number of methods predicted to be plastic (non-plastic) and were actually in the top (bottom) quartile for plasticity. Similarly, the *incorrect_prediction_count* is the number of methods predicted to be plastic (non-plastic) but were actually in the bottom (top) quartile for plasticity.

A grid search (with a pitch of 0.01) was carried out over the three variables: normalized average def-use distance (*normAveDU*); normalized median def-use distance (*normMedDU*); and normalised cyclomatic complexity. The best four predictive models depended only on the first two variables and these models are shown in Table 5.6. We used these models to look for good citizens and rogue methods. For the task of predicting plastic methods, the rogue methods were those that were predicted to be plastic, by models 1) and 2) in Table 5.6, but were actually in the lowest quartile for plasticity. These, deceptively non-plastic methods (four methods for statement edits and five methods for lines with three in common — six unique in total) all had one or more of the following features. The number of methods exhibiting the features is shown in brackets after each listed feature.

- relatively short method length (two methods).
- long statements or inlined classes, whose presence induces a longer def-use distance (measured in lines) while the intra-method dependencies are actually very tight (two methods).
- references to class variables because of the implicit data and control dependencies that they introduce (two methods);
- calls to methods with side-effects (two methods);
- strong implicit control dependencies (three methods).

For the first point above, the average length of the deceptively non-plastic methods is 16 lines whereas the average length for the predictably plastic methods is 21 lines. The use of normalised def-use distances may be noisier on short methods because normalisation amplifies even very short absolute def-use distances.

Corresponding to the second point above, an example of a deceptively non-plastic method is `filterLine` in the baseTestRunner class of the junit4 project (`https://tinyurl.com/y356k3r5`). This method contains a multi-line string array which increases the nominal def-use distance when counted in lines when, in reality, the method has quite tight intra-method dependencies.

Corresponding to the third point above, another deceptive method is the `configure` method in the `StaticFilesConfiguration` class of the spark project (`https://tinyurl.com/yc7wwdyb`). This short method has few references to its parameters, but makes multiple changes to object state that introduce implicit inter-statement dependencies.

For the task of predicting non-plastic methods using models 3) and 4) in Table 5.6, there were only five unique rogues identified (all five were wrongly predicted by the statement-based model — two overlapping by the line-based model). These rogues were less distinct from the good citizens but seemed to exhibit localised dependencies that admit some re-ordering of lines or statements. The `extractObjectFromList` in the `ResultExtractor` class in mybatis (`https://tinyurl.com/2p97jadz`) is one example. This method contains an if-statement with inter-statement dependencies in each clause but some scope for re-ordering of guarded clauses. Another example is the `getParameterType` method in the `MapperAnnotationBuilder` in mybatis (`https://tinyurl.com/yc8xwhv5`), which has strong intra-method dependencies but has nested logic that might only be executed in exceptional circumstances — unless these circumstances are exercised in tests, destructive edits to the innermost edits will not be detected.

For the good citizens, the methods correctly predicted as plastic (20 unique methods — the same set for line and statement edits) exhibit few sequential dependencies — often consisting of long disjoint conditionals or case statements or initialisation of unconnected fields in structures. The good citizens correctly predicted as non-plastic (17 unique methods) exhibited very tight sequential dependencies.

Our broad observations of rogues, in particular, indicate ways in which we can refine (boost) our models to search more broadly for dependencies and adjust our metrics to account for shorter methods. Following the Anna Karenina principle, from Tolstoy:

> All happy families are alike; each unhappy family is unhappy in its own way.

We would expect that there would be a lot of diversity in ways in which methods can be rogues *w.r.t.* any model but the above features might serve as a useful starting point. In addition, there is a lot of potential for using a broader set of features and learning models to empirically derive models of

plasticity. Such models, even if they are of moderate skill, are a promising way to improve the efficacy of APM.

We can now answer RQ5: "Which features predict code's plasticity, its amenability to APM modulo a test suite?". We find that:

Answer to RQ5

No code measure we tested showed a strong correlation with test pass rate. Only cyclomatic complexity and normalised def-use show any correlation with pass rate, and it is weak. The presence of tight sequential dependencies also seems to offer some indication of plasticity.

Our earlier results (Section 5.5) revealed that pass rates fall off quickly as more edits are applied. We might improve pass rates by targeting edits to code regions on which they are more likely to work well (termed "plastic regions" by Harrand et al. (2019)). We have shown that cyclomatic complexity and normalised def-use show a small correlation with APM-ability, but it is not enough to reliably predict an edit's success. We might consider three possible explanations for this result.

First, there is a defining characteristic of APM-able code but our measures do not capture it. We have considered some basic code complexity measures as a basis for a simple linear model. Future work could consider the many other measures that have been applied in software fault prediction (Pandey et al. (2021)) that could be deployed; knots, nesting levels, number of unique operands, and counts of if-then or other structures are a small selection of candidates that could be tested. Higher order and non-linear models should also be considered as part of this work.

Second, it is possible that no set of code measures or rules can be defined to predict APM-ability, and this becomes more likely as additional measures and model types are considered as suggested above but still fail. We note that this kind of "I know it when I see it but I can't define it" problem is where artificial neural networks (ANNs) seem to work well in practice, and indeed ANNs have already shown some potential in software fault prediction (Li et al. (2017)). The trouble with ANNs is having sufficient training data. Larger scale experiments than those in our study are of course possible, but costly. An alternative might be to mine software repositories such as GitHub for examples of small edits, although labelling these would also prove challenging.

Third, it may simply be that the basic CDRS operators that we have considered produce a search space that is too large to thoroughly sample.obscuring any signal from the code measures. In addition to the motivation provided by the answers to our earlier RQs, this further justifies the design of 'fat' operators that reduce that search space (at the cost of limited applicability due to loss of universality); over these reduced search spaces the signal should be more evident. We conclude that measures to improve search by exploring more sophisticated, grammar-aware, and more human-like, operators; must be prioritised.

A complementary approach to increasing search effectiveness is to exploit equivalence classes in the edit-space by immediately pruning from search those edits that produce variants whose behaviour is equivalent to a variant that has already been tested. Varfix Wong et al. (2021) falls under this broad approach by avoiding repeated runs of execution paths by merging the program state induced by different edits in multi-edit patches. We expect that work to further exploit equivalence classes in the edit space has great potential to increase the power of search.

6 Threats to Validity

Our results generalise to the extent to which our corpus is representative. Our corpus has two parts: one we manually selected to be well-suited for our experimental setup and a second that we systematically selected. Although we only use 10, we have used a systematic approach to select the most popular, large, real-world projects, which represent a wide range of Java applications (see Table 4.1 and Table 4.2). Section 4.1 details our selection procedure. Other than our restriction to Java, we have no reason to believe our selection criteria introduced systematic bias. We note that our corpus is the largest used in a landscape study in the arena of automated program modification, both in total number of projects it contains (10) and the size of each project. Despite our focus on Java, comparison with previous work on C suggests that our results might generalise beyond one programming language.

A less standard threat to this study's external validity is our decision to narrow testing to only *hot* methods, as identified by profiling. This focus was necessary to make this landscape study feasible over large, real-world Java programs, due to the high computation cost of APM. Section 4.1 and Table 4.3 answer the question of whether the *hot* methods are representative of methods in general. We found no dramatic difference between *hot* methods and other methods within the corpus; while the differences were statistically significant, they all have small effect sizes (< 0.2). We also considered whether the high test coverage for *hot* methods might be skewed by very short methods such as getters and setters. We found this unlikely: removing getters and setters changed the coverage figures by less than one percentage point for all projects, and reduced the method size by a median of one line and one statement across all projects.

In general, a test suite underapproximates a program's specification and the degree of that underapproximation is usually unknown, even unknowable. Thus, testing can only approximately measure a program's correctness: A program can pass a test suite and still be arbitrarily incorrect. Such programs are said to overfit the test suite. Overfitting is a construct validity threat faced by all APM approaches that use testing as a measure of correctness. In an attempt to better approximate a program's actual specification, we could have employed ideas for enhancing the test suite (e.g. Ye et al. (2021); Xiong et al. (2018); Xin and Reiss (2017); Yang et al. (2017)). We decided against doing

so, because we can only speculate (and hope) that the new tests are consistent with the actual specification and to avoid the computational cost of running them. Like all other work in this space, we rely on the assumption that test-passing programs are unlikely to all overfit, as the fact of meeting the partial specification encoded in a test suite (witnessed by passing its tests), does not entail violating the rest of the specification. Furthermore, test-passing programs sample a smaller search space that all correct programs inhabit.

Concerning the question of whether our restriction to *hot* methods exacerbates this construct threat, we believe that we did not make the problem much worse, as our *hot* method focus restricts testing to *hot* method tests. As a consequence, our study does not run and cannot overfit tests for non-*hot* methods.

7 Related Work

To date, there has only been a handful of analyses of automated program modification search spaces Petke et al. (2019); Harrand et al. (2019) — this stands in stark contrast to the extensive analysis of such spaces in the field of meta-heuristic program synthesis Gulwani (2010). Petke et al. (2019)'s survey on search space landscapes in genetic improvement (including search-based program repair) identified only 14 papers, with none systematically exploring the search space of multi-edit patches for traditional mutation operators, nor comparing line and statement granular edits for Java. Our work thus fills this gap in the literature. Moreover, we formalise the cost of APM, and investigate characteristics of methods that are particularly plastic, *i.e.*, amenable to APM.

7.1 Code Plasticity

Our study focuses on analysing how plastic is Java code under the CDRS operators. Harrand et al. (2019) define plasticity as the "intrinsic capability at being changed to another code, while keeping functional correctness, with respect to a given test suite". Identification and exploration of plastic regions is important, as these, intuitively, represent the places where improvements to non-functional behaviour of software might be found (without changing software's functional behaviour as specified by its test suite). Moreover, Renzullo et al. (2018) show that such regions often form basis for an eventual repair. Although APR tools that generate multi-edit patches exist (Mechtaev et al. (2016); Saha et al. (2019)), in none of the work we found, do they analyse the whole search space of multi-edit CDRS sequences.

The closest work to ours is Harrand et al. (2019)'s study, who sampled the space of six Java projects (four from the commons-collection) to calculate neutral variant rates. They consider three mutation operators, COPY, DELETE and REPLACE, applied at statement granularity, and report NVRs (Equation (2.2)) of between 15.7% and 30% for single edits. This result is consistent with our

finding of NVRs between 9.6% and 30.2%. We much extend their analysis by considering two granularities (line and statement), an additional mutation operator (*i.e.*, SWAP), a larger corpus (systematically chosen), and the space of multi-edit patches. Moreover, we exhaustively analyse the search space of DELETE, the most effective operator (Section 7).

7.2 Size of APM landscapes

Due to the size of the APM search space with CDRS operators, exhaustive studies have only been conducted on toy programs, with a restricted mutation set. For instance, Langdon et al. (2017) conducted an exhaustive study, but they mutated binary comparison operators only. In particular, they used the triangle program, that, given three integers, outputs whether these form the edges of a triangle and, if so, whether that triangle is isosceles, scalene, or equilateral. With 6 comparison operators and overall 17 possible mutation points, this APM search space contains 6^{17} mutation points. This study shows the difficulty of exhaustively exploring APM landscapes.

More recently Wong et al. (2021) proposed *variational execution* to effectively reduce the cost of exploring edit spaces and realised in it VarFix. VarFix uses program transformations that simultaneously encode multiple patches (*i.e.* distinct edits) into Boolean-guarded paths. For each of the transformed program's tests, a variational execution engine Wong et al. (2018) then enumerates the distinct states induced by patched paths leveraging state-merging to avoid a state-explosion. The effectiveness of state-merging rests on the empirical observation that there are often few interactions in a program's state space between individual edits. Their approach succeeded in greatly reducing the number of test runs required to explore combinations of large number of edits. Because VarFix provides a means to speed up search space exploration, its contribution is orthogonal to the current work: its use might speed our landscape traversal but would not change any of the findings of this work.

7.3 APM Granularity

When considering APM landscapes, most research has focused on navigating the space of statement granular operators Petke et al. (2019). However, both line (*e.g.*, Langdon and Harman (2010)) and expression (*e.g.*, Haraldsson et al. (2017); Wen et al. (2018)) granular changes have proved effective at finding software improvements. It is worth noting that expression-granular changes vastly increase the search space, producing even more unviable program variants unless constraints and/or prioritisation strategies are introduced. Central to our work is our effort to exhaustively study CDRS within our resource constraints. Additional granularities would greatly expanded our search space and made our study shallower. Wen et al. (2018) resort to prioritisation strategies to narrow the search space for bug fixes, a tactic that runs counter to goal to

map and characterise the search landscape. This fact, coupled with the prominent use of line and statement granularity in the literature, is why we decided to focus solely on line and statement granularity.

The question of determining at which operator granularity APM is most effective has been tackled in the literature.

For example, Binkley et al. (2019) compared line and statement granularity in the observational slicing context. In particular, they compared a slicer that deletes lines of code with one that removes AST subtrees. They used 14 C and 6 Java programs, with less than 1.5k LoC each. They concluded that although the slices are identical in the majority of cases, the tree-based slices are occasionally larger. They also take longer to obtain. Interestingly, in our work we show that statement granular edits are actually more attractive than line granular edits in terms of effectiveness and time cost. We do, however, focus on smaller edits than commonly encountered in the slicing literature.

An et al. (2018) ran their program repair tool with two sets of APM operators (COPY, DELETE, and REPLACE), at line and at statement granularity, on a 3.6k LoC Python program. They conclude that the statement-granular edits are more effective, finding a correct fix for one more bug than the line-granular ones. However, statement-granular edits are less efficient. They produce more syntactically valid patches than the line granular ones, leading to more test case evaluations, thus taking longer to produce a patch. Their study, however, has a different focus to ours. Rather than direct bug fixes, we are interested in finding neutral program variants. Moreover, also taking compilation costs in Java into account, we find that line-granular edits waste 10% more computation resources than statament-granular ones.

7.4 APR Landscapes

In the field of automated program repair (APR), studies on APM search spaces, understandably, largely focused on finding bug fixes within them, rather than on NVRs. Below we summarise latest work in this direction.

Recently, Etemadi et al. (2022) studied *ca.* 55k commits to statically estimate whether a bug fix is within a search space of current APR tooling. They show that only 1.35% of those bug fixing commits lie in the search space of at least one current APR tool. In order to best choose which tool might be best fit for fixing bug in a given project, they thus proposed a light-weight approach for checking whether previous fixes lie in the search space of a given APR tool.

Along similar lines, Ginelli et al. (2022) empirically investigated the search space of an APR tool that only uses DELETE to generate candidate patches. The authors showed that, not only were candidate patches generated infrequently ($< 3\%$ of bugs), but also they did not correctly fix a given bug in 96% of cases. Of these incorrect fixes, they find 63% is due to the inadequacy of the test suite used. Unlike that work, we exhaustively enumerate the space of all

candidate DELETE operations over the subject program's *hot* methods, which have high test coverage (see Table 4.3).

Furthermore, Ahmad et al. (2022) recently sampled space of variants produced by four program repair tools, and used syntactic and semantic comparison measures. They conclude that searching in the right place is more important than searching broadly, over a semantically and/or syntactically diverse set of variants. This supports our decision to focus on *hot* methods. They suggest that a deeper understanding of how repairs are distributed throughout syntactic and semantic search spaces is needed.

Unlike the aforementioned work, we are not interested in the ability of individual tooling to find a bug fix. We investigate the landscape of primitive APM operators that are universal in the sense that they can produce any variant, paying particular attention to neutral variants. These variants can be exploited to improve both functional software properties, such as bug fixing Renzullo et al. (2018), and non-functional ones Petke et al. (2019).

APR work has largely focused on the Boolean result of test cases when investigating search landscapes. This creates landscapes with plateaus, separated by one test case failure. To address this critique, Yuan and Banzhaf (2020) and Bian et al. (2021) proposed fitness functions that take into account the types of test failures (*e.g.*, NullPointerException ranked lower than incorrect numerical output). Techniques, like this one, that smooth the landscape of the search space would benefit all forms of APM. Unfortunately, followup work has cast doubt on this direction, showing that it does not improve APR effectiveness in practice (Guizzo et al. (2021)). It remains to be seen whether this direction remains promising when relaxing the search away from directly finding patches to neutral variants.

8 Conclusion

Automated program modifications have proven successful in optimising various software properties, both functional, *e.g.*, fixing bugs, and non-functional, *e.g.*, decreasing software's runtime. However, the question of which edit operators should be applied to software and at what level of granularity, is largely unsolved. Therefore, in this work we investigated the Java program search space, to make a step towards answering that question. We have not only considered the effect of the traditional operators, used in genetic improvement, including search-based automated program repair, both at the statement and line-level, but also considered the largely unused SWAP operator and focused on the differences between programs to which GI and APR is applied to.

Our study on the largest APM benchmark to-date shows that statement-level operators yield more program variants that still pass test suites, though among the variants that compile the rate of those that pass is higher for the line-level operators. We also show that the time required to apply and evaluate line edits is much wasted on ineffective patches. Overall, we thus recommend the use of statement-level edits over line-level ones in future APM work.

Similarly to bug fixing work, the DELETE operator is the most effective, yet SWAP, as implemented in the Gin toolbox we use, is second-best effective. Moreover, even though the larger the set of random edits applied to software, the less likely it is to pass its test suite, these edit sequences can also include self-repairs, that recover from previous failures. Perhaps those patterns could be extracted to produce multi-edit operators.

Finally, we saw no correlation between test pass rate and several traditional code metrics, leaving the question of which methods are more amenable to APM, and thus have potential to be effectively improved, yet to be solved.

Declarations

Data Availability Statement All our scripts and data are available in the following repository: `https://github.com/automatedprogrammodification/automatedprogrammodification/`.

Funding and Copyright This work was funded by the UKRI EPSRC grants EP/P023991/1 and EP/J017515/1; Carnegie Trust grant RIG008300; Australian Research Council projects DE160100850, DP200102364, and DP210102670, and by gifts from Facebook and Google. For the purpose of open access, the authors have applied a Creative Commons Attribution (CC BY) license to any Accepted Manuscript version arising.

Conflict of interest The authors declare that they have no conflict of interest.

References

Ahmad H, Cashin P, Forrest S, Weimer W (2022) Digging into semantics: Where do search-based software repair methods search? URL `https://web.eecs.umich.edu/~weimerw/p/weimer-ppsn2022.pdf`

An G, Kim J, Yoo S (2018) Comparing line and AST granularity level for program repair using pyggi. In: Petke J, Stolee KT, Langdon WB, Weimer W (eds) Proceedings of the 4th International Genetic Improvement Workshop, GI@ICSE 2018, Gothenburg, Sweden, June 2, 2018, ACM, pp 19–26, DOI 10.1145/3194810.3194814, URL `https://doi.org/10.1145/3194810.3194814`

Barr ET, Brun Y, Devanbu PT, Harman M, Sarro F (2014) The plastic surgery hypothesis. In: Cheung S, Orso A, Storey MD (eds) Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014, ACM, pp 306–317, DOI 10.1145/2635868.2635898, URL `https://doi.org/10.1145/2635868.2635898`

Bian Z, Blot A, Petke J (2021) Refining fitness functions for search-based program repair. In: 2nd IEEE/ACM International Workshop on Automated Program Repair, APR@ICSE 2021, Madrid, Spain, June 1, 2021, IEEE, pp 1–8, DOI 10.1109/APR52552.2021.00008, URL `https://doi.org/10.1109/APR52552.2021.00008`

Binkley DW, Gold N, Islam SS, Krinke J, Yoo S (2019) A comparison of tree- and line-oriented observational slicing. Empirical Software Engineering 24(5):3077–3113, DOI 10.1007/s10664-018-9675-9, URL `https://doi.org/10.1007/s10664-018-9675-9`

Blot A, Petke J (2021) Empirical comparison of search heuristics for genetic improvement of software. IEEE Trans Evol Comput 25(5):1001–1011, DOI 10.1109/TEVC.2021.3070271, URL https://doi.org/10.1109/TEVC.2021.3070271

Brownlee AEI, Petke J, Alexander B, Barr ET, Wagner M, White DR (2019) Gin: genetic improvement research made easy. In: Auger A, Stützle T (eds) Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2019, Prague, Czech Republic, July 13-17, 2019, ACM, pp 985–993, DOI 10.1145/3321707.3321841, URL https://doi.org/10.1145/3321707.3321841

Brownlee AEI, Petke J, Rasburn AF (2020) Injecting shortcuts for faster running java code. In: IEEE Congress on Evolutionary Computation, CEC 2020, Glasgow, United Kingdom, July 19-24, 2020, IEEE, pp 1–8, DOI 10.1109/CEC48606.2020.9185708, URL https://doi.org/10.1109/CEC48606.2020.9185708

Callan J, Krauss O, Petke J, Sarro F (2022) How do android developers improve non-functional properties of software? Empir Softw Eng 27(5):113, DOI 10.1007/s10664-022-10137-2, URL https://doi.org/10.1007/s10664-022-10137-2

Coelho R, Almeida L, Gousios G, van Deursen A, Treude C (2017) Exception handling bug hazards in android - results from a mining study and an exploratory survey. Empir Softw Eng 22(3):1264–1304, DOI 10.1007/s10664-016-9443-7, URL https://doi.org/10.1007/s10664-016-9443-7

Cohen J (1969) Statistical Power Analysis for the Behavioral Sciences. Academic Press, NY

Etemadi K, Tarighat N, Yadav S, Martinez M, Monperrus M (2022) Estimating the potential of program repair search spaces with commit analysis. J Syst Softw 188:111263, DOI 10.1016/j.jss.2022.111263, URL https://doi.org/10.1016/j.jss.2022.111263

Gazzola L, Micucci D, Mariani L (2019) Automatic software repair: A survey. IEEE Trans Software Eng 45(1):34–67, DOI 10.1109/TSE.2017.2755013, URL https://doi.org/10.1109/TSE.2017.2755013

Gewirtz P (1996) On "I know it when I see it". The Yale Law Journal 105(4):1023–1047, DOI 10.2307/797245, URL http://www.jstor.org/stable/797245

Ginelli D, Martinez M, Mariani L, Monperrus M (2022) A comprehensive study of code-removal patches in automated program repair. DOI 10.1007/s10664-021-10100-7, URL https://doi.org/10.1007/s10664-021-10100-7

Guizzo G, Blot A, Callan J, Petke J, Sarro F (2021) Refining fitness functions for search-based automated program repair - A case study with ARJA and arja-e. In: O'Reilly U, Devroey X (eds) Search-Based Software Engineering - 13th International Symposium, SSBSE 2021, Bari, Italy, October 11-12, 2021, Proceedings, Springer, Lecture Notes in Computer Science, vol 12914, pp 159–165, DOI 10.1007/978-3-030-88106-1_12, URL https://doi.org/10.1007/978-3-030-88106-1_12

Gulwani S (2010) Dimensions in program synthesis. In: Kutsia T, Schreiner W, Fernández M (eds) Proceedings of the 12th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, July 26-28, 2010, Hagenberg, Austria, ACM, pp 13–24, DOI 10.1145/1836089.1836091, URL https://doi.org/10.1145/1836089.1836091

Haraldsson SO, Woodward JR, Brownlee AEI, Siggeirsdottir K (2017) Fixing bugs in your sleep: how genetic improvement became an overnight success. In: Bosman PAN (ed) Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings, ACM, pp 1513–1520, DOI 10.1145/3067695.3082517, URL http://doi.acm.org/10.1145/3067695.3082517

Harrand N, Allier S, Rodriguez-Cancio M, Monperrus M, Baudry B (2019) A journey among Java neutral program variants. Genetic Programming and Evolvable Machines 20(4):531–580, DOI 10.1007/s10710-019-09355-3, URL https://doi.org/10.1007/s10710-019-09355-3

Hassan F, Bansal C, Nagappan N, Zimmermann T, Awadallah AH (2020) An empirical study of software exceptions in the field using search logs. In: Baldassarre MT, Lanubile F, Kalinowski M, Sarro F (eds) ESEM '20: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 5-7, 2020, ACM, pp 4:1–4:12, DOI 10.1145/3382494.3410692, URL https://doi.org/10.1145/3382494.3410692

Kirbas S, Windels E, McBello O, Kells K, Pagano MW, Szalanski R, Nowack V, Winter ER, Counsell S, Bowes D, Hall T, Haraldsson S, Woodward JR (2021) On the introduction of automatic program repair in bloomberg. IEEE Softw 38(4):43–51, DOI 10.1109/MS.2021.3071086, URL https://doi.org/10.1109/MS.2021.3071086

Koyuncu A, Liu K, Bissyandé TF, Kim D, Klein J, Monperrus M, Traon YL (2020) Fixminer: Mining relevant fix patterns for automated program repair. Empir Softw Eng 25(3):1980–2024, DOI 10.1007/s10664-019-09780-z, URL https://doi.org/10.1007/s10664-019-09780-z

Langdon WB, Harman M (2010) Evolving a CUDA kernel from an nvidia template. In: Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2010, Barcelona, Spain, 18-23 July 2010, IEEE, pp 1–8, DOI 10.1109/CEC.2010.5585922, URL https://doi.org/10.1109/CEC.2010.5585922

Langdon WB, Lam BYH (2017) Genetically improved barracuda. BioData Min 10(1):28:1–28:11, DOI 10.1186/s13040-017-0149-1, URL https://doi.org/10.1186/s13040-017-0149-1

Langdon WB, Petke J (2017) Software is not fragile. In: Bourgine P, Collet P, Parrend P (eds) First Complex Systems Digital Campus World E-Conference 2015, Springer, Cham, pp 203–211

Langdon WB, Lam BYH, Petke J, Harman M (2015) Improving CUDA DNA analysis software with genetic programming. In: Silva S, Esparcia-Alcázar AI (eds) Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2015, Madrid, Spain, July 11-15, 2015, ACM, pp 1063–1070, DOI 10.1145/2739480.2754652, URL http://doi.acm.org/10.1145/2739480.2754652

Langdon WB, Veerapen N, Ochoa G (2017) Visualising the search landscape of the triangle program. In: McDermott J, Castelli M, Sekanina L, Haasdijk E, García-Sánchez P (eds) Genetic Programming - 20th European Conference, EuroGP 2017, Amsterdam, The Netherlands, April 19-21, 2017, Proceedings, Lecture Notes in Computer Science, vol 10196, pp 96–113, DOI 10.1007/978-3-319-55696-3_7, URL https://doi.org/10.1007/978-3-319-55696-3_7

Le Goues C, Weimer W, Forrest S (2012) Representations and operators for improving evolutionary software repair. In: Soule T, Moore JH (eds) Genetic and Evolutionary Computation Conference, GECCO '12, Philadelphia, PA, USA, July 7-11, 2012, ACM, pp 959–966, DOI 10.1145/2330163.2330296, URL http://doi.acm.org/10.1145/2330163.2330296

Li J, He P, Zhu J, Lyu MR (2017) Software defect prediction via convolutional neural network. In: 2017 IEEE International Conference on Software Quality, Reliability and Security, QRS 2017, Prague, Czech Republic, July 25-29, 2017, IEEE, pp 318–328, DOI 10.1109/QRS.2017.42, URL https://doi.org/10.1109/QRS.2017.42

Liu K, Koyuncu A, Kim D, Bissyandé TF (2019) Tbar: revisiting template-based automated program repair. In: Zhang D, Møller A (eds) Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019, ACM, pp 31–42, DOI 10.1145/3293882.3330577, URL https://doi.org/10.1145/3293882.3330577

Marginean A, Bader J, Chandra S, Harman M, Jia Y, Mao K, Mols A, Scott A (2019) Sapfix: automated end-to-end repair at scale. In: Sharp H, Whalen M (eds) Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019, IEEE / ACM, pp 269–278, DOI 10.1109/ICSE-SEIP.2019.00039, URL https://doi.org/10.1109/ICSE-SEIP.2019.00039

Martinez M, Monperrus M (2015) Mining software repair models for reasoning on the search space of automated program fixing. Empirical Software Engineering 20(1):176–205, DOI 10.1007/s10664-013-9282-8, URL https://doi.org/10.1007/s10664-013-9282-8

Martinez M, Monperrus M (2018) Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In: Colanzi TE, McMinn P (eds) Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings, Springer, Lecture Notes in Computer Science, vol 11036, pp 65–86, DOI 10.1007/978-3-319-99241-9_3, URL https://doi.org/10.1007/978-3-319-99241-9_3

Mechtaev S, Yi J, Roychoudhury A (2016) Angelix: scalable multiline program patch synthesis via symbolic analysis. In: Dillon LK, Visser W, Williams LA (eds) Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016, ACM, pp 691–701, DOI 10.1145/2884781.2884807, URL https://doi.org/10.1145/2884781.2884807

Meyvis T, Yoon H (2021) Adding is favoured over subtracting in problem solving. nature 592, DOI 10.1038/d41586-021-00592-0, URL https://doi.org/10.1038/d41586-021-00592-0

Monperrus M (2014) A critical review of "automatic patch generation learned from human-written patches": essay on the problem statement and the evaluation of automatic software repair. In: Jalote P, Briand LC, van der Hoek A (eds) 36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014, ACM, pp 234–242, DOI 10.1145/2568225.2568324, URL https://doi.org/10.1145/2568225.2568324

Nejmeh BA (1988) Npath: A measure of execution path complexity and its applications. Commun ACM 31(2):188–200, DOI 10.1145/42372.42379, URL https://doi.org/10.1145/42372.42379

Ochoa G, Tomassini M, Vérel S, Darabos C (2008) A study of NK landscapes' basins and local optima networks. In: Ryan C, Keijzer M (eds) Genetic and Evolutionary Computation Conference, GECCO 2008, Proceedings, Atlanta, GA, USA, July 12-16, 2008, ACM, pp 555–562, DOI 10.1145/1389095.1389204, URL http://doi.acm.org/10.1145/1389095.1389204

Pandey SK, Mishra RB, Tripathi AK (2021) Machine learning based methods for software fault prediction: A survey. Expert Syst Appl 172:114595, DOI 10.1016/j.eswa.2021.114595, URL https://doi.org/10.1016/j.eswa.2021.114595

Petke J (2017) New operators for non-functional genetic improvement. In: Bosman PAN (ed) Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings, ACM, pp 1541–1542, DOI 10.1145/3067695.3082520, URL http://doi.acm.org/10.1145/3067695.3082520

Petke J, Haraldsson SO, Harman M, Langdon WB, White DR, Woodward JR (2018) Genetic improvement of software: A comprehensive survey. IEEE Trans Evol Comput 22(3):415–432, DOI 10.1109/TEVC.2017.2693219, URL https://doi.org/10.1109/TEVC.2017.2693219

Petke J, Alexander B, Barr ET, Brownlee AEI, Wagner M, White DR (2019) A survey of genetic improvement search spaces. In: López-Ibáñez M, Auger A, Stützle T (eds) Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2019, Prague, Czech Republic, July 13-17, 2019, ACM, pp 1715–1721, DOI 10.1145/3319619.3326870, URL https://doi.org/10.1145/3319619.3326870

Qi Z, Long F, Achour S, Rinard MC (2015) An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In: Young M, Xie T (eds) Proceedings of the 2015 International Symposium on Software Testing and Analysis, ISSTA 2015, Baltimore, MD, USA, July 12-17, 2015, ACM, pp 24–36, DOI 10.1145/2771783.2771791, URL https://doi.org/10.1145/2771783.2771791

Reeves CR (1999) Fitness landscapes and evolutionary algorithms. In: Fonlupt C, Hao J, Lutton E, Ronald EMA, Schoenauer M (eds) Artificial Evolution, 4th European Conference, AE'99, Dunkerque, France, November 3-5, 1999, Selected Papers, Springer, Lecture Notes in Computer Science, vol 1829, pp 3–20, DOI 10.1007/10721187_1, URL https://doi.org/10.1007/10721187_1

Reidys CM, Stadler PF (2002) Combinatorial landscapes. SIAM Review 44(1):3–54, DOI 10.1137/S0036144501395952, URL https://doi.org/10.1137/S0036144501395952

Renzullo J, Weimer W, Moses ME, Forrest S (2018) Neutrality and epistasis in program space. In: Petke J, Stolee KT, Langdon WB, Weimer W (eds) Proceedings of the 4th International Genetic Improvement Workshop, GI@ICSE 2018, Gothenburg, Sweden, June 2, 2018, ACM, pp 1–8, DOI 10.1145/3194810.3194812, URL https://doi.org/10.1145/3194810.3194812

Saha S, Saha RK, Prasad MR (2019) Harnessing evolution for multi-hunk program repair. In: Atlee JM, Bultan T, Whittle J (eds) Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019, IEEE /

ACM, pp 13–24, DOI 10.1109/ICSE.2019.00020, URL https://doi.org/10.1109/ICSE.2019.00020

Schulte EM, Fry ZP, Fast E, Weimer W, Forrest S (2014) Software mutational robustness. Genetic Programming and Evolvable Machines 15(3):281–312, DOI 10.1007/s10710-013-9195-8, URL https://doi.org/10.1007/s10710-013-9195-8

Smigielska M, Blot A, Petke J (2021) Uniform edit selection for genetic improvement: Empirical analysis of mutation operator efficacy. In: 10th IEEE/ACM International Workshop on Genetic Improvement, GI@ICSE 2021, Madrid, Spain, May 30, 2021, IEEE, pp 1–8, DOI 10.1109/GI52543.2021.00009, URL https://doi.org/10.1109/GI52543.2021.00009

Soto M, Goues CL (2018) Using a probabilistic model to predict bug fixes. In: Oliveto R, Penta MD, Shepherd DC (eds) 25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018, IEEE Computer Society, pp 221–231, DOI 10.1109/SANER.2018.8330211, URL https://doi.org/10.1109/SANER.2018.8330211

Wen M, Chen J, Wu R, Hao D, Cheung S (2018) Context-aware patch generation for better automated program repair. In: Chaudron M, Crnkovic I, Chechik M, Harman M (eds) Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, ACM, pp 1–11, DOI 10.1145/3180155.3180233, URL https://doi.org/10.1145/3180155.3180233

Wong C, Meinicke J, Lazarek L, Kästner C (2018) Faster variational execution with transparent bytecode transformation. Proc ACM Program Lang 2(OOPSLA):117:1–117:30, DOI 10.1145/3276487, URL https://doi.org/10.1145/3276487

Wong C, Santiesteban P, Kästner C, Goues CL (2021) Varfix: balancing edit expressiveness and search effectiveness in automated program repair. In: Spinellis D, Gousios G, Chechik M, Penta MD (eds) ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021, ACM, pp 354–366, DOI 10.1145/3468264.3468600, URL https://doi.org/10.1145/3468264.3468600

Xin Q, Reiss SP (2017) Identifying test-suite-overfitted patches through test case generation. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Association for Computing Machinery, New York, NY, USA, ISSTA 2017, p 226–236, DOI 10.1145/3092703.3092718, URL https://doi.org/10.1145/3092703.3092718

Xiong Y, Liu X, Zeng M, Zhang L, Huang G (2018) Identifying patch correctness in test-based program repair. In: Proceedings of the 40th International Conference on Software Engineering, Association for Computing Machinery, New York, NY, USA, ICSE '18, p 789–799, DOI 10.1145/3180155.3180182, URL https://doi.org/10.1145/3180155.3180182

Yang J, Zhikhartsev A, Liu Y, Tan L (2017) Better test cases for better automated program repair. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, ESEC/FSE 2017, p 831–841, DOI 10.1145/3106237.3106274, URL https://doi.org/10.1145/3106237.3106274

Ye H, Martinez M, Monperrus M (2021) Automated patch assessment for program repair at scale. Empir Softw Eng 26(2):20, DOI 10.1007/s10664-020-09920-w, URL https://doi.org/10.1007/s10664-020-09920-w

Yuan Y, Banzhaf W (2020) ARJA: automated repair of java programs via multi-objective genetic programming. IEEE Trans Software Eng 46(10):1040–1067, DOI 10.1109/TSE.2018.2874648, URL https://doi.org/10.1109/TSE.2018.2874648

Zhong H, Su Z (2015) An empirical study on real bug fixes. In: Bertolino A, Canfora G, Elbaum SG (eds) 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1, IEEE Computer Society, pp 913–923, DOI 10.1109/ICSE.2015.101, URL https://doi.org/10.1109/ICSE.2015.101