Injecting Shortcuts for Faster Running Java Code

Alexander E.I. Brownlee
Computing Science and Mathematics
University of Stirling
Scotland, UK
sbr@cs.stir.ac.uk

Justyna Petke
Department of Computer Science
University College London
London, UK
j.petke@ucl.ac.uk

Anna F. Rasburn
Computing Science and Mathematics
University of Stirling
Scotland, UK

***Abstract*—Genetic Improvement of software applies search methods to existing software to improve the target program in some way. Impressive results have been achieved, including substantial speedups, using simple operations that replace, swap and delete lines or statements within the code. Often this is achieved by specialising code, removing parts that are unnecessary for particular use-cases. Previous work has shown that there is a great deal of potential in targeting more specialised operations that modify the code to achieve the same functionality in a different way.**

We propose six new edit types for Genetic Improvement of Java software, based on the insertion of break, continue and return statements. The idea is to add shortcuts that allow parts of the program to be skipped in order to speed it up. 10 000 randomly-generated instances of each edit were applied to three open-source applications taken from GitHub. The key findings are: (1) compilation rates for inserted statements without surrounding "if" statements are 1.3–18.3%; (2) edits where the inserted statement is embedded within an "if" have compilation rates of 3.2–55.8%; (3) of those that compiled, all 6 edits have a high rate of passing tests (Neutral Variant Rate), >60% in all but one case, and so have the potential to be performance improving edits. Finally, a preliminary experiment based on local search shows how these edits might be used in practice.

***Index Terms*—Genetic Improvement, GI, Search-Based Software Engineering, SBSE**

I. Introduction

Genetic Improvement of software [1] (GI) is a rapidly expanding area of research. GI uses automated search to improve existing software. It has been used for the purpose of bug fixing (e.g. [2]), runtime improvement (e.g. [3]), optimisation of energy (e.g. [4]) and memory consumption (e.g. [5]), among others. Changes evolved by GI have been incorporated into development [6] and GI-based repair has been incorporated into software development process [7]. Most of these impressive results have been achieved by making relatively simple modifications or *edits* to source code: some combinations of moving, copying or deleting statements or lines. This is rooted in the *plastic surgery hypothesis* [8], that shows the building blocks of the most common human-made patches to code were already present elsewhere in the program.

In the field of automated program repair (APR) there has been more work devoted to finding more efficient mutation operators. In particular, templates have been used, that have been either based on human-evolved patches [9], [10] or abstracted from fix operations at the abstract syntax tree level [11]. More fine-grained operators have also been tried, for example, at the expression level [7], or custom ones [12].

Much less work has been devoted to operators for improvement of non-functional properties [13]. The only exception being work on exchanging Java Collections [5], [14] and tuning of parameters embedded in code (so-called deep parameter tuning [15]).

Nevertheless, both within the GI and APR literature, *delete* seems to be one of the most successful operators [16]. This suggests that in real-world code there are often sections of redundant code, particularly when limiting runs to specific distributions of input data or use-cases. A related field, approximate computing [17], attempts to find trade-offs between accuracy and resource consumption, underpinned by the idea that not every line or statement needs executed every time. This has led to approaches such as loop perforation [18] (skipping some loop iterations). This background motivates finding other approaches to create shortcuts in code: new ways to change control flow while retaining functionality with respect to a software oracle (the original working program) or a test suite that captures the intended behaviour. Edits that insert a `return`, `continue` or `break` statement represent another way to skip sections of the code.

A recent study [16] explored in more depth the concept of *neutral program variants*; i.e. implementations of a program that are equivalent with respect to the test suite. They suggest that where there are many neutral variants, there is a greater chance of finding functional versions of the program that offer some kind of improvement over the original (such as decreased run time). They found that edits that add `return` statements at random locations rarely produced neutral variants, but edits adding `if` statements produced a high number of neutral variants (perhaps to be expected as the condition means they do not fire as often). This leads us to consider whether edits that insert `break` statements within `if` statements will represent a sweet spot: more likely to produce neutral variants and more likely to offer some kind of improvement.

Thus, the goal of this paper is to propose and test new edits for GI of Java code that add early `break`, `continue` or `return` statements, within and without surrounding `if` statements. These edits are tested on three popular open source projects to determine how brittle the code is with respect to these edits, and whether they can realise any improvements to the code in terms of run time.

The edits are implemented and tested in the *Gin* framework [19], [20]. Gin is a lightweight toolkit for experimentation in GI for Java projects. It provides utilities for: parsing and manipulating Java source code as lines of text or as an abstract syntax tree (using JavaParser); modifying, compiling and running test suites of target Java projects implemented using either Gradle or Maven build tools; profiling of a target project to identify hotspots; and sampling the space of possible edits with respect to a given project. The edits proposed in this paper will be made publicly available via the Gin project on GitHub at https://github.com/gintool/gin.

The paper begins in Section II by defining the proposed operators, then describes our experimental procedure in Section III. We then provide our experimental results in Sections IV to VI, before comparing our work to the relevant literature in Section VII. We then draw our conclusions and suggest future work in Section VIII.

II. Operators

A. Basic approach

There are six edit operators. Each is created against a target class $\mathbb{C}$ and a target method $\mathbb{M}$ within $\mathbb{C}$. For each, a block statement s and an insertion point p within s are selected uniformly at random from all block statements in $\mathbb{M}$. One of the six statements from the list below is then inserted at p. The six statements available are:

1) $\mathcal{B}$: **`break;`**
2) $\mathcal{C}$: **`continue;`**
3) $\mathcal{R}$: **`return;`**
4) $\mathcal{B}_{if}$: **`if`** `(a)` **`break;`**
5) $\mathcal{C}_{if}$: **`if`** `(a)` **`continue;`**
6) $\mathcal{R}_{if}$: **`if`** `(a)` **`return;`**

In the above, a can take one of two forms. An in-scope primitive variable v is chosen at random following the procedure noted below. If v is of type **`boolean`** then a is one of either `v` or `!v` (chosen at random). Otherwise, if v is of any other primitive type (i.e. a number or character) then a takes the form `v # 0`, where, `#` is one of the binary operators `<`, `<=`, `==`, `>=`, `>` (again chosen at random).

For simplicity, no attempt is made to detect a variable for the inserted return statements, or a label for break/continue. Neither is an attempt made to target $\mathcal{R}$ and $\mathcal{R}_{if}$ at only void methods, or **`break`** / **`continue`** at blocks where these are applicable. Of course, this will result in more failed compilations, but edits where this is the case can quickly be discarded. (The more problematic case is edits that compile but cause the tests to fail, as this requires the time cost of running the test suite).

B. Detecting local variables

Gin makes use of JavaParser[1] to construct an Abstract Syntax Tree (AST) representing the target source code. We adopt the following approach to find in-scope variables for a given target insertion point, a statement node s in the AST. (The inserted statement will be immediately before the insertion point). From s, we walk recursively back up the JavaParser AST, looking for any variable, field or parameter declarations, until we reach the containing class of s. The procedure is replicated in Figure 1.

```
{Input: Insertion point s}
n ← s
V ← ∅
while n.hasParent() do
  p ← p.getParent()
  for all c ← p.getChildNodes() do
    if c == n then
      break
    else
      if c.isVariableDeclarationExpr() or
      c.isParameter() or c.isFieldDeclaration()
      then
        if c.isPrimitiveType() then
          V ← c
        end if
      end if
    end if
  end for
  n ← p
end while
return V
```

Fig. 1: Algorithm to find a list of in-scope variables V at insertion point represented by AST statement node s

III. Experiments

In our experiments, we applied the edits to three publicly available projects from GitHub:

- jCodec (0.2.3)[2] (135k LoC)
- spark (2.7.2)[3] (15k LoC)
- spatial4j (0.7)[4] (14k LoC)

Gin retrieves the unit tests specified by the Maven build script for the target project. In this case, we have run Gin's profiler, a wrapper for the `hprof` tool[5], to determine the hot methods and the unit tests that make calls to them (this is achieved by sampling the call stack as the unit tests are running; so it not just limited to direct calls from the unit tests). Gin compiles the updated code, runs each unit test, and for each test, retrieves the test result (pass with time taken or fail with failure reason). The unit tests provided with each project were found by the JaCoCo tool[6] to have the following statement coverage over the entire project: 34% for jCodec, 60% for spark, and 72% for spatial4j.

Gin's profiler tool was applied to the test suite for the test applications. The full profiler output (list of target methods,

[1] http://www.javaparser.org

[2] https://github.com/jcodec/jcodec
[3] https://github.com/perwendel/spark
[4] https://github.com/locationtech/spatial4j
[5] https://docs.oracle.com/javase/7/docs/technotes/samples/hprof.html
[6] https://www.eclemma.org/jacoco/

tests calling them, and count of the number of times the profiler detected them on the call stack) is available at the URL provided at the end of the paper. The total number of target methods found for each project were: jCodec: 477; spark: 56; spatial4j: 77. For these targeted methods, test case coverage was 88% for jCodec, 90% for spark, and 93% for spatial4j.

We also report the neutral variant rates (NVRs) as defined in Equation 1.

$$NVR = \frac{\#ProgramVariantsPassingAllTests}{\#ProgramVariantsThatCompile} \quad (1)$$

Our experiments followed three stages, outlined below.

A. Enumeration

We begin with an enumeration: for every possible insertion point in the target methods identified by the profile, we apply each of the three new edits $\mathcal{B}$ (insert **`break;`**), $\mathcal{C}$ (insert **`continue;`**) and $\mathcal{R}$ (insert **`return;`**). The resulting three variants of the code are then compiled, and if this is successful, the unit tests that exercise the respective target methods are called. We report the number of edits generated for each project, as well as the fraction of edits that compiled, and the fraction of those that then also passed the tests.

The other three proposed new edits, $\mathcal{B}_{if}$, $\mathcal{C}_{if}$ and $\mathcal{R}_{if}$, were omitted from this stage because the possible conditions (choice of variable, and choice of comparison) meant the search space was excessively large for an exhaustive exploration.

B. Random Sampling

For each of the 6 edit types, 10 000 instances of the edit were sampled uniformly at random, over the space of all target methods identified by the profiler. Again, these were compiled and run on the corresponding unit tests, and we report the number of edits generated for each project, as well as the fraction of edits that compiled, and the fraction of those that then also passed the tests.

C. Local Search

As a proof-of-concept, once the sampling was completed, we were able to identify target methods with a relatively high neutral variant rate. Such regions were termed *plastic* in [16]; parts of the program where changes are less likely to cause test failures and so are more amenable to editing.

We rank-ordered the target methods according to the results from the profiler (i.e., most often called during testing, and thus potentially fruitful areas to achieve speed up). From this list, we omitted any target methods where no neutral variants were generated during the sampling process (i.e. those methods where edits tended to break something). From the resulting list, we then took the top 5 methods and applied a simple hill-climbing algorithm.

The hill-climber starts with an empty patch. At each iteration it chooses at random to either remove an edit chosen at random from the patch, or add an edit to the patch, sampled at random from the potential insertion points and the 6 edit types proposed here plus the replace/delete/copy/swap statement edits provided with Gin. The resulting patch is then evaluated by compiling, then running it 10 times on the relevant unit tests. If the modified code fails to compile, or the average run time does not represent an improvement, it is discarded. Otherwise, the patch is retained. The procedure repeats until 1000 patches have been evaluated.

A warm-up run of 10 repeats of the test suite for the target program was carried out before the local search runs as an attempt to reduce uncertainty in the run times due to memory caching.

TABLE I: Results of enumeration experiments. Each type of edit was applied to all possible insertion points in the target methods; *count* gives the total number of edits applied.

Project	Edit	Count	#Compile	CR%	#Passing	NVR%
jCodec	$\mathcal{B}$	6604	555	8.4	209	37.7
	$\mathcal{C}$	6604	555	8.4	527	95.0
	$\mathcal{R}$	6604	1045	15.8	678	64.9
spark	$\mathcal{B}$	694	29	4.2	25	86.2
	$\mathcal{C}$	694	29	4.2	28	96.6
	$\mathcal{R}$	694	64	9.2	57	89.1
spatial4j	$\mathcal{B}$	723	19	2.6	10	52.6
	$\mathcal{C}$	723	19	2.6	16	84.2
	$\mathcal{R}$	723	44	6.1	40	90.9

IV. Results: Enumeration

The results of the enumeration experiment are given in Table I and summarised in Figures 2 and 3. The compilation rates for the 3 simpler edits ($\mathcal{B}$, $\mathcal{C}$ and $\mathcal{R}$) are low across all projects, never exceeding 16%. $\mathcal{R}$ is consistently higher than the other two; given that a **`return`** can appear anywhere within a **`void`** method's body whereas **`break`** and **`continue`** have a narrower range of possible locations, this seems reasonable. The low compilation rates can also be partly attributed to Java failing to compile where 'unreachable code' is detected, such as after a **`return`** statement.

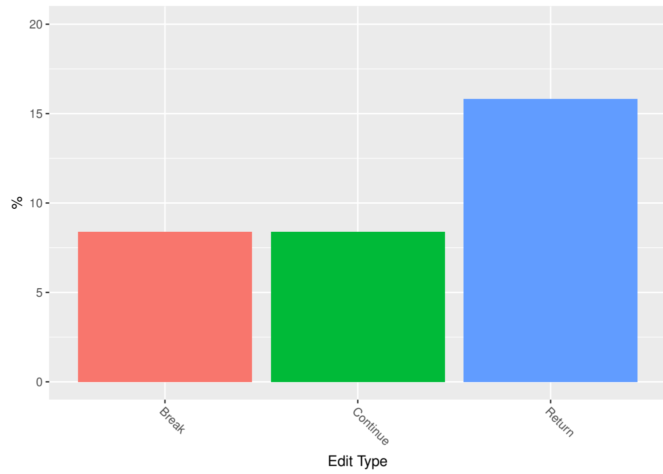
There is no surprise that compilation rates for $\mathcal{B}$ and $\mathcal{C}$ are the same: the only situation where **`break`** is allowed, and **`continue`** is not, is in a **`switch`** statement (and none of the target methods contain switches).

Of the edits that compiled successfully, a large fraction passed the unit tests (in fact, the majority for all except $\mathcal{B}$ in jCodec). This is similar to the neutral variant rates in previous work [16], where for the three traditional operators (including **`delete`**) NVRs were between 15.7% and 30%, while for the three proposed operators in the work (i.e., **`loop flip`**, **`add method invocation`** and **`swap subtype`**), the rates were between 58% and 73%.

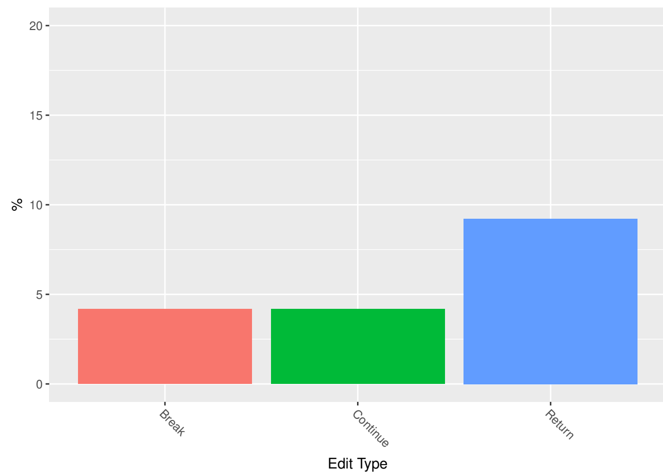
These results show that there is some promise in edits that insert **`break`**, **`continue`** and **`return`** statements; where compilable code results from their insertion, it seems likely to still pass the tests.

V. Results: Sampling

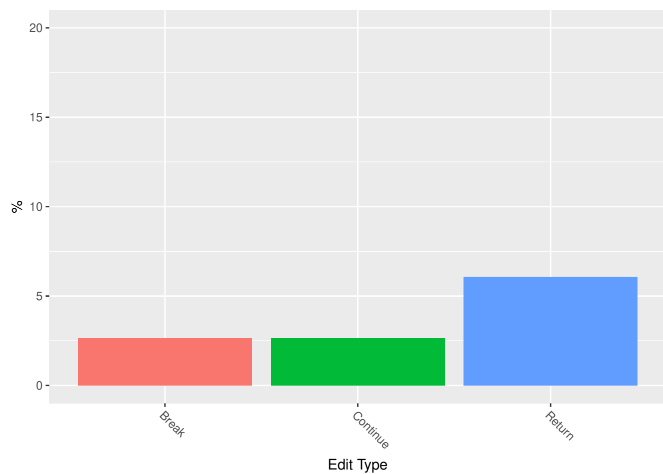
The results for the sampling experiment are given in Table II. As there are 10 000 edits of each type, there is some



(a) jCodec

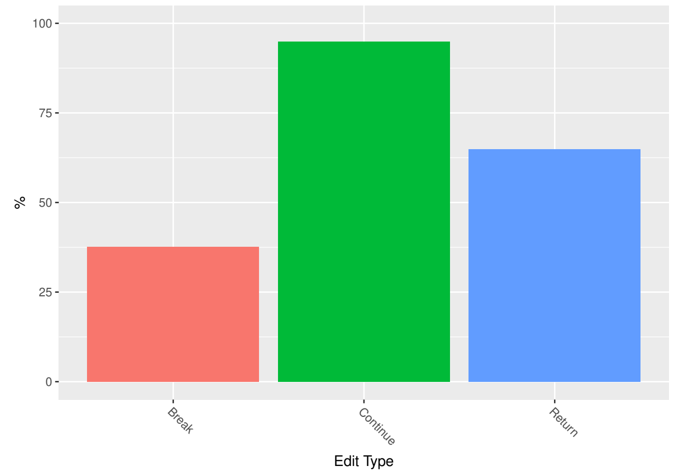


(b) spark

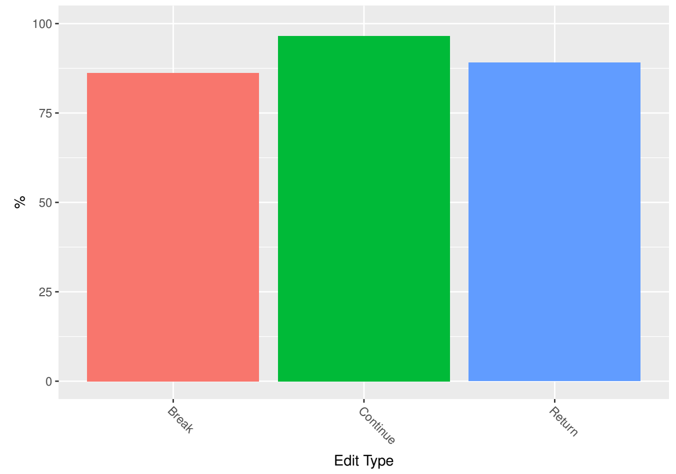


(c) spatial4j

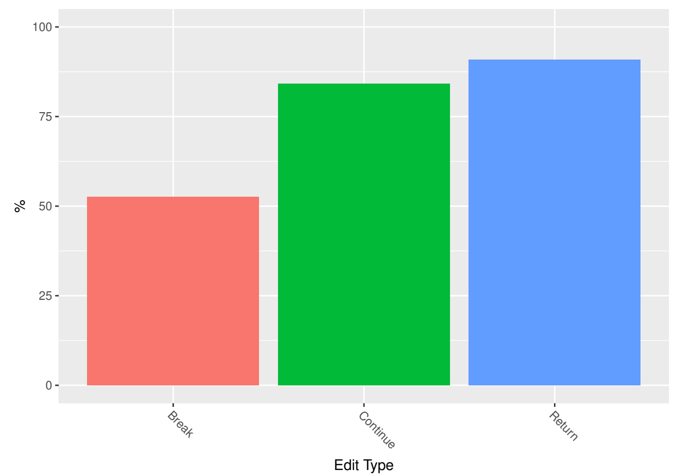
Fig. 2: Enumeration: % of edits that compiled



(a) jCodec

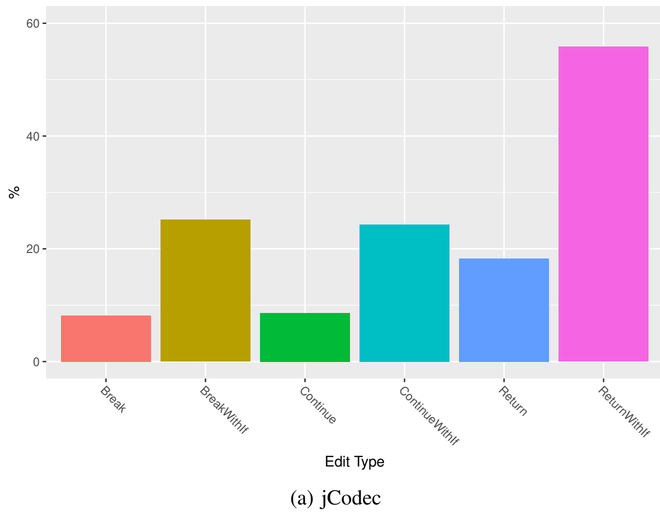


(b) spark

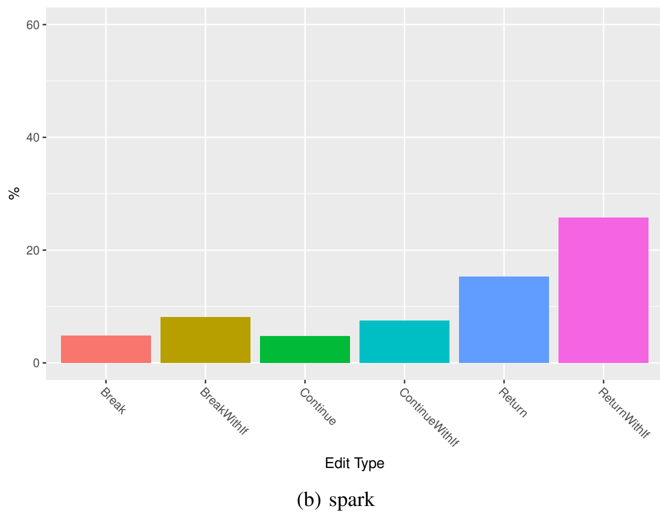


(c) spatial4j

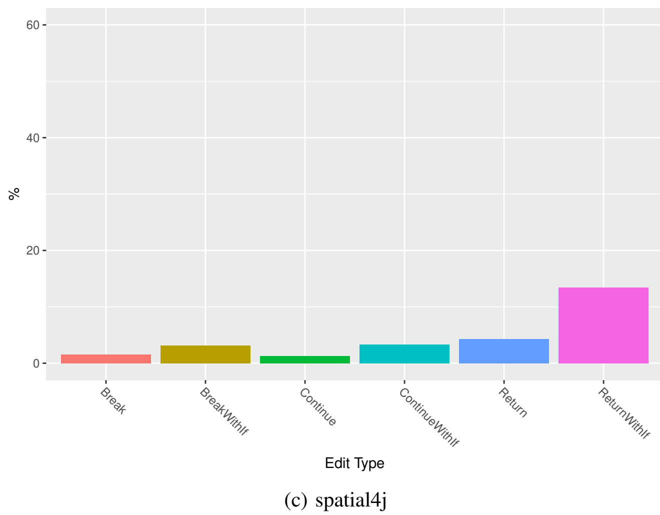
Fig. 3: Enumeration:% of edits that produce neutral variants; i.e., those that compiled that then passed all unit tests



(a) jCodec

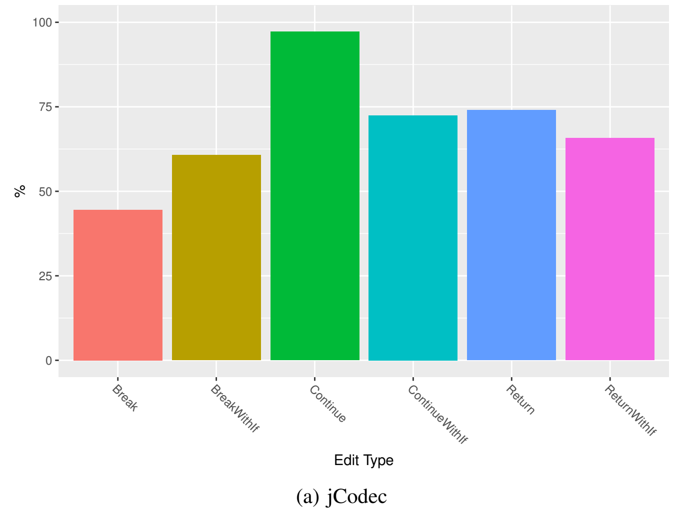


(b) spark

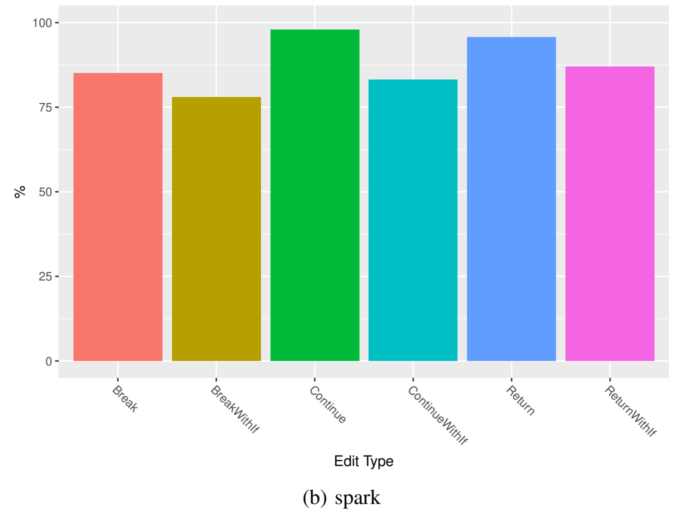


(c) spatial4j

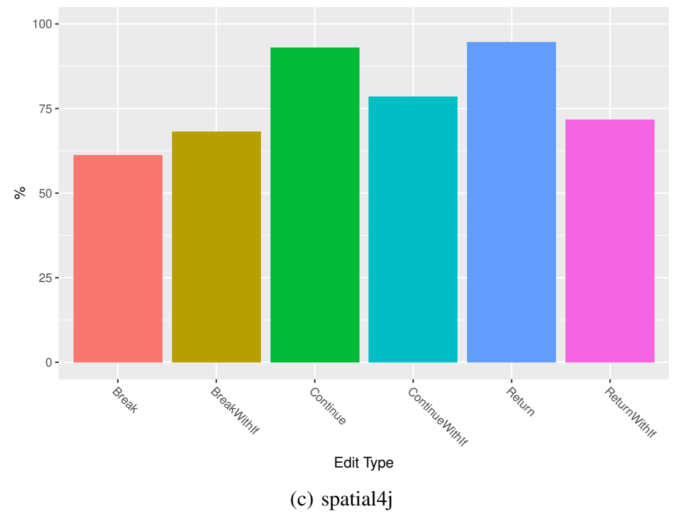
Fig. 4: % of 10 000 edits of each type that compiled



(a) jCodec



(b) spark



(c) spatial4j

Fig. 5: Neutral Variant Rate: % of edits that compiled that passed all unit tests

TABLE II: Results of sampling experiments. 10 000 of each edit type was applied to each project, distributed uniformly at random over all target methods. #Unique shows the number of unique edits of each type; the other counts and percentages are with respect to the full 10 000 edits including duplicates.

Project	Edit	#Unique	#Compile	CR	#Passing	NVR
jCodec	$\mathcal{B}$	3830	816	8.2	363	44.5
	$\mathcal{C}$	3845	863	8.6	839	97.2
	$\mathcal{R}$	3911	1827	18.3	1352	74.0
	$\mathcal{B}_{if}$	8364	2516	25.2	1528	60.7
	$\mathcal{C}_{if}$	8325	2430	24.3	1760	72.4
	$\mathcal{R}_{if}$	8380	5582	55.8	3668	65.7
spark	$\mathcal{B}$	656	486	4.9	413	85.0
	$\mathcal{C}$	663	470	4.7	460	97.9
	$\mathcal{R}$	665	1526	15.3	1461	95.7
	$\mathcal{B}_{if}$	2141	810	8.1	632	78.0
	$\mathcal{C}_{if}$	2126	750	7.5	623	83.1
	$\mathcal{R}_{if}$	2175	2573	25.7	2238	87.0
spatial4j	$\mathcal{B}$	645	152	1.5	93	61.2
	$\mathcal{C}$	635	130	1.3	121	93.1
	$\mathcal{R}$	642	431	4.3	408	94.7
	$\mathcal{B}_{if}$	3595	318	3.2	217	68.2
	$\mathcal{C}_{if}$	3715	335	3.4	263	78.5
	$\mathcal{R}_{if}$	3669	1343	13.4	963	71.7

oversampling of the space (especially for the three simpler edits where the full space is less than this number). The number of unique edits among the 10 000 of each type is given in the table. The figures for #Compile and #Passing, and the corresponding percentages, are with respect to the full 10 000. Note that the figures for **`break`** and **`continue`** edits are not quite equal here (compared to the enumeration results) due to random sampling variation.

These results show that the main obstacle still seems to be passing compilation; if an edit results in code that compiles, it will also often pass all unit tests. The compilation rates are higher for $\mathcal{B}_{if}$, $\mathcal{C}_{if}$ and $\mathcal{R}_{if}$ edits than for their equivalents without the **`if`**: typically around a third more of the edits compile. As noted earlier, this is because Java will fail to compile where 'unreachable code' exists; the presence of the **`if`** allows following code to be reached conditionally. The higher compilation rates then underpin higher numbers of the **`if`** edits passing the unit tests. Despite the higher numbers of edits passing the tests, the NVR (percentage of compiling edits that pass tests) actually decreases slightly for the **`if`** edits in most cases. However, NVR remains high for all edits on all projects: 44.5% for $\mathcal{B}$ on jCodec and over 60% for all other edits and projects. These high figures reflect those seen in [16] for other insertion types (including **`if`** statements) but that work found much lower rates for insertion of returns. A possible explanation here is that we are only adding **`void return`** statements, which simply skip some part of execution. In [16], the experiments considered insertion of **`return`** statements with a type; returning an incorrect value being a possible cause of test failure.

The key points to draw from these results are as follows. (1) Edits that insert **`break`**, **`continue`** and **`return`** statements have higher compilation rates when embedding the statement in an **`if`**. (2) All six edits have a high Neutral Variant Rate (>44.5%), offering a potentially ripe source of performance improving edits.

A. Sampling run times on jCodec[7]

The sampling experiment was repeated for jCodec. 7000 patches were generated: each patch contained one edit, sampled uniformly at random from the 6 proposed edits, and the possible locations for each. These were reduced to only those for which the patched code compiled and passed the tests. Each of these patches was then run on the unit tests 30 times; after each repeat the original unpatched code was also run on the same unit tests. It is hoped that by interleaving the runs this way, any impact of caching etc. is minimised.

3509 of the 7000 patches compiled and passed the tests (i.e. neutral variants). Of these, 1366 offered a significant (i.e. t-test $p < 0.05$) decrease in run time over the original code for the 30 repeats. 84 patches produced a significant increase in run time. The mean percentage reduction in run time for these patches was 15.1%. In terms of the specific edit types ($\mathcal{B}, \mathcal{C}, \mathcal{R}, \mathcal{B}_{if}, \mathcal{C}_{if}, \mathcal{R}_{if}$), the numbers of each offering a statistically significant speedup were: 134, 8, 352, 166, 121, 585. So, all the operators appear to offer some potential for speedup, and $\mathcal{R}_{if}$ appears the most fruitful.

VI. Results: Local Search

Our final experiment is simply intended to show the potential of these kinds of edits. We conducted a single run of a hill-climber on each of the 5 target methods (the 5 methods ranked highest by the profiler for which more than zero neutral variants were found during the sampling runs). Here we focused on jCodec only as it had the highest numbers of edits both compiling and passing the tests.

The methods targeted were those with IDs 1, 3, 4, 5, and 6 in the data sets provided via the URL at the end of the paper. Specifically these are:

1) `org.jcodec.scale.BaseResampler.resample(Picture,Picture)`
3) `org.jcodec.codecs.h264.decode.CoeffTransformer.idct4x4(int[])`
4) `org.jcodec.codecs.h264.decode.MBlockDecoderBase.predictChromaInter(Frame[][],MvList,int,int,int,Picture,PartPred[])`
5) `org.jcodec.codecs.h264.decode.deblock.DeblockingFilter.filterBlockEdgeVert(Picture,int,int,int,int,int,int,int)`
6) `org.jcodec.codecs.h264.decode.BlockInterpolator.getBlockChroma(byte[],int,int,byte[],int,int,int,int,int,int)`

No improving edit was found in 1000 iterations of the local search for target methods 1 and 5.

[7]The results in this section were obtained after acceptance of this paper

For method 3, speedup of 3.58% (final run time 64.5s for 10 repeats of the tests). This was a **continue** at the end of a **for** loop and did not in practice do anything. For method 4, speedup of 3.88% (final run time 62.1s for 10 repeats of the tests). This was also a **continue**, embedded within an existing **if** statement, but at the end of that statement, which in turn was at the end of a loop so not in practice having any effect. For method 6, speedup of 4.05% (final run time 62.8s for 10 repeats of the tests). The resulting patch consisted of two edits, highlighted in Figure 6. The first of these does have the effect of skipping loop iterations, although further analysis is needed to understand its full impact. With a relatively small improvement of run time it is possible this is still the effect of random noise.

In practice these edits are only making small improvements to the code run time; but this proof of concept run shows that even a small scale run is able to find an edit that still passes the tests for the class and is worth further investigation.

VII. Related Work

Genetic improvement has proven to be a successful technique in finding test-suite adequate patches that lead to bug fixes and various efficiency improvements [21]. Those patches usually fall in the three types: copy, delete or replace, applied to either source code, assembly or even binary code. The choice of these three types has its origins in Genetic Programming, the first search technique applied in GI. However, a question arises whether more effective mutation operators exist. Several researchers have looked into this issue by proposing tuning parameters embedded in code [7], [15], or replacement of Java Collections [5], [14]. In related, automated program repair (APR) field, more targeted mutations have been proposed, for example, borrowing from human-evolved patches [9], [11], or programming language specific ones [12].

In the GI work to-date the most effective operator type has been deletion. This has been observed in other fields. For example, loop perforation has been used to achieve quick speedups [18] (up to 3-fold), while producing up to 10% decrease in output quality. This idea of approximation by weighing non-functional properties of software against functional ones has also been explored in work on reducing energy consumption using genetic improvement [4]. For instance, 33% energy reduction was achieved with loss of less than 4% of accuracy, for one of the applications for the application test set used.

It is now well-established that software is not fragile [22], [23]. In fact, there exist a high amount of program variants that are neutral with regards to the given test suite. This observation led to automatic exploration of these plateaus in the program search space in the hope of finding software that improves upon non-functional properties. The question is how to find such improvements efficiently. Given the success of the delete operator we propose here six new edits.

The closest to our work is the work by Harrand et al. [16]. They investigate the search space of program variants achieved with the traditional mutation operators. Based on these observations they propose three new operators: **add method invocation**, **swap subtype** and **loop flip** . These achieve, accordingly, 66.29%, 58.26%, and 73% neutral variant rates. They have, however, not investigated the impact of these on non-functional properties, such as run time.

VIII. Conclusions

We have proposed six new edit types for Genetic Improvement of Java software, based on the insertion of **break**, **continue** and **return** statements. 10 000 randomly-generated instances of each of these edits were applied to three open-source applications taken from GitHub.

There are several key findings: (1) compilation rates for the inserted statements without surrounding **if** statements are low (1.5–18.3%). (2) Edits that insert **break**, **continue** and **return** statements have higher compilation rates when embedding the statement in an **if** (3.2–55.8%). (3) All six edits have a high Neutral Variant Rate (>44.5%), offering a potentially ripe source of performance improving edits. We note here that all these results are with respect to the existing test suites for each project.

Furthermore, a preliminary experiment based on local search showed how these edits might be used in practice.

Several future research directions naturally follow these findings. An obvious extension to the edits would be to have non-zero return values for the *if* conditions. This is, however, non-trivial: it cannot simply be a uniform sampling in the range of the variable's type as in many cases this is likely to be outside the likely values the variable will take. A related question here is how to choose a suitable distribution from which to sample. The $\mathcal{R}$ and $\mathcal{R}_{if}$ edits could also check the return type of methods and add a return variable to improve compilation rates (though as per [16] this may actually reduce the NVR). More interesting is a static analysis of the code to better inform the creation of the edits: limiting insertions to valid locations (though this is possibly more effort than just trying to compile them). In this context it would also be interesting to find the trade-off between only generating valid insertions and just testing by compilation (somewhat akin to handling infeasible solutions in combinatorial optimisation).

Acknowledgments

This work was partially funded by a grant from the Carnegie Trust (TOGA: Towards Grammar-Aware Operators for Genetic Improvement of Software) and the EPSRC (grant number EP/P023991/1).

Data Access Statement

Data sets generated in this work can be found at http://hdl.handle.net/11667/146. The operators have been integrated with the gin tool at https://github.com/gintool/gin.

References

[1] J. Petke, B. Alexander, E. T. Barr, A. E. I. Brownlee, M. Wagner, and D. R. White, "A survey of genetic improvement search spaces," in *Proceedings of GECCO Companion*, 2019, pp. 1715–1721.

```
1 private int invert(int startOff, int level, int prefix, IntArrayList values, IntArrayList
      valueSizes) {
2     int tableEnd = startOff + 256;
3     values.fill(startOff, tableEnd, -1);
4     valueSizes.fill(startOff, tableEnd, 0);
5     int prefLen = level << 3;
6     for (int i = 0; i < codeSizes.length; i++) {
7        if ((codeSizes[i] <= prefLen) || (level > 0 && (codes[i] >>> (32 - prefLen)) != prefix
            ))
8           continue;
9        int pref = codes[i] >>> (32 - prefLen - 8);
10       int code = pref & 0xff;
11       int len = codeSizes[i] - prefLen;
12       if (len <= 8)
13          if (len == 0)
14           continue;
15          for (int k = 0; k < (1 << (8 - len)); k++) {
16             values.set(startOff + code + k, i);
17             valueSizes.set(startOff + code + k, len);
18          }
19          if (code <= 0)
20           continue;
21       } else {
22          if (values.get(startOff + code) == -1) {
23             values.set(startOff + code, tableEnd);
24             tableEnd = invert(tableEnd, level + 1, pref, values, valueSizes);
25          }
26       }
27    }
28    return tableEnd;
29 }
```

Fig. 6: Best patch found for target method 6.

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

[3] W. B. Langdon and M. Harman, "Optimizing existing software with genetic programming," *IEEE Trans Evol Comp*, vol. 19, no. 1, pp. 118–135, 2015.

[4] B. R. Bruce, J. Petke, and M. Harman, "Reducing energy consumption using genetic improvement," in *Proceedings of GECCO*, Madrid, Spain, 2015, p. 1327–1334.

[5] M. Basios, L. Li, F. Wu, L. Kanthan, and E. T. Barr, "Darwinian data structure selection," in *ESEC/SIGSOFT FSE*. ACM, 2018, pp. 118–128.

[6] W. B. Langdon, B. Y. H. Lam, J. Petke, and M. Harman, "Improving CUDA DNA analysis software with genetic programming," in *Proc. of the GECCO, GECCO*. ACM, 2015, pp. 1063–1070.

[7] S. O. Haraldsson, J. R. Woodward, A. E. I. Brownlee, and K. Siggeirsdottir, "Fixing bugs in your sleep: how genetic improvement became an overnight success," in *GECCO, Companion Material Proc.* ACM, 2017, pp. 1513–1520.

[8] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proc of the 22nd ACM SIGSOFT International Symp on Found of Softw Eng*, 2014, pp. 306–317.

[9] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *ICSE*. IEEE Computer Society, 2013, pp. 802–811.

[10] M. Martinez and M. Monperrus, "Mining software repair models for reasoning on the search space of automated program fixing," *EMSE*, vol. 20, no. 1, pp. 176–205, 2015.

[11] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 727–739.

[12] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *IEEE Trans on Softw Eng*, 2018.

[13] J. Petke, "New operators for non-functional genetic improvement," in *Proc GECCO Companion*. ACM, 2017, pp. 1541–1542.

[14] A. E. I. Brownlee, N. Burles, and J. Swan, "Search-based energy optimization of some ubiquitous algorithms," *IEEE Trans. Emerging Topics in Comput. Intellig.*, vol. 1, no. 3, pp. 188–201, 2017.

[15] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke, "Deep parameter optimisation," in *GECCO*. ACM, 2015, pp. 1375–1382.

[16] N. Harrand, S. Allier, M. Rodriguez-Cancio, M. Monperrus, and B. Baudry, "A journey among java neutral program variants," *Gen Prog and Evol Mach*, vol. 20, no. 4, pp. 531–580, 2019.

[17] S. Mittal, "A survey of techniques for approximate computing," *ACM Computing Surveys (CSUR)*, vol. 48, no. 4, pp. 1–33, 2016.

[18] H. Hoffmann, S. Misailovic, S. Sidiroglou, A. Agarwal, and M. Rinard, "Using code perforation to improve performance, reduce energy consumption, and respond to failures," 2009.

[19] D. R. White, "GI in no time," in *Genetic and Evolutionary Computation Conference, Berlin, Germany, July 15-19, 2017, Companion Material Proceedings*, P. A. N. Bosman, Ed. ACM, 2017, pp. 1549–1550. [Online]. Available: http://doi.acm.org/10.1145/3067695.3082515

[20] A. E. I. Brownlee, J. Petke, B. Alexander, E. Barr, M. Wagner, and D. White, "Gin: Genetic Improvement Research Made Easy," in *Proc. of the Genetic and Evolutionary Computation COnference*, Prague, Czech Republic, 2019, pp. 985–993.

[21] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, *IEEE Trans Evol Comp*, vol. 22, no. 3, pp. 415–432, 2018.

[22] E. Schulte, Z. P. Fry, E. Fast, W. Weimer, and S. Forrest, "Software mutational robustness," *Genetic Programming and Evolvable Machines*, vol. 15, no. 3, pp. 281–312, 2014.

[23] W. B. Langdon and J. Petke, "Software is not fragile," in *First Complex Systems Digital Campus World E-Conference 2015*. Springer, 2017, pp. 203–211.