Uniform Edit Selection for Genetic Improvement: Empirical Analysis of Mutation Operator Efficacy

Marta Smigielska
University College London
London, United Kingdom
marta.smigielska.19@ucl.ac.uk

Aymeric Blot
University College London
London, United Kingdom
a.blot@cs.ucl.ac.uk

Justyna Petke
University College London
London, United Kingdom
j.petke@ucl.ac.uk

***Abstract*—Genetic improvement (GI) tools find improved program versions by modifying the initial program. These can be used for the purpose of automated program repair (APR). GI uses software transformations, called *mutation operators*, such as deletions, insertions, and replacements of code fragments. Current edit selection strategies, however, under-explore the search spaces of insertion and replacement operators. Therefore, we implement a uniform strategy based on the relative operator search space sizes. We evaluate it on the QuixBugs repair benchmark and find that the uniform strategy has the potential for improving APR tool performance. We also analyse the efficacy of the different mutation operators with regard to the type of code fragment they are applied to. We find that, for all operators, choosing expression statements as target statements is the most successful for finding program variants with improved or preserved fitness (50.03%, 33.12% and 23.85% for deletions, insertions and replacements, respectively), whereas choosing declaration statements is the least effective (3.16%, 10.82% and 3.14% for deletions, insertions and replacements).**

***Index Terms*—genetic improvement, mutation operators**

I. INTRODUCTION

Genetic improvement (GI) is a software engineering technique aiming to improve both functional and non-functional properties of software by utilising search-based techniques [1]. GI can be used as a strategy for automated program repair (APR), where software bugs, usually exposed by failing test cases within a test suite, are fixed in an automatic way [2]. Test suite based APR tools aim to find a patch, that would modify the software in a way such that it passes all the test cases from the test suite [3], [4].

Search-based APR tools use software transformations — *mutations* — to generate new program variants for evaluation. The most commonly used mutation operators are: deletion (removes a fragment of the program), insertion (copies a fragment of the program and inserts it in a different place) and replacement (replaces a fragment of the program with another fragment) [2]. Fragments are typically a line in the program source code or a node in the program's abstract syntax tree (AST), usually a statement.

One possible area for improvement of GI-based APR tools is in the strategies for selecting and applying those mutation operators. The state-of-the-art for edit selection strategy leads to an unbalanced exploration of all possible ways the different operators can be applied. Our idea is to implement and evaluate a more fair strategy, that will allow for the spaces of all operators to be uniformly explored. We also want to explore and analyse the efficacy of applying the operators to different parts of the program source code (for example, conditional statements, loops, return statements). We aim to present recommendations for more effective edit selection strategies, which would consider where in the program the mutation operator is being applied.

In order to achieve our aims we implemented the uniform edit selection strategy in the PyGGI [5] GI tool and ran it on one of the more challenging benchmark sets for program repair: QuixBugs [6]. Our results show that:

- For the programs that could *not* be repaired using a single edit targeting a comparison operator modification, PyGGI performed better with the uniform edit selection strategy than the standard strategy.
- Out of the three standard operators, the deletion operator is the most effective at improving program fitness, even though it is not present in many repairing patches. This could be caused by deletions removing program statements that include the buggy code fragments. The insertion operator, however, is the most effective at preserving program behaviour with respect to the test suite.
- For program repair, expression statements are very effective (50.03%, 33.12%, 23.85%) and declaration statements are the least effective (3.16%, 10.85%, 3.14%) as target nodes for all three standard operators — deletions, insertions and replacements, respectively.
- For preserving program behaviour, expression statements are the most effective (16.53%, 14.14%, 9.33%) and declaration statements are the least effective (2.41%, 5.16%, 2.56%) for deletions, insertions and replacements, respectively.

II. BACKGROUND

The research areas of automated program repair (APR) and genetic improvement (GI) are closely interlinked, as the roots of GI also lie in program synthesis and automatic programming as well as in genetic programming. GI techniques have a broader use than just automatic bug repair, as they also include improvement of non-functional properties, such as memory usage, execution time or energy consumption [1].

GI-based program repair tools typically generate hundreds or thousands program variants, using software transformations,

at the granularity level of lines, statements, expressions, or even binary code. Next, a search strategy is employed to navigate the space of the generated software variants, evaluated using a given fitness function, until all test cases pass. For the purpose of program repair the fitness typically takes into account the number of passing and failing tests.

The state-of-the-art for choosing and applying the mutation operators is a two-step selection process [7]. The mutation operator is chosen first, each mutation operator having an equal probability of being selected, and its parameters, including the location at which the operator will be applied, second. Typically, if there are n statements in the program's AST, then there are n possible statement deletions, n^2 statement insertions, and n^2 statement replacements.[1] The search space induced by the deletion operator is significantly smaller than the search spaces of the insertion and replacement operators. Therefore, when operators are selected with equal probabilities the search space of deletions ends up being explored much more thoroughly. Note that throughout this paper, *search space* always refers to the finite set of possible single edits induced by a given mutation operator and not the infinite space of all sequences of edits.

The idea we investigate is to implement and evaluate a fair uniform strategy for the edit selection process, based on the size of the induced search spaces. With such a strategy, each individual way of the modification of the program would have an equal probability of being selected and applied and the spaces of all operators would be equally explored.

III. Experimental Design

For this study on the effectiveness of the uniform strategy for mutation edit selection and on the efficacy of mutation operators to create successful new program variants, we formulate the following two research questions:

RQ1. How effective is the uniform edit selection strategy at finding test suite adequate patches for faulty programs using GI?

For RQ1, we want to analyse the number of test suite adequate patches that a GI tool can find for defected programs from a standard APR benchmark using the uniform edit selection strategy. We want to compare those results to the results produced by the initial edit selection strategy to determine whether the uniform strategy improves tool's performance.

RQ2. Which combinations of GI mutation operators and types of program statements are the least disruptive, creating useful new program variants?

For RQ2, we aim to analyse the efficacy of each operator to create improved program variants, i.e., with better fitness than the variant on which it is applied, or *preceding variant*. Each program variant is represented as an edit list, meaning a list of modifications to be applied to the original program. Additionally, we focus on analysing which operators applied to which types of program statements lead to program variants with better fitness, with unchanged fitness, and with worse fitness than their preceding variant.

[1] The exact numbers depend on the exact implementation used.

A. Dataset

In this study, we use faulty Java programs from the QuixBugs benchmark, that provides 40 programs, each exposing a single one-line bug. We chose the QuixBugs benchmark as it proved to be particularly challenging for APR tools [8]. We exclude programs for which there is no positive test case, without which APR tools cannot ensure the intended behaviour of the program is preserved. Additionally we only target programs that have bugs causing incorrect output, excluding 5 programs with infinite loops and array index errors. Overall, we target 27 Java programs from the QuixBugs benchmark.

B. PyGGI setup

The experiment is conducted using a modified version of PyGGI [5] — a GI tool for multiple programming languages that can also be used for APR. Our version has a new mutation operator and the proposed uniform selection strategy. PyGGI uses SrcML to obtain an XML-based representation of the program's AST. The types of tags we consider are `break`, `continue`, `decl_stmt`, `do`, `expr_stmt`, `for`, `goto`, `if`, `return`, and `switch`, for statements; and `operator_comp` for operators.

PyGGI implements the three standard mutation operators: deletion, insertion, and replacement. For the sake of simplicity, to avoid overlaps in the search space, and to ensure fair comparison, we restrict the insertion operator to modification points preceding statements — other GI tools such as GenProg [9] and Gin [10] implement insertion before only. In the experiments we also use a mutation operator which we call comparison operator modification, which was not originally implemented in PyGGI. It targets comparison operators (that is, $==$, $!=$, $<$, $>$, $<=$, $>=$), replacing the operator with any other possible comparison operator. We decided to add the comparison operator modification as it proved to be effective with different datasets [11], [12], [13] including successful application in industry [14].

Our experimental setup is based on the setup chosen for the evaluation of QuixBugs benchmark by PyGGI 2.0 [5]. The local search algorithm is run with a budget of 500 iterations with a time limit of 10 seconds for test suite execution. Each patch is represented as a list of edits. PyGGI by default implements a hill-climbing local search algorithm. We use the standard mutation in which with equal probabilities either a new edit is generated and applied to the current edit list, or an existing edit is removed unless the list is already empty.

Experiments are repeated 20 times, during which we gather data needed for the research questions. For RQ1, after each repair attempt we report the number of iterations, the best patch that was found, and its fitness. For RQ2, in each iteration we report which mutation operator was added to the patch, and what type of nodes it targets (both target and ingredient modification points). We also report on the fitness, run status, and run information, for both the current and preceding patches.

TABLE I
INITIAL EDIT SELECTION STRATEGY RESULTS

Program	Patches		Iterations			
	all	unique	avg	min	max	stdev
DFS	14	6	192	22	393	126.8
KHEAP-SORT	1	1	63	63	63	-
KNAP-SACK	17	5	102	4	495	128.6
LIS	13	9	179	18	442	128.3
MERGE-SORT	1	1	95	95	95	-
PASCAL	15	4	153	2	490	148.4
QUICK-SORT	20	4	25	1	64	18.7
SIEVE	3	3	256	65	389	169.8
Avg	10.50	4.13	133.13	-	-	-
Stdev	7.63	2.64	75.68	-	-	-

TABLE II
UNIFORM EDIT SELECTION STRATEGY RESULTS

Program	Patches		Iterations			
	all	unique	avg	min	max	stdev
DFS	19	5	83	10	222	51
KHEAP-SORT	0	0	-	-	-	-
KNAP-SACK	9	2	159	1	397	157.3
LIS	17	6	171	10	451	154.3
MERGE-SORT	3	3	393	330	474	73.7
PASCAL	1	1	430	430	430	-
QUICK-SORT	13	2	251	17	499	173
SIEVE	6	6	312	161	467	113.1
Avg	8.50	3.13	257.0	-	-	-
Stdev	7.25	2.3	128.3	-	-	-

C. Uniform strategy implementation

To simplify the process of implementation of the uniform selection strategy, we decided to keep the two-step selection process and skew the probabilities of selecting the operator during the first step, as such implementation is equivalent to the theoretical one-step uniform selection strategy. Such implementation is easily reproducible for future studies.

Overall, with n the number of statements and m the number of comparison operators in the program, there are n unique possible statement deletions, n^2 unique statement insertions, n^2 unique statement replacements, and $5m$ unique comparison operator modifications, as each one can be replaced by any of the other possible five.

Combined, that makes a total of $T = n + n^2 + n^2 + 5m$ possible unique edits, and therefore skewed probabilities of $n/_T$, $n^2/_T$, $n^2/_T$, and $5m/_T$ are used so that selection is uniform over the possible unique edits rather than simply uniform over the edit type.

IV. Evaluation

Next, we present the experimental results for both research questions. RQ1 evaluates the effectiveness of the uniform edit selection strategy as compared to the standard one. For RQ2 we analyse the efficacy of different mutation operators considering the type of statements they are applied to.

A. RQ1: Uniform edit selection strategy results

For this RQ we attempt to find test suite adequate patches for the programs in the QuixBugs benchmark using PyGGI with two edit selection strategies to compare the results.

At first, we run PyGGI with the QuixBugs benchmark using the initial two-step edit selection strategy. The local search algorithm is able to find 84 test suite adequate patches for 8 programs among the 27 programs we evaluate, with 33 patches syntactically unique. Table I shows detailed results for the programs with successful patches, including statistics on the number of iterations spent before a successful patch is found.

Compared to the PyGGI 2.0 release work [5], also using the QuixBugs benchmark, the search algorithm found test suite adequate patches for five more programs, taking into account only results for the statement level for Java programs.

Next, we run PyGGI using the newly implemented uniform edit selection strategy. This time, the local search algorithm is able to find 68 test suite adequate patches for 7 programs. Out of those 68 patches, 25 patches are syntactically unique. Table II shows the detailed results.

For four out of eight programs (*kheapsort*, *knapsack*, *pascal*, *quicksort*), PyGGI with the initial edit selection strategy had more successful repair attempts that found a test-adequate patch than the uniform strategy. For all those four programs, PyGGI with the initial strategy took less iterations on average to find a successful patch. The explanation partly lies in the type of bug exposed in three of these programs, for which a comparison operator modification is required: due to the relatively small size of the induced search space, that operator ends up being selected less often in the uniform edit selection strategy. For the other four programs (*dfs*, *lis*, *mergesort*, *sieve*), PyGGI with the uniform edit selection strategy had more successful repair attempts.

We can also observe, as shown in Table I and Table II, that the standard deviation of the average iterations needed to find a successful patch is around twice the one for the uniform strategy than for the initial strategy, whereas the number of found patches has comparable values of presented statistics. This can be explained by the fact that the programs that can be fixed with one comparison modification operator took much longer to find a patch for, as that operator is selected in the uniform strategy less often than in the initial strategy.

The comparison of unique patches found by both strategies is shown in Table III, together with the number of other APR tools reported to find test suite adequate patches in the recent QuixBugs study [8]. In particular, PyGGI was able to find test suite adequate patches for three more programs (*kheapsort*, *pascal*, *sieve*) than any of the ten tools used in the study.

Next, we analyse the patches from both initial and uniform edit selection strategies. There are 73 individual edits within the 33 unique test suite adequate patches that were generated by PyGGI with the initial selection strategy, which gives an average patch length of 2.12 edits. For the uniform strategy, there are 49 edits within the 25 unique test suite adequate patches, giving an average patch length of 1.96 edits.

Interestingly, as shown in Table IV, only 12.3% of edits

TABLE III
PyGGI effectiveness compared to other APR tools

Program	Standard strategy	Uniform strategy	# other APR tools [8]
DFS	✓ (6)	✓ (5)	6
KHEAPSORT	✓ (1)	(0)	0
KNAPSACK	✓ (5)	✓ (2)	1
LIS	✓ (9)	✓ (6)	5
MERGESORT	✓ (1)	✓ (3)	1
PASCAL	✓ (4)	✓ (1)	0
QUICKSORT	✓ (4)	✓ (2)	6
SIEVE	✓ (3)	✓ (6)	0

TABLE IV
Occurrences of mutation operators in unique successful patches

Operator	Initial strategy		Uniform strategy	
	patches	edits	patches	edits
All	33	73	25	49
Deletion	18.2%	12.3%	0%	0%
Insertion	45.4%	21.9%	40%	24.5%
Replacement	45.4%	24.7%	68%	57.1%
Comparison op mod	63.6%	41.1%	32%	18.4%

from successful unique patches generated by the initial edit selection strategy have deletion as the mutation operator, even though every operator has the same chance of getting selected. The comparison operator modification is the most successful, which is understandable, as the incorrect comparison bug appears in 5 programs from the QuixBugs benchmark.

We compare our results to the work by Le Goues et al. [15], which includes the effectiveness of different operators at repairing buggy programs. Interestingly, they found that deletions appeared in the repair patches more often than insertions and replacement, with replacements appearing the least often. This is in contradiction with our results, as in our study deletions are the least common, whereas insertions and replacements are comparably common. An explanation might be the type of programs and bugs to be fixed, as they used a much bigger benchmark including eight C programs with over 5 million of lines and 105 defects. The operator distribution for initial repairs from their work is **1.7 : 1 : 1.45**, in contrast to **1 : 2.5 : 2.5** in our experiments (for deletions, insertions, and replacements, respectively).

For the uniform edit selection strategy, probably due to the reduced associated probability, there are no deletion edits within the successful patches. Replacements are the most successful, they appear in 68% of unique successful patches, followed by insertions at 40%, then comparison operator modifications at 32%.

To sum up our findings for RQ1, for programs that could not be repaired using comparison operator modification only, PyGGI performed better with the uniform edit selection strategy, finding successful test suite adequate patches during more repair attempts than the initial operator strategy. This shows that a strategy in which the search spaces of operators are more uniformly explored can improve performance of GI tools. Our results suggest to skew the probability distribution from deletions in favour of insertions and replacements.

TABLE V
Mutation operators efficacy

Operator	Patches	Success	Fitness			
			<%	>%	=%	==%
Deletion	7189	47.8%	3.9	20.2	23.7	9.5
Insertion	81110	28.7%	0.7	5.0	23.0	10.5
Replacement	74303	22.8%	1.5	8.4	12.9	6.3
Comp op mod	5329	89.2%	13.6	36.2	39.4	17.9

B. RQ2: Mutation operators efficacy

For this RQ we aim to analyse the efficacy of different mutation operators in finding new program variants. We analyse data from 167,931 iterations of the local search algorithm, excluding the comparison operator modifications in which the target and ingredient are identical.

Table V shows statistics regarding mutation operator efficacy, in the context of addition to the current patch. For each operator we describe the number of analysed patches, the percentage of patches that run successfully without regard to their fitness (i.e., excluding compilation and runtime errors), then the percentage of patches with a better (<), worse (>), and equal (=) fitness (i.e., number of failing test cases) that their predecessor, and finally the percentage of patches with equal fitness and exact same output than their predecessor.

Mutation operators that produce program variants with test suite preserving behaviour are an important factor for improvement of both functional and non-functional properties of software [16]. Indeed, the presence of neutral mutations within the patches plays an important role in the process of finding correct patches [17]. At the same time, when improving non-functional properties such as execution time or memory usage, it is crucial to preserve the behaviour of the original program.

Comparison operator modifications are the most effective at improving the fitness of the program, with 13.57% effectiveness; this is probably an artefact due to the 5 QuixBugs programs for which the one-line defect is precisely an incorrect comparison operator. Comparison operator modifications are also the most effective at finding behaviour preserving variants, with 17.88% patches producing the same output as their preceding variants. Interestingly, the delete operator was the least present within the final successful patches analysed in RQ1; a possible explanation for this results comes from insufficient test suite for the analysed program, as intuitively in most cases change of the comparison operator in logical expressions should change the behaviour of the program.

As for the three standard operators, the deletion operator is the most effective. Out of patches with deletions as the lastly added edit, 3.95% patches have better fitness than their predecessor, as compared to 0.66% for insertions and 1.49% for replacements. Possible interpretation of this result is that using deletion operators often leads to patches that get stuck

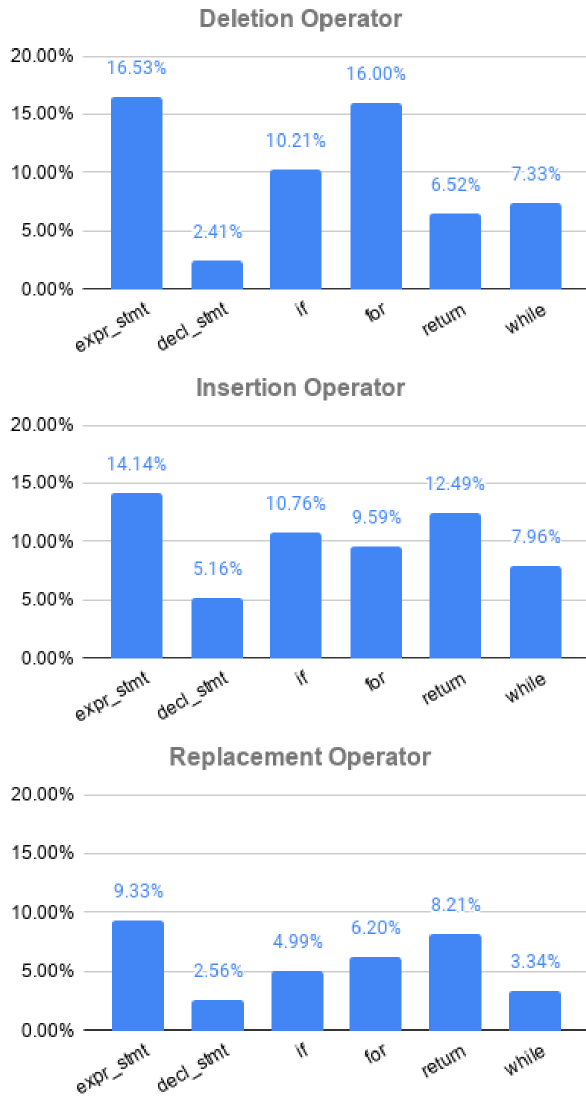

Fig. 1. Percentage of modifications that create neutral program variants

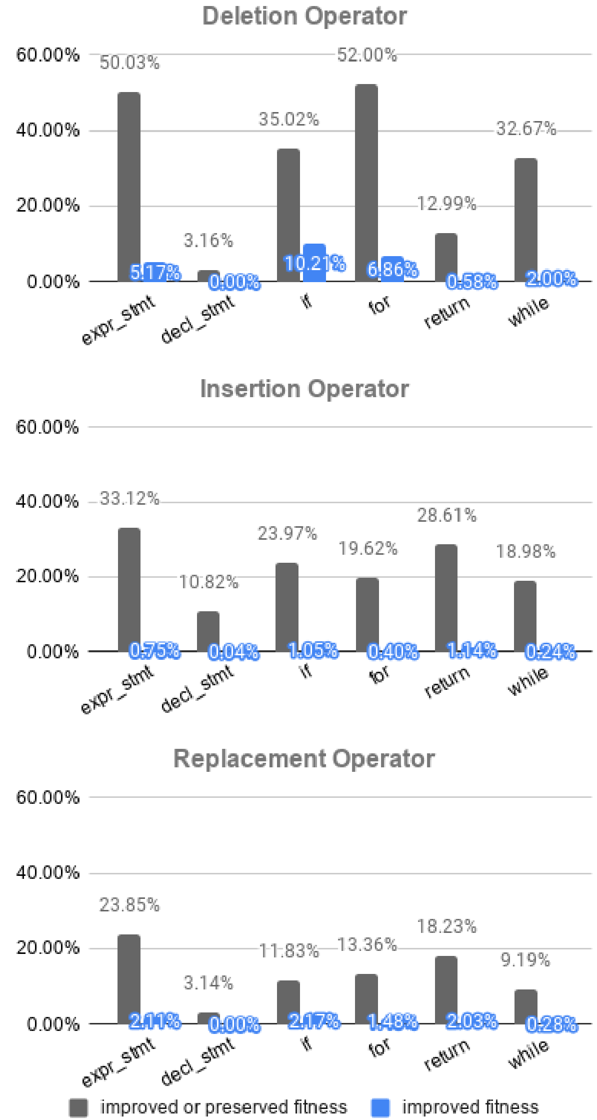

Fig. 2. Percentage of modifications that create variants with improved or preserved fitness

in local minima and don't lead to actual repairs. Deletions that improve fitness may remove the program statements that cause the fault to appear, which could possibly show what place in the program is causing the defect, but not actually fix it.

We also analyse the efficacy of mutation operators considering the type of target statement they were applied to. Charts presented in Figure 1 and Figure 2 show how often each modification operator applied to different types of statements led to program variants with fitness no worse than their preceding variants and to program variants with test suite preserved behaviour. *Continue statements*, despite being in percentage by far the most effective, are excluded from all charts due to the extremely low support, as show in Table VI.

Choosing *expression statements* as target nodes for any operator showed to be the most effective as compared to other types of nodes. 50.03%, 33.12% and 23.85% of modifications with *expression statements* as target nodes, for deletion, insertion, and replacement, respectively, led to program variants with improved or preserved fitness in comparison to their preceding variants. Deleting, inserting before and replacing *declaration statements* was consistently the least effective, as only 3.16% of deletions, 10.82% of insertions and 3.14% replacements applied to this type of statements created program variants with fitness no worse than their predecessors.

TABLE VI
ANALYSED PATCHES FOR EACH OPERATOR AND ITS TARGET STATEMENT

Operator	expr stmt	decl stmt	if	for	return	while	continue
Deletion	1,645	1,615	1,342	700	1,733	150	4
Insertion	21,809	19,413	13,325	8,695	16,072	1,633	163
Replacement	20,020	17,841	12,293	8,044	14,526	1,436	143

For the deletion operator, another type of nodes that showed to be effective are *for loops* and *if statements*, with 52% and 35.02% of program variants with improved or preserved

fitness, respectively. Interestingly choosing *expression statements* (5.17%), *if statements* (10.21%) and *for loops* (6.86%) as target nodes for the deletion operator is the most effective at creating variants with improved fitness, as compared to all other operators and types of nodes. For both the insertion and replacement operator, *return statement* is another type of target node that is effective, with 28.61% and 18.23% effectiveness in creating variants with no worse fitness. It is also worth noting that compared to insertions and replacements, the deletion operator has over 10 times bigger variance amongst effectiveness on different types of target nodes.

The relative results for finding neutral program variants are similar to the effectiveness of operators and target nodes in creating variants with no worse fitness than their predecessors. Choosing *expression statements* is the most successful in preserving program behaviour for all modification operators, with effectiveness of 16.53%, 14.14% and 9.33% for deletions, insertions and replacements respectively. Similarly, the deleting, inserting before or replacing the *declaration statements* is the least effective.

The chart in Figure 3 presents the effectiveness of the comparison operator modification in creating program variants with improved or preserved fitness depending on the target and ingredient comparison operators chosen. For each bar on the chart, there is a label that corresponds to the number of comparison operator modification edits with specific target and ingredient nodes. The chart shows that modifying the less or equal comparison operator was consistently the most successful, with effectiveness greater than 90% for each ingredient comparison operator. Modification of the greater or equal comparison operator also shows to be consistently successful, with effectiveness over 70% for all but not equal ingredient comparison operators. Another effective target and ingredient comparison operators combinations are changing greater comparison operators to not equal and not equal to less, with 100% and 82% effectiveness, respectively. Replacing greater operators with less or equal and equals operators was the least effective, both having effectiveness less than 15% in creating program variants with improved or preserved fitness. Note there are 69 comparison operators within the 27 programs we analyse, as compared to 352 program statements.

We also compare our results on creating neutral program variants with regards to the given test suite to work by Harrand et al. [16]. Their study focuses on finding regions within Java programs that are likely to transform into neutral program variants. They use 6 large open source Java projects, and they distinguish slightly different program statement types as compared to our work. For the deletion operator, they reported the effectiveness of *if statements* to range between 10% and 20% for the 6 projects, as compared to 10.21% effectiveness result in our study. They also reported the effectiveness of loop statements to range between 0% and 15%, as compared to 16% for *for loops* and 7.33% for *while loops* in our study and the effectiveness of *return statements* to range from 10% to 25% as compared to 6.25% in our results.

For the insertion operator, Harrand et al. report the effectiveness of ingredient nodes, whereas we focus on target nodes for that operator. However, we can compare the results. They found *if statements* and *loops* to be comparably successful at creating neutral program variants as ingredient nodes, with effectiveness ranging from 40% to over 60%. They found *try statements*, which we did not analyse, to be the most successful with effectiveness over 70% for three Java projects and *return statements* to be the least successful with effectiveness from 10% to 30%. We found *return statements* and *expression statements* to be relatively successful with 12.49% and 14.14% effectiveness respectively. In our study, *declaration statements* were the least successful (5.16%) as target nodes for insertions.

Summarising our findings for RQ2:

- Out of the three standard operators, deletion is the most effective at improving program fitness, even though it is present in very few final patches, which could be caused by deletions removing program statements that include buggy code fragments. Insertion, however, is the most effective at preserving program behaviour with respect to the test suite.
- For program repair, *expression statements* are very effective (50.03%, 33.12%, 23.85% for deletions, insertions, and replacements, respectively) and *declaration statements* are the least effective (3.16%, 10.85%, 3.14%) as target nodes for all three standard operators. For deletions, *for loops* are the most successful with 52% effectiveness. For insertions and replacements *return statements* are also effective (28.61%, 18.23%).
- For preserving program behaviour, with respect to the given test-suite, *expression statements* are the most effective (16.53%, 14.14%, 9.33%, for deletions, insertions, and replacements) and *declaration statements* are the least effective (2.41%, 5.16%, 2.56%).
- Comparison operator modification was the most effective out of all four operators at both improving and preserving program behaviour. It is important to note that every program from the QuixBugs benchmark has a one-line defect, and for 5 programs that defect is an incorrect comparison operator. Modifying the "<=" comparison operator was consistently the most successful at improving or preserving fitness, with effectiveness greater than 90% for each ingredient comparison operator. Replacing ">" operators with "<=" or "==" operators was the least effective, both having effectiveness less than 15%.

V. Threats to Validity

We use the QuixBugs benchmark to evaluate our work. This benchmark includes small programs (up to 60 lines of code) that implement popular algorithms, each program having one simple one-line bug. The benchmark does not include more complex defects, therefore our results may not generalise to bigger software with more complex bugs. 5 out of 27 programs we analyse from the benchmark and 4 out of 8 programs we found successful patches for have faults that include a single incorrect comparison operator. Therefore, positive results for the comparison operator modification may be inflated.

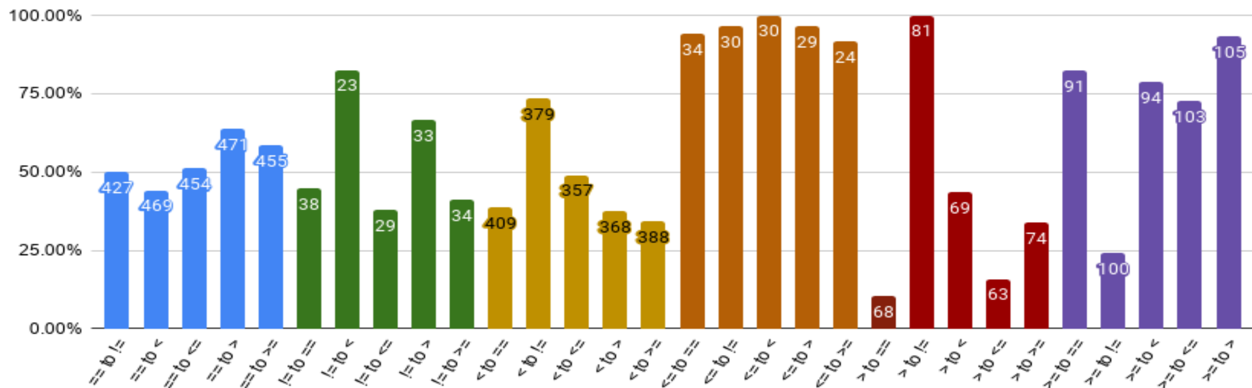

Fig. 3. Percentage of comparison operator modifications that create variants with improved or preserved fitness

The ability of mutations to improve program fitness during search before the final repair is found does not necessarily correlate with their usefulness in reaching optimal fitness and the desired fix. For insertions and replacements we analyse and present the results for different types of target nodes but do not take into consideration the ingredient node for each target node, which could provide more detailed data.

Finally, we do not provide a manual analysis of the resultant test-suite adequate patches, due to the large number of patches found. However, manual analysis would be needed to establish which patches are true fixes.

To mitigate these threats we make our implementation freely available, so that future researchers can extend the study to other benchmarks and programming languages. Thanks to using PyGGI, a multi-lingual GI tool, the study can be easily extended to Python, C#, C, or C++ programs.

VI. Related Work

The exponential growth of work in GI and APR started in the late 2000s when Arcuri et al. [18], [19], Weimer et al. [7], [20], and Debroy and Wong [21] published their work on automated software repair using genetic programming. Since then many APR tools and techniques were released, and applied in the industry [22], [14]. In addition to APR, GI tools are also used for the improvement of non-functional properties of software, as introduced by White et al. [23] with their work on reduction of energy consumption. Since then, GI tools have been applied to improve different non-functional properties such as execution time [24], [25] or memory consumption [12].

We performed a literature review for the area of mutation operators within fields of genetic improvement and automated program repair. We found several papers on the improvement of different sub-areas of APR techniques, including previously mentioned work by Le Goues at al. on representations and operators [15], which we found the most relevant to our paper, the only focusing on this aspect of the GI process. We also found work on improving patches produced by APR tools [26], [27], using high-order mutations in GI [28], [29], fault localisation [30], [31], [32] and the impact of test suite metrics on the APR performance [33]. Finally, Petke and al. [34] provide a recent survey of GI search spaces.

VII. Conclusions and Recommendations

Changing the mutation edit selection strategy is a potential area of improvement of GI-based APR tools. The state-of-the-art approach for edit selection leads to unbalanced exploration of the program modification space, with the search spaces of the insertion and replacement operators being under-explored. We implemented the uniform edit selection strategy in the PyGGI framework [5], using the Java versions of programs from the QuixBugs benchmark for evaluation [6]. We used four mutation operators: deletion, insertion, and replacement of statements, as well as a comparison operator replacement. We implemented the uniform edit selection strategy, keeping the two-step edit selection process by adjusting the probabilities of selecting individual mutation operators. We compared this strategy with the standard one.

The standard edit selection strategy found 33 unique test suite adequate patches for 8 programs altogether, whereas the uniform strategy found 25 unique test suite adequate patches for 7 programs. The performance of the uniform strategy greatly depends on the type of defect in the program. For four programs where the defect can be fixed with one comparison operator modification, the uniform strategy was less successful as this type of operator ends up being selected much less often. For other programs, the uniform selection strategy had more successful repair attempts (42), i.e., more test-suite adequate patches found, than the standard strategy (31), which suggests that moving towards uniform exploration of spaces of the three standard modification operators can improve the performance of the GI search algorithm.

The deletion operator was present in only 18.2% of repairing patches for the standard strategy (as compared to 45.4% for insertions, 45.4% for replacements and 63.6% for comparison operator modifications), and was present in no patches for the uniform strategy. Even without deletion operators in the final patches, the uniform strategy had more successful repair attempts. Possibly the optimal edit selection strategy lies somewhere in between the standard approach and the pure uniform selection strategy.

Interestingly, we found that during the execution of the search algorithm, deletion operators were more successful than insertions and replacements at obtaining program variants with improved fitness. Possible explanation of that result is that deletion operators lead to better fitness because they remove statements that contain the faulty code. When analysing the efficacy of operators as applied to different types of program statements, we found that choosing an *expression statement* as target node was consistently the most successful (50.02% for deletions, 33.12% for insertions, 23.85% for replacements) at finding program variants with improved or preserved fitness, whereas choosing a *declaration statement* was the least effective (3.16% for deletions, 10.82% for insertions, 3.14% for replacements, respectively).

A possible direction for future work is to evaluate different edit selection strategies that skew the probability distribution of edit selection away from deletions and towards insertions, replacements, and comparison operator modifications. Additionally, after a specific operator gets selected, different types of program statements could be selected with different probabilities, skewed towards statements that were shown to be more successful at creating improved program variants.

Artefact: https://github.com/SOLAR-group/gi2021artefact

Funding: supported by UK EPSRC Fellowship EP/P023991/1

REFERENCES

[1] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic improvement of software: A comprehensive survey," *IEEE Trans. Evol. Comput.*, vol. 22, no. 3, pp. 415–432, 2018.

[2] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Trans. Softw. Eng.*, vol. 45, no. 1, pp. 34–67, 2019.

[3] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comp. Surveys*, vol. 51, no. 1, pp. 17:1–17:24, 2018.

[4] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for Java programs," in *International Conference on Software Engineering*. ACM, 2020, pp. 615–627.

[5] G. An, A. Blot, J. Petke, and S. Yoo, "PyGGI 2.0: Language independent genetic improvement framework," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 1100–1104.

[6] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "Quixbugs: a multi-lingual program repair benchmark set based on the quixey challenge," in *ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. ACM, 2017, pp. 55–56.

[7] S. Forrest, T. Nguyen, W. Weimer, and C. Le Goues, "A genetic programming approach to automated software repair," in *Genetic and Evolutionary Computation Conference*. ACM, 2009, pp. 947–954.

[8] H. Ye, M. Martinez, T. Durieux, and M. Monperrus, "A comprehensive study of automatic program repair on the quixbugs benchmark," in *Intelligent Bug Fixing*. IEEE, 2019, pp. 1–10.

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

[10] A. E. I. Brownlee, J. Petke, B. Alexander, E. T. Barr, M. Wagner, and D. R. White, "Gin: genetic improvement research made easy," in *Genetic and Evolutionary Computation Conference*. ACM, 2019, pp. 985–993.

[11] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. R. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 34–55, 2017.

[12] F. Wu, W. Weimer, M. Harman, Y. Jia, and J. Krinke, "Deep parameter optimisation," in *Genetic and Evolutionary Computation Conference*. ACM, 2015, pp. 1375–1382.

[13] M. Martinez and M. Monperrus, "Coming: A tool for mining change pattern instances from git commits," in *International Conference on Software Engineering*. IEEE / ACM, 2019, pp. 79–82.

[14] S. O. Haraldsson, J. R. Woodward, A. E. Brownlee, and K. Siggeirsdottir, "Fixing bugs in your sleep: How genetic improvement became an overnight success," in *Genetic and Evolutionary Computation Conference*. ACM, 2017, pp. 1513–1520.

[15] C. Le Goues, W. Weimer, and S. Forrest, "Representations and operators for improving evolutionary software repair," in *Genetic and Evolutionary Computation Conference*. ACM, 2012, pp. 959–966.

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

[17] J. Renzullo, W. Weimer, M. Moses, and S. Forrest, "Neutrality and epistasis in program space," in *International Workshop on Genetic Improvement*. ACM, 2018, pp. 1–8.

[18] A. Arcuri, "Evolutionary repair of faulty software," *Appl. Soft Comput.*, vol. 11, no. 4, pp. 3494–3514, 2011.

[19] D. R. White, A. Arcuri, and J. A. Clark, "Evolutionary improvement of programs," *IEEE Trans. Evol. Comput.*, vol. 15, no. 4, pp. 515–538, 2011.

[20] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *International Conference on Software Engineering*. IEEE, 2009, pp. 364–374.

[21] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *International Conference on Software Testing, Verification and Validation*. IEEE, 2010, pp. 65–74.

[22] W. B. Langdon, B. Y. H. Lam, J. Petke, and M. Harman, "Improving CUDA DNA analysis software with genetic programming," in *Genetic and Evolutionary Computation Conference*. ACM, 2015, pp. 1063–1070.

[23] D. R. White, J. A. Clark, J. Jacob, and S. M. Poulding, "Searching for resource-efficient programs: low-power pseudorandom number generators," in *Genetic and Evolutionary Computation Conference*. ACM, 2008, pp. 1775–1782.

[24] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, "Using genetic improvement and code transplants to specialise a C++ program to a problem class," in *European Conference on Genetic Programming*, ser. LNCS, vol. 8599. Springer, 2014, pp. 137–149.

[25] ——, "Specialising software for different downstream applications using genetic improvement and code transplantation," *IEEE Trans. Softw. Eng.*, vol. 44, no. 6, pp. 574–594, 2018.

[26] M. Soto, "Improving patch quality by enhancing key components of automatic program repair," in *Automated Software Engineering*. IEEE, 2019, pp. 1230–1233.

[27] M. Wen, J. Chen, R. Wu, D. Hao, and S. Cheung, "Context-aware patch generation for better automated program repair," in *International Conference on Software Engineering*. ACM, 2018, pp. 1–11.

[28] C. Wong, J. Meinicke, and C. Kästner, "Beyond testing configurable systems: Applying variational execution to automatic program repair and higher order mutation testing," in *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 749–753.

[29] Y. Jia, F. Wu, M. Harman, and J. Krinke, "Genetic improvement using higher order mutation," in *Genetic and Evolutionary Computation Conference*. ACM, 2015, pp. 803–804.

[30] S. Sun, J. Guo, R. Zhao, and Z. Li, "Search-based efficient automated program repair using mutation and fault localization," in *Computer Software and Applications Conference*. IEEE, 2018, pp. 174–183.

[31] T. Xu, "Improving automated program repair with retrospective fault localization," in *International Conference on Software Engineering*. IEEE / ACM, 2019, pp. 159–161.

[32] X. Xu, C. Zou, and J. Xue, "Every mutation should be rewarded: Boosting fault localization with mutated predicates," in *International Conference on Software Maintenance and Evolution*. IEEE, 2020, pp. 196–207.

[33] J. Yi, S. H. Tan, S. Mechtaev, M. Böhme, and A. Roychoudhury, "A correlation study between automated program repair and test-suite metrics," vol. 23, no. 5, 2018, pp. 2948–2979.

[34] J. Petke, B. Alexander, E. T. Barr, A. E. Brownlee, M. Wagner, and D. R. White, "A survey of genetic improvement search spaces," in *Genetic and Evolutionary Computation Conference*. ACM, 2019, pp. 1715–1721.