Representations and Operators for Improving Evolutionary Software Repair

Claire Le Goues
University of Virginia
Charlottesville, VA 22903
legoues@cs.virginia.edu

Westley Weimer
University of Virginia
Charlottesville, VA 22903
weimer@cs.virginia.edu

Stephanie Forrest
University of New Mexico
Albuquerque, NM 87131
forrest@cs.unm.edu

ABSTRACT

Evolutionary computation is a promising technique for automating time-consuming and expensive software maintenance tasks, including bug repair. The success of this approach, however, depends at least partially on the choice of representation, fitness function, and operators. Previous work on evolutionary software repair has employed different approaches, but they have not yet been evaluated in depth. This paper investigates representation and operator choices for source-level evolutionary program repair in the GenProg framework [17], focusing on: (1) representation of individual variants, (2) crossover design, (3) mutation operators, and (4) search space definition. We evaluate empirically on a dataset comprising 8 C programs totaling over 5.1 million lines of code and containing 105 reproducible, human-confirmed defects. Our results provide concrete suggestions for operator and representation design choices for evolutionary program repair. When augmented to incorporate these suggestions, GenProg repairs 5 additional bugs (60 vs. 55 out of 105), with a decrease in repair time of 17–43% for the more difficult repair searches.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; F.2.2 [**Artificial Intelligence**]: Search

General Terms

Algorithms

Keywords

Representation, crossover, mutation, search-based software engineering, software repair, genetic programming

1. INTRODUCTION

Software maintenance, including error repair, refactoring, performance or optimization, entails considerable economic cost [8, 23]. Accordingly, there is significant interest in applying evolutionary computation (EC) to repair or otherwise transform existing software. Unlike earlier work on Genetic Programming (GP), such approaches focus on large corpora of legacy software written in traditional languages [16, 28]. EC has also been applied recently to program-transformation problems such as shader simplification [25], CUDA kernel synthesis [14], and other languages including byte and assembly codes [19, 22].

Success in these domains depends to some degree on choices of representation, fitness functions, genetic operators, and the effective search space. Fitness functions for software repair have been studied using test cases [7, 28], formal specifications [27], or both [4, 9]. However, there has been little work comparing representations, operators or the effective search spaces in this domain.

Recent research has proposed a number of options. Representation choices include: Abstract Syntax Trees (ASTs) with a weighted list of statements to mutate [28] (referred to in this paper as AST/WP); lists of edits to an initial program [1, 16] (referred to as PATCH); or lists of bytecode [20] or assembly statements [22]. Mutation operators range from highly specialized (e.g., changing relational operators or variable names [1]); to pre-specified by the GP engine based on language-specific primitives (e.g., [5]); to grammar-based rewrite rules [15]; to generic insertions, replacements, or deletions of statements in an AST [28]. Certain applications appear to demand a prominent treatment of particular operators (e.g., a semantics-preserving crossover operator for Java bytecode [20]). Popular crossover choices include crossback [5, 28], one-point [1, 22], and variations of uniform crossover [16]. Parameter choices vary, and tend to be selected heuristically or by deferring to previous work. By and large, Arcuri *et al.*'s [5] summary is characteristic: "In general, we set the parameters with values that are common in literature . . . the ones reported are the best we found."

The goal of this paper is to conduct an in-depth study of several critical representation, operator and other choices used for evolutionary program repair at the source code level. We use GenProg, a previously established evolutionary software repair framework [17], and a benchmark set proposed in previous work [16] to conduct these controlled comparative studies. We claim the following contributions:

- A direct comparison of two source-level representations and an analysis of which features contribute most highly to success rate. We find that the PATCH representation outperforms the AST/WP in terms of repair

GECCO'12, July 7-11, 2012, Philadelphia, Pennsylvania, USA.

success rate for all tested crossover operators. A semantic check is critical to repair success in either representation.

- An empirical evaluation of competing crossover operators. We find that one-point crossover provides the best tradeoff between success rate and repair time, requiring approximately 25% less time for successful repairs and improving success rates by 4% compared to a default baseline.
- An empirical study of the role of mutation operators in successful repairs and the effect of varying the probabilities with which different operators are applied. For the most challenging repairs, an unequal operator selection function improves success rate by 9% and decreases repair time by 40%.
- An evaluation and analysis of the assumption made in earlier work [7, 28] that genetic modifications should focus on statements executed exclusively by buggy inputs (fault localization). A more balanced weighting scheme increases the number of bugs repaired, improves success rate per trial, and decreases repair time by 50–75% on the most challenging repair searches.

We conclude with several concrete recommendations for operator and design choices for EC-based program repair. Overall, we find that these choices individually matter, especially for the more difficult repair scenarios in terms of both repair success and time. These otherwise disjoint features combine synergistically: we show that an EC that incorporates our recommendations can automatically repair more bugs (60 vs. 55 out of 105), with a decrease of 17–43% in repair time for the more difficult repair searches as compared to previous work.

2. RELATED WORK

This study builds on earlier work [17], which uses GP to repair extant software. Arcuri [5] also proposed a GP for co-evolving defect repairs and unit test cases, relying on formal specifications. In his representation, the operators and representation are defined by a pre-existing GP engine based on toy language primitives. Recently, Debroy and Wong independently validated that mutations targeted to statements that are likely faulty can repair bugs successfully [7]. Their work uses an out-of-the-box fault localizer [12] to weight statements for possible mutation, but does not compare different mutation operators, representations, or weighting schemes. As mentioned earlier, EC has been extended to new languages [19, 1, 22] and domains [25, 14]. White *et al.* used GP to improve non-functional program properties, such as execution time [29].

Much of this previous work focuses on adapting EC to new domains, with operators and representation choices receiving attention only as necessary. For example, Orlov and Sipper outline a semantics-preserving crossover operator for Java bytecode [19]; Ackling *et al.* propose a patch-based representation to encode Python rewrite rules [1]; Le Goues *et al.* briefly examined two representation choices [16], and Forrest *et al.* quantified operator effectiveness, and compared crossback to traditional crossover [10]. However, a dedicated analysis of competing operators and representations choices for this domain has not yet been undertaken.

Recent work described a large benchmark set of real-world bugs [16], which formed the basis of a systematic software engineering-focused study of EC-based program repair. That work used one set of parameters and operators to facilitate an unbiased study but also considered a second representation and set of operators for the sake of scalability and efficiency. The second approach's greater performance was left unexplained, raising a set of questions, which we study in detail in the following sections. We adopt the benchmarks and default parameter set from this earlier work, but focus directly on fundamental representation, operator and related search space choices, measuring how they relate to success rate and repair time.

The field of Search-Based Software Engineering (SBSE) [11] uses evolutionary and related methods for software testing, e.g., to develop test suites [26, 18]. SBSE also uses evolutionary methods to improve software project management and effort estimation [6], find safety violations [3], and in some cases re-factor or re-engineer large software bases [24]. SBSE focuses primarily on improvements to fitness functions instead of on representation and operator choices.

3. TECHNICAL APPROACH

We investigate operators and representation choices for evolutionary repair of bugs in C programs using the GenProg tool, as outlined in Le Goues *et al.* [17]. In this approach, GP begins with a working program that has an identified bug, and searches for a program that avoids the bug while retaining required behavior. Example bugs include: crashing on a malformed input, looping forever, or producing the wrong result. The input to GenProg is a C program, a *negative* test on which the program behaves incorrectly, and a set of regression (*positive*) test cases encoding required functionality. Each individual in the GP is a program that can be evaluated by its performance on the positive and negative test cases. We use tournament selection and compute individual fitness as the weighted sum of the positive and negative test cases that the variant passes. Mutation and crossover can be restricted to certain segments of the program, for example using fault localization methods in software engineering (e.g., [12]). GenProg terminates after a fixed number of iterations, or when it finds a program that passes all test cases, known as an *initial repair*. Delta-debugging [30] is applied post-facto to minimize differences between the repaired version of the program and the original. This results in the *final repair*. The minimization can be applied either to the mutation operations that produced the final variant or to the source code of the variants themselves [28] using tree-based structural differencing [2].

This general outline highlights four key areas of algorithm design that are the foci of this paper: representation (Section 3.1), crossover (Section 3.2), mutation and selection (Section 3.3), and search space (Section 3.4). The rest of this section outlines these areas in more detail. Section 4 gives empirical results.

3.1 Representation

We restrict our attention to source-level GP representations in GenProg, and investigate the two best-established options.

The *Abstract Syntax Tree/Weighted Path* (AST/WP) representation (e.g., [28]) defines a program variant as a pair consisting of its AST (with each statement uniquely numbered) and a *weighted path* through it, typically defined by the statements executed by the failing test case. The weighted path is defined for the original program and remains unchanged during the search, even if introduced variations change control flow. Fitness is evaluated by pretty-printing the AST to produce source code, which is then compiled and run on the test cases.

The PATCH representation defines an individual as a sequence of edits to the original program source. Ackling *et al.* [1] intoduced this method, storing the edit list as a bitvector where each bit indexes an array of possible mutations to the underlying code. An alternative approach represents each patch as a variable-length sequence of edits to the statements of the original program's AST [16]. We study the latter representation for several reasons: (1) it admits a wider range of mutations, (2) it does not require pre-enumeration of all possible mutations (improving scalability), and (3) it is directly comparable to related work, especially the AST/WP representation in the context of the GenProg implementation. To evaluate fitness, each edit in the list is applied to the original program, the resulting AST is printed as source code, and that code is compiled and run on the test cases.

3.2 Crossover

Program source representations support several crossover operators, only some of which we study here. For example, earlier work proposed a *crossback* operator [5, 28], but subsequent work found it equivalent to traditional one-point crossover [10].

In the AST/WP representation, one-point crossover applies to the Weighted Path. Given two parents p and q, a point along the weighted path is selected at random, and all statements after that point are swapped between the parents to produce two offspring. Only statements along the weighted path are affected.

The *patch subset* operator is a variant of uniform crossover for the PATCH representation [16]. This operator takes as input two parents p and q. The first (resp. second) offspring is created by appending p to q (resp. q to p) and then removing each element randomly with 50% probability. This operator allows edits to similar ranges of the program to be combined into one individual (e.g., parent p inserts B after A and parent q inserts C after A). This contrasts with AST/WP one-point crossover, where each offspring can receive only one edit to statement A.

One-point crossover on the patch representation [21] selects crossover points p_n and q_m in parents p and q. The first half of p is appended to the second half of q, and vice versa, to create two offspring.

3.3 Mutation

In both representations, mutation is restricted to AST nodes corresponding to C statements. A destination statement d is chosen from the set of permitted statements according to a probability distribution (see Section 3.4). One of the available mutation operators is then selected (with equal probability in previous work). The available mutation operators are **delete**, **insert**, and **swap** operators. In some recent papers, **replace** is substituted for **swap** because **swap** was found to be up to an order of magnitude less successful than the others [10, Tab. 2]. If **insert** or **swap/replace** are selected, a second statement s is also selected randomly. Statement d is then either swapped with s, replaced with s, or replaced with a new statement consisting of d followed by an inserted s. These changes are either applied directly to the AST (in the AST/WP representation) or appended to a list of edits (in the PATCH representation).

3.4 Search space

The search space of possible programs in a given representation is infinite, but existing techniques restrict the search to a smaller space that is likely to contain a repair. We propose a parameterization of these restrictions along three dimensions:

- **Fault space.** The search space is reduced by restricting mutations to program locations associated with incorrect behavior. These locations (i.e., program statements) can be weighted to influence their probability of being mutated, such as by Tarantula [12], suspiciousness values [7], or using heuristics. As an example, one heuristic weights statements visited only by the negative test cases 10 times more highly than those visited by both the negative and positive test cases [28]. The fault space size (defined as the sum of all weights) appears related to the repair success [10].
- **Mutation space.** The search space is further constrained by the set of mutations that are possible at each location and their selection probability. Typically, mutations are selected with equal random probability.
- **Fix space.** To review, earlier work on EC program repair typically copies (inserts/replaces/swaps) or deletes code, rather than generating truly random mutations. In the case of a copy, unlike a simple delete, a statement must be selected as the *source* to be copied. We refer to this as *fix localization.* Candidate fixes are restricted to those within the original program. A *semantic check* can also be applied to the fix space [16]. This eliminates the possibility of copying or moving statements that reference variables that will be out-of-scope in the new location. The semantic check is a special case of the operators proposed by Orlov and Sipper for well-typed Java bytecode mutation [19], applied to weakly typed C programs.

Thus, the search space is defined by the locations that can be changed, the mutations that can be applied, and the statements that can serve as sources of the repair. The search is further constrained by probability distributions defined over the space, e.g., the probability that a given location is selected, a given mutation is applied, or a given source statement is selected. These decisions constrain the mutation operators regardless of the representation choice.

In AST/WP, the fault space is explicitly represented by the *weighted path*, which defines the site of mutation operations and the basic unit (gene) used in crossover. The PATCH representation uses a similar weighting to guide selection of possible edit sites, although the weights are stored explicitly with the individual variant.

4. EXPERIMENTS

This section presents experimental results on four algorithmic and parameter choices for EC-based program repair:

- **Representation:** Does AST/WP or PATCH give better results in terms of finding repairs, and which representation features contribute most to success?
- **Crossover:** Which crossover operator is best for EC-based program repair?

Program	Fault	LOC	Tests
`gcd`	infinite loop	22	6
`uniq-utx`	segfault	1146	6
`look-utx`	segfault	1169	6
`look-svr`	infinite loop	1363	6
`units-svr`	segfault	1504	6
`deroff-utx`	segfault	2236	6
`nullhttpd`	buffer exploit	5575	7
`indent`	infinite loop	9906	6
`flex`	segfault	18775	6
`atris`	buffer exploit	21553	3
average		**6325**	**5.8**

Table 1: Benchmark C programs [28, Fig. 4], each with one defect and several human- or fuzz- generated test cases. Col. 1 gives the name of the program, Col. 2 the type of bug in the program, Col. 3 gives the size of the program in lines of code, and Col. 4 gives the number of test cases.

Program	Description	kLOC	Tests	Bugs
`fbc`	Basic comiler	97	773	3
`gmp`	math library	145	146	2
`gzip`	data compression	491	12	5
`libtiff`	image manipulation	77	78	24
`lighttpd`	web server	62	295	9
`php`	web coding	1,046	8,471	44
`python`	general coding	407	355	11
`wireshark`	network analyzer	2,814	63	7
total		**5,139**	**10,193**	**105**

Table 2: 105 historical C defects [16, Tab. 1], with test suites. Each benchmark contains at least 2 testable defects. Col. 1 gives the program name, Col. 2 describes the program's functionality, Col. 3 gives the program size (in lines of code × 1000), Col. 4 the number of test cases, and Col. 5 the number of defects considered in the program.

- **Operators:** Which operators contribute the most to repair success, and how should the operators be selected?
- **Search space:** How should the representation weight program statements to best define the search space?

4.1 Experimental Setup

This subsection describes two benchmark sets and establishes baseline parameters and operators for our experiments.

Table 1 shows the first set of ten benchmark programs, taken from the work that introduced the AST/WP representation for program repair [28]. We include these benchmarks because the AST/WP representation does not scale (i.e., we cannot run it with our computational resources) to a number of the bugs from the larger benchmark set — the smaller benchmarks thus allow direct comparisons between PATCH and AST/WP. The second benchmark set, shown in Table 2 is an order-of-magnitude larger set of 105 real-world, human-repaired defects in open-source C programs [16]. This benchmark set was systematically generated by looking at human-repaired bugs associated with existing test suites in open-source C programs, and is thus viewed as more indicative. We prefer this benchmark set where possible because of its size and diversity.

For the purposes of comparison, we define default parameters and operator choices. The mutation operators are **replace**, **delete**, and **insert**. We evaluated a parameter set that used **swap** instead of replace; the results do not vary significantly (data not shown). We included **replace** to maintain compatibility with published results on the dataset [16]. The population size is 40. The GP is run for a maximum of 10 generations or 12 wall-clock hours,[1] whichever comes first. The tournament size is 2. The mutation rate is 1 mutation per individual per generation. Each individual variant undergoes crossover once per generation, regardless of the crossover operator. When a mutation is applied, one of the mutation operators is selected with equal random probability. The fault localization scheme assigns a weight of 1.0 to statements executed by only the failing test case and 0.1 to statements executed by both the failing negative test case and passing positive test cases. The fix localization scheme includes the *semantic check* described earlier. Fitness is the weighted sum of all tests passed, where the negative test cases are weighted twice as heavily as the total contribution from positive test cases. Variants that do not compile receive a fitness of 0.0. The fitness function samples a random 10% of the positive tests for the larger benchmarks: If a variant passes all tests in the sample as well as all of the negative test cases, it is then tested against the full suite.

On this "baseline" parameter set using the PATCH representation, GenProg repairs 55/105 of the bugs from Table 2 and 10/10 bugs in Table 1.

The primary metrics in all experiments are *success rate* (fractiohn of trials that produce a successful repair) and average number of *fitness evaluations to find a repair*, which is a proxy for repair time. Results are averaged over a number of repair *trials* (a *trial* is one run of the GP) each with different random seeds. Results on the smaller benchmarks in Table 1 are computed over 100 trials per tested repair scenario. The number of trials per bug for experiments using the larger benchmarks in Table 2 ranges from 10 to 20. We limited the number of trials to conserve resources, as we ran these experiments in a commercial cloud environment. We performed sufficient trials per experiment to report statistically significant results, and we report significance in terms of α (probability of an outcome under the null hypothesis) where applicable.
probability of the outcome under the null hypothesis)

4.2 Representation

In this subsection, we directly compare the PATCH and AST/WP representations. We ran GenProg on each benchmark in Table 1, testing both representations with and without the WP One-Point crossover operator (for direct comparison) and with and without the semantic check operator. We implement the WP one-point crossover in the patch representation by mapping the edits in the patch list to their corresponding statements in what would be the Weighted Path in the AST/WP representation (determined via the execution of the test cases), choosing a point along that path,

[1] These experiments were run in the Amazon EC2 cloud computing environment; wall-clock time is thus conservative to compensate for slower virtualized machine I/O.

Representation	Crossover	Semantic Check	Success Ratio
AST/WP	Yes	No	0.85
	No	Yes	0.94
	Yes	Yes	1.00
PATCH	Yes	No	0.95
	No	Yes	1.01
	Yes	Yes	1.14

Table 3: Repair success ratios between AST/WP and PATCH representation, with and without crossover and semantic check. Success ratios are normalized to the best performance of the AST/WP representation (with crossover and the semantic check). Higher is better. Experiments conducted on Table 1 benchmarks.

Crossover Operator	Success	Fitness Evals Req'd
No Crossover	54.4%	82.43
Patch Subset	61.1%	163.05
WP One-Point	63.7%	114.12
Patch One-Point	65.2%	118.20

Table 4: Success rates and effort to repair for different crossover operators using the PATCH representation. Higher success rates are better; lower number of fitness evaluations to find a repair are better.

and crossing over the edits that affect statements before and after that point.

Table 3 shows results, in terms of success rate ratio.[2] Results are normalized to the AST/WP representation including both crossover and the semantic check The semantic check strongly influences the success rate of both representations. Overall, PATCH outperforms AST/WP. We conclude that the PATCH representation, in addition to its scalability advantages for larger programs, provides a significant advantage over the AST/WP representation for the purposes of source-level evolutionary program repair.

4.3 Crossover

This subsection evaluates the role of crossover in the patch representation. We compare **Patch Subset**, **Patch One-Point**, **WP One-Point** (simulated in the patch representation as described in Section 4.2), and **No Crossover** repair scenarios. To isolate the influence of crossover, we restricted attention to programs with runs (using default parameters) that produced minimized repairs consisting of multiple mutations (suggesting that crossover might be important). We then reran GenProg on each such program, testing the different crossover operators. FIXME-Claire: Add a footnote documenting which programs we used.

Table 4 reports results in terms of GP success rate (higher is better) and repair effort (fitness evaluations; lower is better). **Patch One-Point** crossover finds repairs more often than all other options ($\alpha < 0.05$), and the difference in repair time between it and **WP Patch** is not statistically significant. By contrast, the **Patch Subset** operator leads to significantly longer repair times ($\alpha < 0.05$). Interestingly, leaving out crossover reduces repair time ($\alpha < 0.05$), but also significantly reduces the GP success rate ($\alpha < 0.01$). A possible explanation for these two results is that the bugs on which repair takes longer are also less likely to be repaired without crossover.

	Initial Repairs			Final Repairs		
Program	Ins	Del	Rep	Ins	Del	Rep
`fbc`	1.00	5.00	3.00	0.00	1.00	1.00
`gmp`	2.50	1.00	2.00	0.00	0.50	1.00
`gzip`	0.63	0.88	0.38	0.38	0.38	0.25
`libtiff`	0.54	0.97	0.76	0.25	0.39	0.47
`lighttpd`	0.61	1.22	1.14	0.04	0.31	0.65
`php`	0.26	0.46	0.48	0.18	0.39	0.45
`python`	0.33	0.70	0.17	0.00	0.80	0.20
`wireshark`	0.60	0.70	0.80	0.22	0.33	0.56
average	0.51	0.85	0.74	0.13	0.51	0.57

Table 5: GP operator frequency per repair on the baseline parameter set. "Initial" reports the frequency of insertions, deletions, and replacements in initial repairs; "Final" reports the same information for minimized repairs.

These results suggest that **Patch One-Point** crossover is preferable because it affords the best compromise between success rate and repair effort. It performs comparably to **WP One-Point** crossover in terms of repair effort (and is about 30% faster than the **Patch Subset** operator), but it significantly outperforms the other operators in terms of success rate by 2–10%.

4.4 Operators

This subsection quantifies the contribution of the different mutation operators in successful repairs and empirically evaluates the effect of an unequal operator selection probability function.

By default, GenProg selects between the three mutation types with equal random probability. However, the operations do not appear equally often in successful runs on the larger benchmark suite. Table 5 reports results, both for initial and for minimized repairs. The operator distribution for initial repairs on this benchmark set is closer to uniform (**insert** : **delete** : **replace** :: 1 : 1.7 : 1.45) than was observed in previous work [10]. We hypothesize that this is a result of a larger and more indicative benchmark set. For final repairs, however, the distribution appears much more strongly skewed in favor of deletions and replacements (1 : 3.84 : 4.3). Note that although performance does not change significantly when **swap** is used instead of **replace**, **replace** appears to be more important to these repairs than **swap** was in previous work.

These results suggest that the different mutation types are not equally important to the repair search. To test this hypothesis, we modified the operator selection probabilities to match the observed distribution (using 10-fold cross validation [13] to mitigate the threat of overfitting). We then ran new repair trials on a subset of the benchmarks.

Results are shown in Figure 1 (success rate) and Figure 2 (repair time). This setup did not lead to repairs to previously unrepaired bugs. However, both figures show that results vary significantly by the difficulty of the search prob-

[2] Average number of fitness evaluations to find a repair is equivalent between representations on this benchmark set, and therefore not shown.

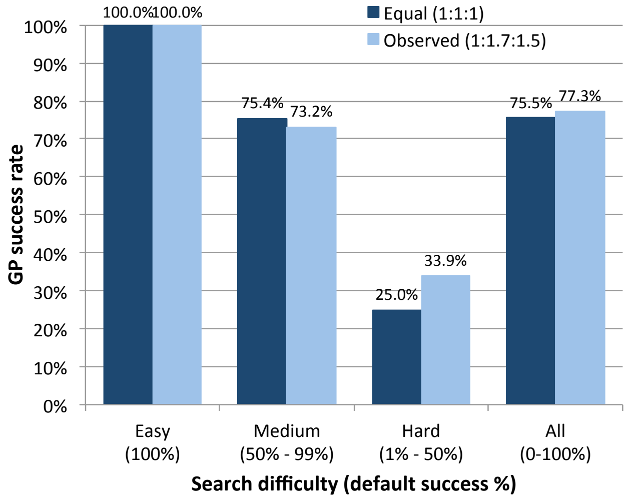

Figure 1: GP success rate for two mutation operator selection functions, binned by initial default success rate. Higher is better. The "Equal" selection function chooses between possible mutation operators with equal random probability. The "Observed" selection function is weighted to match the observed distribution of operators in automatic repairs.

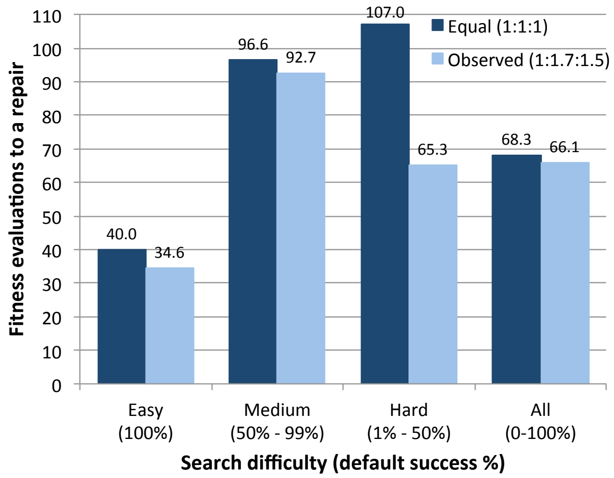

Figure 2: Fitness evaluations to a repair for two mutation operator selection functions, binned by success rate with default parameters. Lower is better. The "Equal" selection function chooses between possible mutation operators with equal random probability. The "Observed" selection function is weighted to match the observed distribution of operators in automatic repairs.

lem, as measured by success rate using the default parameter set. On balance, the search benefits from tuning the operator selection function. For bugs that GenProg repairs easily on the default parameter set, modifying the operator selection function does not significantly alter performance. On the most difficult bugs, defined as those on which the default parameter set is the least successful, tuned mutation distribution increased success rate by 8.9% and decreased repair time by 40%, respectively ($\alpha < 0.05$ for both).

4.5 Search Space

This subsection studies the weighting scheme used to direct mutation operators to particular areas of a program under repair. It seems intuitive that statements executed exclusively by the negative test cases are the most likely source of error and therefore should be the target of genetic modifications. The default parameter set weights such statements ten times more highly than those executed by both negative and positive test cases.

Table 6 classifies the statements modified in actual repairs according to whether they are executed exclusively by the negative test case or by both the positive and the negative test cases.[3] The data are extremely noisy. However, we observe that the vast majority of repairs do not follow the 10:1 ratio of the default weighting scheme. Over all repairs, the ratio of statements executed solely by the negative test cases to those executed by both negative and positive test cases averages to 1 : 1.85. FIXME-Claire: Can you uncomment the following and explain more clearly. SF couldn't follow enough to rewrite for you.

The results in Table 6 suggest that the 10:1 path weighting scheme of previous work may not be optimal. To test this hypothesis, we tried two additional weighting schemes on a subset of the benchmarks: a *Realistic* weighting scheme approximating the observed average, and an *Equal* weighting scheme, in which all statements along the negative path receive weight 1.0 (to mitigate against overfitting, as the data are extremely noisy, and to establish a baseline).

Figure 3 and Figure 4 show results in terms of success rate and number of fitness evaluations needed to find a repair. As with previous experiments, results are more striking for more difficult repairs. Both new weighting schemes let the GP repair a bug on which the default weighting scheme failed. With the exception of searches on which the default weighting scheme acheived 100% success, the alternative weighting schemes significantly ($\alpha < 0.05$ for both metrics) outperform the default scheme: success rates increase, and repair times decrease. The differences on the 100% bugs, while statistically significant, are small in terms of both percentages and wall-clock time. On the other hand, the new weighting schemes require only 25% and 50% of the time taken by the default weighting scheme on the more difficult repairs.

In practice, automatically-generated repairs do not modify the "exclusively negative" statements ten times more often than they do the "negative and positive" statements. On our dataset, both alternative weighting schemes provide performance comparable to the 10:1 default on the bugs where the default succeeds easily, while outperforming it, in some cases considerably, on more "difficult" bugs, to the point of finding one additional repair.

4.6 Limitations

Our results provide concrete suggestions for operator and representation choices for evolutionary program repair. However, several caveats apply. The benchmark programs, particularly those used to compare the PATCH and AST/WP representations, may not be representative of the spectrum of defective programs. We mitigate this threat by selecting

[3] We report data for initial (unminimized) repairs; the distribution for minimized repairs is similar.

Program	Num repairs	Neg. Only	Neg. or Pos.
`fbc`	1	50%	50%
`gmp`	2	14%	86%
`gzip`	8	36%	64%
`libtiff`	136	72%	28%
`lighttpd`	34	99%	1%
`php`	181	52%	48%
`python`	6	71%	29%
`wireshark`	9	100%	0%

Table 6: Location of successful repairs. "Num Repiars" shows the number of unique repairs. "Neg. Only" reports the average percentage of the statements modified by a repair that are executed only by the negative test case. "Neg. or Pos." is similar, but reports percentages for statements executed by both positive and negative test cases.

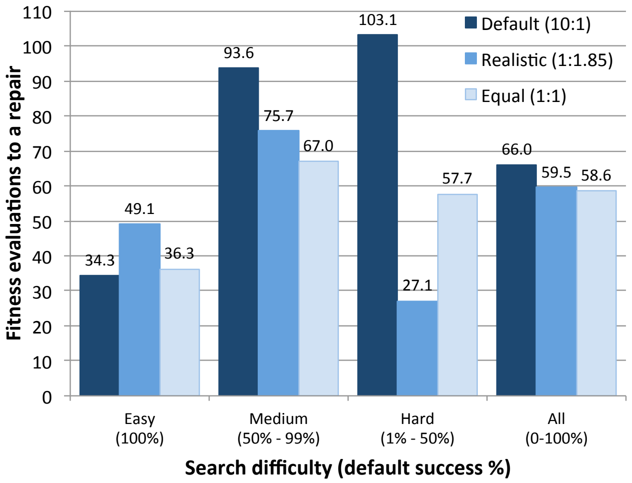

Figure 3: Repair effort (number of fitness evaluations to first repair) for the "Default", "Realistic", and "Equal" path weighting schemes, binned by success on the default parameter set. Lower is better.

these programs from a variety of domains and to be explicitly comparable to previous work [10, 28]. The benchmark set used for the majority of our experiments is large, generated systematically, and taken from real-world bugs in C programs, and thus is hopefully more indicative. This EC technique by necessity is limited in terms of what kind of bugs it can address; we do not claim that these results are generalizable to all bugs in all programs. Different choices may result in better performance on different bug classes. Additionally, in the pursuit of a controlled study and in the interest of brevity, we have made heuristic choices for a number of parameters, such as population size, tournament size, and mutation rate. We expect that an EC-based program repair approach is likely also sensitive to these parameter choices, but leave a parameter sweep for future work.

To guard against the risks of inappropriate or unindicative statistical results, we used nonparametric tests on smaller datasets to mitigate the risk of data that are not normally distributed. While our computational setup limits the number of trials we can run, we quantify α wherever applicable to increase confidence that the results are significant. We use cross- validation to mitigate the threat of overfitting where applicable and a uniform baseline for the search space experiments, when the distribution is too noisy to be significant.

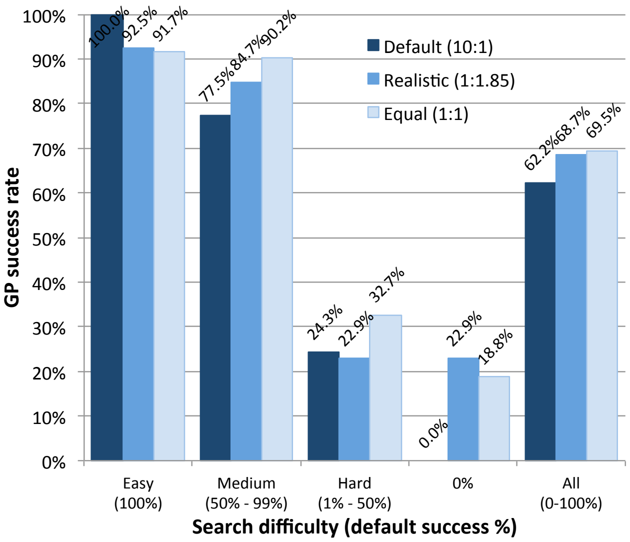

Figure 4: Repair success for the "Default", "Realistic", and "Equal" path weighting schemes, binned by success on the default parameter set. Higher is better for success rate. Both the Equal and Realistic weighting schemes outperform Default in terms of overall success rate ($\alpha < 0.05$).

5. CONCLUSIONS

EC-based software transformation is a promising technique for a number of applications. Despite its recency, several different representation and operator choices have already arisen, and it is timely to evaluate them in depth as the field matures. We focused on representation and operator choices at the source-code level and conclude with several concrete suggestions for evolutionary program repair in a framework like GenProg.

First, in addition to the scalability benefits afforded by storing individual variants as small lists of edits instead of entire program source trees, PATCH is more effective than the AST/WP representation in terms of repair success. The semantic check contributes importantly to repair success regardless of representation.

Second, a one-point crossover operator over the patch representation (**Patch One-Point**) offers the best tradeoff between repair success and time. It found repairs 28% faster than the **patch subset** crossover operator with a 4% improvement in success rate. Omitting crossover also decreases repair time, but it does so at the expense of success rate.

Third, **delete** is the most useful mutation operator of those we tested, followed by **replace** or **swap** (which are roughly equivalent), followed by **insert**. A biased mutation selection technique improves both success rate and repair time, particularly for more challenging repairs. On these examples, a biased operator selection function improved success rate from 25.0% to 33.9%, and decreases repair time by 40%.

Finally, the assumption that statements executed exclusively by negative test cases should be weighted much more heavily than those executed by both the positive and negative test cases is flawed: The actual distribution of modifications in final repairs, by and large, does not agree with this assumption. Changing this distribution improves both success rates and repair time, particularly on the more challenging bugs; on such examples, time to repair improves by up to 70%. Doing so also allows GenProg to repair bugs on which the default setup fails.

Taken together, these results suggest that operator choice, program representation, and probability distributions do individually matter, especially for difficult bugs. Although the features that contribute to bug repair difficulty warrant further study, the results in this paper suggest strongly that there are clear distinctions, and the more difficult examples are the most sensitive to such choices. In the end, these apparently disjoint features combine synergistically. When we modify our default parameter set with the recommendations outlined above and rerun the repair scenarios from [16], GenProg automatically repairs 5 additional bugs (60 vs. 55), and repair time decreases by 17–43% for the more difficult search scenarios.

6. ACKNOWLEDGMENTS

The authors gratefully acknowledge the support of the National Science Foundation (SHF-0905236, CCF-0954024, CCF-0905373), Air Force Office of Scientific Research (FA9550-07-1-0532, FA9550-10-1-0277), DARPA (P-1070-113237), DOE (DE-AC02-05CH11231) and the Santa Fe Institute.

7. REFERENCES

[1] T. Ackling, B. Alexander, and I. Grunert. Evolving patches for software repair. In *Genetic and Evolutionary Computation*, pages 1427–1434, 2011.

[2] R. Al-Ekram, A. Adma, and O. Baysal. diffX: an algorithm to detect changes in multi-version XML documents. In *Conference of the Centre for Advanced Studies on Collaborative research*, pages 1–11. IBM Press, 2005.

[3] E. Alba and F. Chicano. Finding safety errors with ACO. In *Genetic and Evolutionary Computation Conference*, pages 1066–1073, 2007.

[4] A. Arcuri. Evolutionary repair of faulty software. *Applied Soft Computing*, 11(4):3494–3514, June 2011.

[5] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *IEEE Congress on Evolutionary Computation*, pages 162–168, 2008.

[6] A. Barreto, M. de O. Barros, and C. M. Werner. Staffing a software project: a constraint satisfaction and optimization-based approach. *Computers and Operations Research*, 35(10):3073–3089, 2008.

[7] V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *International Conference on Software Testing, Verification, and Validation*, pages 65–74, 2010.

[8] L. Erlikh. Leveraging legacy system dollars for e-business. *IT Professional*, 2(3):17–23, 2000.

[9] E. Fast, C. Le Goues, S. Forrest, and W. Weimer. Designing better fitness functions for automated program repair. In *Genetic and Evolutionary Computation Conference*, 2010.

[10] S. Forrest, W. Weimer, T. Nguyen, and C. Le Goues. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computation Conference*, pages 947–954, 2009.

[11] M. Harman. The current state and future of search based software engineering. In *International Conference on Software Engineering*, pages 342–357, 2007.

[12] J. A. Jones and M. J. Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *Automated Software Engineering*, pages 273–282, 2005.

[13] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *International Joint Conference on Artificial Intelligence*, 14(2):1137–1145, 1995.

[14] W. B. Langdon and M. Harman. Evolving a CUDA kernel from an nVidia template. In *Congress on Evolutionary Computation*, pages 1–8, 2010.

[15] W. B. Langdon, M. Harman, and Y. Jia. Efficient multi-objective higher order mutation testing with genetic programming. *Journal of Systems and Software*, 83(12):2416 – 2430, 2010.

[16] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *International Conference on Software Engineering (to appear)*, 2012.

[17] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automated software repair. *Transactions on Software Engineering*, 38(1):54–72, 2012.

[18] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *Transactions on Software Engineering*, 27(12):1085–1110, 2001.

[19] M. Orlov and M. Sipper. Flight of the finch through the java wilderness. *IEEE Transactions on Evolutionary Computation*, 15(2):166–192.

[20] M. Orlov and M. Sipper. Genetic programming in the wild: Evolving unrestricted bytecode. In *Genetic and Evolutionary Computation Conference*, pages 1043–1050, 2009.

[21] J. E. Rowe and N. F. McPhee. The effects of crossover and mutation operators on variable length linear structures. In *Genetic and Evolutionary Computation Conference*, pages 535–542, 2001.

[22] E. Schulte, S. Forrest, and W. Weimer. Automatic program repair through the evolution of assembly code. In *Automated Software Engineering*, pages 33–36, 2010.

[23] R. C. Seacord, D. Plakosh, and G. A. Lewis. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[24] O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Genetic and Evolutionary Computation Conference*, pages 1909–1916, 2006.

[25] P. Sitthi-Amorn, N. Modly, W. Weimer, and J. Lawrence. Genetic programming for shader simplification. *ACM Transactions on Graphics*, 30(5), 2011.

[26] S. Wappler and J. Wegener. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In *Genetic and Evolutionary Computation Conference*, pages 1925–1932, 2006.

[27] W. Weimer. Patches as better bug reports. In *Generative Programming and Component Engineering*, pages 181–190, 2006.

[28] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering*, pages 364–367, 2009.

[29] D. R. White, A. Arcuri, and J. A. Clark. Evolutionary improvement of programs. *IEEE Trans. on Evolutionary Computation*, 15(4):515 –538, aug. 2011.

[30] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Foundations of Software Engineering*, 1999.