# Designing Better Fitness Functions for Automated Program Repair

Ethan Fast
University of Virginia
ejf3z@virginia.edu

Claire Le Goues
University of Virginia
legoues@virginia.edu

Stephanie Forrest
University of New Mexico
forrest@cs.unm.edu

Westley Weimer
University of Virginia
weimer@virginia.edu *

## ABSTRACT

Evolutionary methods have been used to repair programs automatically, with promising results. However, the fitness function used to achieve these results was based on a few simple test cases and is likely too simplistic for larger programs and more complex bugs. We focus here on two aspects of fitness evaluation: efficiency and precision. Efficiency is an issue because many programs have hundreds of test cases, and it is costly to run each test on every individual in the population. Moreover, the precision of fitness functions based on test cases is limited by the fact that a program either passes a test case, or does not, which leads to a fitness function that can take on only a few distinct values. This paper investigates two approaches to enhancing fitness functions for program repair, incorporating (1) test suite selection to improve efficiency and (2) formal specifications to improve precision. We evaluate test suite selection on 10 programs, improving running time for automated repair by 81%. We evaluate program invariants using the Fitness Distance Correlation (FDC) metric, demonstrating significant improvements and smoother evolution of repairs.

## Categories and Subject Descriptors

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

## General Terms

Algorithms

## Keywords

Software repair, genetic programming, software engineering

*This research was supported in part by National Science Foundation Grants CCF-0621900, CCF-0905236, CCF-0954024, CCR-0331580, and CNS-0716478. It was also supported by Air Force Office of Scientific Research MURI grant FA9550-07-1-0532 and gifts from Microsoft Research.


*GECCO'10,* July 7–11, 2010, Portland, Oregon, USA.


## 1. INTRODUCTION

Maintaining and repairing buggy code is a significant component of the software life cycle [22], and several research efforts are exploring how to automate certain aspects of program repair, particularly in the software engineering and program language communities. Evolutionary computation is one promising approach to this problem (e.g., [7, 27]). Although the results reported in the original papers are promising, the fitness function is overly simplistic and unlikely to scale up to industrial size problems. In this paper we focus on fitness function enhancements, using the representation proposed by Forrest *et al.* [7].

There are two main approaches to assessing program correctness that are relevant to program repair evolution: test suites [7, 19, 27] and formal specifications [2, 26]. For test suites, the time taken to produce a repair scales with the product of the number of candidates investigated and the time to run the test suite, a problem of fitness function *efficiency*. In practice, complex or critical programs have large, long-running test suites [14, 21]. Additionally, test suites exhibit "all-or-nothing" behavior, making partial solutions difficult to reward. Consider, for example, a bug that can be repaired by adding two distinct function calls: a lock acquisition, and a lock release. A candidate repair formed by adding only the lock acquisition should, ideally, have a higher fitness value than the original program. Using standard test cases, however, it will presumably have the same fitness or lower (e.g., as the unbalanced locking call may introduce deadlocks). We refer to this as a problem of fitness function *precision*. In this paper, we propose and evaluate methods to address both of these problems, although we have not yet integrated them into a single unified solution.

We address fitness function *efficiency* by adapting software engineering techniques for test suite selection [20, 23], allowing fitness evaluation to incorporate hundreds of test cases. Each individual is evaluated on only a small random subset of the test suite. Once the population has found an individual that can pass its subset, it is then tested against the entire test suite, thus guaranteeing that correctness is not sacrificed for efficiency. This reduces the cost of each fitness evaluation, but introduces a risk of "leading the search astray" by introducing noise into the fitness function. Our results show that this is not a problem in practice, a result corroborated by a large body of work in change impact analysis (e.g., [13, 17, 18, 20]), which demonstrates that localized changes need not affect a program's behavior on test cases unrelated to those changes.

We address fitness function *precision* by augmenting fit-

ness functions based on test cases with automatically learned program invariants. For example, even though a partial repair may not fully pass a test case, it could correctly maintain an invariant such as `x != 0`. In a pre-processing step, we derive a set of predicates that predict buggy behavior in the baseline program, using its positive and negative test cases. We then instrument programs to record the values of each predicate at run-time [6]. Finally, we combine this information using measures such as "fraction of the important predicates that should be `true` that are made `true` by this variant" and use them to augment the fitness function. This leads to a smoother fitness gradient, which we quantify using Fitness Distance Correlation (FDC) [11].

Enhancing the fitness function is a key step for scaling evolutionary approaches to program repair, so that they can tackle larger programs and more complex bugs. This paper addresses this goal with following contributions:

- Fitness function enhancements for evolutionary program repair, which improve efficiency and precision.
- An empirical evaluation of fitness function performance. We replicate repairs from prior work with 81% improvement in running time. We measure precision using FDC, obtaining an improvement from 0.04 to 0.65 over earlier fitness functions. We show that the enhanced function is more precise in terms of mutation distance of variants from a repair.
- New repairs for programs with large test suites. We repair 10 programs with 1206 total test cases, an order of magnitude more than the 58 total test cases reported in previous work [27].

# 2. BACKGROUND

This section describes relevant previous work in genetic programming for automated program repair, noisy fitness functions, test suite selection, dynamic predicates, and fitness distance correlation.

## 2.1 Automated Program Repair using GP

Earlier work on automated program repair demonstrated a framework for GP-based patch generation [27] and explained the evolutionary characteristics of the approach [7]. In this work, a GP evolves programs that avoid a particular bug. The GP system begins with a working program in C, an input that causes the program to behave incorrectly, and a set of regression tests that the program passes. Individuals in the GP system's population are variants of the original buggy program. An abstract syntax tree (AST) represents each program variant. Test cases serve as the fitness function, where a *test case* consists of input to the program (e.g., an image to be processed, or an HTTP request to be served) and an *oracle comparator* function that defines the correct program response [4]. A program *passes* a test case if it produces the expected output when run on the input, as defined by the oracle comparator; otherwise, it *fails* the test case. A *positive test case* is a standard (regression) test case that encodes correct program behavior; the program's existing test suite comprises the positive test cases. A *negative test case* is a program input that demonstrates the bug and a comparator that detects it. To compute a variant's fitness, its AST is printed as source code, compiled, and then run against test cases in a sand box. The weighted sum of the total number of positive and negative test cases passed is its fitness. A *repair* is a program variant that passes all test cases. As a post-processing step, redundant code can be eliminated from the repair with program minimization techniques (the *final repair*).

## 2.2 Fitness Function Design

Test suite selection introduces noise into fitness evaluation and it is well known that evolutionary algorithms perform well in the presence of noise e.g., [8]. There has been significant work assessing the impact of noisy fitness functions in different contexts (e.g., [10]), but most of this work focuses on evaluating an individual several times to get a better estimate of the true fitness. Our approach addresses the slightly different problem of partial function evaluation through sampling [28]. This resembles function approximation, where the candidate function (individual) is evaluated on a finite number of inputs. Although sampling methods have been studied in the context of Search Based Software Engineering (e.g., [24, 25]), they have not to our knowledge been used previously on the problem of automated software repair. The closest work to our predicate-based fitness evaluation is that of Arcuri [2] who proposed using formal methods in a GP to automate software repair, demonstrating the idea on a hand-coded example of bubble sort.

## 2.3 Test Suite Selection

Reducing the costs associated with testing is a mature area of software engineering research. We use the term *prioritization* to describe techniques that permute test orders to find defects more quickly and *reduction* (e.g., [9]) to describe techniques that remove test cases from consideration and thus reduce execution time. Of special interest are *impact analysis* techniques that use static and dynamic information to identify tests that could possibly be affected by a source code change.

We consider two sampling algorithms, both of which are parametric with respect to the desired test suite size. *Random sampling* chooses test cases uniformly at random, without replacement, and *time-aware test suite reduction* as described by Walcott *et al.* [24] uses the GA to select a test suite that optimizes for both coverage and efficiency.

Li *et al.* empirically evaluated greedy algorithms for test suite prioritization [14]. Notably, greedy algorithms that found local minima within the search space sometimes produced suboptimal results. A greedy algorithm applied to the scenario considered in this paper suffers from all the difficulties they describe, such as concerns about fitness metric choice and landscape properties. Our experiments showed that a naïve greedy sampling algorithm performed similarly to a purely random one.

## 2.4 Impact Analysis

Although test suite reduction has a number of advantages, it is not always appropriate (e.g., even though reduction approaches can maintain 100% requirement coverage, they may have different fault detection behavior from the original test suite [23]). Thus, developers often prefer *impact analysis*-based approaches [13, 17, 18, 20] that use static and dynamic analyses to determine which tests could possibly be affected by a source code change.

The CHIANTI tool [20] is an example of a safe [21] change impact analysis. The tool decomposes program edits into sets of atomic changes and produces two types of informa-

tion. First, it enumerates a set of all possibly-affected test cases for the set of changes. Second, for each test case, it produces a list of which changes are responsible for the change in behavior. Empirical results show that an affected unit test is often influenced by only 4% of atomic changes. Impact analysis is relevant to evolutionary program repair because CHIANTI's atomic actions are similar to the actions of our mutation operator [27]. The DEJAVU tool of Rothermel and Harrold is another example of a safe analysis; it emphasizes statement-level coverage [21].

## 2.5 Dynamic Predicates

There are several mature techniques for automatically learning program invariants, or *predicates*, that characterize normal or erroneous program executions [15]. These techniques are used in specification mining [6], dynamic repair of deployed software in an N-variant system [19], and error isolation [15], among other applications. In this work, a program is instrumented to track predicates over program values, such as those contained in branch conditions. The instrumented program is then executed on an indicative workload or monitored during normal operation. Each run produces both program output and a set of predicates that were true over the course of the run.

These sets can be statistically analyzed to identify which are most strongly correlated with program failure. Given a predicate $P$, Liblit *et al.* [15] define $Failure(P)$ as the probability that the program will fail given that $P$ is true, $Context(P)$ to be the probability that the program will fail given that the line on which $P$ is tested is ever reached, and $Increase(P)$ to be the amount that $P$ being true increases the probability of failure. The result of this analysis is a set of facts about the program that are correlated with failure, such as "the branch condition at line 5 in file `foo.c` is often true." In Section 4.4 we adapt these measurements to produce a finer-grained fitness function.

## 2.6 Fitness Distance Correlation

*Fitness distance correlation* (FDC) measures the correlation between a fitness function and an estimate of the true distance between an individual and the global optimum [11]. In bit-based genetic algorithms, distance is typically estimated by the Hamming Distance between the individual and the global optimum. Applying FDC to program repair presents two complications: (1) because we are operating in the space of ASTs, Hamming Distance is not a natural metric; and (2) there may be many global optima (i.e., many repairs that pass all the test cases). These complications are addressed in Section 4.4.

Jones and Forrest used the correlation coefficient $r$ (covariance divided by the product of two standard deviations) between fitness and true distance to approximate the difficulty of a GA search problem [11]. Their results suggested that $(-0.15 < r < 0.15)$ signifies a difficult GA problem, where the fitness function is not well correlated with true distance.[1] We use FDC to evaluate the original fitness function described in [7, 27] and compare the results to those of our enhanced fitness function which includes automatically derived program invariants.

[1]Desired numbers vary depending on whether minimization or maximization problems are considered; we treat program repair as a minimization problem in our FDC analysis.

# 3. FITNESS FUNCTION ENHANCEMENTS

As stated earlier, our goal is to improve the fitness function for GP-based program repair by increasing both its efficiency and its precision. In the original work, fitness evaluation was limited to six [27] to twenty test cases [7]. If the fitness function does not provide useful information about partial solutions, the GP system is reduced to random search because there is no fitness gradient correlated to program structure. Complex repairs that require combining separate portions of a solution from two distinct variants will thus benefit from a good fitness function. We hypothesize that the original fitness function based on a small number of all-or-nothing test cases is not sufficient for this purpose.

## 3.1 Fitness Efficiency: Test Suite Selection

We use *sampling* to reduce test suite size for positive tests. When a program variant's fitness is evaluated, a subset of the positive test suite is chosen, and the program is run against that subset together with all negative test cases. If the variant passes all test cases in its subset, it is then tested against all remaining test cases (the *final test*) to determine if it is correct. If the variant fails the final test, its assigned fitness value is its score against its original sample. The original method is a special case of this framework in which a *retest-all* sampling strategy is used. Note that in the new scheme, every individual is assigned its own sample on each generation, potentially introducing noise to the fitness evaluation. A sampling-based fitness function will save time if the reduced cost outweighs the increased number of fitness evaluations required due to noise.

This framework handles positive and negative test cases asymmetrically: individuals are always evaluated against all negative test cases to ensure a maximally correct repair. The final test guarantees that the sampling-based fitness function is as correct as the original algorithm: A candidate repair is only reported if it passes all test cases, just as with the original algorithm.

## 3.2 Fitness Precision: Dynamic Predicates

In a pre-processing step, we instrument and run the buggy program on all test cases to record observed program predicates. This produces several interesting sets:

- *Increase Set*: predicates $P$ such that $Increase(P) > 0$.
- *Context Set*: predicates $P$ such that $Context(P) > 0$.
- Four *Universal Sets* of predicates: (1) those always true on all test case executions, (2) those always false on all test case executions, (3) those true only on executions that produce the correct output, and (4) those false only on executions that produce the correct output.

We write $B$ ("baseline") to refer to the set of those six predicate sets for the original buggy input program. We also observe predicate behavior for each variant $i$ in a GP population and write $V_i$ for the set of predicate sets thus observered. We characterize the difference between a variant and the original program by comparing $V_i$ to $B$. This characterization estimates the distance to a hypothetical repaired program.

We can use the sets $B$ and $V_i$ to define a number of concrete metrics that may characterize important differences in program behavior. For example, we can ask which or how

many predicates that once predicted failure are no longer observed on failing runs in variant $i$; or how many predicates that were never observed on failing runs are observed on failing runs in variant $i$. We can simply count the cardinality of these set differences; we can compute a weighted sum of the *Context* or *Increase* scores of the contained predicates; or we can weight the result by the number of successful or failing runs on which the predicates are observed; and so forth.

In total, considering all possible difference sets, weightings, and test cases passed and failed, each individual $i$ is described by a large set of scalar features $f_i$. Since we don't know *a priori* how these features relate to fitness, we use learning to identify a linear combination of the features. A training set of example variants is used to learn global coefficients $c_j$ ($j$ indexes metrics in the feature set $f_i$) and a global intercept $c_0$, which are then used to compute the fitness for each individual $i$:

$$\mathsf{predicate\_fitness}(i) = c_0 + \sum_j c_j f_{i,j}$$

This learning is applied only once per buggy program and is detailed in Section 4.4. We hypothesize that good variants change the feature values of predicates correlated with failed runs and are unlikely to change those of predicates correlated with successful runs. We further conjecture that if the features $f_{i,j}$ are combined as just described, the resulting fitness function will improve the FDC. The experimental results in the next section support this claim.

## 3.3 Optimal Fitness

Computing the FDC requires knowing the distance between any individual and the *optimal fitness*. When evolving strings of bits, the Hamming distance between an individual and the optimal solution often suffices. For numerical optimization, the Euclidean distance may serve. In the domain of program repair, however, there are often multiple solutions and the abstract syntax tree representation does not have an obvious metric.

We define a tree-structured distance metric for quantifying the distance between two program variants. A natural approach might employ graph isomorphism algorithms. However, they are rarely used on ASTs or control flow graphs because of efficiency concerns (e.g., subgraph isomorphism is NP-complete and even a 5000-line program may have thousands of AST nodes). Tree-structured difference algorithms are a common substitute. They typically gain speed at the expense of true accuracy by making assumptions (e.g., planar graphs or trees) or using heuristics (e.g., specially weighting leaf nodes when looking for a correspondence). We use the DIFFX structural differencing algorithm [1] which measures the difference between two ASTs in terms of a number of insertions, deletions and moves required to transform one into the other. These operations closely approximate those used by our GP system, so the tree-structured difference between a variant and a solution is essentially the number of atomic mutation operations required to reach the solution from the variant.

However, merely counting mutations is misleading, as some mutations have no semantic effect. Consider the example mentioned earlier of a program that requires two mutations to fix (e.g., inserting one locking function and one unlocking function call). If all mutations are weighted equally, a variant that adds one required function call and also inserts three no-ops will appear to be farther away from a repair than the original, when in fact it should have a higher fitness value. By minimizing the output of DIFFX we can identify extraneous changes and weight them lower than essential changes. We have found that a weighting factor of 1/10 for extraneous changes works well. Given a number of known final repairs, the optimal fitness $d_{opt}$ of a variant is thus its minimal weighted DIFFX distance to any of those known final repairs.

# 4. EXPERIMENTS

In this section, we present results from several experiments. First, we investigate performance gains from test suite sampling, demonstrating dramatic speedups in fitness evaluation. Second, we evaluate parent/offspring fitness correlation to explore the potential benefit of optimal safe change impact analysis. This result suggests that even a perfect understanding of parent/offspring fitness correlation would not admit equivalent performance gains. Third, we measure the FDC of the predicate-based fitness function, showing that predicates based on program invariants dramatically improve fitness function precision. The final experiment explores the shape of the fitness function in the vicinity of a valid repair, showing that dynamic predicates provide a smoother signal to the GP process than the previous approach.

## 4.1 Experimental Setup

The experimental benchmarks are shown in Figure 1. Some benchmarks were chosen for the purposes of direct comparison to previous work [7, 27], and appear here with expanded test suites; others are presented for the first time. The `gcd` program is a simple example; `imagemagick` and `tiff` are image processing applications; `leukocyte` is a computational biology program; `nullhttpd` and `lighttpd` are web servers; `zune` is the code responsible for the locking up over two million Microsoft music-playing devices on December 31st, 2008; the remaining programs are operating system utilities. The test suites for `deroff`, `gcd`, `look`, `uniq`, and `zune` are taken from Miller *et al.*'s work on *fuzz testing* [16], in which programs are subjected to random inputs. The web server tests consist of captured traffic requests from the University of Virginia Computer Science Dept. web server; the remaining programs use developer-provided or otherwise built-in test suites. Each program has a single defect exercised by a single negative test case. The defect and negative test case were taken from public forums or black hat security lists for `nullhttpd`[2], `lighttpd`[3], `zune` [3], `tiff`[4], `leukocyte`, and `imagemagick`[5].

We adopt GP parameters from previous work [27]. Population size is 40; the GP is run for a maximum of ten generations; *stochastic universal sampling* [5] selects individuals to propagate; the crossover rate is 1.0. A *trial* consists of at most two serial invocations of the GP loop using each of the following sets of values for path weight $W_{Path}$ and mutation

[2] http://nvd.nist.gov/nvd.cfm?cvename=CVE-2002-1496, http://www.mail-archive.com/bugtraq@securityfocus.com/msg09178.html

[3] http://nvd.nist.gov/nvd.cfm?cvename=CVE-2007-4727, http://www.milw0rm.com/exploits/4437

[4] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2285

[5] http://trac.imagemagick.org/changeset/511, http://www.shaik.com/pngsuite

| Program | Total LOC | Module LOC | # Tests | Test Suite Description | Defect Description |
|---|---|---|---|---|---|
| `deroff` utx 4.3 | 2236 | 2236 | 100 | fuzz input (as typesetting directives) | segfault |
| `look` utx 4.3 | 1169 | 1169 | 100 | fuzz input (for both needle and haystack strings) | segfault |
| `uniq` utx 4.3 | 1146 | 1146 | 100 | fuzz input (as duplicate-containing text file) | segfault |
| `zune` | 28 | 28 | 100 | fuzz input (as days since 1 January 1980) | infinite loop |
| `gcd` | 22 | 22 | 100 | fuzz input (as pairs of integers) | infinite loop |
| `lighttpd` 1.4.15 | 51895 | 3829 | 200 | HTTP requests from `cs.virginia.edu` | buffer overrun |
| `nullhttpd` 0.5.0 | 5575 | 5575 | 200 | HTTP requests from `cs.virginia.edu` | buffer overrun |
| `leukocyte` | 6718 | 6718 | 100 | video microscopy images | segfault |
| `tiff` 3.8.2 | 84067 | 1084 | 106 | image suite bundled with `tifflib` | segfault |
| `imagemagick` 6.5.2 | 450416 | 5858 | 100 | images from `pngsuite` test suite | wrong output |
| total | 603272 | 27665 | 1206 | | |

**Figure 1: Benchmark programs, with size in lines of code (LOC). "Total" shows total program size; "Module" shows the size of the portion of the program considered for repair. "# Tests" shows the total number of positive test cases.**

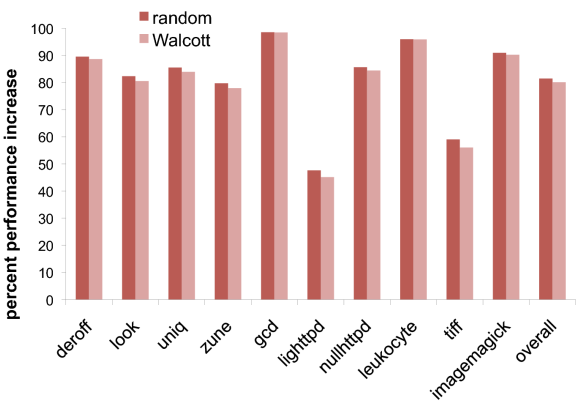


**Figure 2: Performance gains achieved by sampling-based fitness functions. "random" performs an unbiased random selection of the test suite; "Walcott" shows results using the Walcott *et al.* algorithm.**

chance $W_{mut}$, stopping early if a repair is discovered:

$$\{W_{Path} = 0.01, W_{mut} = 0.06\}$$
$$\{W_{Path} = 0.00, W_{mut} = 0.03\}$$

The primary metric used in the first two experiments is "average test case evaluations per successful repair", or `atce/r`. The `atce/r` value is the average number of positive test cases executed during a successful repair divided by the fraction of repair trials that succeed. For example, if 20,000 individual test case evaluations are required to produce a repair, but the GP system only produces a repair one time in three, the `atce/r` value is 60,000. `atce/r` approximates the average time and effort necessary to generate one successful repair. We report "test cases" rather than "wall clock seconds" to fairly compare algorithmic gains independent of how long a particular program's test cases take to run. A lower `atce/r` number indicates a more efficient repair.

## 4.2 Sampling-Based Fitness Performance

The first experiment measured the effect of test case sampling on GP performance. We studied two algorithms: *Random* which selects a subset of the test suite at random, without replacement, and *Walcott* which uses the test suite reduction algorithm of Walcott *et al.* [24]. Performance gains are reported in `atce/r` and are relative to the unmodified retest-all fitness function used in previous work [27].

In Figure 2 each bar represents the performance gain when 2% of the test suite is sampled on each fitness evaluation, averaged over at least 5 different repair trials. Performance gains scale almost linearly with the percentage of the test suite unsampled, up to 98% (data not shown). For both sampling strategies, all candidate final repairs were subsequently tested on all remaining test cases; the cost of this final test is included in Figure 2.

The results show that sampling reduces the effort required to produce a repair by 81%. The heavier-weight Walcott *et al.* method performs slightly worse because of its computational overhead, not because of the quality of the subsets selected. A typical single trial repair may require as many as 1520 fitness evaluations, each one of which requires selecting a sample. The overhead of Walcott *et al.*'s sampling algorithm as measured by `atce/r` is therefore negligible for programs with long-running test cases, such as `leukocyte`, but significant for test suites consisting of individually rapid tests, such as `lighttpd`. The Walcott algorithm's performance equals or exceeds that of Random when its overhead is ignored.

`lighttpd` and `tiff` are significant outliers, and represent worst-case scenarios for sampling test cases. The repairs for these are typically produced in under twenty fitness evaluations, minimizing the potential for improvement.

The performance impact of the sample-based fitness function is significant. The ten programs in Figure 1 were repaired in 1.8 minutes each, on average. On the same hardware, `leukocyte` was repaired in 6 minutes instead of over 90 minutes for the retest-all method, and `imagemagick` was repaired in 3 minutes instead of 36. The repairs produced by both approaches are equivalent.

## 4.3 Safe Impact Analysis

The second experiment investigates the hypothesis that the performance benefits obtained with the sampling-based fitness function arise from high parent/offspring fitness correlation. To test this hypothesis, we analyzed the effect of a perfect *change impact analysis* on the test suite and used the results to compare parent/offspring overlap. As mentioned earlier, change impact analysis determines which tests (regression, unit, etc.) will potentially be affected by a given program change (and therefore need to be rerun). Tests that are unaffected can be skipped, since their previous re-

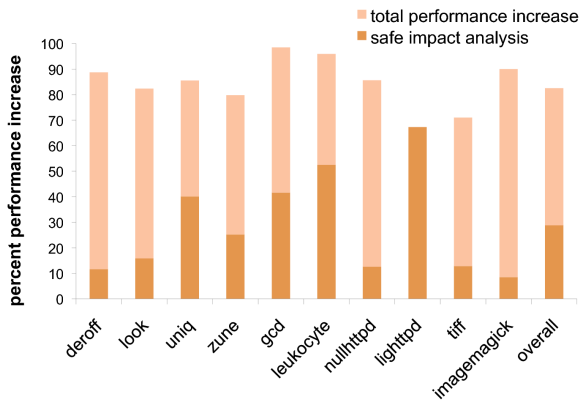


**Figure 3: Cumulative performance gain from optimal safe impact analysis compared to the fitness function described in Section 3.1. Optimal safe impact analysis improves performance (i.e., reduces the effort to construct a repair) by just over 29%, while the sampling-based fitness function improves performance by 81%.**

sults will remain unchanged. In the context of our system the fraction of test cases that are the same between parent and offspring[6] represents the maximal performance savings that a hypothetical perfect safe change impact analysis could provide to a fitness function.

We ran each benchmark program's entire test suite on each variant generated during a representative repair run. We measured the percentage of an offspring's individual test case results that were the same as its parent's. We repeated this process at least five times for each benchmark and calculated the average percentage of test case results shared between all parents and their offspring.

Figure 3 shows that even an optimal safe analysis (e.g., [20, 21]) could only reduce the average number of test case evaluations per successful repair by 29%, the average proportion of the test suite that remains unchanged between a parent and its offspring. This is significantly less than the 81% average reduction achieved with random sampling. Our fitness function can be more aggressive because of the final test and because GP is more noise tolerant than regression testing.

The `lighttpd` program is an outlier, for which many individuals are very likely to have the same test case behavior as their offspring, and thus an optimal safe impact analysis could account for all of the performance increase observed. Note that the efficacy of impact analysis is not solely a function of program type or test suite: Both `lighttpd` and `nullhttpd` are web servers, and both use exactly the same test suite in these experiments, but impact analysis would only improve performance on `nullhttpd` by 10%. A key contributing factor is the nature of the defect and the localization provided by the weighted path. The bug in `lighttpd` is well-localized, while the bug in `nullhttpd` is not. The poorer localization in `nullhttpd` means that variants and their offspring differ by a wider range of statements and are thus less likely to have correlated test case behavior.

We also tested the hypothesis that test-suite sampling is effective because of high overlap in the test suites. For example, if all test cases were the same, choosing a subset of them at random would save time with no cost. We constructed high-overlap and low-overlap suites and used them to conduct the repairs of all 10 programs. Test case overlap is measured roughly by the number of executed lines of code they have in common. To construct test suites with high or low overlap, we subsampled the original test suite for a given benchmark to find, for example, the 50 test cases with the highest overlap, as well as the 50 test cases with the least overlap. Overall there was only a 3% performance difference between low- and high-overlap test suites (1.02 vs. 0.99 compared to a 1.00 baseline of the time taken to find the repair on the original test suite).

We conclude that change impact analysis and test suite dependence alone cannot explain the overall performance gain we observe, even though there is a non-trivial overlap between parent and offspring fitness.

## 4.4 Predicate-based Fitness Function

The third experiment explores the effect of extending fitness evaluation to include learned runtime predicates. We focus on a detailed analysis of the `nullhttpd` benchmark because it has a rich, hand-crafted set of test cases, encoding required functionality, that are separate from the test suite in Figure 1. In addition, it contains a real-world error that is neither very easy nor very hard for the GP to repair [7] and is thus a good exemplar.

We first combined the predicate metrics defined in Section 2.5 to produce a fitness function that correlates well with the $d_{opt}$. We used fourteen known repairs for the `nullhttpd` program as the basis for the optimal fitness distance $d_{opt}$ (see Section 3.3). We collected the predicate set information $B$ for the original (buggy) program. We then generated 1772 unique variants of `nullhttpd` using our GP operators. For each variant $i$, we collected its associated predicate set measurements $V_i$, and thus its scalar features $f_{i,j}$.

We used linear regression on training subsets of these 1772 data points to construct a model, consisting of a set of coefficients $c_j$ and an intercept $c_0$ that, when applied to the predicate-based features, yields a fitness function designed to approximate the distance between a variant $i$ and its closest potential repair (see Section 3.2). We used cross-validation and avoided testing and training on the same data (see Section 5). A per-feature analysis of variance substantiates the validity of the model, indicating with high confidence ($p < 0.05$) that the predicate metrics and the test cases passed are predictive ($F$ substantially greater than 1.0) of the ideal fitness $d_{opt}$.

We used FDC to estimate fitness function precision. In this setting an ideal fitness function would correlate perfectly with the optimal distance $d_{opt}$ ($r = 1.0$). We evaluated a random fitness function as a baseline. As expected, it does not correlate well with $d_{opt}$: $-0.04 \leq r = 0.01 \leq 0.06$ (95% confidence interval). We used testcase$(1, 10)$ to denote the fitness function from our previous work [7, 27] because it weights each positive test case by one and each negative test case by 10. The testcase$(1, 10)$ function also does not correlate well with $d_{opt}$: $0.00 \leq r = 0.04 \leq 0.09$, suggesting that it describes a *difficult* GA search problem [11].

Because testcase$(1, 10)$ uses somewhat arbitrary weights for positive and negative test cases (i.e., 1 and 10), we con-

[6] In crossover, each variant is combined with the original program to produce two offspring: this *crossback* operator [27] allows each offspring to have one associated parent.

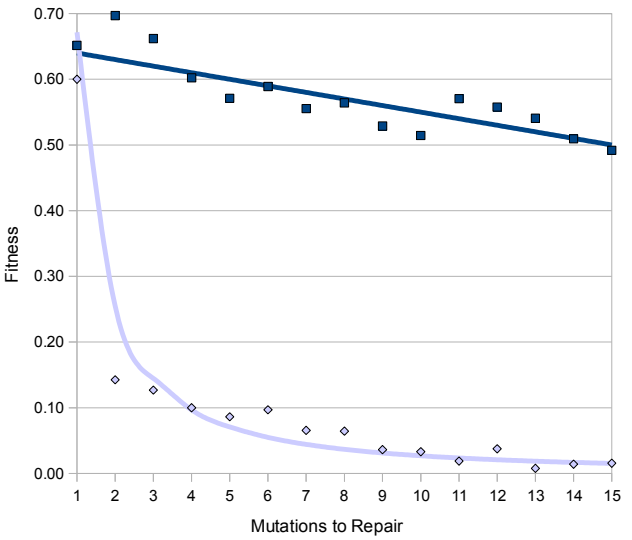


**Figure 4: Fitness as mutations to repair increases. The light line shows** testcase$(10, 1)$**; the dark line shows** predicate_fitness**. Each data point averages 40 variants.**

jecture that setting these more carefully would lead to an increased FDC, thus reducing search difficulty even without adding new predicates. To test this, we trained a second linear model to learn coefficients for the testcase fitness function. The resulting testcase$(-3.7, 14.0)$ function correlates much more strongly with optimal fitness: $0.39 \leq r = 0.43 \leq 0.47$, suggesting that even this simple enhancement could make the search easier. Although additional research is necessary to understand why this scheme weights positive test cases negatively, a possible explanation is that such a weighting facilitates escape from the local maximum of the original program, which passes all positive tests.

The predicate_fitness function demonstrates an even stronger FDC: $0.63 \leq r = 0.65 \leq 0.67$, a substantial improvement over the best result obtained with the testcase family of fitness functions. According to Jones and Forrest [11], this improvement is sufficient to transform the program-repair problem out of the realm of a *difficult* search problem.

## 4.5 Predicate-based Repair Evolution

The final experiment shows that predicate_fitness decreases more smoothly than testcase$(10, 1)$ as the mutation distance between a variant and a repair increases. We generated variants of `nullhttpd` that are a specified number of mutations from a known repair. A mutation is a swap, delete, or insert of a statement at some point on the weighted path. We consider only variants that compile but fail at least one testcase.

Figure 4 shows how mutation distance to a repair compares to fitness for both predicate_fitness and testcase$(10, 1)$. Solid lines plot the learned functions. predicate_fitness is linearly related to mutation distance via $f(x) = -0.01x + 0.65$, while testcase$(10, 1)$ is modeled by $f(x) = 0.67x^{-1.40}$. Confidence in the fit and slope for both this and predicate_fitness exceeds 0.9995.

Most variants close to a solution receive a testcase$(10, 1)$ fitness of zero, e.g., light blue points in figure at $x = 2$. The variance of testcase$(10, 1)$is low (standard deviation of 0.07) at all points $> 1$, but high for variants one mutation from repair (0.42). The standard deviation of predicate_fitness over all points is 0.12. Figure 4 supports the hypothesis that the all-or-nothing nature of a testcase impedes the ability of testcase$(10, 1)$ to provide an adequate signal to the GP repair process, especially near solutions (compare distance 1 with distance 2).

Figure 4 suggests that predicate_fitness evaluates fitness more smoothly, precisely and consistently, and is more precise than is possible with the testcase$(10, 1)$ function. This enhancement may allow the GP repair system to tackle more complicated repairs than those attempted to date. The linear decrease with predicate_fitness as a variant's mutation distance increases is encouraging.

# 5. DISCUSSION AND LIMITATIONS

These results extend our earlier work in four key ways. First, we demonstrated repairs on four new programs (e.g., `lighttpd`, `leukocyte`, `tiff`, `imagemagick`) from different domains. Second, we improved the efficiency of GP-based program repair by incorporating sampling and test suite reduction into the fitness function. We observed an 81% performance improvement over earlier results while retaining equivalent repair quality. Third, we gave evidence of GP-based automated program repair scaling to programs with non-trivial numbers of test cases, by presenting repairs on ten programs with at least one hundred test cases each in 1.8 minutes on average. Fourth, we proposed an enhanced fitness function, predicate_fitness, that includes the behavior of dynamic program predicates in the fitness calculation. We showed that its FDC is 0.64, compared to 0.04 for the weighted test case function employed in previous work; this improved FDC value transforms the GP search problem so that it is no longer in the *difficult* problem category. We also showed that predicate_fitness provides a more consistent signal to the GP process, and more accurately evaluates intermediate program variants.

Although the results suggest that aggressive sampling-based fitness functions can improve performance, they may not generalize to all programs. First, the benchmark programs and the faults they contain may not be representative of the spectrum of buggy programs. We attempted to mitigate this concern by choosing programs from many application domains containing multiple types of faults (e.g., crashes, infinite loops, incorrect output, buffer overruns). However, the programs and bugs used in our experiments are of the same order as those reported earlier [7, 27]. Also, the test suites we used may not be indicative of industrial practice. Although the suites are of varied character (e.g., fuzz tests, web server requests, images, language conformance tests, etc.), they are freely available and may not be of equivalent size or quality to those commonly used in industry. Also, the experiments used only test suites with uniform costs: our results might not apply to heterogeneous test suites. We note, however, that the Walcott *et al.* algorithm includes both test coverage and test cost and is only 2% worse, on average, than the random sampling function. Thus the technique could be applied directly to heterogeneous test suites.

One limitation of the predicate-based fitness function study is potential over-fitting of the model to the training data. We addressed this concern with 10-fold cross validation [12]; bias is suspected if the average results of cross-validation

(over many random partitionings) differ from the original. Our delta was 0.03%, indicating little or no bias.

Finally, the predicate-based fitness function experiments were conducted on a single benchmark program, a limitation that we hope to address in future work. However, our evaluation used a large population of unique variants; the measured statistics demonstrate significance at or above the 95% confidence level. Importantly, the benchmark includes a real-world error and set of hand-crafted test cases that exercises critical functionality. This is likely both the most indicative and unforgiving of our available benchmarks.

# 6. CONCLUSION

Automated program repair with GP is a promising technique for reducing debugging costs. In this setting, the efficiency and precision of the fitness function are primary concerns, We aspire to apply automated repair to complicated bugs (which require a precise fitness function) in interesting, critical programs (which require efficient fitness evaluation). The previous work in this area considered only small test suites and a fitness function with only a few discrete values.

We presented two fitness function enhancements: test suite sampling and dynamic predicates. Overall, we note that both enhancements exploit the tradeoff between efficiency and precision. Sampling increases efficiency but reduces precision. However, we demonstrated empirically that the sampling enhancements can produce the same repairs in 81% less time on 10 benchmark programs with 1206 test cases. The dramatic performance increase allows GP-based automated repair to scale to more realistic systems (1206 test cases vs. 58 in earlier work) and to incorporate more test cases for correctness guarantees. We performed experiments to account for the source of this performance increase, and found that even an optimal safe impact analysis that fully exploits parent/offspring correlation could provide only 29% of the performance gains of the sampling approach. Moreover, little of the performance gain are attributable to test case dependence.

Conversely, dynamic predicates increase precision but reduce efficiency. Recording predicate information introduces an overhead of 3% [15]. However, our detailed case study showed that dynamic predicates can increase the FDC from 0.04 to 0.65. The new fitness function is a significant improvement over weighted test cases, even when the weights are optimized with learning. We measured the fitness of variants as they were mutated away from a known repair to show that a predicate-based fitness function more smoothly and precisely evaluates the fitness of intermediate variants than the function employed in earlier work.

# 7. REFERENCES


[1] 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.

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

[3] BBC News. Microsoft zune affected by 'bug'. In `http://news.bbc.co.uk/2/hi/technology/7806683.stm`, Dec. 2008.

[4] R. V. Binder. *Testing object-oriented systems: models, patterns, and tools*. Addison-Wesley, 1999.

[5] A. Eiben and J. Smith. *Introduction to Evolutionary Computing*. Springer, 2003.

[6] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.

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

[8] J. Grefenstette and J. Fitzpatrick. Genetic search with approximate fitness evaluations. In *Int. Conf. on Genetic Algorithms and Their Applications*, pages 112–120, 1985.

[9] M. J. Harrold, R. Gupta, and M. L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.

[10] Y. Jin. A comprehensive survey of fitness approximation in evolutionary computation. *Soft Computing*, 9(1):3–12, 2005.

[11] T. Jones and S. Forrest. Fitness distance correlation as a measure of problem difficulty for genetic algorithms. In *ICGA*, pages 184–192, 1995.

[12] R. Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. *IJCAI*, 14(2):1137–1145, 1995.

[13] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *ICSE*, pages 308–318, 2003.

[14] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Trans. Softw. Eng.*, 33(4):225–237, 2007.

[15] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *PLDI*, pages 12–15, 2005.

[16] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.

[17] A. Orso, T. Apiwattanapong, and M. J. Harrold. Leveraging field data for impact analysis and regression testing. *SIGSOFT Softw. Eng. Notes*, 28(5):128–137, 2003.

[18] A. Orso, T. Apiwattanapong, J. Law, G. Rothermel, and M. J. Harrold. An empirical comparison of dynamic impact analysis algorithms. In *ICSE*, pages 491–500, 2004.

[19] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *SOSP*, pages 87–102, October 2009.

[20] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. *SIGPLAN Not.*, 39(10):432–448, 2004.

[21] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, 1997.

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

[23] A. M. Smith and G. M. Kapfhammer. An empirical study of incorporating cost into test suite reduction and prioritization. In *ACM symposium on Applied Computing*, pages 461–467, 2009.

[24] K. Walcott, M. Soffa, G. Kapfhammer, and R. Roos. Time-aware test suite prioritization. In *ISSTA*, 2006.

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

[26] W. Weimer. Patches as better bug reports. In *Generative Programming and Component Engineering*, 2006.

[27] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–367, 2009.

[28] T.-L. Yu, D. E. Goldberg, and K. Sastry. Optimal sampling and speed-up for genetic algorithms on the sampled onemax problem. In *GECCO*, pages 1554–1565, 2003.