# Automatic Program Repair with Evolutionary Computation

By Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen

## Abstract

**There are many methods for detecting and mitigating software errors but few generic methods for automatically repairing errors once they are discovered. This paper highlights recent work combining program analysis methods with evolutionary computation to automatically repair bugs in off-the-shelf legacy C programs. The method takes as input the buggy C source code, a failed test case that demonstrates the bug, and a small number of other test cases that encode the required functionality of the program. The repair procedure does not rely on formal specifications, making it applicable to a wide range of extant software for which formal specifications rarely exist.**

## 1. INTRODUCTION

Fixing bugs is a difficult, time-consuming, and manual process. Some reports place software maintenance, traditionally defined as any modification made on a system after its delivery, at up to 90% of the total cost of a typical software project.[22] Modifying existing code, repairing defects, and otherwise evolving software are major parts of those costs.[19] The number of outstanding software defects typically exceeds the resources available to address them. Mature software projects are forced to ship with both known and unknown bugs[13] because they lack the development resources to deal with every defect.

In this paper, we describe how to combine this problem by combining program analysis methods with evolutionary computation to automatically repair bugs in off-the-shelf legacy C programs. *Genetic programming* (GP) is a computational method inspired by biological evolution which evolves computer programs tailored to a particular task.[12] GP maintains a population of individual programs, each of which is a candidate solution to the task. Each individual's suitability is evaluated using a task-specific fitness function, and the individuals with highest fitnesses are selected for continued evolution. Computational analogs of biological mutation and crossover produce variations of the high-fitness programs, and the process iterates until a high-fitness program is found. GP has solved an impressive range of problems (e.g., Schmidt and Lipson[21]), but it has not been used to evolve off-the-shelf legacy software. As the 2008 *Field Guide to Genetic Programming* notes, "while it is common to describe GP as evolving programs, GP is not typically used to evolve programs in the familiar Turing-complete languages humans normally use for software development. It is instead more common to evolve programs (or expressions or formulae) in a more constrained and often domain-specific language." (Poli et al.[18] as quoted by Orlov and Sipper[15]).

Our approach assumes that we have access to C source code, a negative test case that exercises the fault to be repaired, and

several positive test cases that encode the required behavior of the program. The C program is represented as an abstract syntax tree (AST), in which each node corresponds to an executable statement or control-flow structure in the program. With these inputs in hand, a modified version of GP evolves a candidate repair that avoids failing the negative test case while still passing the positive ones. We then use structural differencing[1] and delta debugging[25] techniques to minimize the size of the repair, providing a compact human-readable patch.

A significant impediment for an evolutionary algorithm like GP is the potentially infinite-size search space it must sample to find a correct program. To address this problem we introduce two key innovations. First, we restrict the algorithm so that all variations introduced through mutation and crossover reuse structures in other parts of the program. Essentially, we hypothesize that even if a program is missing important functionality (e.g., a null check) in one location, it likely exhibits the correct behavior in another location, which can be copied and adapted to address the error. Second, we constrain the genetic operations of mutation and crossover to operate only on the region of the program that is relevant to the error, specifically the AST nodes on the execution path that produces the faulty behavior. Instead of searching through the space of all ASTs, the algorithm searches through the much smaller space of nodes representing one execution path. In practice, the faulty execution path has at least an order of magnitude fewer unique nodes than the AST. Combining these insights, we demonstrate automatically generated repairs for eleven C programs totaling 63,000 lines of code.

The main contributions of the work reported in Forrest et al.[8] and Weimer et al.[24] are:

- Algorithms to find and minimize program repairs based on test cases that describe desired functionality. The algorithms are generic in the sense that they can repair many classes of bugs.
- A novel and efficient representation and set of operations for applying GP to program repair. This is the first published work that demonstrates how GP can repair unannotated legacy programs.
- Experimental results showing that the approach gener-

ates repairs for several classes of defects in 11 production programs taken from multiple domains.
- Experiments to analyze how algorithm performance scales up with problem size and the relative contribution of different components of the evolutionary algorithm.

In the remaining sections of the paper we first give an overview of our technical approach (Section 2), illustrating it with a recent bug in Microsoft's Zune media player (Section 3). In Section 4 we report results obtained for repairs of several benchmark programs and study how algorithm performance scales with problem size. We place the work in the context of prior contributions in Section 5 and discuss our experiences, caveats, and thoughts for future work in Section 6, concluding in Section 7.

## 2. TECHNICAL APPROACH
The core of our method is an evolutionary algorithm that repairs programs by selectively searching through the space of related program variants until it discovers one that avoids known defects and retains key functionality. We use a novel GP representation and make assumptions about the probable nature and location of the necessary repair, improving search efficiency. Given a defective program, there are several issues to be addressed:

1. **What is it doing wrong?** We take as input a set of *negative test cases* that characterizes a fault. The input program fails all negative test cases.
2. **What is it supposed to do?** We take as input a set of *positive test cases* that encode functionality requirements. The input program passes all positive test cases.
3. **Where should we change it?** We favor changing program locations visited when executing the negative test cases and avoid changing program locations visited when executing the positive test cases.
4. **How should we change it?** We insert, delete, and swap program statements and control flow using existing program structure. We favor insertions based on the existing program structure.
5. **When are we finished?** We call the first variant that passes all positive and negative test cases a primary repair. We minimize the differences between it and the original input program to produce a final repair.

To present the repair process, we first describe our program representation (Section 2.1) and fault localization (Section 2.2) choices. We then detail the GP-based repair strategy (Section 2.3), discussing the genetic operators (Section 2.4), which modify the representation, and the fitness function (Section 2.5), which uses test cases to evaluate the results of the modifications. Finally, a postprocessing step is used to minimize the resulting repair (Section 2.6).

### 2.1. Representation
There are a number of commonly accepted structures for representing programs, such as control-flow graphs (CFGs) and abstract syntax trees (ASTs). We chose ASTs because they can losslessly represent all structured programs and

tree operations are well studied in GP. ASTs can be expressed at multiple levels of abstraction or granularity, and our program representation reflects the trade-off between expressive power and scalability. For example, C programs contain both *statements*, such as the conditional statement "`if(!p){x = 0;}`" and *expressions*, such as "0" or "`(!p)`". For scalability, we treat the statement as the basic unit, or gene. Thus, we never modify "`(!p)`" into "`(p||error_flag)`" because doing so would involve changing the inner structure of an expression. Instead, when manipulating compound statements, we operate on entire AST subtrees. For example, we might delete the entire "`if ...`" statement, including its then-branch and else-branch children. Finally, we never directly modify low-level control-flow directives such as `break`, `continue`, or `goto`, although statements around them can be modified.

### 2.2. Fault localization
We assume that software defects are local and that fixing one does not require changing the entire program. This assumption narrows the search space by limiting code changes to portions of the program likely to contain the defect. We bias modifications toward statement nodes that were visited when running the negative test cases but not visited when running the positive test cases. We find this information by assigning each statement a unique ID and instrumenting the program to print out the ID of each statement visited.[14] This allows our approach to scale to larger program sizes. For example, while the `atris` program contains a total of 8068 statement nodes (Table 1), we use this fault localization information to bias the search toward 34 statement nodes that are likely to matter, a reduction of over two orders of magnitude.

Formally, each program variant is a pair containing:

1. An *abstract syntax tree* (*AST*) including all of the statements $s$ in the program.
2. A *weighted path* through that program. The weighted path is a list of pairs $\langle s, w_s \rangle$, each containing a statement in the program visited on the negative test case and the associated weight for that statement.

The default *path weight* of a statement is 1.0 if it is visited in the negative test case but not on any positive test case. Its weight is 0.1 if it is visited on both positive and negative test cases. All other statements have weight 0.0. The weight represents an initial guess of how relevant the statement is to the bug. This approach is related to the union/intersection model of fault localization.[20] The *weighted path length* is the sum of statement weights on the weighted path. This scalar gives a rough estimate of the complexity of the search space and is correlated with algorithm performance (Section 4). We return to the issue of fault localization in Section 6.

### 2.3. Genetic programming
We use GP to maintain a population of program variants. Each variant, sometimes referred to as an *individual*, is represented as an AST annotated with a weighted path (fault localization information). We modify variants using two genetic operations, mutation and crossover. Mutation makes random changes to the nodes along the weighted path, while

crossover exchanges subtrees between two ASTs (see below for details). Each modification produces a new AST and weighted program path. The fitness of each variant is evaluated by compiling the AST and running it on the test cases. Its final fitness is a weighted sum of the positive and negative test cases it passes. Once the fitnesses have been computed for each individual, a *selection phase* deletes the bottom-ranked 50% of the population.[a] The new population is formed by first crossing over the remaining high-fitness individuals with the original program. Each crossover produces a single child. We add the children to the population and retain the parents unchanged, maintaining a constant population size. Finally, all surviving individuals are mutated.

The repair process terminates either when it finds a candidate solution that passes all its positive and negative test cases, or when it exceeds a preset number of generations. The first variant to pass all test cases is the *primary repair*.

## 2.4. Genetic operators
As mentioned above, we apply GP operators to a given variant to produce new program variants, thus exploring the search space of possible repairs. A key operator is *mutation*, which makes random changes to an individual. Because the primitive unit (gene) of our representation is the statement, mutation is more complicated than the simple bit flip used in other evolutionary algorithms. Only statements on the weighted path are subject to the mutation operator. Each location on the weighted path is considered for mutation with probability equal to its path weight multiplied by a *global mutation rate*. A statement selected for mutation is randomly subjected to either *deletion* (the entire statement and all its substatements are deleted: $s \leftarrow \{\}$), *insertion* (another statement is inserted after it: $s \leftarrow \{s; s';\}$), or *swap* of ($s \leftarrow s'$ while $s' \leftarrow s$). Note that a single mutation step in our scheme might contain multiple statement-level mutation operations along the weighted path.

The second operation for manipulating variants is *crossover*, which in GP exchanges subtrees chosen at random between two individuals. Although our initial experiments used a more complicated form of crossover, we have seen that the results do not depend on the particular crossover operator used.[8] During each generation, every surviving variant undergoes crossover.

Finally, there are a number of other C program components not touched by the GP operators, such as datatype definitions and local and global variable declarations. Because these are never on the weighted path, they are never modified by mutation or crossover. This potentially limits the expressive power of the repairs: If the best fix for a bug is to change a data structure definition, GP will not discover that fix. For example, some programs can be repaired either by reordering the data structure fields, or by changing the program control flow; our technique finds the second repair. Ignoring variable declarations, on the other hand, can cause problems with ill-formed variants. Because of the constraints on mutation and crossover, GP never generates syntactically ill-formed

programs (e.g., it will never generate unbalanced parentheses). However, it could move the use of a variable outside of its declared scope, leading to a semantically ill-formed variant that does not type check and thus does not compile.

## 2.5. Fitness function
In GP, the *fitness* function is an objective function used to evaluate variants. The fitness of an individual in a program repair task should assess how well the program avoids the program bug while still doing "everything else it is supposed to do." We use test cases to measure fitness. For our purposes, a *test case* consists of input to the program (e.g., command-line arguments, data files read from the disk, etc.) and an *oracle comparator* function that encodes the desired response. A program $P$ is said to *pass* a test case $T$ iff the oracle is satisfied with the program's output: $T_{oracle}(P(T_{input})) =$ pass. Test cases may check additional behavior beyond pure functional correctness (e.g., the program may be required to produce the correct answer within a given time bound or otherwise avoid infinite loops). Such testing accounts for as much as 45% of total software lifecycle costs,[17] and finding test cases to cover all parts of the program and all required behavior is a difficult but well-studied problem in the field of software engineering.

We call the defect-demonstrating inputs and their anomalous outputs (i.e., the bug we want to fix) the *negative test cases*. We use a subset of the program's existing test inputs and oracles to encode the core functionalities of the program, and call them the *positive test cases*. Many techniques are available for identifying bugs in programs, both statically (e.g., Ball and Rajamani[3] and Hovemeyer and Pugh[10]) and dynamically (e.g., Forrest et al.[7] and Liblit et al.[13]). We assume that a bug has been identified and associated with at least one negative test case.

The fitness function takes a program variant (genotype), compiles the internal representation into an executable program, and runs it against the set of positive and negative test cases. It returns the weighted sum of the test cases passed. The sum is weighted so that passing the negative test cases is worth at least as much as passing the positive test cases. Intuitively, this weighting rewards the search for moving toward a possible repair. Programs that do not compile are assigned fitness zero.

## 2.6. Minimizing the repair
Because the GP may introduce irrelevant changes, we use program analysis methods to trim unnecessary edits from the primary repair. For example, in addition to the repair, the GP might produce dead code ($x = 3; x = 5;$) or calls to irrelevant functions. We use tree-structured difference algorithms and delta debugging techniques in a postprocessing step to generate a simplified patch that, when applied to the original program, causes it to pass all of the test cases.

Using tree-structured differencing,[1] we view the primary repair as a set of changes against the original program. Each *change* is a tree-structured operation such as "take the subtree of the AST rooted at position 4 and move it so that it becomes the 5th child of the node at position 6". We seek to find a small subset of changes that produces a program that still passes all of the test cases.

Let $C_p = \{c_1, ..., c_n\}$ be the set of changes associated with

---

[a] We obtained results qualitatively similar to those reported here with a more standard method known as tournament selection.

the primary repair. Let $Test(C) = 1$ if the program obtained by applying the changes in $C$ to the original program passes all positive and negative test cases; let $Test(C) = 0$ otherwise. A *one-minimal subset* $C \subseteq C_p$ is a set such that $Test(C) = 1$ and $\forall c_i \in C.\ Test(C \backslash \{c_i\}) = 0$. That is, a one-minimal subset produces a program that passes all test cases, but dropping any additional elements causes the program to fail at least one test case. Checking if a set is valid involves a fitness evaluation (a call to *Test*).

We use *delta debugging*[25] to efficiently compute a one-minimal subset of changes from the primary repair. Delta debugging is conceptually similar to binary search, but it returns a set instead of a single number. Intuitively, starting with $\{c_1, ..., c_n\}$, it might check $\{c_1, ..., c_{n/2}\}$: if that half of the changes is sufficient to pass the *Test*, then $\{c_{1+n/2}, ..., c_n\}$ can be discarded. When no more subsets of size $n/2$ can be removed, subsets of size $n/4$ are considered for removal, until eventually subsets of size 1 (i.e., individual changes) are tested. Finding the overall minimal valid set by brute force potentially involves $\mathcal{O}(2^n)$ evaluations; delta debugging finds a one-minimal subset in $\mathcal{O}(n^2)$.[25, Proposition 12] However, we typically observe a linear number of tests in our experiments. This smaller set of changes is presented to the developers as the *final repair* in the form of a standard program patch. In our experiments, the final repair is typically at least an order-of-magnitude smaller than the primary repair.

## 3. ILLUSTRATION

On 31 December 2008, a bug was widely reported in Microsoft Zune media players, causing them to freeze up.[b] The fault was a bug in the following program fragment:[c]

```
1   void zunebug(int days) {
2      int year = 1980;
3      while (days > 365) {
4         if (isLeapYear (year)){
5            if (days > 366) {
6               days -= 366;
7               year += 1;
8            }
9            else {
10           }
11        }
12        else {
13           days -= 365;
14           year += 1;
15        }
16     }
17     printf("the year is %d\n", year);
18  }
```

[b] See *Microsoft Zune affected by "Bug,"* BBC News, December 2008, http://news.bbc.co.uk/2/hi/technology/7806683.stm.
[c] Downloaded from http://pastie.org/349916 (Jan. 2009). Note that the original program source code does not make lines 9–10 explicit: our AST represents missing blocks, such as those in **if** statements without **else** clauses, as blocks containing zero statements.

When the value of the input days is the last day of a leap year (such as 10,593, which corresponds to 31 December 2008), the program enters an infinite loop on lines 3–16.

We now walk through the evolution of a repair for this program. We first produce its AST and determine the weighted path, using line numbers to indicate statement IDs. The positive test case zunebug (1,000) visits lines 1–8, 11–18. The negative test case zunebug (10,593) visits lines 1–16, and then repeats lines 3, 4, 8, and 11 infinitely.

For the purposes of this example, the negative test cases consist of the inputs 366 and 10,593, which cause an infinite loop (instead of the correct values, 1980 and 2008), and the positive test cases are the inputs 1,000, 2,000, 3,000, 4,000, and 5,000, which produce the correct outputs 1982, 1985, 1988, 1990, and 1993.

We consider one variant, $V$, which is initialized to be identical to the original program. In Generation 1, two operations mutate $V$: the conditional statement "**if (days > 366) {days -= 366; year += 1;}**" is inserted between lines 6 and 7 of the original program; and the statement "**days -= 366**" is inserted between lines 10 and 11. Note that the first insertion includes not just the **if** but its entire subtree. This produces the following code fragment:

```
5   if (days > 366)    {
6      days -= 366;
7      if (days > 366) {    // insert #1
8         days -= 366;      // insert #1
9         year += 1;        // insert #1
10     }                    // insert #1
11     year += 1;
12  }
13  else {
14  }
15  days -= 366;            // insert #2
```

This modified program passes the negative test case 366 (year 1980) and one positive test case 1000.

Variant $V$ survives Generations 2, 3, 4, 5 unchanged, but in Generation 6, it is mutated with the following operations: lines 6–10 are deleted, and "**days -= 366**" is inserted between lines 13 and 14:

```
5   if (days > 366)    {
6      // days -= 366;           // delete
7      // if (days > 366) {      // delete
8      //    days -= 366;        // delete
9      //    year += 1;          // delete
10     // }                      // delete
11     year += 1;
12  }
13  else {
14     days -= 366;              // insert
15  }
16  days -= 366;
```

At this point, $V$ passes all of the test cases, and the search terminates with $V$ as the primary repair. The minimization step is invoked to discard unnecessary changes. Compared to the original program (and using the line numbers from the original), there are three key changes: $c_1$ = "days $-=$ 366" deleted from line 6; $c_2$ = "days $-=$ 366" inserted between lines 9 and 10; and $c_3$ = "days $-=$ 366" inserted between lines 10 and 11. Only $c_1$ and $c_3$ are necessary to pass all tests, so change $c_2$ is deleted, producing the final repair:

```
1   void zunebug_repair (int days) {
2     int year = 1980;
3     while (days > 365) {
4       if (isLeapYear (year)) {
5         if (days > 366) {
6           // days -= 366;   // deleted
7           year += 1;
8         }
9         else {
10        }
11        days -= 366;        // inserted
12      } else {
13        days -= 365;
14        year += 1;
15      }
16    }
17    printf ("the year is %dn", year);
18  }
```

On average, constructing and minimizing a repair for the Zune fragment shown here takes our prototype a total 42 s, including the time to compile and evaluate variants against a suite of five positive and two negative tests.

## 4. RESULTS

To date, we have repaired 20 defects in modules totaling 186kLOC from 20 programs totaling 2.3MLOC (not all shown here). We have repaired defects from eight classes: infinite loop, segmentation fault, heap buffer overrun, nonoverflow denial of service, integer overflow, invalid exception, incorrect output, and format string vulnerability. Constructing a repair requires 1428 s on average, most of which is spent performing an average of 3903 fitness evaluations. In Table 1, we summarize results for 11 benchmark programs reported in Forrest et al.[8] and Weimer et al.[24] The benchmark programs, test cases, GP code, and the supporting infrastructure used to generate and reproduce these results are available at: http://gen prog.adaptive.cs.unm.edu/.

In all of our experiments, a standard *trial* uses the following setup. The population size is 40, and GP runs for a maximum of 20 generations. For the first 10 generations, the global mutation rate is 0.06. If no primary repair is found, the current population is discarded, the global mutation rate is halved to 0.03, and, if possible, the weighted path is restricted to statements visited solely during the negative test case, and the GP is run for 10 additional generations. These results show that GP can automatically discover repairs for a variety of documented bugs in production C programs.

Table 1. Eleven defects repaired by Genetic Programming, summarized from previous work. The size of each program is given in lines of code as well as weighted path units (see Section 2.2). Each repair used five or six positive tests and one or two negative tests. The "Time" column gives the total wall-clock time required to produce and minimize the repair (on a successful trial). The "Fitness Evals" column lists the number of times the entire fitness function was called before a repair was found (averaged over only the successful trials). The "Repair Size" column gives the size of the final minimized repair, as measured in lines by the Unix diff utility.

| Program | Lines of Code | Weighted Path | Description | Fault | Time (s) | Fitness Evals | Repair Size |
|---|---|---|---|---|---|---|---|
| gcd | 22 | 1.3 | Euclid's algorithm | Infinite loop | 153 | 45.0 | 2 |
| zune | 28 | 2.9 | MS Zune excerpt | Infinite loop | 42 | 203.5 | 4 |
| uniq utx 4.3 | 1146 | 81.5 | Duplicate filtering | Segmentation fault | 34 | 15.5 | 4 |
| look utx 4.3 | 1169 | 213.0 | Dictionary lookup | Segmentation fault | 45 | 20.1 | 11 |
| look svr 4.0 | 1363 | 32.4 | Dictionary lookup | Infinite loop | 55 | 13.5 | 3 |
| units svr 4.0 | 1504 | 2159.7 | Metric conversion | Segmentation fault | 109 | 61.7 | 4 |
| deroff utx 4.3 | 2236 | 251.4 | Document processing | Segmentation fault | 131 | 28.6 | 3 |
| nullhttpd 0.5.0 | 5575 | 768.5 | Webserver | Heap buffer overrun | 578 | 95.1 | 5 |
| indent 1.9.1 | 9906 | 1435.9 | Source code formatting | Infinite loop | 546 | 108.6 | 2 |
| flex 2.5.4a | 18775 | 3836.6 | Lexical analyzer generator | Segmentation fault | 230 | 39.4 | 3 |
| atris 1.0.6 | 21553 | 34.0 | Graphical tetris game | Stack buffer overrun | 80 | 20.2 | 3 |
| Total | 63277 | 8817.2 | | | 2003 | 651.2 | 39 |

The trial terminates if it discovers a primary repair. We performed 100 trials for each program, memoizing fitnesses such that within one trial, two individuals with different ASTs but the same source code are not evaluated twice. Similarly, individuals that are copied to the next generation without change are not reevaluated.

Once a primary repair has been located, the process of minimizing it to a final repair is quite rapid, requiring less than 5 s on average. Final repairs, expressed in `patch` format, varied in size from four lines (e.g., `zune`: one insert, one delete, and one context-location line for each edit) to 11 lines (e.g., `look utx 4.3`).

Not all trials lead to a successful repair; in the repairs shown here, an average of 60% of trials produce a primary repair. The "Time" and "Fitness Evals" columns in Table 1 measure the effort taken for a successful trial. Since all trials are independent, a number of trials can be run in parallel, terminating when the first successful trial yields a primary repair. In addition, the fitnesses of different variants and the results of different test cases for a given variant can all be evaluated independently, making our approach easy to parallelize on multicore architectures. The measurement in Table 1 was made on a quad-core 3 GHz machine.

Over half of the total time required to create a repair is spent evaluating fitness by running compiled variants through test cases. For programs with large test suites, this cost can be considerable. A dominant factor in the scalability of our approach is thus the number of such fitness evaluations that must be made to find a repair. The number of fitness evaluations required is related to the size of the search space and the efficacy of the search strategy: each fitness evaluation represents a single probe. We hypothesize that the size of the weighted path is a good representation of the search space size; recall that we only modify statements along the path (Sections 2.2 and 2.4). Figure 1 shows the results of an empirical investigation into this relationship, plotting the average number of fitness evaluations required to produce each of 18 repairs against the length of their weighted paths (note log–log scale). Although more data points are needed before strong conclusions can be drawn, the plot suggests that the number of fitness evaluations, and thus the search time, may scale as a power law of the form $y = ax^b$ where $b$ is the slope of the best fit line (0.78). This suggests that the time to find a repair scales nearly linearly with the size of the weighted path for a fixed number of test cases.

Our approach has also failed to repair some defects, including those that require many simultaneous edits or changes that cannot be made directly at the statement level (e.g., `matmul(b, a)` should be `matmul(a, b)`). We return to the issue of repair quality in Section 6.

## 5. RELATED WORK

Arcuri[2] proposed the idea of using GP to repair software bugs automatically, and Orlov and Sipper experimented with evolving Java bytecode.[15] However, our work is the first to report substantial experimental results on real programs with real bugs. The field of Search-Based Software Engineering (SBSE)[9] uses evolutionary and related methods for software testing, e.g., to develop test suites, improve

Figure 1. GP search time scales with weighted path size. Data are shown for 18 programs successfully repaired by GP (gcd and zune examples omitted; figure includes several additional programs to those listed in Table 1), with best linear fit. The x-axis is the base-10 logarithm of the weighted path length, and the y-axis shows the logarithm of the total number of fitness evaluations performed before the primary repair is found (averaged over 100 runs).
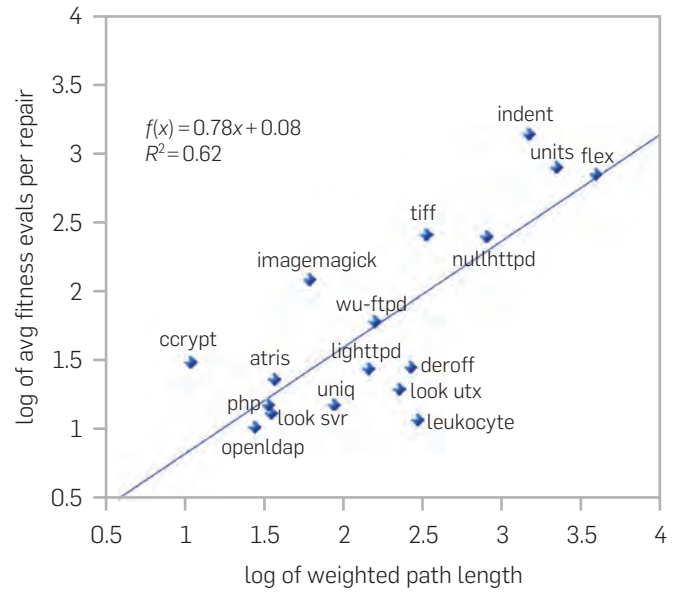


software project management, and effort estimation find safety violations and in some cases refactor or reengineer large software bases. In SBSE, most innovations in the GP technique involve new kinds of fitness functions, and there has been less emphasis on novel representations and operators, such as those explored here.

Our approach automatically repairs programs without specifications. In previous work, we developed an automatic algorithm for soundly repairing programs with specifications.[23] However, formal specifications are not always available (e.g., there were no formal specifications available for any of the programs repaired here), so the present work focuses on test cases to check and ensure correctness.

Trace and fault localization, minimization, and explanation (e.g., Jones and Harrold[11]) projects also aim to elucidate faults and ease debugging. These approaches typically narrow down an error symptom to a few lines (a potential cause). Our work extends this work by proposing a concrete repair. In addition, these other algorithms are usually limited to the given trace or source code and will thus never localize the "cause" of an error to a missing statement or suggest that a statement be moved. Our approach can infer new code that should be added, deleted, or swapped: 6 of the 11 repairs in Table 1 required insertions or swaps.

Demsky et al.[5] present a technique for data structure repair. Given a formal specification of data structure consistency, they modify a program so that if the data structures ever become inconsistent, they can be modified back to a consistent state at runtime. Their technique does not modify the program source code in a user-visible way. Instead, it inserts runtime monitoring code that "patches

up" inconsistent state so that the buggy program can continue to execute. Thus, their programs continue to incur the runtime overhead after the repair is effected. Another difference from our work is that their data structure repair requires formal specifications. Finally, their technique is limited to data structures and does not address the full range of logic errors. The gcd infinite loop in Section 3, for example, is outside the scope of this technique. However, this technique complements ours: in cases where runtime data structure repair does not provide a viable long-term solution, it may enable the program to continue to execute while our technique searches for a long-term repair.

Clearview[16] automatically detects and patches assembly-level errors in deployed software. Clearview monitors a program at runtime, learns invariants that characterize normal behavior, and subsequently flags violations for repair. Candidate patches that make the implicated invariant true are generated and tested dynamically. Although the performance overhead of Clearview is high, it has successfully been applied to buggy versions of Mozilla Firefox and evaluated against a Red Team of hackers. However, Clearview can repair only those errors that are relevant to selected monitors. Our method is more generic, providing a single approach to repair multiple classes of faults without the need for specific monitors, and we do not require continual runtime monitoring (and the incumbent slowdown) to create and deploy repairs.

This body of work illustrates a burgeoning interest in the problem of automated software repair and some of the many possible approaches that might be tried. There are several other recent but less mature proposals for automatically finding and repairing bugs in software, e.g., Dallmeier et al.,[4] suggesting that we can expect rapid progress in this area over the next several years.

## 6. DISCUSSION

The results reported here demonstrate that GP can be applied to the problem of bug repair in legacy C programs. However, there are some caveats.

**Basic limitations.** First, we assume that the defect is reproducible and that the program behaves deterministically on the test cases. This limitation can be mitigated by running the test cases multiple times, but ultimately if the program behavior is random it will be difficult for our method to find or evaluate a correct patch. We further assume that positive test cases can encode program requirements. Test cases are much easier to obtain than formal specifications or code annotations, but if too few are used, a repair could sacrifice important functionality. In practice, we are likely to have too many test cases rather than too few, slowing down fitness evaluation and impeding the search. We also assume that the path taken along the negative test case is different from the positive path. If they overlap completely, our weighted representation will not guide GP modifications as effectively. Finally, we assume that the repair can be constructed from statements already extant in the program; in future work, we plan to extend our method to include a library of repair templates.

**Evolution.** One concern about our results to date is the role of evolution. Most of our repairs result from one or two random modifications to the program, and they are often found within the first few generations or occasionally, not at all. We have conducted some experiments using a brute force algorithm (which applies simple mutation operations in a predetermined order) and random search (which applies mutation operations randomly without any selection or inheritance of partial solutions). Both these simpler alternatives perform as well or better than the GP on many, but not all, of our benchmark programs. We do not fully understand what characteristics, either of the program or the particular bug, determine how easily a solution can be found through random trial and error. However, thus far GP outperforms the other two search strategies in cases where the weighted path is long (i.e., where the fault is difficult to localize). There are several interesting questions related to the design of our GP algorithm, but the overall process proved so successful initially that we have not experimented carefully with parameter values, selection strategies, and operator design. These all could almost certainly be improved.

**Fault localization.** As Figure 1 shows, the time to find a solution varies with the length of the weighted path. Since the weighted path is a form of fault localization, we could use off-the-shelf fault localization techniques (e.g., Jones and Harrold[11] and Renieres and Reiss[20]) or dynamically discovered invariants,[13] in the style of Daikon[6] to further narrow the search space. Predicates over data might help in cases where faults cannot be localized by control flow alone, such as cross-site scripting or SQL injection attacks. In addition, our recent experiments have shown that the location of the fault (i.e., where to insert new code) is rarely the same as source of the fix (i.e., where to find code to insert). Since more than half of our repairs involve inserting or swapping code, locating viable fixes is of critical importance but remains poorly understood.

**Fitness function.** Our current test suite fitness function has the advantage of conceptual simplicity: a variant that passes all test cases is assumed to be correct, and a variant that does not compile or fails all tests is rejected. However, it may not be accurate in the middle ranges or precise enough to guide the evolutionary search in more complex problems. Consider a program with a race condition, for which the fix consists of inserting separate lock and unlock calls. A variant with a partial solution (e.g., just an inserted lock) may unfortunately pass fewer test cases (e.g., by deadlocking), thus "deceiving" the evolutionary algorithm. Fitness function design could be enhanced in several ways, for example, by weighting individual test cases, dynamically choosing subsets of test cases to be included, or by augmenting the test case evaluation with other information. For example, if a simple predicate like x == y is true at a particular point on all positive test cases, but false for the negative test, a variant that causes it to be true for the negative test might be given a higher fitness value.

**Repair quality.** We are interested in how much repairs vary after minimization, and how repair quality compares to human-engineered solutions. In our experiments to date, many, but not all, repairs look identical after the minimization step. For example, we have isolated 12 distinct repairs for the null-httpd fault, but much overlap exists (e.g., two repairs may insert the same statement into different points in the same basic block). Repair quality depends on the presence of a

high-quality set of positive test cases that encode program requirements. In other work, we experimented with held-out indicative workloads, fuzz testing, and held-out exploits to demonstrate that our server repairs address the causes of problems without being fragile memorizations of the negative input and without failing common requests (i.e., because of the positive tests). Much remains to be done in this area, however, such as automatically documenting or proving properties of the generated repairs.

**Future work.** Beyond these immediate steps, there are other more ambitious areas for future work. For example, we plan to develop a generic set of repair templates so the GP has an additional source of new code to use in mutation, beyond those statements that happen to be in the program. Our AST program representation could be extended in various ways, for example, by including data structure definitions and variable declarations. Similarly, we are currently experimenting with assembly- and bytecode-level repairs. Finally, we are interested in testing the method on more sophisticated errors, such as race conditions, and in learning more about bugs that need to be repaired, such as their size and distribution, and how we might identify which ones are good candidates for the GP technique.

## 7. CONCLUSION

We credit much of the success of this technique to design decisions that limit the search space. Restricting attention to statements, focusing genetic operations along the weighted path, reusing existing statements rather than inventing new ones, and repairing existing programs rather than creating new ones, all help to make automatic repair of errors using GP tractable in practice.

The dream of automatic programming has eluded computer scientists for at least 50 years. The methods described in this paper do not fulfill that dream by evolving new programs from scratch. However, they do show how to evolve legacy software in a limited setting, providing at least a small down payment on the dream. We believe that our success in evolving automatic repairs may say as much about the state of today's software as it says about the efficacy of our method. In modern environments, it is exceedingly difficult to understand an entire software package, test it adequately, or localize the source of an error. In this context, it should not be surprising that human programming often has a large trial and error component, and that many bugs can be repaired by copying code from another location and pasting it in to another. Such debugging is not so different from the approach we have described in this paper.

## Acknowledgments

**C**

### References

1. Al-Ekram, R., Adma, A., Baysal, O. diffX: an algorithm to detect changes in multi-version XML documents. In *Conference of the Centre for Advanced Studies on Collaborative Research*. IBM Press, 2005, 1–11.
2. Arcuri, A., Yao, X. A novel co-evolutionary approach to automatic software bug fixing. In *IEEE Congress on Evolutionary Computation* (2008), 162–168.
3. Ball, T., Rajamani, S.K. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking of Software* (May 2001), 103–122.
4. Dallmeier, V., Zeller, A., Meyer, B. Generating fixes from object behavior anomalies. In *International Conference on Automated Software Engineering* (2009).
5. Demsky, B., Ernst, M.D., Guo, P.J., McCamant, S., Perkins, J.H., Rinard, M. Inference and enforcement of data structure consistency specifications. In *International Symposium on Software Testing and Analysis* (2006), 233–244.
6. Ernst, M.D., Perkins, J.H., Guo, P.J., McCamant, S., Pacheco, C., Tschantz, M.S., Xiao, C. The Daikon system for dynamic detection of likely invariants. *Sci. Comput. Program* (2007).
7. Forrest, S., Hofmeyr, S.A., Somayaji, A., Longstaff, T.A. A sense of self for Unix processes. In *IEEE Symposium on Security and Privacy* (1996), 120–128.
8. Forrest, S., Weimer, W., Nguyen, T., Le Goues, C. A genetic programming approach to automated software repair. In *Genetic and Evolutionary Computing Conference* (2009), 947–954.
9. Harman, M. The current state and future of search based software engineering. In *International Conference on Software Engineering* (2007), 342–357.
10. Hovemeyer, D., Pugh, W. Finding bugs is easy. In *Object-Oriented Programming Systems, Languages, and Applications Companion* (2004), 132–136.
11. Jones, J.A., Harrold, M.J. Empirical evaluation of the tarantula automatic fault-localization technique. In *International Conference on Automated Software Engineering* (2005), 273–282.
12. Koza, J.R. *Genetic Programming: on the Programming of Computers by Means of Natural Selection*. MIT Press, 1992.
13. Liblit, B., Aiken, A., Zheng, A.X., Jordan, M.I. Bug isolation via remote program sampling. In *Programming Language Design and Implementation* (2003), 141–154.
14. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W. Cil: An infrastructure for C program analysis and transformation. In *International Conference on Compiler Construction* (Apr. 2002), 213–228.
15. Orlov, M., Sipper, M. Genetic programming in the wild: Evolving unrestricted bytecode. In *Genetic and Evolutionary Computation Conference* (ACM, 2009), 1043–1050.
16. Perkins, J.H., Kim, S., Larsen, S., Amarasinghe, S., Bachrach, J., Carbin, M., Pacheco, C., Sherwood, F., Sidiroglou, S., Sullivan, G., Wong, W.-F., Zibin, Y., Ernst, M.D., Rinard, M. Automatically patching errors in deployed software. In *ACM Symposium on Operating Systems Principles* (Oct. 2009), 87–102.
17. Pigoski, T.M. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley & Sons, Inc., 1996.
18. Poli, R., Langdon, W.B., McPhee, N.F. *A Field Guide to Genetic Programming*. Published via http://lulu.com and freely available at http://www.gp-field-guide.org.uk, 2008.
19. Ramamoothy, C.V., Tsai, W.-T. Advances in software engineering. *IEEE Comput. 29*, 10 (1996), 47–58.
20. Renieres, M., Reiss, S.P. Fault localization with nearest neighbor queries. In *International Conference on Automated Software Engineering* (Oct. 2003), 30–39.
21. Schmidt, M., Lipson, H. Distilling free-form natural laws from experimental data. *Science 324*, 5923 (2009), 81–85.
22. Seacord, R.C., Plakosh, D., Lewis, G.A. *Modernizing Legacy Systems: Software Technologies, Engineering Process and Business Practices*. Addison-Wesley, 2003.
23. Weimer, W. Patches as better bug reports. In *Generative Programming and Component Engineering* (2006), 181–190.
24. Weimer, W., Nguyen, T., Le Goues, C., Forrest, S. Automatically finding patches using genetic programming. In *International Conference on Software Engineering* (2009), 364–367.
25. Zeller, A., Hildebrandt, R. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng. 28*, 2 (2002), 183–200.

**Westley Weimer** (weimer@virginia.edu), University of Virginia.

**Claire Le Goues** (legoues@virginia.edu), University of Virginia.

**Stephanie Forrest** (forrest@cs.unm.edu), University of New Mexico.

**ThanhVu Nguyen** (tnguyen@cs.unm.edu), University of New Mexico.