A Genetic Programming Approach to Automated Software Repair

[Submitted to the Genetic Programming Track]

ABSTRACT

Genetic programming is combined with program analysis methods to repair bugs in off-the-shelf legacy C programs. Fitness is defined using negative test cases that exercise the bug to be repaired and positive test cases that encode program requirements. Once a successful repair is discovered, structural differencing algorithms and delta debugging methods are used to minimize its size. Several modifications to the GP technique contribute to its success: (1) genetic operations are localized to the nodes along the execution path of the negative test case; (2) high-level statements are represented as single nodes in the program tree; (3) genetic operators use existing code in other parts of the program, so new code does not need to be invented. The paper describes the method, reviews earlier experiments that repaired 11 bugs in over 60,000 lines of code, reports results on new bug repairs, and describes experiments that analyze the performance and efficacy of the evolutionary components of the algorithm.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.3.1b [**Programming Languages**]: Syntax; I.2.2 [**Artificial Intelligence**]: Automatic Programming

General Terms

Algorithms

Keywords

Software repair, genetic programming, software engineering

1. INTRODUCTION

Despite its many successes Genetic Programming (GP) has not replaced human programmers, who still develop, maintain, and repair computer programs largely by hand. In this paper, we describe how GP can be combined with program analysis methods to repair bugs in off-the-shelf legacy C programs. We assume that we have access to the 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.

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 [2] and delta debugging [23] techniques to minimize the repair, mitigating code bloat. The 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.The genetic operators are restricted to AST nodes on the execution path that produces the faulty behavior. The GP problem is thus reduced: Instead of searching through the space of all nodes in the AST, the algorithm searches through the much smaller space of nodes representing one execution path. In practice, the faulty execution path has an order of magnitude fewer unique nodes than the AST.

The primary contribution of the paper is a demonstration of GP successfully applied to the problem of software repair. To accomplish this, we introduce the idea of localizing genetic operations to the buggy execution path. Finally, we report results analyzing how the GP search proceeds and documenting the contribution of various parts of the algorithm.

In the remainder of the paper, we first describe the technical approach (Section 2). We illustrate the approach on a recent bug in Microsoft's Zune program (Section 3.1), and we summarize earlier results obtained on multiple programs (Section 3.2). Next, we report results that explore GP's performance, including the role of crossover (Section 3.3), the effect of adding test cases to the fitness function (Section 3.4), and the success of the different mutation operations in the search. We also address the question of scalability by comparing GP search time with execution path length (Section 3.6). Finally, we review related work and discuss some of the implications and future prospects for this line of inquiry (Section 5).

2. TECHNICAL APPROACH

GP is used to generate and evaluate program variants. The variants, or individuals, are AST representations of C programs. Mutation and crossover operators are applied to statements that lie along a weighted execution path through the AST. We find this execution path by running an instrumented version of the program on a "negative" input that exercises a bug in the code. The first generation is created by making multiple identical copies of the original program, with the bug intact, and then applying mutation to each individual, before proceeding with the fitness evaluation.

Our GP follows the traditional algorithmic structure but uses a nontraditional form of crossover and strong elitism. It maintains a population of chromosomes (programs), selects a pool of individuals based on their fitness, and modifies them with mutation and crossover. Selection deletes the bottom-ranked 50% of the population (20 individuals in our standard runs). The new population is formed by first crossing over the remaining high 20 individuals with the original program. We refer to this as *crossing back*. Each such crossover produces a single child. We add the 20 children to the population and retain the 20 parents unchanged, bringing the population total back to 40. Finally, all surviving individuals are mutated.[1] The program 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*. It will likely contain irrelevant changes, so we use program analysis methods to minimize the repair, producing the *minimized repair*.

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) [1]. We chose ASTs because they are efficient,[2] and sufficiently powerful to losslessly represent all structured programs. Moreover, tree operations are well-studied in genetic programming.

ASTs can be expressed at multiple levels of abstraction or granularity, and the chromosome representation reflects the tradeoff between expressive power and scalability. In particular, C programs contain both *statements*, such as the conditional statement "`if (!p) { x = (1-2)*3; }`," and *expressions*, such as "`(1-2)`" or "`(!p)`". For example, the **`atris`** program described in Section 3.2 is 21553 lines of C code, but its AST contains 32474 expression nodes and 8068 statement nodes. For scalability, we treat the statement as the basic unit, or gene. Thus we never modify "`(!p)`" into "`(p || error_flag)`" because that would involve changing the structure of an expression. We might, however, delete the entire "`if` ..." statement or replace it with a function call statement.

A few details remain. First, note that when the program uses structured control flow, statements can contain other statements. For example, the "`if` ..." statement above contains the statement "`x = (1-2)*3;`" as its then-branch. If the conditional statement is deleted from the AST, the contained assignment statement, which is in its subtree, will necessarily be removed as well. Second, we never directly modify low-level control-flow directives such as `break`, `continue` or `goto`, although statements around them can be modified. For the **`atris`** program, this reduces the number of statement nodes of interest from 8068 to 6470. Third, we assume that software defects are local, rather than spanning an entire program. Thus, we consider only code that is visited when the bug is exercised, ignoring the rest of the program. Finally, we restrict attention to statement nodes that were visited when running the negative test cases but not visited when running the positive test cases (see Section 2.2). In the **`atris`** program, only 34 statements meet those requirements. We find this information by assigning each statement a unique ID, and instrumenting the program to print out the ID of each statement visited [17].

Informally, this example demonstrates why our approach scales to real-world program sizes: rather than considering all 32474 expression nodes in **`atris`**, the GP search is localized to the 34 statement nodes that are likely to matter, a reduction of three orders of magnitude.

Formally, each chromosome is a pair containing:

1. An *abstract syntax tree* 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 an associated weight for that statement.

The *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 can be viewed as having weight 0.0. The weight represents an initial guess of how relevant the statement is to the bug.

The *weighted path length* is the weighted 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 3.6).

Finally, there are a number of other C program components that are not mutable parts of our GP representation. For example, programs contain 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 a change to a data structure definition, the GP will not discover that fix. In practice, this has not been problem. For example, the heap-based buffer overflow defect in **`nullhttpd`** (Section 3.2) 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, does cause problems with ill-formed variants. Because of the constraints on mutation and crossover, the GP will never produce *syntactically* ill-formed programs (i.e., it will never generate unbalanced parentheses). However, it could move the use of a variable outside of its declared scope, which leads to a *semantically* ill-formed variant that does not type check and thus does not compile. We return to this issue in Section 3.2.

2.2 Fitness Function

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 [11]. A program P is said to *pass* a test case T iff the oracle is satisfied with the program's output: $T_{oracle}(P(T_{input})) = pass$. Such testing accounts for as much as 45% of total software lifecycle cost [18], and finding a set of test cases that covers 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 input and its anomalous output (i.e., the bug we want to fix) the *negative test case*. 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., [7, 15]) and dynamically (e.g., [13, 16, 19]). We assume that a bug has been identified and associated with at least one negative test case.

The fitness function takes a chromosome, 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. Programs that do not compile, as well as those whose runtimes exceed a predetermined threshold (currently five seconds for most programs), are assigned fitness zero.

[1] We obtained results qualitatively similar to those reported here with tournament selection.

[2] In the worst case, any context-free grammar can be parsed into an AST in $\mathcal{O}(n^3)$ time. In practice, languages such as C can be parsed in near-linear time using optimized techniques such as LALR(1). [1]

2.3 Genetic Operators

Because the primitive unit (gene) is the statement, the mutation operator is more complicated than a simple bit flip. It consists either of a deletion (the entire statement and all its sub-statements are deleted), an insertion (another statement is inserted after it), or a swap with another statement from the same program. 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. Thus, statements occurring in negative test case paths but not on positive test case paths are mutated with high probability, and statements that occur on both positive and negative test case paths are less likely to be mutated.

Even though genetic operators are focused on the weighted path, the rest of the program remains important. We use the term *C-Bank* (for code bank) to refer to the set of all statements of interest in the program, even those not on the weighted path. Statements in the C-Bank are weighted equally. In the **`atris`** example described in Section 2.1, there are 34 statements in the weighted path and 6470 statement in the code bank.

Each statement s in the negative path is mutated with probability m_s as follows, where w_s is the weight assigned to s in the weighted path, $0.0 < p_m < 1.0$ is the global mutation rate.

Once it is determined that a mutation will occur at a given location, a mutation type is chosen uniformly at random: *delete* ($s \leftarrow \{\}$), *replace* with a random statement s' from the C-Bank ($s \leftarrow s'$), and *insert* a random statement s' from the C-Bank ($s \leftarrow \{s; s';\}$). Note that in a typical mutation, several independent mutations might be applied in a single step (see Table 4).

The *crossover* operator is unusual in two ways. First, an individual is always *crossed back* with the original parent program. Second, one-point crossover is used to determine the crossover point, and then a biased coin is tossed for each gene in the first segment to determine which genes are actually swapped. A location is identified in the weighted path (the crossover point), which partitions the path into two segments. This point is chosen uniformly at random. The existing variant V is then crossed over with the *original program* O to produce two child variants. The variant V can be viewed as $Pre \circ V_1 \circ V_2 \circ Post$, where Pre and $Post$ are pre- and post-amble code in the program but not in the weighted path, and V_1 and V_2 are the two parts of the weighted path, split at the crossover point. Similarly, $O = Pre \circ O_1 \circ O_2 \circ Post$. This produces $Pre \circ O_1 \circ V_2 \circ Post$ and $Pre \circ V_1 \circ O_2 \circ Post$, and both these offspring are copied into the next generation. All statements in Pre and $Post$ are left untouched. When constructing $O_1 \circ V_2$ from $V_1 \circ V_2$ and the original program, each statement $s \in V_1$ is swapped with its counterpart in the original program with probability equal to its weight.[3]

The crossover rate is 1.0; that is, during each generation, every surviving variant undergoes crossover. Individuals are paired for crossover randomly without replacement.

This is a nonstandard version of crossover, and in Section 3.3, we compare its performance with a more traditional implementation. The intuition behind crossing back to the original program is similar to the intuition behind elitist strategies — some mutations could cause irretrievable damage, and this provides a way to preserve the original functionality of the program. We use the weighted path to bias the probability of exchanging a single gene (statement) because we most want to change statements with higher weights. This further protects the positive functionality from damange.

2.4 Minimizing the repair

[3]Because of insertions and deletions, some of these statements may be empty.

The search terminates when GP discovers a *primary repair* that passes both the positive and the negative test cases. However, the primary repair typically contains at least an order-of-magnitude more changes than are necessary to repair the program. For example, GP might produce no-op statements (**`x=x-0;`**), dead code (**`x=3; x=5;`**) or calls to irrelevant functions. We use program analysis techniques to minimize the primary repair to produce the *final repair*.

Using tree-structured differencing techniques [2], we can 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". Applying all of the changes to the original program produces the primary repair, while applying none of the changes leaves the original program. We seek to find a small subset of changes that produce a program that still passes all of the test cases.

Let $C_p = \{c_1, \ldots, c_n\}$ be the set of changes associated with 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. We have $Test(C_p) = 1$ and $Test(\{\}) = 0$ (i.e., the primary repair passes all test cases, the original program does not). A *one-minimal subset* $C \subseteq C_p$ is a set such that $Test(C) = 1$ and $\forall c_i \in C.\ Test(C \setminus \{c_i\}) = 0$. That is, a one-minimal subset of changes causes the program to pass all test cases, but dropping any single change from it fails at least one test case.

We use *delta debugging* [23] to efficiently compute a one-minimal subset of changes from the primary repair. Checking if a set is valid involves a fitness evaluation (a call to $Test$). Delta debugging is conceptually similar to binary search, but it returns a set instead of a single number. Intuitively, starting with $\{c_1, \ldots, c_n\}$, it might first check $\{c_1, \ldots, c_{n/2}\}$: if that half of the changes is sufficient to pass the $Test$, then $\{c_{1+n/2}, \ldots, c_n\}$ 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 minimal valid set by brute force potentially involves $\mathcal{O}(2^n)$ evaluations; delta debugging is $\mathcal{O}(n^2)$ in the worst case [24, 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.

3. RESULTS

In this section, we first illustrate how GP repairs bugs using the well-known recent bug in Microsoft's Zune audio player [8]. Next, we summarize earlier results on repairs in ten additional programs totaling over 60,000 lines of code. We then report on experiments that explore various aspects of GP performance, including the role of crossover, the effect of varying the number of test cases in the fitness function, the relative importance of the different genetic operations, and the effect of path length on time to solution.

In all of the experiments, a standard *trial* uses the following setup. The population size is 40, and GP runs for a maximum of twenty generations. For the first ten generations, the global mutation rate is $p_m = 0.06$, and statements visited on both the positive and negative test cases are given a weight of 0.01. If no primary repair is found, the current population is discarded, the global mutation rate is lowered to $p_m = 0.03$, statements visited on both sets of test cases receive a weight of 0.00, and the process is restarted for ten more generations.

The trial terminates if it discovers an initial repair. We performed

100 trials for each program. We memoize fitnesses such that 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.

3.1 Example: Repairing the Zune Bug

On December 31st, 2008 a widely reported bug was discovered in the Microsoft Zune media players, causing them to freeze up [8]. The fault was a bug in the following program fragment:[4]

```
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("current year is %d\n", year);
18 }
```

When the value of of the input **days** is the last day of a leap year (such as 10593, which corresponds to Dec 31, 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(1000)** visits lines 1–8, 11–18. The negative test case **zunebug(10593)** visits lines 1–16, and then repeats lines 3, 4, 8, and 11 infinitely.

For the purposes of this example, our negative test case consists of the inputs 366 and 10593, which cause an infinite loop (instead of the correct values, 1980 and 2008), and our positive test cases are the inputs 1000, 2000, 3000, 4000, and 5000, which produce the correct outputs 1982, 1985, 1988, 1990 and 1993.

Here, we focus on one variant, V. V 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.

[4] Note that the original program source code does not make lines 9–10 explicit: the AST represents missing blocks, such as those in **if** statements without **else** clauses, as blocks containing zero statements.

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. The resulting program is shown below:

```
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 initial 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:

```
5    if (days > 366) {
6      year += 1;
7    }
8    else {
9      // days -= 366; // deleted c2
10   }
11   days -= 366;
```

This produces the final repair, shown below. This is one of the many possible repairs that the search might produce.

```
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; // repair deletes
7            year += 1;
8          }
9          else {
10         }
11         days -= 366;        // repair inserts
12       } else {
13         days -= 365;
14         year += 1;
15       }
16     }
17     printf("current year is %d\n", year);
18   }
```

Figure 1 shows how the average fitness of the population changes over time in one GP trial. In this run, we used five positive test cases (weight 1 each) and two negative test cases (weight 10 each). Also shown in Figure 1 is the fitness trajectory of the primary repair V beginning with Generation 1, in which V is the original program, and continuing up to Generation 7, when the primary repair is discovered.

3.2 Other Repairs

We tested the method on ten programs in addition to the Zune bug, generating repairs in every case. These results are summarized in Figure 2; portions of this figure are reproduced from [22]. The results show that GP can automatically discover repairs for a wide variety of documented bugs in production C programs. The

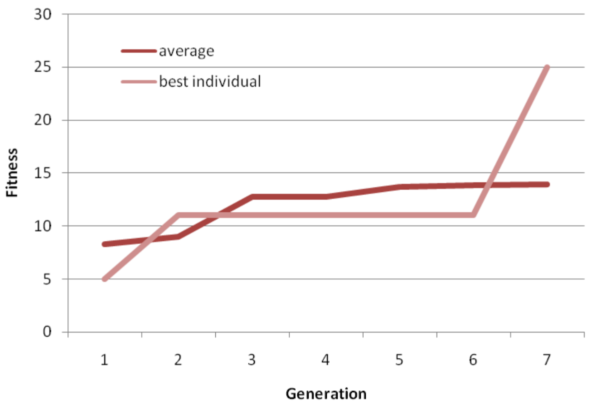

Figure 1: Evolution of the Zune bug repair for one successful GP trial. The blue box curve plots the average fitness of the population, and the red diamond curve plots the fitness of the individual V that becomes the primary repair.

results raise many interesting questions about how the repairs are discovered by GP, test case selection, scalability to larger problems, and repair quality. We address the first three of these issues in the following subsections and return to the question of repair quality in Section 5.

3.3 The Role of Crossover

Crossover is an important search operator in GP, creating new individuals by recombining partial solutions (subtrees) from different individuals. The original implementation (described in Section 2.3) does not take advantage of the potential power of crossover because individuals are always *crossed back* to the original parent program. In Figure 2 we report data comparing the performance of this implementation with a traditional GP crossover operator, which takes two individuals as input, chooses a random position (i.e., statement) from each one, swaps their contents, and returns two new chromosomes.

Although the data are not conclusive, the two implementations appear to be comparable: each outperforms the other in some instances. A potential explanation of these results is that crossover is not contributing enough to the search for it to matter, regardless of which version we use. We explore this question in Section 3.5.

3.4 Varying the Number of Test Cases

The results in Figure 2 typically involve six test cases. limiting the fitness function to six discrete values. This could limit the complexity of repair that can be evolved as it provides a relatively coarse signal to GP. Also, programs may have more critical functionality than a few test cases can capture. Typically, programs have too many test cases rather than too few, and test case selection and time-aware test suite prioritization are active research areas (e.g., [20]).

In this section, we ask how GP performance changes when more test cases is used. Figure 3 shows the averaged results of 70 distinct trials on the Zune bug, using a fitness function with 24 test cases: 20 positive test cases and 4 negative test cases. The error bars represent one standard deviation.

Ideally, the test cases would be independent. In this case they were selected by taking the original five (which were 1000, 2000, 3000, 4000, and 5000) and adding the following: one arbitrary negative number (-100); one negative number that if it were positive would cause the program to hang (-366); one extremely large

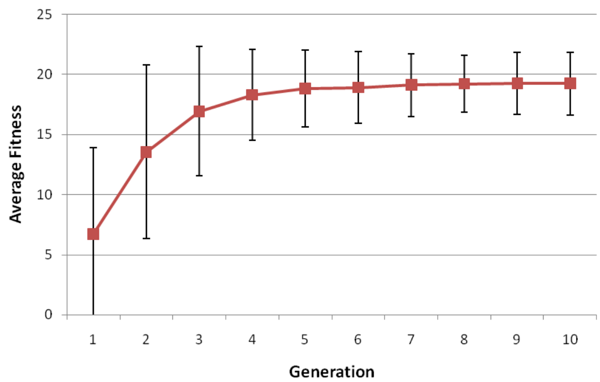

Figure 3: Evolution of the Zune bug repair with 20 positive test cases and 4 negative test cases, all equally weighted. The boxes represented the average over 70 distinct trials; the error bars represent one standard deviation.

number (100000000); selecting several arbitrary numbers near leap years and then finding the numbers around those dates that exercise the bugs. The four negative testcases include the original bug (that caused all the zunes to crash in December) as well as several other leap years: 1980 (366), 1984, and 2012.

Unsurprisingly, early generations have fitness values with high variance, and in later generations the variance decreases. The original program passes the positive test cases but fails the negative test cases; it thus has a fitness of 20. Note that over all generations, the average fitness is below the baseline of 20, indicating that the majority of individuals are worse than the original program. Thus, the primary repair is discovered by first losing fitness and then regaining it on the way to the global optimum.

Intuitively, additional test cases could reduce success rate by overly constraining the search space. However, the opposite happened in this example. Using seven test cases, the average success rate is 72%, while the average success rate using 24 test cases is 75%. However, adding test cases does dramatically increase the total running time of the algorithm: with seven test cases, the average time to discover the primary repair is 56.1 seconds; with 24, this time increases to 641.0 seconds. This makes sense: Every fitness evaluation potentially involves running all of the test cases. Therefore, in general, we prefer a fitness function with a small number of test cases.

3.5 Genetic Operators

There are several unusual features of our implementation. This section studies the relative contribution of the different operators and estimates how many genetic changes are needed to accomplish a repair. Figure 4 reports data on several aspects of the GP search for a representative sample of the programs we have repaired, averaged over 20 trials.

The second column reports the percentage of unique program variants that fail to compile. We memoize results so as to avoid recompiling a program that has already failed, which is why our numbers look so low.

The remaining columns report data on the number of genetic operations involved per fitness evaluation and per successful repair. The average number of genetic operations per fitness evaluation is 2.19 (sum of Cols. 3, 4, 5, and 7), and the average number of operations to produce a successful repair (sum of Cols. 8, 9, 10, and 12) is 4.63. Summing over all individuals in the population (40) and considering that a repair is discovered on average within 3.6

Program	Version	Stmt Nodes / Lines of Code	Pos/Neg Test cases	Weighted Path Length	Crossover Success	Trad. Crossover Success	Final Repair
`zune`	example	14 / 28	5/2	1.1	71%	58%	4
`gcd`	example	10 / 22	5/1	1.3	54%	24%	2
`uniq`	ultrix 4.3	81 /1146	5/1	81.5	100%	100%	4
`look-u`	ultrix 4.3	90 /1169	5/1	213.0	99%	100%	11
`look-s`	svr4.0 1.1	100 /1363	5/1	32.4	100%	100%	3
`units`	svr4.0 1.1	240 /1504	5/1	2159.7	7%	5%	4
`deroff`	ultrix 4.3	1604 /2236	5/1	251.4	97%	97%	3
`nullhttpd`	0.5.0	1040 /5575	6/1	768.5	36%	47%	5
`indent`	1.9.1	2022 /9906	5/1	1435.9	7%	34%	2
`flex`	2.5.4a	3635 /18775	5/1	3836.6	5%	4%	3
`atris`	1.0.6	6470 /21553	2/1	34.0	82%	82%	3

Figure 2: Program Repairs for eleven programs. "Stmt Nodes" gives the total number of statement nodes in the AST (see Section 2.1) while "Lines of Code" is a traditional measure of program size. "Pos/Neg Test cases" lists the number of positive and negative test cases used in the fitness function. "Weighted Path Length" (see Section 2.1) is informally the amount of the AST considered by mutation and crossover. The next columns report on GP performance. The "Crossover" heading refers to the crossover implementation described in Section 2.3. The "Trad. Crossover" heading refers to a traditional crossover operator (Section 3.3). For both, "Success" lists the success rate, the percentage of trials that produced a repair. "Final Repair" gives the size of the repair as represented by the Unix `diff` utility, measured in lines.

Program	Xover	Mut	Fit Fun	Pos Test	Neg Test	`gcc`
`gcd`	0.2	0.1	0.7	66.1	69.7	12.5
`uniq`	1.1	0.3	1.1	5.5	17.2	8.0
`look-u`	1.0	0.3	1.0	7.2	24.9	8.7
`look-s`	0.9	0.3	0.3	21.2	22.1	6.6
`units`	1.0	1.2	4.6	22.2	39.7	38.0
`deroff`	3.6	3.0	7.7	11.5	45.9	60.0
`nullhttpd`	6.3	8.8	30.2	273.5	117.4	67.4
`indent`	11.6	28.3	53.2	90.4	111.1	238.1
`flex`	38.6	17.0	53.3	11.2	42.7	92.0
`atris`	19.6	3.6	39.0	0.3	0.4	22.4
Average	8.4	6.3	19.1	50.9	49.1	55.4

Figure 5: GP performance. Average length of time, in seconds, spent executing each operator type in runs that converged on a successful repair.

generations (Col. 13), the search on average requires 667 genetic operations to discover the primary repair. When we consider the individual operations, it is difficult to discern a clear pattern and make definite conclusions about the relative importance of the different operators. (The Delete operator appears to be the most effective, except that its average is skewed by one example (indent).)

Overall, however, we can conclude that GP is routinely discovering successful repairs with a surprisingly small amount of search. This suggests that much of the cleverness in repairing bugs arises from the problem representation, the fitness function, and the minimization step. And, it raises the question about how well the approach will scale up to more complex problems, which we address in the next subsection.

3.6 GP Performance and Scalability

To assess efficiency, we first consider the relative execution times of various portions of the algorithm. Figure 5 reports the average running time for trials that led to successful repairs. The experiments were conducted on a quad-core 3 GHz machine; with a few exceptions, the process was CPU-bound. The GP prototype is itself single-threaded, with only one fitness evaluation at a time, during a fitness evaluation execute all test cases are executed in parallel.

The "Xover" and "Mut" columns give the time, in seconds, spent manipulating the internal representation to perform crossover and mutation operations (e.g., to generate a random number to select the crossover point). The "Fit Fun" column includes the time to compute the fitness function plus the time to pretty-print the AST and memoize results based on a hash of the printed source code. The "Pos Test" and "Neg Test" columns denote the time spent running the user-supplied test cases to evaluate the fitness function. Note that test cases come with timeouts, and many of them involve explicit internal delays (e.g., ad hoc instructions to wait two seconds to get the web server "up and running" and listening on a port before requests are sent to it). The last column gives the time spent compiling variants. Overall, more than half of the time is spent executing the positive and negative test cases. Almost thirty percent of the time is spent compiling. The initial implementation makes no attempt at incremental compilation and explicitly invokes `gcc` each time to produce an executable, so this is a potential source of future optimizations.

In order to assess the practicality of this approach, we need to know how the algorithm scales with problem size, and we need to know the expected size of the problems we would want to solve.

Figure 6, plots weighted path length against search time, measured as the average number of fitness evaluations until the first repair. On a log-log scale, the relationship is roughly linear with slope 1.26 (90% confidence: [0.90, 1.63]), including the two outliers (**`atris`** and **`uniq`**). Although we do not have enough data to draw strong conclusions, the plot suggests that search time may scale as a power law of the form $y = ax^b$ where b is the slope of the best fit line (1.26) and $b = 1$ would indicate that search time grew linearly. This is encouraging because it suggests that search time grows as a small polynomial of the weighted execution path and not as an exponential.

A second question concerns the size distribution of bugs. In short, what is the expected length of the execution path for bugs that the GP might be expected to repair? Although we do not have solid data on this question, we note that in 2004 Van Belle documented the size distribution of revisions checked in to several large open-source repositories and discovered that it resembles a power

Program	% Don't Compile	Genetic Ops per Fitness Eval.					Genetic Ops per Repair					Generations per Repair
		Ins	Del	Swaps	Muts	Xover	Ins	Del	Swaps	Muts	Xovers	
`zune`	5.33	0.36	0.27	0.38	0.72	0.27	2.50	0.50	0.00	2.25	2.25	3.75
`gcd`	3.70	0.53	0.08	0.10	0.66	0.34	3.75	0.00	0.00	3.00	1.75	3.50
`uniq`	25.37	0.29	0.53	0.25	0.76	0.23	0.00	1.67	0.33	1.17	0.67	1.83
`look-u`	31.73	0.44	0.95	0.48	0.95	0.05	0.22	1.78	0.22	1.00	0.00	1.00
`look-s`	38.72	0.49	0.71	0.72	0.93	0.07	0.60	1.20	0.60	1.40	0.40	1.60
`units`	12.85	0.16	0.16	0.14	0.54	0.46	1.67	1.00	0.67	4.33	3.17	6.33
`deroff`	61.10	1.10	1.34	1.05	0.81	0.19	0.00	4.67	0.00	1.00	0.00	1.00
`nullhttpd`	57.93	1.27	1.24	1.44	0.62	0.38	0.00	4.50	0.00	1.25	0.25	7.54
`indent`	65.07	1.49	1.52	1.61	0.56	0.43	2.00	8.17	1.33	2.50	0.17	5.17
`flex`	37.14	0.40	0.41	0.40	0.73	0.27	0.00	2.80	0.00	1.20	0.00	1.20
`atris`	25.10	0.34	0.33	0.20	0.72	0.28	0.00	2.00	0.00	1.00	0.00	6.40
Average	32.19	0.62	0.68	0.62	0.73	0.27	0.98	2.57	0.29	1.83	0.79	3.57

Figure 4: GP Operators: The "% Don't Compile" column reports the percentage of unique program variants that failed to compile. The "Genetic Ops per Fitness Eval" columns (Insertions, Deletions, Swaps, Mutation and Crossovers) show the total average number of genetic changes per individual between fitness evaluations for each evaluated program variant, in units of genetic operations. Note that one Mutation typically involves multiple Insertions, Deletions and/or Swaps (see Section 2.3). The "Genetic Ops per Repair" columns report the average number of times each type of evolutionary operation was used in the evolution of the first primary repair discovered in each successful program run. The last column designates the number of generations spent on each successful instance.

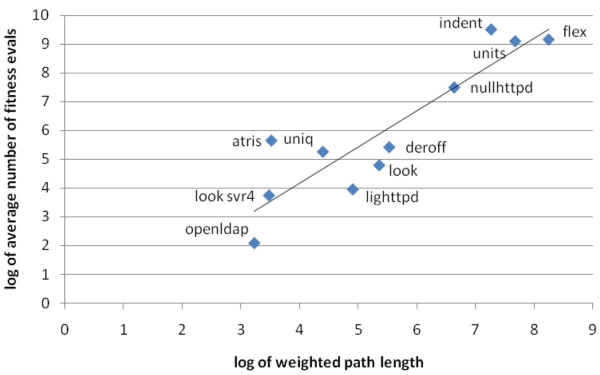

Figure 6: GP Search Time Scales with Execution Path Size. Data are shown for 11 programs successfully repaired by GP. The x-axis is the natural logarithm of the weighted path length, and the y-axis shows the natural logarithm of the total number of fitness evaluations performed before the primary repair is found (averaged over 100 runs).

law distribution [10, 9]. Although he was unable to conclude exactly which distribution best fit his data, the trend clearly showed that there were many more small changes than large ones. This is an important area for future work, but if most bug repairs turn out to be localized to a single function, we could be much more optimistic about the practicality of the GP approach.

4. RELATED WORK

To our knowledge, GP has not previously been used to evolve off-the-shelf legacy software. Arcuri [3, 5, 4] proposed the idea of using GP to automate the repair of software bugs, demonstrating the idea on a hand-coded example of the bubble sort algorithm. However our experiments appear to be the first to report results on real programs with real bugs. Our approach differs in several details from Arcuri's. For example, we do not rely on formal specifications; we constrain the search space to regions where the defects occur instead of evolving the entire program tree; and we control code bloat once at the end of the search rather than incrementally. Localizing the search space allow the search to scale to large programs and despite recent advances in specification mining [14], formal specifications are rarely available in practice (e.g., none of the experimental programs used in this paper have formal specifications available). Several aspects of Arcuri's work could be combined with ours, including his use of co-evolutionary techniques to select test cases.

Previous work on automatic patch generation used the finite state machine structure of certain formal safety specifications to generate repairs [21]. The GP approach reported here addresses some of the limitations of this previous work. First, we introduced positive test cases to prevent repairs that sacrifice functionality of the original program. Second, the GP approach can handle a wider range of defects; for instance repairing infinite loops such as the Zune bug. Third, the GP approach does not rely on formal specifications of the policy being violated by the fault.

Finally, Demsky and Rinard [12] use specifications to repair data structures at run-time Their technique works at the level of data structures and not at the level of program logic, and it may be viewed as addressing an orthogonal problem. For example, their approach does not generate a source code patch that fixes the problem; instead, they restore data invariants at run-time to keep the program running. Our techniques are complementary: Their repair approach might be used to keep a critical program running while our technique searches for a long-term repair.

5. DISCUSSION AND CONCLUSIONS

The results reported here demonstrate that GP can be applied to the problem of bug repair in legacy C programs. To date, GP has succeeded at every repair task we have attempted, but as can be seen in Figure 2 this is only eleven programs. Although encouraging, the results raise many interesting questions, which we hope to address in future work. For example, we are interested in how much repairs vary after minimization, and how repair quality compares to human-engineered solutions. In our experiments to date, most repairs look identical after the minimization step. There are several interesting questions related to the GP component of the process, for example: Is crossover essential to the search? Is it sufficient to control code bloat at the end of the run? Are we using the optimal GP design? Are we using optimal parameter settings? Because the process proved so successful initially, we have not experimented with parameter values, selection strategies, and operator design. These all could almost certainly be improved. Similarly, there are many ways that the fitness function design could be enhanced, say by different weightings on the test cases or by dynamically choosing test cases to be included in the fitness function.

Beyond these immediate steps, there are other areas for more ambitious future work. For example, we plan to develop a generic set of repair templates so the GP has one source of new code to use in mutation, beyond those statements that happen to be in the program being repaired. Another possibility is to use more sophisticated bug localization techniques (e.g., [6]) to help control the size of the weighted execution path. We could potentially extend the representation to include data structure definitions and variable declarations. We are also interested in the question of code bloat and whether our two strategies for dealing with it (using execution paths and minimizing the primary repair) could be applied to other GP settings. 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 candidates for the GP technique.

The dream of automatic programming has eluded computer scientists for at least 50 years. Although the methods described in this paper are a small initial step towards this goal, our success at repairing bugs automatically may say as much about the state of today's software as it says about the efficacy of our methods. In today's environments, it is exceedingly difficult to understand an entire software package, test it adequately, or to localize the source of an error. In this context, it should not be surprising that programming has a large trial and error component, and that many bugs are repaired by copying code from another location and pasting it in to another. This is not so different from the approach we have described here.

6. REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (second edition)*. Pearson Education, 2006.

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

[3] A. Arcuri. On the automation of fixing software bugs. In *Proceedings of the Doctoral Symposium of the IEEE International Conference on Software Engineering*, 2008.

[4] A. Arcuri, D. R. White, J. Clark, and X. Yao. Multi-objective improvement of software using co-evolution and smart seeding. In *Proceedings of the International Conference on Simulated Evolution And Learning*, pages 61–70, 2008.

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

[6] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: localizing errors in counterexample traces. *SIGPLAN Not.*, 38(1):97–105, 2003.

[7] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN Workshop on Model Checking of Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 103–122, May 2001.

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

[9] T. V. Belle. *Modularity and the Evolution of Software Evolvability*. PhD thesis, University of New Mexico, Albuquerque, NM, 2004.

[10] T. V. Belle and D. H. Ackley. Code factoring and the evolution of evolvability. In *GECCO '02: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1383–1390, San Francisco, CA, USA, 2002. Morgan Kaufmann Publishers Inc.

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

[12] B. Demsky and M. C. Rinard. Automatic data structure repair for self-healing systems. In *Object-Oriented Programming, Systems, Languages, and Applications*. 2003.

[13] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *IEEE Symposium on Security and Privacy*, pages 120–128, 1996.

[14] M. Gabel and Z. Su. Symbolic mining of temporal specifications. In *International Conference on Software Engineering*, pages 51–60, 2008.

[15] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Companion to the conference on Object-oriented programming systems, languages, and applications*, pages 132–136, 2004.

[16] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *Programming Language Design and Implementation*, June 9–11 2003.

[17] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: An infrastructure for C program analysis and transformation. In *International Conference on Compiler Construction*, pages 213–228, Apr. 2002.

[18] T. M. Pigoski. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley & Sons, 1996.

[19] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[20] K. Walcott, M. Soffa, G. Kapfhammer, and R. Roos. Time-aware test suite prioritization. In *International Symposium on Software Testing and Analysis*, pages 1–12, 2006.

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

[22] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. In *International Conference on Software Engineering*, 2009.

[23] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Foundations of Software Engineering*, pages 253–267, 1999.

[24] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.