Factors Affecting the Use of Genetic Algorithms in Test Suite Augmentation

Zhihong Xu, Myra B. Cohen and Gregg Rothermel
Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE, USA
{zxu, myra, grother}@cse.unl.edu

ABSTRACT

Test suite augmentation techniques are used in regression testing to help engineers identify code elements affected by changes, and generate test cases to cover those elements. Researchers have created various approaches to identify affected code elements, but only recently have they considered integrating, with this task, approaches for generating test cases. In this paper we explore the use of genetic algorithms in test suite augmentation. We identify several factors that impact the effectiveness of this approach, and we present the results of a case study exploring the effects of one of these factors: the manner in which existing and newly generated test cases are utilized by the genetic algorithm. Our results reveal several ways in which this factor can influence augmentation results, and reveal open problems that researchers must address if they wish to create augmentation techniques that make use of genetic algorithms.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Algorithms, Experimentation

Keywords

Regression testing, test suite augmentation, genetic algorithms, empirical studies

1. INTRODUCTION

Software engineers use regression testing to validate software as it evolves. To do this more cost-effectively, they often begin by reusing existing test cases. Existing test cases, however, are often inadequate to retest code or system behavior that is affected by changes. *Test suite augmentation techniques* (e.g., [1, 16, 22]) help with this, by identifying where new test cases are needed and then creating them.

Despite the importance of augmentation, most research on regression testing has focused instead on test suite reuse. There has been research on algorithms for identifying where new test cases are needed by *identifying affected elements* [1, 3, 7, 14, 16], but these approaches do not then generate the required test cases, leaving that task to engineers. There has been research toward generating test cases given pre-supplied coverage goals (e.g., [5, 12, 13, 17, 19]), but approaches for integrating the test generation task with reuse of existing tests and with techniques that identify affected elements have not been explored.

In [22], we present a *directed test suite augmentation technique* that uses a regression test selection algorithm [15] to identify where new test cases are needed and existing test cases applicable to those needs, while integrating this information with a concolic approach [17] for test case generation.

Search-based algorithms can also be used to generate test data. Thus, we have been researching a test suite augmentation technique that uses genetic algorithms. In the course of this work, however, we have discovered that there are several factors that can influence the performance of such augmentation techniques beyond the normal parameters for tuning a genetic algorithm, through effects on the algorithm's population size and its diversity. These factors include: (1) the algorithm used to identify affected elements; (2) the size and composition of the existing test suite; (3) the order in which program elements are considered while generating test cases; and (4) the manner in which existing and newly generated test cases are harnessed by the genetic algorithm.

We believe that understanding the foregoing factors is essential if we wish to create cost-effective test suite augmentation techniques using genetic algorithms. To begin to build such an understanding, we have conducted a case study exploring the fourth factor, which we believe is likely to be the most significant in its effects. Our results show that the factor *can* affect the cost and effectiveness of test suite augmentation techniques that use a genetic algorithm for test case generation. This has implications for researchers wishing to create and study such techniques.

2. BACKGROUND AND RELATED WORK

Let P be a program, P' be a modified version of P, and T be a test suite for P. Regression testing is concerned with validating P'. To facilitate this, engineers may use the *retest-all* technique, re-executing all viable test cases in T on P', but this can be expensive. *Regression test selection* (RTS) techniques (e.g., [3, 15]) use information about P, P' and T to select a subset T' of T with which to test P'.

Test suite augmentation techniques, unlike RTS techniques, are not concerned with reuse of T. Rather, they are concerned with the tasks of (1) *identifying affected elements* (portions of P' or its specification for which new test cases are needed), and then (2) *creating or guiding the creation of test cases that exercise these requirements.*

Various algorithms have been proposed for identifying affected elements in software systems following changes. Some of these [4] operate on levels above the code such as on models or specifications, but most operate at the level of code, and in this paper we focus on these. Code level techniques [3, 7, 14] use various analyses such as slicing on program dependence graphs to select existing test cases that should be re-executed, while also identifying portions of the code that are related to changes and should be tested. However, these approaches do not provide methods for generating actual test cases to cover the identified code.

Four recent papers [1, 13, 16, 22] specifically address test suite augmentation. Two of these [1, 16] present an approach that combines dependence analysis and symbolic execution to identify test requirements that are likely to exercise the effects of changes, using specific chains of data and control dependencies to point out changes to be exercised. A potential advantage of this approach is a fine-grained identification of coverage needs; however, the papers present no specific algorithms for generating test cases. A third paper [13] presents a more general approach to program differencing using symbolic execution that can be used to identify requirements more precisely than [1, 16], and yields constraints that can be input to a solver to generate test cases for those requirements. However, this approach is not integrated with reuse of existing test cases.

The test suite augmentation approach presented in [22] integrates an RTS technique [15] with an adaptation of the concolic test case generation approach presented in [17]. This approach leverages test resources and data obtained from prior testing sessions to perform both the identification of coverage requirements and the generation of test cases to cover these. A case study shows that it can be effective and efficient; however, the approach is applicable only in cases in which concolic testing can be applied cost-effectively.

In other related work [23], Yoo and Harman present a study of *test data augmentation.* They experiment with the quality of test cases generated from existing test suites using an heuristic search algorithm. While their work is similar to the technique that we consider in this paper in that it uses a search algorithm and starts with existing test cases, their definition of augmentation differs from ours. They focus on duplicating coverage in a single release in order to improve fault detection, not on obtaining coverage of affected elements in a subsequent release.

Genetic algorithms have been used previously in regression testing [10] and for structural test case generation which begins with an initial (often randomly generated) test data population and evolves the population toward targets that can be blocks, branches or paths in the CFG of a program [8, 11, 12, 18, 20]. To apply such an algorithm to a program, we first provide a representation of the test problem in the form of a chromosome, and a fitness function that defines how well a chromosome satisfies the intended goal. The algorithm proceeds iteratively by evaluating all chromosomes in the population and then selecting a subset of the fittest to mate. These are combined in a crossover stage to generate a new population of which a small percentage of chromosomes in the new population are mutated to add diversity back into the population. This concludes a single generation of the algorithm. The process is repeated until a stopping criteria has been met and the solution has converged.

3. FACTORS AFFECTING AUGMENTATION

We have identified several factors that are independent of genetic algorithm parameters, but that could potentially affect how well such algorithms perform by impacting both population size and diversity. We now describe these factors.

F1: Algorithm for identifying affected elements. As discussed in Section 2, various algorithms have been proposed for identifying affected parts of software systems following changes. The numbers and complexity of identified affected elements can clearly impact the cost of subsequent test generation efforts by affecting the numbers of inputs that genetic algorithms must generate, and the complexity of the paths through code that the algorithms must target.

F2: Characteristics of existing test suites. Test suites can differ in terms of size, composition, and coverage achieved. Such differences in characteristics could potentially affect augmentation processes. For example, the extent to which an existing suite achieves coverage prior to modifications can affect the number and locations of coverage elements that must be targeted by augmentation. Furthermore, test suite characteristics can impact the size and diversity of the starting populations utilized by genetic algorithms.

F3: Order in which affected elements are considered. For genetic algorithms that utilize existing test cases to generate new ones, the order in which a set of affected elements are considered can affect the overall cost and effectiveness of test generation, and thus, the cost and effectiveness of augmentation. For example, if elements executed earlier in a program's course of execution are targeted first, and if test cases are found to reach them, these may enlarge the size and diversity of the population of test cases reaching other affected elements later in execution, giving the test generation algorithm additional power when it considers those – power that it would not have if elements were considered in some other order.

F4: Manner in which test cases are utilized. Given a set of affected elements, a set of existing test cases, and an augmentation algorithm that uses existing test cases to generate new ones, there are several ways to interleave the use of existing and newly generated test cases in the augmentation process. Consider, for example the following approaches:

1. For each affected element, let the augmentation approach work with all existing test cases.
2. For each affected element, analyze coverage of existing test cases to determine those that are likely to reach it and let the augmentation approach use these.
3. For each affected element, let the augmentation approach use existing test cases which, based on their execution of the modified program, can reach it.
4. For each affected element, let the augmentation approach use existing test cases that can reach it in the modified program (approach 3), together with new test cases that have been generated thus far and reach it.
5. For each affected element, begin with approach 4 but select some subset of those test cases, and let the augmentation approach use these.

Each of these approaches involves different tradeoffs. Approach 1 incurs no analysis costs but may overwhelm a genetic algorithm approach by providing an excessively large population. Approach 2 reduces the test cases used by the genetic algorithm but in relying on prior coverage information may be imprecise. Approach 3 passes a more precise set of test cases on to the genetic algorithm, but requires that these first be executed on the modified program. None of the first three approaches takes advantage of newly generated test cases as they are created, and thus they may experience difficulties generating test cases for new elements due to lack of population diversity. Approaches 4 and 5 do use newly generated test cases along with existing ones, and also use new coverage information, but differ in terms of the number of new test cases used, again affecting size and diversity.

Among these four factors, we believe that F4 is of particular interest, because it provides a range of approaches potentially differing in cost and effectiveness for using genetic algorithms in augmentation tasks. We thus set out to perform a study investigating this factor.

4. CASE STUDY

To investigate factor F4, we fix the values of other factors at specific settings as discussed below. The research questions we address are:

RQ1: How does factor F4 affect the cost of augmentation using a genetic algorithm?

RQ2: How does factor F4 affect the effectiveness of augmentation using a genetic algorithm?

4.1 Objects of Analysis

For our experiment, we chose a non-trivial Java software application, `Nanoxml`, from the SIR repository [6]. `Nanoxml` has multiple versions and more than 7000 lines of code. `Nanoxml` is an XML parser that reads string and character data as input. It has many individual components which realize different functionality. Drivers are used to execute the various components. We focused on performing augmentation as the system goes through three iterations of evolution, from versions v_0 to v_1, v_1 to v_2, and v_2 to v_3. In other words, we augmented the test suite for v_1 using the suite for v_0, augmented the test suite for v_2 using the suite for v_1 and augmented the test suite for v_3 using the suite for v_2. The test suites for v_0, v_1, and v_2 contain 235, 188, and 234 specification-based test cases applicable to the following versions, respectively. These test cases cover 74.7%, 83.6% and 78.5% of the branches in versions v_0, v_1, and v_2, respectively.

4.2 Genetic Algorithm

To investigate our research questions we required an implementation of a genetic algorithm tailored to fit our object program. We used an approach suitable to the object, that could be modified to work on other string-based programs.

Our chromosome consists of strings containing two parts: test drivers that invoke an application and input files (XML files) that are the target of the application. The driver is a single gene in the chromosome. The XML files give way to a set of genes; one for each character in the file.

We treat each part of the chromosome differently with respect to crossover and mutation. For the test drivers, we use a pool of drivers that are provided with the application. We do not modify this population, but rather modify how it is combined with the input files that are evolved. We do not perform crossover on the drivers; we use only mutation. When a chromosome's driver gene is selected for mutation, the entire driver is replaced with another (randomly selected) valid driver from our pool of potential drivers. This prevents invalid drivers from being generated.

In the XML part of our chromosome, we perform a one point crossover by randomly selecting a line number that is between 0 and the number of lines of the smaller file. We then swap everything between files starting at that row to the end of the file. We do not test the file for well-formed XML, but rather use it as-is. During mutation, each character in the input file is considered a unit. We randomly select the new character from the set of all ASCII upper and lower case letters combined with the set of special characters found in the pool of input files, such as braces and underscores.

Our search targets are branches in the program, therefore for our fitness function we use the *approach level* described in [21]. For our initial implementation, for the sake of simplicity and due to the instrumentation overhead required, we did not combine this with branch distance. We nonetheless achieved good convergence on these programs; still, research suggests that branch distance is an important part of the fitness function [2] and we intend to consider it in the future.

The approach level is a discrete count measuring how far we were from the predicate controlling the target branch in the CFG when we deviated course. The further away we are from this predicate, the higher the fitness, therefore we are trying to minimize this value. If we reach the predicate leading to our target, the approach level is 0.

For selection, we select the best half of the population to generate the next generation; we keep the selected chromosomes in the new generation. We rank the chromosomes and divide them into two parts. The first chromosome in the first half is mated with the first chromosome in the second half, and the second chromosome in the first half with the second chromosome in the second half, etc.

We use a three stage mutation. First we select 30% of the test cases in the population for mutation and mutate the driver for this test case. Next we select 30% of the lines in the file part of the chromosome for these test cases, and then select 30% of the genes in these rows for mutation. Our stopping criterion is coverage of the required program branch or ten generations of our genetic algorithm, whichever is reached first.

Note that we manually tuned the parameters used by our algorithm so that we can cover as many branches of a program for a straight test case generation problem before starting our experiments, and this process also led us to choose the values of 30/30/30. However, we performed this tuning for normal test case generation, not augmentation, and we did it on the base version of the program. This is appropriate where augmentation is concerned, because in a regression testing setting, a test generation algorithm can be tuned on prior versions of the system before being utilized to augment test suites for subsequent modified versions.

4.3 Factors, Variables, and Measures

We describe our factors, variables and measures next.

4.3.1 Fixed Factors

Our goal being to consider only the effects of factor F4, we selected settings for the other factors described in Section 3 and held these constant.

For better understanding, we use the example in Figure 1 to explain factors. The figure shows portions of two versions of a program, in a control flow graph notation. The graph on the left corresponds to an initial version a and the graph on the right corresponds to a subsequent version b. Nodes represent statements within methods, and root nodes are indicated by labels $m1$ through $m6$. Solid lines represent control flow within methods and dashed lines represent calls. Labels on dashed lines represent test cases in which the associated method call occurs. From version a to b, changes occur in method $m3$ in which one branch is added to call a new method $m6$. Other methods remain unchanged.

F1: Algorithm for identifying affected elements.

As affected elements we use a set of potentially impacted coverage elements in P'. To calculate these, we use the analysis at the core of the DejaVu regression test selection technique [15]. This analysis compares control-flow graph (CFG) representations of the methods in programs P and P', where P' is a modified version of P, by performing a simultaneous depth-first traversal the CFGs. This analysis identifies for each method m the pairs of corresponding edges, $e \in CFG(m, P)$ and $e' \in CFG(m, P')$, that reach statements that have been modified. We call e (or e') a *dangerous* edge because it leads to code that may cause program executions to exhibit different behavior. We treat methods containing dangerous edges as "dangerous methods", and then apply an algorithm that walks the interprocedural control flow graph for P' to find the set of affected methods that can be reached via control flow paths through one or more of the dangerous methods. All branches contained in affected methods are targets for augmentation.

In our example, $m3'$ contains a dangerous edge, so it is a dangerous method, and $m4$ and $m6$ are reachable via interprocedural control flow from the dangerous edge in $m3'$, so they are affected methods. Further, $m3$'s return value to $m1$ is affected, so $m1$ is also affected. Method $m2$ is called along the path from $m3$ to the exit node of $m1$, so it too is affected. Continuing to propagate impact, $m5$ and $m4$ are called by $m2$, so they are both affected. In this example all methods and all branches contained in them are affected, but in general this may not be the case.

F2: Characteristics of existing test suites.

Our test suites T are those provided with `Nanoxml`. As described above, they are specification-based and operate at the application testing level, and they achieve branch coverage levels ranging from 74.7% to 83.6% on our versions.

F3: Order in which affected elements are considered.

As an ordering, we used an approach that causes individual methods to be considered in top-down fashion in control flow, thus approximating the consideration of affected elements in such a fashion. The approach applies a depth first ordering to all affected methods in the call graph for P'. The effect of this approach is to cause augmentation to be applied to a particular method only after its predecessors in the call graph have been considered, which may allow test cases generated earlier to cover methods addressed later. Note, however, that this approach may be imprecise in relation to cycles, and in the order in which it considers individual branches within individual methods. As an example, in Figure 1, the ordering of methods in version b imposed by our approach is $m1$, $m3'$, $m2$, $m5$, $m4$ and $m6$.

4.3.2 Independent Variables

Our independent variable is factor F4, the "treatment of test cases" factor, and we use the five treatments described in that section, more precisely described here. To facilitate the description, Table 1 presents information on the disposition of test cases achieved by the treatments, applied to the example in Figure 1.

Treatment 1.

For each affected element e in method m, all existing test cases in T are used to compose the initial population for the genetic algorithm. In this case we may have a large population for the genetic algorithm, which may cause it to take a relatively long time to complete the augmentation task for P'. However, this approach does increase the variety in the population which could improve the effectiveness of the search. In Figure 1, for all target branches, we use all four test cases $t1$ to $t4$ to compose the initial population.

Treatment 2.

For each affected element e in method m, all existing (old) test cases that used to reach m in P, denoted by $T_{old:P}$, are used to compose the initial population for the genetic algorithm. In this case since we are using old coverage information, we avoid running all existing test cases on P' first and focus on the changes from P to P'. However, if we have new methods in P', since there are no existing test cases available to reach them, we lose opportunities to perform augmentation for them and may lose some coverage.

In Figure 1, in this case, for $m1$, $m2$, $m3$ and $m4$ we use all test cases to form the initial population, since all the test cases reach them in version a. For $m5$, we use just $t1$ and $t2$. In this case, since there is no method $m6$ in version a and there are no existing test cases that reach it in that version, we cannot do augmentation for $m6$ directly.

Treatment 3.

For each affected element e in method m, all existing test cases that reach m in P', denoted by $T_{old:P'}$, are used to compose the initial population for the genetic algorithm. In this case we need to run all existing test cases on P' first and then we use the new coverage information, which is more precise than in treatment 2 since these test cases are near to our target, and this helps the genetic algorithm in its search. Also, $T_{old:P'} \subseteq T$, so we may lose some variety in the population, but we may save time in the entire process since we have fewer test cases to execute in each iteration.

Considering Figure 1, when we run all existing test cases on version b, some of them take new execution paths. Methods $m3$, $m4$ and $m6$ contain uncovered branches after checking the coverage of all existing test cases on b. For $m3$, all existing test cases still reach it in b so they are used in its initial population. Because of the change in $m3$, all test cases that used to reach $m3$ take different paths and reach $m6$ so they are used to form the initial population for $m6$. There are only two test cases, $t2$ and $t4$ from $m2$, that reach $m4$ and they are used to form the initial population for $m4$.

Treatment 4.

For each affected element e in method m, all existing test cases that reach m in P' ($T_{old:P'}$) and all newly generated test cases that obtain new coverage in version b, denoted

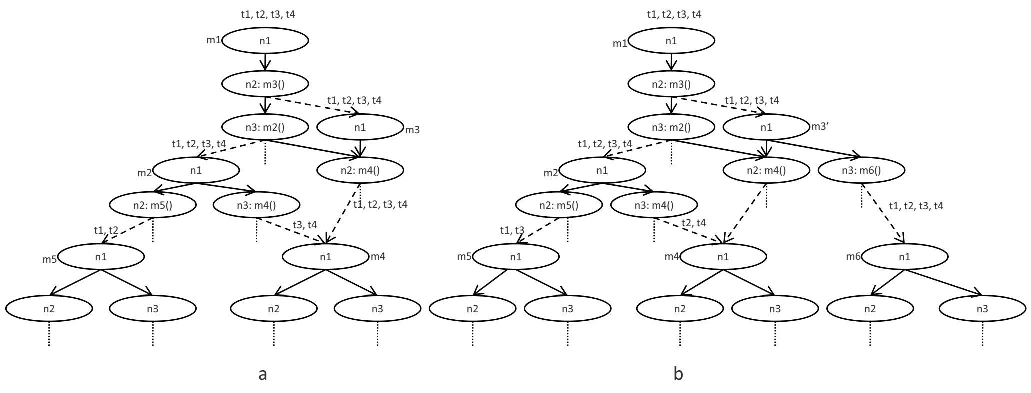

Figure 1: Partial control flow graphs for two versions of a program

Table 1: Disposition of Test Cases Under the Five Treatments for the Example of Figure 1

Treatment	$m1$	$m2$	$m3$	$m4$	$m5$	$m6$
1	$t1$, $t2$, $t3$, $t4$	$t1$, $t2$, $t3$, $t4$	$t1$, $t2$, $t3$, $t4$	$t1$, $t2$, $t3$, $t4$	$t1$, $t2$, $t3$, $t4$	$t1$, $t2$, $t3$, $t4$
2	$t1$, $t2$, $t3$, $t4$	$t1$, $t2$, $t3$, $t4$	$t1$, $t2$, $t3$, $t4$	$t1$, $t2$, $t3$, $t4$	$t1$, $t2$	
3	-	-	$t1$, $t2$, $t3$, $t4$	$t2$,$t4$	-	$t1$, $t2$, $t3$, $t4$
4	-	-	$t1$, $t2$, $t3$, $t4$	$t2$, $t4$, $t1'$	-	$t1$, $t2$, $t3$, $t4$
5	-	-	$t1$, $t2$, $t3$, $t4$	$t2$, $t1'$	-	$t1$, $t2$, $t3$, $t2'$

by $T_{new:P'}$, are used to compose the initial population for the genetic algorithm. Here, we also need to run all existing test cases first to obtain their new coverage information. Adding new test cases brings greater variety to the population, which increases the size of the population but may increase running time.

In Figure 1, the same test cases used in treatment 3 are used to form the initial population for $m3$, since when we do augmentation for $m3$ there have not been test cases generated. We generate a test case $t1'$ for $m3$ to cover the branch that calls $m4$, so when we do augmentation for $m4$ we include $t1'$ with $t2$ and $t4$ to form the initial population for it. For $m6$, $t1'$ does not reach it so we still use only the existing test cases that reach it in its initial population.

Treatment 5.

For each affected element e in method m, all existing and generated test cases generated that reach m in P' ($T_{old:P'} \cup T_{new:P'}$) are considered applicable, but before being utilized they are considered further. A reasonable size of population is determined (in our case we chose the size that would be required by using treatment 3) and initial test cases are selected from the applicable test cases to compose the population. In this case, a good selection technique should be used to choose test cases that form a population which has the best variety for genetic algorithm. In our case, we chose test cases according to their branch coverage information on P'. More precisely, if we need to pick s test cases, we do the following:

- Find all paths from the root of P's call graph to m.
- Put the methods along these paths, including m, into set M_{pre}.
- Find branches in all methods in M_{pre}.
- Order the candidates on these branches in terms of coverage
- Pick the first s of the ordered candidates.

In Figure 1, $m3$ is in the same situation as with treatment 4, so the same test cases are used here. For $m4$, when we perform augmentation for $m3$ we generate thousands of test cases, some that increase coverage such as $t1'$ and others that cover branches covered by other existing test cases. Next, we order all test cases that reach $m3$ and select two that cover most of the branches in $m1$, $m2$, $m3$ and $m4$. We select $t2$ and $t1'$ here, since they both pass through $m1$, cover one branch in $m2$ and $m3'$ separately, and pass through one of the branches in $m4$. The same procedure is followed on $m6$. For example, $t2'$ and $t3'$ are generated and reach $m6$ and including these with all existing test cases we select $t1$, $t2$, $t3$ and $t2'$ to form the initial population for $m6$.

4.3.3 Dependent Variables and Measures

We chose two dependent variables and corresponding measures to address our research questions. The first variable relates to costs associated with employing the different test case treatments and the second relates to the effectiveness associated with the different treatments.

Cost of employing treatments.

To measure the cost of employing treatments, one approach is to measure the execution time of the augmentation algorithm under each treatment. However, measuring time in a manner that facilitates fair comparison requires the use of identical machines, and for the sake of parallelism we ran our experiments on a set of machines and under different system loads.

An alternative approach to cost measurement involves tracking, under each test case treatment, the number of invocations by augmentation techniques of the operations that most directly determine technique cost. For the augmentation technique that we consider the operation that matters most involves the execution of test cases by the genetic algorithm, because if that algorithm finds a target soon it will use fewer iterations, execute fewer test cases and require less running time. Thus, in this study, we use the number of test cases executed by the genetic algorithm as a measure of cost.

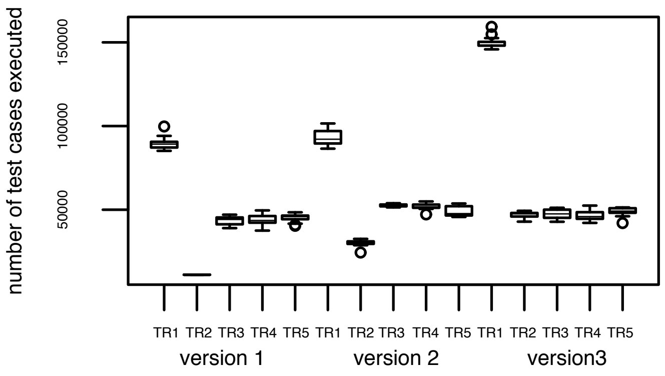

Figure 2: Costs of applying the five treatments, per treatment and version

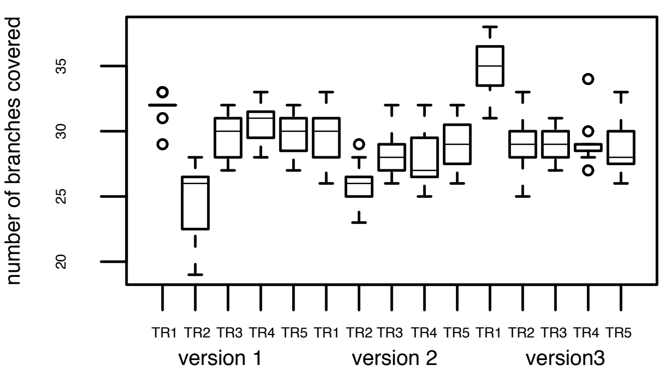

Figure 3: Coverage obtained in applying the five treatments, per treatment and version

Effectiveness of employing treatments.

To assess the effectiveness of using different test case treatments, we measure the progress that augmentation can make toward its coverage goal under each treatment in terms of the numbers of branches covered.

4.4 Experiment Setup

To conduct our study we performed the following steps. For each v_k ($0 \leq k \leq 2$) we instrumented and created the CFG for v_k using Sofya [9]. We then executed v_k on the test suite T_k for v_k, collecting test coverage for use in the next step. Next, we created the CFG for v_{k+1} and determined the affected methods and target coverage elements (branches) in v_{k+1} using the methodology described above. These target elements are the affected elements we attempt to cover with our genetic test case generation algorithm under the different test case treatments. Further, because a genetic algorithm can fare differently on different runs, for each test case treatment we executed the test case generation algorithm fifteen times, and we consider data taken from all of these runs in our subsequent analysis.

4.5 Study Limitations

There are several limitations to our results. The first is the representativeness of our object program, versions, and test suites. We have examined only one system, coded in Java, and other systems may exhibit different cost-benefit tradeoffs. We have considered only three pairs of versions of this subject, and others may exhibit different tradeoffs. A second threat pertains to algorithms; we have utilized only one variant of a genetic test case generation algorithm, hand-tuned, and under particular settings of factors F1, F2, and F3. Subsequent studies are needed to determine the extent to which our results generalize.

Another limitation involves possible faults in the implementation of the algorithms and in tools we use to perform evaluation. We controlled for this through extensive functional testing of our tools.

Finally, there are other metrics that could be pertinent to the effects studied. Given tight implementations and controls over environments, time could be measured. Costs of engineer time in employing methods could also matter.

4.6 Results and Analysis

Figures 2 and 3 present boxplots showing the data gathered for our independent variables. The first figure plots the number of test cases executed (vertical axis) against each treatment (TR1, TR2, TR3, TR4, and TR5) per version (v_1, v_2 and v_3). The second figure plots the number of covered branches against each treatment per version.

Table 2: Results of ANOVA Analysis

	Df	Sum Sq	Mean Sq	F value	Pr
v1_cost	4	4.04e+10	1.01e+10	109.43	<2.2e-16
v2_cost	4	3.18e+10	7.96e+09	1027.30	<2.2e-16
v3_cost	4	6.13e+10	6.13e+10	68.40	<4.4e-12
v1_cov	4	459.15	114.79	36.84	<2.2e-16
v2_cov	4	124.86	31.21	7.83	2.9e-0.5
v3_cov	4	427.20	106.80	35.19	6.6e-16

4.6.1 RQ1: Costs of augmentation

To address RQ1 (cost of the treatments) we compare the number of test cases executed by the treatments. As the boxplots show, in all cases the number of test cases executed by TR1 is substantially greater than the number executed by the other four treatments. On versions v_1 and v_2, but not v_3, TR2 results in the execution of the fewest test cases. TR5 appears to differ slightly from other treatments on v_2 and v_3, but in other cases treatment results appear similar.

We performed per version ANOVAs on the data for a significance level of 0.05; Table 2 reports the results. The first three rows pertain to cost comparisons. As the $p-$values in the rightmost column show, there is enough statistical evidence to reject the null hypothesis on all three versions; that is, the mean costs of the five different treatments are different in each case.

The ANOVA evaluated whether the treatments differ, and a multiple comparison procedure using Bonferroni analysis quantifies how the treatments differ from each other. Table 3(A) presents the results of this analysis for the three versions considering treatment cost, ranking the treatments by mean. Grouping letters (in columns with header "Gr") indicate differences: treatments with the same grouping letter were not significantly different. In v_1 the five treatments are classified into three groups: TR1 and TR2 are most and least costly, respectively, while TR3, TR4 and TR5 are in a single group intermediate in cost. In v_2 the treatments are classified into four groups; TR1 remains most costly and TR2 least costly, but TR3, TR4, and TR5 form two overlapping classes in terms of cost, with TR3 significantly more

Table 3: Results of Bonferroni Means Test on Cost and Coverage

(A) cost									(B) coverage								
v1			v2			v3			v1			v2			v3		
	Mean	Gr		Mean	Gr		Mean	Gr		Mean	Gr		Mean	Gr		Mean	Gr
TR2	11136	A	TR2	29960	A	TR4	46522	A	TR1	31.9	A	TR1	29.4	A	TR1	35.0	A
TR3	43355	B	TR5	49347	B	TR2	46752	A	TR4	30.6	A, B	TR5	28.9	A	TR3	29.1	B
TR4	43914	B	TR4	51811	B, C	TR3	47262	A	TR5	29.9	B	TR3	28.1	A	TR4	29.1	B
TR5	45086	B	TR3	52569	C	TR5	48961	A	TR3	29.7	B	TR4	27.9	A	TR2	29.0	B
TR1	84302	C	TR1	93048	D	TR1	149856	B	TR2	24.7	C	TR2	25.7	B	TR5	28.6	B

costly than TR5. In v_3, TR1 is most costly and other techniques are classified into a single less costly group.

4.6.2 RQ2: Effectiveness of augmentation

Next we explore RQ2, which involves the effectiveness of the five treatments in terms of achieving branch coverage when augmenting test suites. As mentioned above, after running all existing test cases we found that 68 branches needed to be covered for v_1, 77 for v_2 and 100 for v_3. Among these, several branches are difficult to cover in each version, since `Nanoxml` is a parser for XML and often requires specific characters at specific positions which can be difficult to satisfy. Also, in v_2 and v_3, since the test drivers we used are for previous versions and we did not mutate them to trigger some methods in the new version that are important for improving coverage, we were unable to cover 13 and 3 branches, respectively.

The boxplots in Figure 3 show the numbers of branches covered by each treatment in the fifteen runs for the three versions. On the three versions, TR1 covers the most branches. For v_1 and v_2, TR2 covers the fewest branches and TR3, TR4 and TR5 have similar results, while in v_1, TR4 appears better and in v_2 TR5 appears better. For v_3, the other four treatments return similar results.

Table 2 displays the results of ANOVAs on coverage data for the versions (bottom three rows). The $p-$values indicate that the five treatments do have significant effects on coverage for all three versions.

Table 3(B) presents the results of the Bonferroni comparison. The results differ somewhat across versions. In all versions, TR1 is among the most effective treatments, though it shares this with TR4 on v_1 and with all but TR2 on v_2. Similarly, TR2 is always among the least effective treatments, though sharing this with others on v_3. TR3, TR4, and TR5 are always classified together.

5. DISCUSSION

Our results show that, for the object program and versions considered, TR1 consistently requires significantly more time to execute but also achieves the best coverage (in terms of means, with significance on one version) than the other treatments. TR2 is also significantly less costly and effective on two of the three versions than other treatments, and on the third version is in the equivalence class of least costly and least effective treatments. The other three treatments behave somewhat differently across versions and we now explore reasons for some of the observed behaviors.

Across all versions, TR4 works comparatively well in terms of cost and coverage according to Table 3(A) and Table 3(B). It uses a smaller population than TR1, and this allows it to save time. Compared to TR3, it has more test cases which does bring greater diversity into its population, since these test cases improve coverage and help the genetic algorithm find targets sooner. However, it is no more costly than TR3, and this is arguably due to the presence of many unreachable branches. When the genetic algorithm tries to cover a branch there are two stopping criteria: either finding a test case to cover the target or reaching the maximum number of iterations without covering the target. For these unreachable branches TR4 may have a larger population than TR3; however, since the branches are unreachable the additional test cases are not useful but require time to run. Therefore the time consumed counteracts the time that is saved by covering other branches sooner.

The data shows that on v_3, all five treatments improve coverage by only 30%, which leaves a lot of branches uncovered. We checked all the uncovered branches. Other than ten determinably unreachable branches, many of the uncovered branches are new in v_3 and no existing test cases reach them. We believe that this relates to factor $F2$, the characteristics of existing test suites. The existing test suite for v_3 covers a relatively small portion of v_3's code, and thus greater effort is required to augment the test suite for that version. At the same time, this relatively poor test suite offers little diversity in terms of coverage of v_3, and this restricts the genetic algorithm's performance. We believe this is the reason that all treatments achieve lower coverage on v_3 than on the other versions.

The foregoing can also can explain why TR2 behaves similar to treatments TR3, TR4, and TR5 on v_3. After updating all the existing test cases' coverage on v_3, many new methods in that version are still unreachable using the existing test cases. In this situation, TR3 is similar to TR2. Since we do not generate many new test cases, when we use TR4, the few new test cases do not add much diversity.

In v_3, TR1 is most effective but is three times more expensive than other techniques, while on the other two versions TR1 is less than two times more expensive than other techniques. We believe this is because the relatively poor starting test suite leaves many affected methods unreachable. In TR1 for all targets in these methods, we use all existing test cases as the base for the genetic algorithm. Since they never reach these methods, our fitness function treats them all equally (the fitness function measures their performance in the method only). This leaves nothing to guide the evolution. For these branches, TR1 just iterates until it reaches the maximum numbers as explained above, which potentially increases its cost. To solve this problem, in addition to a better starting test suite, we may need to find a fitness function that works interprocedurally.

Treatment TR5 did not work as expected. We had conjectured that it would have strengths common to both TR3 and TR4, namely, greater diversity in initial population and smaller size. However, its cost and effectiveness are not significantly different than those of TR3 and TR4. We may require a better technique for selecting test cases to compose the initial population for the genetic algorithm. For example, genetic algorithms require diversity in the chromosome itself, containing all elements required in the application, instead of simply considering its coverage on the code.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we described four factors that we believe can influence the cost-effectiveness of test suite augmentation using genetic algorithms, providing reasons for this belief. We presented the results of a case study exploring one of those factors, involving the treatment of existing and newly generated test cases, that we believe could be particularly significant in its effects. Our results show that different treatments of test cases can affect the augmentation task when applying a genetic algorithm for test case generation during augmentation.

At present, the primary implications of this work are for researchers. Our results indicate that when attempting to integrate genetic test case generation algorithms into the test suite augmentation task, it is important to consider the treatment of existing and newly generated test cases, and it may also be important to consider the other factors that we have presented. Furthermore, when *empirically studying* such techniques, to facilitate repeatability and understanding of results, researchers need to explicitly identify the settings chosen for the various factors.

In future work, we intend ourselves to study the factors we have identified more closely, both by extending the set of objects we study, and by varying different factors and studying their influence on augmentation tasks. We also wish to improve our fitness function by the including branch distance, and to study the use of other forms of test case generation algorithms, and compare them to genetic algorithms in the augmentation context. Through such study we hope to influence not just research on, but also the practice of test suite augmentation.

Acknowledgments

This work was supported in part by the National Science Foundation under Awards CNS-0454203 and CCF-0747009, by the Air Force Office of Scientific Research through award FA9550-09-1-0129, and by the Defense Advanced Research Projects Agency.

7. REFERENCES

[1] T. Apiwattanapong, R. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold. Matrix: Maintenance-oriented testing requirements identifier and examiner. In *Test.: Acad. Ind. Conf. Pract. Res. Techn.*, pages 137–146, Aug. 2006.

[2] A. Arcuri. It does matter how you normalise the branch distance in search based software testing. In *Intl. Conf. on Soft. Test., Verif. and Valid.*, 2010.

[3] D. Binkley. Semantics guided regression test cost reduction. *IEEE Trans. Softw. Eng.*, 23(8), Aug. 1997.

[4] S. Bohner and R. Arnold. *Software Change Impact Analysis.* IEEE Computer Society Press, Los Alamitos, CA, 1996.

[5] E. Díaz, J. Tuya, R. Blanco, and J. Javier Dolado. A tabu search algorithm for structural software testing. *Comp. Op. Res.*, 35(10):3052–3072, 2008.

[6] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Emp. Softw. Eng.: Int'l J.*, 10(4):405–435, 2005.

[7] R. Gupta, M. Harrold, and M. Soffa. Program slicing-based regression testing techniques. *J. Softw. Test., Verif., Rel.*, 6(2):83–111, June 1996.

[8] B. Jones, H. Sthamer, and D. Eyres. Automatic structural testing using genetic algorithms. *Softw. Eng. J.*, 11(5), 1996.

[9] A. Kinneer, M. Dwyer, and G. Rothermel. Sofya: A flexible framework for development of dynamic program analyses for Java software. Technical report, U. Neb. Lincoln, 2006.

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

[11] P. McMinn. Search-based software test data generation: a survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, 2004.

[12] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Softw. Test., Verif. Rel.*, 9:263–282, Sept. 1999.

[13] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu. Differential symbolic execution. In *Int'l. Symp. Found. Softw. Eng.*, pages 226–237, Nov. 2008.

[14] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *Int'l Symp. Softw. Test. Anal.*, Aug 1994.

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

[16] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Auto. Softw. Eng.*, Sept. 2008.

[17] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Int'l Symp. Found. Softw. Eng.*, pages 263–272, Sept. 2005.

[18] P. Tonella. Evolutionary testing of classes. In *Intl. Symp. Softw. Test. Anal.*, pages 119–128, 2004.

[19] H. Waeselynck, P. Thévenod-Fosse, and O. Abdellatif-Kaddour. Simulated annealing applied to test generation: Landscape characterization and stopping criteria. *Emp. Softw. Eng.*, 12(1):35–63, 2007.

[20] S. Wappler and F. Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *Conf. Gen. Evol. Comp.*, pages 1053–1060, 2005.

[21] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *IST*, 43(14):841 – 854, 2001.

[22] Z. Xu and G. Rothermel. Directed test suite augmentation. In *Asia-Pac. Softw. Eng. Conf.*, Dec. 2009.

[23] S. Yoo and M. Harman. Test data augmentation: Generating new test data from existing test data. Technical Report TR-08-04, Dept. of Computer Science, King's College London, July 2008.