Empirically Studying the Role of Selection Operators During Search-Based Test Suite Prioritization

Alexander P. Conrad
Department of Computer Science
University of Pittsburgh
conrada@cs.pitt.edu

Robert S. Roos
Gregory M. Kapfhammer
Department of Computer Science
Allegheny College
{rroos,gkapfham}@allegheny.edu

ABSTRACT

Regression test suite prioritization techniques reorder test cases so that, on average, more faults will be revealed earlier in the test suite's execution than would otherwise be possible. This paper presents a genetic algorithm-based test prioritization method that employs a wide variety of mutation, crossover, selection, and transformation operators to reorder a test suite. Leveraging statistical analysis techniques, such as tree model construction through binary recursive partitioning and kernel density estimation, the paper's empirical results highlight the unique role that the selection operators play in identifying an effective ordering of a test suite. The study also reveals that, while truncation selection consistently outperformed the tournament and roulette operators in terms of test suite effectiveness, increasing selection pressure consistently produces the best results within each class of operator. After further explicating the relationship between selection intensity, termination condition, fitness landscape, and the quality of the resulting test suite, this paper demonstrates that the genetic algorithm-based prioritizer is superior to random search and hill climbing and thus suitable for many regression testing environments.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; I.2.8 [**Computing Methodologies**]: Problem Solving, Control Methods, and Search

General Terms

Experimentation, Algorithms, Verification

Keywords

test prioritization, coverage testing, genetic algorithm

1. INTRODUCTION

Since developers inevitably introduce errors while implementing software systems, they often use software testing to detect and isolate these defects. As the source code grows in size, test cases are written for the new functionality. However, these new tests do not obviate the old ones. In an attempt to ensure both the correctness of new code and its proper integration into the system, every existing test case is executed in a test suite $T = \langle t_1, t_2, t_3, \ldots, t_n \rangle$ [27]. Yet, high test suite execution times often make regression testing challenging for large commercial applications, with a single run frequently spanning days or weeks [7]. Prioritization may accelerate the fault detection rate of a test suite, allowing engineers to detect and begin correcting faults sooner.

Following Harman and Clark, we note that there are many different metrics, such as the average percentage of faults detected [7] and coverage effectiveness [25], for evaluating the quality of a test suite ordering [9]. Yet, because even small test suites have a substantial number of different orderings, it is too expensive to compute the effectiveness of every potential test ordering produced by a brute force search. For instance, a test suite of only 15 test cases has $15! = 1,307,674,368,000$ different orderings. These characteristics – a very large solution space, no efficient algorithm for constructing an optimal solution, a suitable fitness function, and many potential solutions – mark test suite prioritization as a prime candidate for the application of a search-based method like the genetic algorithm [4].

As an extension of [14], this paper implements and empirically evaluates a genetic algorithm (GA)-based prioritizer that features six mutation operators, seven approaches to crossover, and three methods for performing selection [12]. Using the "higher is better" coverage effectiveness (CE) metric [25] as a fitness function, the prioritization method incorporates both the execution time and requirement coverage of each test in order to evolve a test suite that rapidly covers the test requirements. Since high coverage tests are often correlated to high fault detection [8, 22], a genetic algorithm that uses the CE metric has the potential to identify test suite orderings that find faults faster.

While the empirical results ultimately demonstrate that the genetic algorithm finds test suites with high CE scores, the GA-based approach has many parameters, thus making it more challenging for testers to determine the best configuration of this technique for their own applications. To mitigate this concern, this paper shows how to use automatically generated tree models [2] to discover the best configuration of genetic algorithm-based approaches to reordering a test suite. In particular, we use the binary recursive partitioning algorithm [2, 5] to construct tree models that reveal how the explanatory variables (e.g., the selection operator) im-

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

pact the CE value of the resulting prioritization. The use of these explanatory trees, in conjunction with a kernel density estimator that visualizes the shape of the distribution for a method's CE scores, reveals the important role of the selection operator in evolving high quality test orderings. Since the empirical results also show that the GA-based prioritizer is better than both random search and hill climbing, this paper establishes the genetic algorithm as an appropriate choice for regression testing environments in which testers must repeatedly run high quality test orderings.

The important contributions of this paper include:

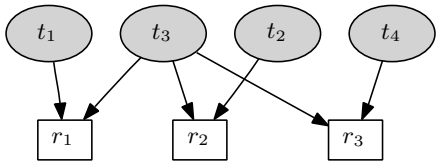
1. The description of a comprehensive genetic algorithm-based test suite prioritization framework that uses the coverage effectiveness score for fitness and includes numerous operators for mutation, crossover, and selection (Sections 2 through 4).
2. An empirical study that leverages established statistical analysis techniques to explicate the role of the selection operator during search-based test prioritization and demonstrate the benefits of using a GA over random search and hill climbing (Sections 5 and 6).
3. Suggestions for future work that may yield improvements to the efficiency and effectiveness of search-based test suite prioritizers (Section 7).

In addition, the prioritization technique described in this paper, called *GELATIONS* (*GE*netic a*L*gorithm b*A*sed *T*est su*I*te pri*O*ritizatio*N* *S*ystem), is completely implemented as free and open source software and is available at `http://gelations.googlecode.com/`. We released our framework, along with all of the data sets needed to reproduce our empirical results, in order to enhance the reproducibility of our experiments and to encourage and facilitate future research into search-based test suite prioritization.

2. MOTIVATION

Suppose that a test suite $T = \langle t_1, t_2, t_3, t_4 \rangle$ covers a set of three test requirements $\mathcal{R}(T) = \{r_1, r_2, r_3\}$. Each requirement represents partial fulfillment of a test adequacy criterion, such as the traversal of paths in a calling context tree [18] or edges in a control flow graph [32], and a test is said to cover a requirement when it satisfies that requirement's criterion. Figure 1 illustrates coverage relationships (e.g., t_1 covers r_1), and gives the execution times for each test case. In this example, $time(t_i)$ is the execution time of test t_i and $\text{CE}(T)$ is the coverage effectiveness of test suite T, which is 0.54 before prioritization (see Section 3 for more details about the computation of the CE score).

First, consider prioritizing T with a greedy algorithm that uses the ratio of coverage to execution time as a greedy choice metric [25]. Since t_3 has the highest ratio value, the greedy algorithm will produce a prioritized test suite $T' = \langle t_3, t_1, t_2, t_4 \rangle$ with a slightly improved coverage effectiveness score of 0.55. Next, suppose that we prioritize T using the *GELATIONS* framework that employs a search-based approach. Since the genetic algorithm performs global optimization, it is likely to overcome local optima that the greedy algorithm becomes stuck on due to "greedy fooling" patterns in coverage data. As shown in Figure 1, the GA presented in Section 4 produces a prioritized test suite $T'' = \langle t_1, t_2, t_4, t_3 \rangle$ that features a $\text{CE}(T'')$ score of 0.63, a value that is higher than the one for both the original ordering and the greedily re-ordered test suite.



$time(t_1) = time(t_2) = time(t_4) = 1$
$time(t_3) = 2.45$

$\text{CE}(T = \langle t_1, t_2, t_3, t_4 \rangle) = 0.54$
$\text{CE}(T' = \langle t_3, t_1, t_2, t_4 \rangle) = 0.55$
$\text{CE}(T'' = \langle t_1, t_2, t_4, t_3 \rangle) = 0.63$

Figure 1: A "Greedy Fooling" Test Suite.

Other than avoiding certain kinds of suboptimal solutions, the GA has three additional advantages over greedy techniques. Previous theoretical and empirical studies have shown that genetic algorithms are often amenable to parallelization [3, 33]. Given the increasing use of multi-core CPUs and graphics processing units (GPUs) for general computation, parallelization has the potential to effectively reduce the cost of GA-based methods. GAs can also be interrupted during their execution, thus enabling the identification of the test ordering that is currently the best and the use of a "human in the loop" prioritization model where a person effectively guides the search algorithm [29].

A third advantage of the GA-based method concerns the degree to which it can construct diverse test orderings that achieve equivalent coverage effectiveness scores [30]. If the test coverage report and the execution time of the tests does not change, then multiple prioritizations of a given test suite produced by a greedy algorithm will always be identical. In contrast, the genetic algorithm is likely to yield different orderings. It is more desirable to use different orderings of tests to cover the same requirements than it is to repeatedly use an unchanged test ordering. This activity ensures that latent properties of the tests that are not reflected in the requirements will be brought to bear on the application, possibly increasing the test suite's capability to find faults not connected to the adequacy criterion. The GA is ideally suited for this task because it can produce different test orderings that have similar CE scores.

3. EVALUATING TEST SUITES

A test coverage monitor allows software testers to characterize the behavior of a program while the test suite executes. Given a test suite $T = \langle t_1, t_2, t_3, \ldots, t_n \rangle$, a test coverage monitor identifies a set of covered requirements $\mathcal{R}(T) = \{r_1, r_2, r_3, \ldots, r_m\}$. Each test case t_i is associated with a non-empty subset of requirements $\mathcal{R}(t_i) \subseteq \mathcal{R}(T)$ that t_i is said to cover [11, 25]. The coverage effectiveness (CE) metric evaluates a prioritized test suite by determining the cumulative coverage of the tests over time [25].

As defined in Equation (1) and depicted in Figure 2, the cumulative coverage function $C(T, l)$ takes as input a test suite T and a time l and returns the total number of requirements covered by T after running for l time units. To guarantee $\text{CE} \in (0, 1)$, the integral of $C(T, l)$ is divided by the integral of the ideal cumulative coverage function $\bar{C}(T, l)$ that Equation (2) defines as immediate coverage of all the requirements. Equation (3) shows that these integrals are

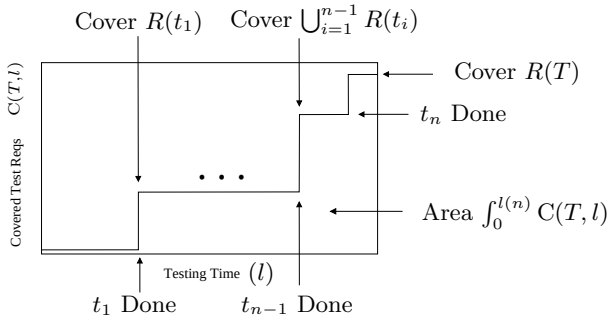


Figure 2: Coverage Effectiveness.

taken within the closed interval 0 to $l(n)$ where $l(i)$ is the time required to execute tests $t_1, t_2, ..., t_i$ and there are a total of n test cases in the entire suite.

Due to the fact that the prioritized test suite T' still covers all of the requirements exercised by the original suite T (i.e., $\mathcal{R}(T') = \mathcal{R}(T)$), CE will never take on the value of zero. Furthermore, the CE value for an ordering will not be equal to one because Equations (1) through (3) stipulate that the first group of requirements are not covered until the first test finishes execution. In fact, the CE metric conservatively credits each test with the coverage of its requirements when it finishes execution since many test coverage monitoring tools do not record the point in time when a test case covers a requirement [11, 21, 28]. While CE may be unfair to high coverage tests with extended running times, the metric does furnish a time sensitive measurement of effectiveness. In contrast, prior metrics such as APFD [7], APBC, APDC, and APRC [14], do not factor time into the evaluation of a prioritization technique. Thus, our work extends [14] by incorporating test case time into the coverage metric. Unlike existing evaluation metrics that do incorporate time (e.g., APFD_c [16] and NAPFD [23]), CE does not require the use of fault information when calculating an effectiveness rating.

$$C(T,l) = \begin{cases} 0 & l < l(1) \\ |\mathcal{R}(t_1)| & l \in [l(1), l(2)) \\ \vdots & \vdots \\ |\bigcup_{i=1}^{n-1} \mathcal{R}(t_i)| & l \in [l(n-1), l(n)) \\ |\mathcal{R}(T)| & l \geq l(n) \end{cases} \quad (1)$$

$$\bar{C}(T,l) = |\mathcal{R}(T)| \quad (2)$$

$$\mathrm{CE}(T) = \frac{\int_0^{l(n)} C(T,l)}{\int_0^{l(n)} \bar{C}(T,l)} \quad (3)$$

Figure 4 provides an empirical cumulative distribution function (ECDF) that shows the range of coverage effectiveness scores for the example test suite from Figure 3. The ECDF curve represents the probability that the coverage effectiveness value is less than or equal to a specific value on the horizontal axis. After enumerating and evaluating each of the $4! = 24$ different test orderings, we find that the initial test suite, $\langle t_1, t_2, t_3, t_4 \rangle$ has the lowest CE value of 0.343. In contrast, the test suite $\langle t_2, t_4, t_3, t_1 \rangle$ has the highest CE score of 0.743. Figure 4 also reveals that 60% of the test suite orders have a CE value that is less than or equal to 0.52, further demonstrating that the value of the coverage effectiveness metric may vary considerably and thus motivating the need for test prioritization methods.

	r_1	r_2	r_3	r_4	r_5	Execution Time
t_1	✓	✓	✓	✓		4
t_2			✓	✓		1
t_3		✓				1
t_4	✓				✓	1

Figure 3: Coverage Effectiveness Example.

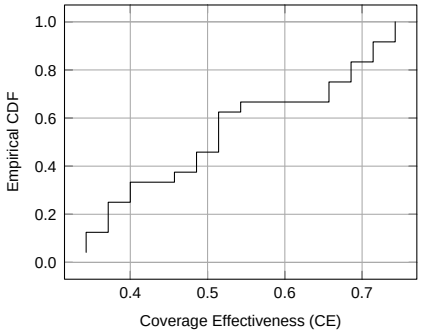


Figure 4: CE Values for the Example Test Suite.

4. TEST SUITE PRIORITIZATION

The input to the prioritizer includes information concerning the requirement coverage and execution time of each test case in the suite undergoing prioritization, as seen in Figure 3. During initialization of the genetic algorithm, an initial set of orderings, or population, is randomly generated. Four main phases characterize the primary loop of the genetic algorithm. First, the GA calculates the fitness value of each individual in the population. While we use the coverage effectiveness score from Section 3 for this paper's empirical study, *GELATIONS* supports the integration of other fitness functions such as the average percentage of faults detected [7]. Using memoization, the algorithm stores the fitness values of previously evaluated orderings in a hash table, so that they will not need to be evaluated again.

After computing CE scores, a selection operator chooses individual orderings from the population to become the parents of the next generation. Next, a crossover operator combines pairs of parent orderings into new child orderings. As an example, partially-mapped crossover picks sub-tuples within two parents, constructs a mapping between the tests in these sub-tuples, and creates children by iteratively swapping tests according to the mapping [12]. Finally, a fraction of the new child orderings are mutated, or altered in some random way. For instance, the insertion mutation operator randomly moves one of the test cases to a different location in the ordering [12]. The algorithm continues until it reaches the termination condition, which is the stagnancy of the population's fitness. Intuitively, after a number of generations have passed in which no new most fit individual emerges, the algorithm exits and returns the best test suite.

As noted in Figure 5, the GA-based prioritizer uses three different types of selection operators. The first is roulette selection [31], a fitness-proportionate selection strategy which assigns a selection probability to each ordering based on its coverage effectiveness score. In particular, the framework contains a regular roulette selection operator (ROU), an exponential transformation operator in which the square root is taken of each fitness value (ROUE), and a linear ranking

Operator Type	Abbreviation	Full Name
Mutation [12]	DM	Displacement mutation
	EM	Exchange
	ISM	Insertion
	IVM	Inversion
	SIM	Simple inversion
	SM	Scramble
Crossover [12]	CX	Cycle
	MPX	Maximal preservative
	OX1	Order
	OX2	Order-based
	PMX	Partially-mapped
	POS	Position-based
	VR	Voting recombination
Selection	ROU	Roulette [31]
	TRU	Truncation [15]
	TOU	Tournament [20, 31]
Transformation (Roulette only)	ROUE	Exponential [13]
	ROUL	Linear ranking [13]

Figure 5: Genetic Algorithm Operators.

Parameter Type	Values
Mutation Rate	0.10, 0.33, 0.67
Density of New Children	0.5, .75, 1.0
Population Size	75, 150, 225
Maximum Stagnancy	20, 30, 40

Figure 6: Genetic Algorithm Parameters.

operator in which each fitness value is assigned based on the individual's rank within the population (ROUL) [13]. The second type of operator, truncation selection [15], picks a specified fraction of the most fit orderings in a population. The operator duplicates this subset of the population until the desired number of parent orderings is achieved. We truncate the population at 40% (TRU40), 50% (TRU50), and 60% (TRU60). Our operator sorted each population by fitness in order to perform the truncation. The third, called tournament selection [20], picks a specified number of randomly chosen orderings from the population, creating a tournament. From the tournament, the ordering with the highest fitness value is chosen as a parent. The operator conducts tournaments until it reaches the desired number of parent orderings. Currently, the tournament selection operator handles tournament sizes of two (TOU2), three (TOU3), four (TOU4), and five (TOU5) individuals.

Figure 5 reviews all of the mutation, crossover, selection, and transformation operators implemented in *GELATIONS*. In addition to a choice for each of these operators, the genetic algorithm accepts a value for several parameters, as given in Figure 6. In this case, the mutation rate specifies the average percentage of each group of new children that should be subject to the mutation operator while the density of new children gives the percentage of a population to replace with the offspring from crossover. Finally, population size stands for the number of individuals in each population and maximum stagnancy sets the number of generations over which the genetic algorithm may fail to produce a new test suite prioritization with a higher fitness than those already in the population. Even though *GELATIONS* supports parameter values different than those in Figure 6, we use these values for this paper's empirical study since preliminary experiments demonstrated their capability to construct a test suite with a high coverage effectiveness score.

				Suite CE Scores	
Name	NCSS	$\|T\|$	$\|\mathcal{R}(T)\|$	Initial	Reverse
DS	1243	66	40	0.8169	0.6093
GB	1455	55	88	0.4389	0.7810
JD	2716	57	783	0.6686	0.6911
LF	215	14	6	0.9903	0.9903
RM	569	16	19	0.2612	0.9215
RP	6822	79	221	0.6957	0.9154
SK	628	28	117	0.8847	0.7520
TM	748	27	46	0.8464	0.8002

Figure 7: Case Study Applications.

5. DESIGN OF THE EMPIRICAL STUDY

Case Study Applications. Our experiment utilized coverage and timing data gathered from eight case study applications and their JUnit test suites. Figure 7 gives information about each application, in terms of non-commented source statements (NCSS), test suite size ($|T|$), and the number of requirements in the coverage report ($|\mathcal{R}(T)|$). We also furnish the CE scores of the initial and reverse ordering of the test suite since these values can serve as a useful baseline. Even though *GELATIONS* supports a coverage report for a wide variety of adequacy criteria, this paper's empirical study uses the coverage of the paths in a calling context tree (CCT) [18], as detected by instrumentation probes inserted into the applications prior to testing [11]. Although a CCT-based adequacy criterion may be criticized for not incorporating the (i) source code or parameters of the methods under test and (ii) state of the program, it has been shown to perform closely to other test adequacy criterion with respect to common fault detection metrics [17]. In fact, in a recent empirical study, test suites that had been reduced using call trees as a coverage metric were 97-100% likely to detect each known fault in the evaluated program [19].

Analysis Techniques. This paper uses automatically generated tree models, such as those in Figure 10, to describe the trends in the results. In particular, we use the binary recursive partitioning algorithm [2, 5] to determine how the explanatory variables (e.g., the choice of a mutation operator) impact the response variable (e.g., a metric such as CE). We selected this type of hierarchical model because it furnishes a simple and easy to understand view of the interactions between the explanatory and response variables [2, 5]. Moreover, tree-based methods are non-parametric in nature, thus enabling the study of search-based prioritizers without making assumptions concerning the relationship between the explanatory and response variables [26]. Tree models are also easy to interpret even if the underlying data set contains a mixture of numerical (e.g., mutation rate) and categorical (e.g., selection operator) variables.

The root of a tree corresponds to the most important explanatory variable for the given data set. By following a path from the root to a leaf node, it is possible to determine the mean value for the specified subset of the data. In the split points of the trees (e.g., "selection_method: TOU3, TOU4, TOU5" in the first tree model of Figure 10), the word before the colon is a categorical explanatory variable and the word(s) to the right are the variable levels associated with the left sub-tree. Moreover, the right sub-tree always corresponds to the remaining variable levels (i.e., the ones not included in the description of the split). Similarly, a tree model can contain a split point for a numerical variable (e.g., "child_density < 0.875"), which abides by the same interpretation as the previously described categorical

Random Number	Input	Output
$\{1,2,3,4\}$ 2	$\langle t_1, t_2, t_3, \vert t_4\rangle$	$\langle t_1, t_4, t_3, t_2\rangle$
$\{1,2,3\}$ 3	$\langle t_1, t_4, \vert t_3, t_2\rangle$	$\langle t_1, t_4, t_3, t_2\rangle$
$\{1,2\}$ 1	$\langle t_1, \vert t_4, t_3, t_2\rangle$	$\langle t_4, t_1, t_3, t_2\rangle$

$T' = \langle t_4, t_1, t_3, t_2\rangle$

Figure 8: Random Test Suite Prioritization.

variables. In an effort to simplify the notation and concepts, the remainder of the discussion about the construction of explanatory tree models focuses on categorical variables since similar methods handle the numerical ones.

Our implementation of the binary partitioning algorithm aims to discover the manner in which each of the explanatory variables, denoted E, and its corresponding levels, written as *levels*(E), impacts a response variable R. If E stands for the mutation operator with *levels*$(E) = \{\text{DM}, \text{EM}, \ldots, \text{SM}\}$ and R is the coverage effectiveness score, then the partitioning procedure must find the level of E that best explains the variations in R. During each iteration, the algorithm examines every variable E and all of E's levels in order to further "grow" the tree model by picking the variable and level combination that leads to the greatest decrease in the deviance $D(\tau)$ for tree model τ.

Since a leaf in τ corresponds to the arithmetic mean of the values for R in a given configuration of the explanatory variables, Equation (4)'s definition of deviance focuses on each leaf λ in the set of τ's leaves, denoted $\Lambda(\tau)$. Given a leaf λ and the set of response variable values associated with λ, designated R_λ, Equation (4) sums the square of the difference between $\rho \in R_\lambda$ and $\mu(R_\lambda)$, a function defined by Equation (5) as the arithmetic mean of the values in R_λ. The iterative process of constructing the tree by choosing the best explanatory variable and level continues until either the algorithm obtains no further reduction in $D(\tau)$ or the size of each R_λ is too small to warrant further division (we used $|R_\lambda| < 10$ as the cut-off for the trees in this paper [5]).

$$D(\tau) = \sum_{\lambda \in \Lambda(\tau)} \sum_{\rho \in R_\lambda} (\rho - \mu(R_\lambda))^2 \tag{4}$$

$$\mu(R_\lambda) = \frac{\sum_{\rho \in R_\lambda} \rho}{|R_\lambda|} \tag{5}$$

As shown in Figures 12 through 14, this paper uses a beanplot to support the visual comparison of the distribution of CE scores for various techniques. The beanplot uses a Gaussian kernel density estimator to show the shape of the distribution for the CE values of test suites resulting from a specific prioritizer [10]. In particular, the beanplot employs the non-parametric kernel density estimator to create a smooth curve where a wide area indicates a concentration of values, such as a CE score, and narrow regions suggest the relative dearth of points. The thick black line in each bean represents the arithmetic mean of CE values for a given technique, with the dashed line across the entire plot standing for the mean across all individual beans.

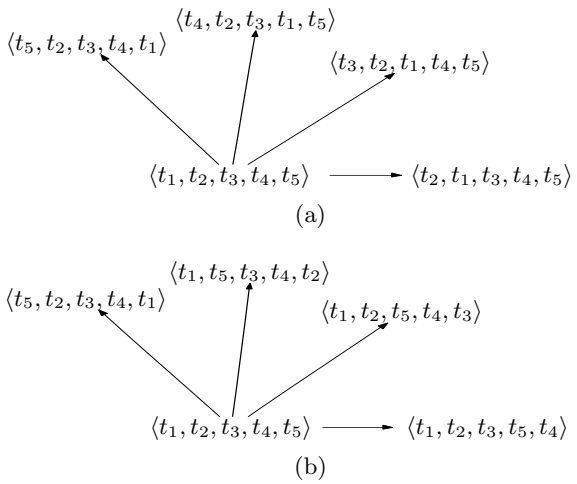


Figure 9: Hill Climbing Neighborhood Generation.

Control Methods. To serve as a form of experimental control for GA-based prioritization, we developed a method for randomly reordering a test suite [7]. Following the modern implementation of the Fisher-Yates shuffle [6], this prioritizer iterates through the test suite in reverse order, swapping the current test case with a randomly chosen test. As shown in Figure 8, when the random prioritizer shuffles $\langle t_1, t_2, t_3, t_4\rangle$, it initially focuses on t_4 and randomly picks test index 2 from the index set $\{1, 2, 3, 4\}$, resulting in the output $\langle t_1, t_4, t_3, t_2\rangle$ that serves as the input to the next round. After completing three iterations, the algorithm constructs the reordered test suite $T' = \langle t_4, t_1, t_3, t_2\rangle$. For the purpose of the empirical study in this paper, the random prioritizer constructed 50, 500, and 5000 different test suite orders for each of the eight case study applications that are listed in Figure 7. While it is possible for the random prioritizer to generate more than 5000 randomly shuffled test suites, the results in Section 6 suggest that the mean coverage effectiveness value does not vary much for the different number of randomly generated orders.

As a final experimental control, we implemented a hill climbing (HC)-based prioritizer that uses two different neighborhood generators [14]. After accepting an initial ordering of a test suite T, the hill climber enumerates all of T's neighbors using either the swap-first or swap-last neighborhood generators. Figure 9(a) shows that the swap-first approach takes the test suite $T = \langle t_1, t_2, t_3, t_4, t_5\rangle$ as input and creates a neighborhood of four orderings produced by swapping t_1 with the other available test cases. As Figure 9(b) reveals, the swap-last method proceeds in a similar fashion, except for the fact that the generator swaps the last test, t_5, with each of the other tests. The steepest ascent (SA) hill climber evaluates the coverage effectiveness score of each test suite in the neighborhood and picks the one yielding the largest improvement in effectiveness. Alternatively, the first ascent (FA) approach to hill climbing immediately chooses a test ordering that offers an improvement in CE over the current test suite. The HC-based prioritizer iteratively continues this process of neighborhood generation, evaluation, and the choice of a new ordering until none of the test suites offer a better CE score, thus requiring the algorithm to return the current suite as the final prioritization.

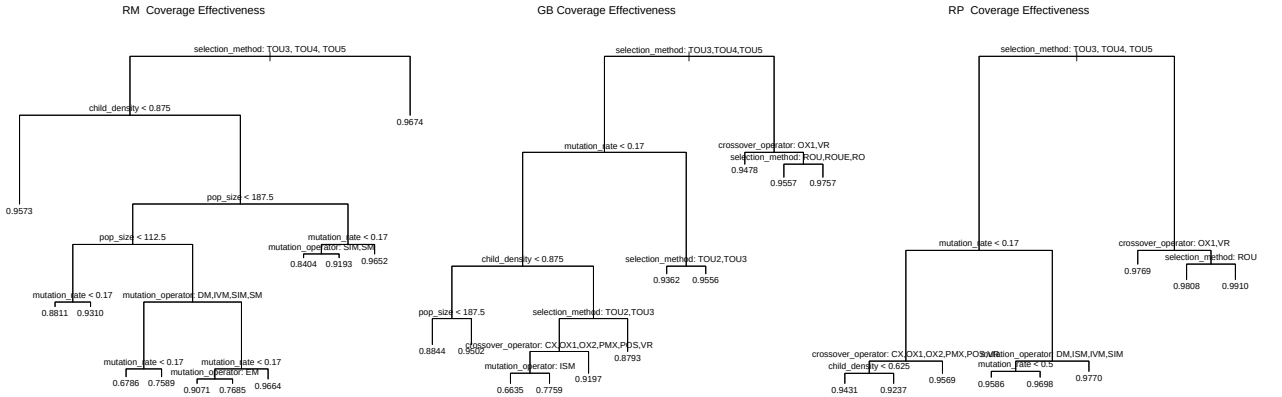


Figure 10: Tree Models Highlighting the Factors that Influence the CE Values for RM, GB, and RP.

Threats to Validity. A number of threats to the validity of this study should be mentioned. Software defects in our prioritizer are an ever-present internal validity threat. To control this concern, we implemented a suite of JUnit test cases to evaluate the core classes of the *GELATIONS* framework. Secondly, since this paper does not specifically investigate the role of crossover operators, we limited the study of this operator's configurations. For instance, some crossover operators always selected between 25% to 50% of the test suite from one of the parents. We picked this baseline configuration in order to ensure a roughly equal contribution of genetic material from each parent. Although we do not think that our implementation of the crossover operators compromised our empirical results, future studies could configure the crossover operators differently.

The relatively small sizes of the case study applications is an external threat to the conclusions of this study. The largest application, RP, consisted of 6822 non-commented source statements, which is rather small compared to other commercial and open-source software applications. We intend to incorporate larger case study applications in our future empirical studies of search-based prioritizers. A final limitation of this study concerns the number of requirements produced by the test adequacy criterion. The greatest number of requirements for any single application was 783 for JD. Additional experiments could use coverage data based on different test adequacy criteria, such as branch or definition-use coverage, that may yield more requirements.

6. EXPERIMENTAL RESULTS

Figure 10 highlights the factors that influence the coverage effectiveness score of the test suite produced by various configurations of the GA-based prioritizer. Since "selection_method" is at the root of each tree, these models reveal that the most important configuration parameter (i.e., the one that leads to the greatest drop in deviance) is the choice of the selection operator. Furthermore, certain operators, such as truncation and roulette selection, tend to enable the genetic algorithm to construct better test suites. For instance, the left sub-trees of the RM model have CE scores ranging between 0.7685 and 0.9664 while the right sub-tree representing the different ROU and TRU operators has a superior average CE score of 0.9674. With the exception of the smallest application, LF, for which all prioritizer configurations find effective test suites, this trend also holds for the trees associated with the other case study applications. When examined in conjunction with the CE scores from Figure 7, the tree models also demonstrate that the search-based prioritizers produce test suites that are normally better than both the initial and reverse orderings.

Name	ROUE	ROUL	TRU60	TRU40	TOU2	TOU5
DS	0.9742	0.9837	0.9893	**0.9915**	0.9514	0.9706
GB	0.9500	0.9572	0.9668	**0.9700**	0.9062	0.9402
JD	0.9247	0.9328	0.9431	**0.9451**	0.8993	0.9192
LF	0.9903	0.9903	0.9903	**0.9903**	0.9903	0.9903
RM	0.9665	0.9670	0.9681	**0.9682**	0.9328	0.9475
RP	0.9774	0.9824	0.9868	**0.9879**	0.9570	0.9705
SK	0.9859	0.9878	0.9911	**0.9915**	0.9667	0.9763
TM	0.9585	0.9605	0.9662	**0.9672**	0.9503	0.9579
Avg.	0.9659	0.9702	0.9752	**0.9765**	0.9443	0.9591

Figure 11: CE Scores Across Selection Operators.

Blickle and Thiele define selection intensity as the change in the average fitness of a population due to selection [1]. According to our results, within each type of operator an increase in selection intensity corresponds to an improvement in the quality of the solution. For instance, increasing the tournament size or decreasing the percentage of the population chosen by truncation selection and applying the linear ranking transformation for roulette selection each theoretically increase the selection intensity for the appropriate type of operator [1]. Ranking tournament selection operators by average empirical fitness of the solution or by theoretical selection intensity produces the same ordering of the operators. This phenomenon also holds for truncation and roulette selection, as illustrated by the average CE scores presented in Figure 11 for a subset of the operators.

However, this trend does not extend across operator types. While TOU4 and TOU5 have greater selection intensities than any of the truncation operators, truncation consistently outperforms tournament in terms of solution quality, as shown by the bold average CE scores in Figure 11. Due to the fact that our roulette selection operators did not precisely match the template of a fitness-proportionate selection scheme used in [1], we are unable to compare their selection intensity with that of tournament and truncation.

In theory, strong selection leads to rapid convergence of a genetic algorithm [1]. Yet, the GA evolved more generations, and thus converged more slowly, as selection intensity increased for each type of selection operator. The beanplot in Figure 13 shows that, as theoretical selection intensity increases, so does the number of generations evolved when prioritizing the GB application's test suite. Although not

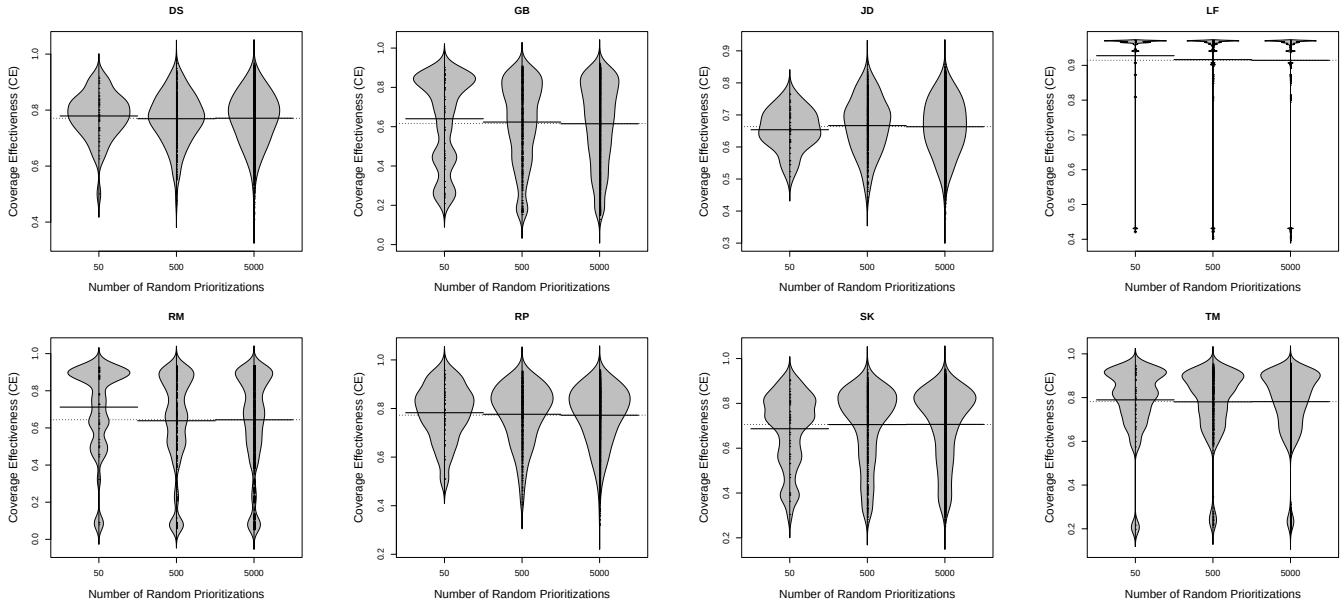


Figure 12: Coverage Effectiveness of Random Prioritization Across All Applications.

shown in the paper due to space constraints, this trend generally holds across all of the other case study applications.

One explanation for this behavior lies in the relationship between the stagnation termination condition and strong selection pressure. A low intensity selection operator causes the algorithm to meander about the search space, looking for a particularly good local optimum to climb, apparently causing the fitness to stagnate. A high intensity selection operator, on the other hand, tends to focus on climbing a local optimum of high quality rather than looking for the hard-to-find global optimum, causing the fitness scores to increase rather than stagnate. We anticipate that, if execution time or number of generations were instead used as termination conditions, many of the lower intensity selection operators would outperform the high intensity operators in terms of the quality of prioritization, given a sufficiently large execution time or number of generations. The noticeable effectiveness of strong selection operators leads us to conclude that the fitness landscape of CE scores has many local optima that correspond to good test suite prioritizations. Thus, our results do not contradict the established theory, but rather contribute a more nuanced understanding of the interplay between selection operators and termination conditions.

The beanplots in Figure 12 demonstrate that the arithmetic mean of the CE scores for the randomly produced test suites ranges between 0.6 and 0.8. The random prioritizers also construct test orderings with a modest variation in their CE values, as evidenced by the fact that the beans often have pronounced grey areas above and below the line designating the average CE score. In contrast, the GA-based prioritizer with order-based crossover, truncation selection, and any value for the other parameters exhibited a standard deviation of 0.00051 across multiple trials and all applications. This result suggests that the GA consistently produces results that are better than random search.

Figure 14 furnishes the beanplots for the first and steepest ascent hill climbers that use the swap-first neighborhood

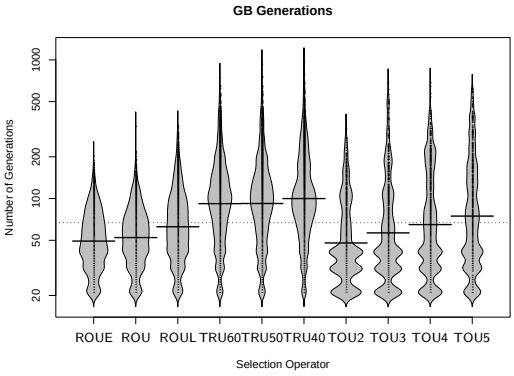


Figure 13: Generations Across Operators.

generators (the swap-last approach always yielded worse results than the given configurations). These plots show that first ascent and steepest ascent hill climbers, respectively denoted HC-FA-FN and HC-SA-FN, construct test suites with an average CE score of 0.8 and 0.84 across all applications. However, Figure 11 indicates that the TRU40 selection operator creates test orderings with an average CE value of 0.9765 over every application. Since these beanplots also reveal modest variability in the CE scores produced by the hill climbers, this empirical outcome highlights the benefits of using a global optimizer like the genetic algorithm.

7. CONCLUSION AND FUTURE WORK

This paper presents and empirically evaluates a genetic algorithm-based prioritization technique, thus generally relating it to prior work such as Elbaum et al. [7]. As an extension to Li et al. [14], this paper describes a comprehensive GA that employs a wide variety of mutation, crossover, and selection operators. Using automatically generated tree models, the empirical study reveals the unique role that the selection operator plays in constructing an effective ordering of a test suite. The results suggest that high selection

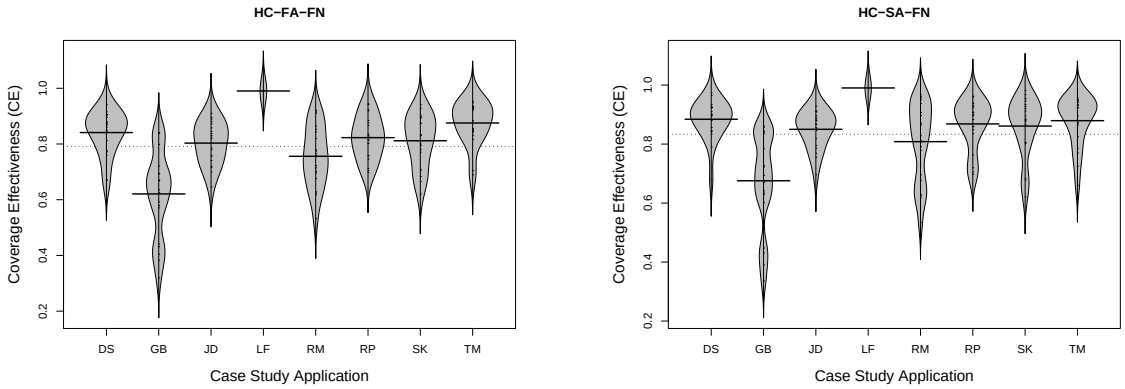


Figure 14: Coverage Effectiveness of Hill Climbing Across All Applications.

intensity and selection elitism are both important for producing good test orderings, as evidenced by the fact that Figure 11 gives TRU40 as the selection operator yielding the best average CE scores across all applications.

Although not the primary focus of this paper, we note that while the GA exhibited execution times similar to the random search and hill climbing methods, it did so with a greater amount of variability. Yet, since the GA consistently produces the most effective test suite orderings, it is useful in development environments that repeatedly run the same test suite on an application whose coverage report does not markedly change across subsequent versions [7, 24]. As part of future work, we intend to further investigate ways to improve the performance of the GA, such as reducing fitness calculation time by persisting CE scores across multiple runs of the prioritizer if coverage and test timing data does not change. Leveraging existing designs [3, 33], we will also implement and evaluate parallel genetic algorithms that reorder test suites. After further study of the GA's various operators and configurations, we will conduct additional experiments with other fitness functions, termination conditions, test adequacy criteria, and case study applications.

8. REFERENCES

[1] T. Blickle and L. Thiele. A comparison of selection schemes used in evolutionary algorithms. *Evol. Comp.*, 4, 1997.

[2] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Chapman and Hall/CRC, 1998.

[3] E. Cantu-Paz. *Efficient and Accurate Parallel Genetic Algorithms*. Kluwer Academic Publishers, 2000.

[4] J. Clarke, J. J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, K. Rees, and M. Roper. Reformulating software engineering as a search problem. *Proc. Softw.*, 150, 2003.

[5] M. J. Crawley. *The R Book*. John Wiley & Sons, Inc., 2007.

[6] R. Durstenfeld. Algorithm 235: Random permutation. *Commun. of the ACM*, 7(7), 1964.

[7] S. Elbaum, A. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *Trans. on Softw. Eng.*, 28(2), 2002.

[8] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *Trans. on Softw. Eng.*, 19(8):774–787, 1993.

[9] M. Harman and J. Clark. Metrics are fitness functions too. In *METRICS*, 2004.

[10] P. Kampstra. Beanplot: A boxplot alternative for visual comparison of distributions. *Journ. of Stat. Softw., Code Snippets*, 28(1), 10 2008.

[11] G. M. Kapfhammer and M. L. Soffa. Database-aware test coverage monitoring. In *ISEC*, 2008.

[12] P. Larranaga, C. M. H. Kuijpers, R. Murga, I. Inza, and S. Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Art. Intell. Rev.*, 13, 1999.

[13] J. Lässig, K. H. Hoffmann, and M. Enachescu. Threshold selecting: best possible probability distribution for crossover selection in genetic algorithms. In *GECCO*, 2008.

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

[15] C. F. Lima, F. G. Lobo, and M. Pelikan. From mating pool distributions to model overfitting. In *GECCO*, 2008.

[16] A. G. Malishevsky, J. Ruthruff, G. Rothermel, and S. Elbaum. Cost-cognizant test case prioritization. Technical Report TR-UNL-CSE-2006-0004, University of Nebraska - Lincoln, 2006.

[17] S. McMaster and A. Memon. Call stack coverage for GUI test-suite reduction. In *ISSRE*, 2006.

[18] S. McMaster and A. M. Memon. Call stack coverage for test suite reduction. In *ICSM*, 2005.

[19] S. McMaster and A. M. Memon. Fault detection probability analysis for coverage-based test suite reduction. In *ICSM*, 2007.

[20] B. L. Miller and D. E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Compl. Sys.*, 9, 1995.

[21] J. Misurda, J. A. Clause, J. L. Reed, B. R. Childers, and M. L. Soffa. Demand-driven structural testing with dynamic instrumentation. In *ICSE*, 2005.

[22] A. S. Namin and J. H. Andrews. The influence of size and coverage on test suite effectiveness. In *ISSTA*, 2009.

[23] X. Qu, M. B. Cohen, and G. Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *ISSTA*, 2008.

[24] D. S. Rosenblum and E. J. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *Trans. on Soft. Eng.*, 23(3), 1997.

[25] A. M. Smith and G. M. Kapfhammer. An empirical study of incorporating cost into test suite reduction and prioritization. In *SAC*, 2009.

[26] C. Strobl, A.-L. Boulesteix, A. Zeileis, and T. Hothorn. Bias in random forest variable importance measures: Illustrations, sources and a solution. *BMC Bioinf.*, 8, 2007.

[27] N. J. Wahl. An overview of regression testing. *SIGSOFT Softw. Eng. Notes*, 24(1), 1999.

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

[29] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In *ISSTA*, 2009.

[30] S. Yoo, M. Harman, and S. Ur. Measuring and improving latency to avoid test suite wear out. In *SBST*, 2009.

[31] J. Zhong, X. Hu, J. Zhang, and M. Gu. Comparison of performance between different selection strategies on simple genetic algorithms. In *CIMCA*, volume 2, 2005.

[32] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *Comput. Surv.*, 29(4), 1997.

[33] B. Zorman, G. M. Kapfhammer, and R. S. Roos. Creation and analysis of a JavaSpace-based genetic algorithm. In *PDPTA*, 2002.