5

Test Suite Prioritization with Greedy and Genetic Algorithms: An Empirical Study

Abstract

Regression test suite prioritization is an important part of real-world software testing plans. Traditionally, regression test suite prioritization has been performed with greedy algorithms. However, little work has been done to explore new prioritization methods. Based on recent work conducted during the summer, I judge that a genetic algorithm can produce prioritizations of significantly higher quality within a reasonable execution time. My proposed research will focus upon three approaches. First, I propose to continue to develop the genetic algorithm-based prioritization tool that I developed during the summer through the addition of new modules and refactoring of existing code, improving the quality of its performance, output, and reusability by future research projects. Second, I propose to incorporate existing empirical data concerning greedy prioritization techniques into an expanded empirical study in which greedy and genetic algorithm-based techniques will be quantitatively compared. Finally, I propose to develop a comprehensive framework for visualization and statistical analysis of regression test prioritization techniques. This study has the potential to guide future regression test suite prioritization research by demonstrating the viability of genetic prioritizers. This framework will provide others who are conducting research in this area with a set of generalizable functions that will allow them to easily perform statistical analyses and visualizations of their data.

1 Introduction

Software testing is a vital component of any software development process. By revealing faults and establishing a confidence in the correctness of code,

testing is invaluable for even the smallest software development tasks. As the size and complexity of a software project expands, the challenges associated with testing that software grow rapidly. One such challenge is preventing the emergence of regressions within the program being developed. Altering one part of a large system can create unexpected faults in other parts of the system, breaking previously functional code. This is known as a software regression [18]. As many modifications occur to a piece of software throughout the course of its life, regressions are likely to become a serious problem.

In order to reveal and guard against software regressions, software engineers use a technique known as regression testing [18]. By rerunning all existing test cases on an entire software system, regression testing reveals regressions as well as other faults within the project. For large software projects, however, the costs of regression testing are often extremely high. For example, the regression test suite for a commercial project of approximately 20,000 lines of code takes seven weeks to run [5]. In the 1990's, it was estimated that over 70% of the costs of software development stemmed from the maintenance phase [18], of which regression testing is a major part. There is no reason to believe that this percentage has significantly declined in recent years. On the contrary, the increasing complexity of software implies an increase in the costs associated with testing.

One approach to handling the immense costs of regression testing is to reduce the test suite by removing some of the test cases. However, certain types of minimization may seriously compromise a test suite's ability to detect certain kinds of faults [23]. Test suite prioritization is a safer means for mitigating the costs associated with regression testing. By prioritizing a test suite, typically with regard to metrics reflecting code coverage or fault detection capability of individual cases, faults are, on average, detected earlier in the test suite's execution. While prioritization does not reduce the overall costs of regression testing, by finding faults faster more information is made available sooner concerning the fault profile of the program, and software engineers are able to address and correct problems more rapidly than would otherwise be possible.

Typically, test suite prioritization systems use greedy algorithms to produce new permutations. Being extremely fast, greedy algorithms are ideal when a test suite prioritization is required very quickly [26]. However, preliminary investigation comparing greedy [25] and search-based test suite prioritization techniques [2] indicates that, in exchange for a somewhat greater execution time, search-based techniques can produce orderings that are significantly better than those produced by current greedy-based techniques.

Thus far, there has been only a single paper published on the topic of search-based test suite prioritization [19]. This field seems to be prime for exploration.

2 Background and Related Work

2.1 Regression Testing

As has already been discussed, regression testing is a means of revealing regression faults within a program Z. By definition, a regression test suite T of size n is composed of a sequence of test cases $(t_1, t_2, \ldots, t_n)$. As T is executed, each of the member test cases is executed sequentially.

2.2 Reduction and Prioritization

For every test suite T, there exists a set of all possible permutations called PT. T covers a set of requirements $r = \{r_1, r_2, \ldots, r_m\}$. Each test case t_i within T is said to cover a certain subset of the requirements in r. As discussed earlier, prioritization seeks to reorder the test cases within T in order to discover a better ordering with respect to the rate at which requirements are covered. Thus, the definition of the problem of test case prioritization is that of finding $T' \in PT$ such that $\forall T'' \in PT$, $T'' \neq T'$ and T' covers the set of requirements at least as rapidly as T'' [4]. These requirements can represent any number of different quantities, such as faults within the program [3] or nodes and edges of a program's control flow graph [11].

Also known as test suite minimization, reduction strives not to find a specific permutation of T within PT, but rather to find a minimal subset of the test cases within T. Given a set of requirements r covered by T and a set of all subsets of T known as ST, the definition of the problem of test suite minimization is that of finding $T_s \in ST$ such that T_s covers all requirements in r, and $\forall T_u \in ST$, $T_u \geq T_s$ in terms of either the number of test cases within each or the execution time of each.

2.3 Metrics

In order to prioritize a test suite, a metric is needed in order to rank potential orderings. A number of such metrics are in existence. The APFD (Average Percentage of Faults Detected) metric, discussed in [24], ranks test cases based on fault coverage data collected during previous executions of the test suite. By reading a dataset indicating which test cases reveal which faults,

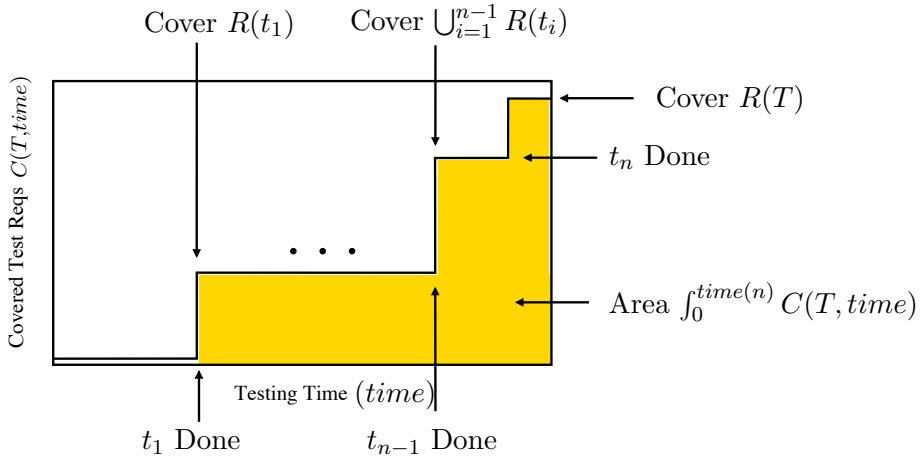

Figure 1: Coverage Effectiveness Graph of a Test Suite T. (used with permission of the author)

a prioritizer operating with respect to APFD reorders a test suite in order to reveal faults quickly, in terms of number of test cases executed. APFD values can range from 0 to 100, where larger values are better. However, only the number of test cases are taken into consideration and not the execution time of each test case; APFD assumes an equal execution time for each test case.

Another limitation of the APFD metric is that all faults are assumed to be of equal severity. An extension of APFD, known as APFD_c [3], is a "cost-cognizant" version of the APFD metric. APFD_c takes into account both the runtime costs of each individual test case as well as the severity of each fault detected. Thus, rather than seeking to maximize the rate of exposed faults with respect to number of test cases executed, APFD_c seeks to maximize the rate of exposed fault severity while minimizing execution time of the test suite. Like APFD, this new cost aware version also requires fault detection information gained from previous executions of the test suite.

As an alternative to fault coverage of individual test cases, other metrics exist concerning code or requirement coverage instead. APBC, APDC, and APSC are additional metrics which measure the Average Percentage of Block, Decision, or Statement Coverage, respectively [19]. Like APFD, these metrics are limited in that they each assume equal execution time for each test case.

A new metric, coverage effectiveness (CE), takes into account both the coverage as well as execution time of individual test cases when prioritizing

a test suite [10]. As this metric was only first published in 2007, so far there has only been a single study [26] published investigating the coverage effectiveness of test suite prioritizations. Figure 1 provides an illustration of a coverage effectiveness graph for a test suite T. In Figure 1, the horizontal axis represents execution time of T, while the vertical axis reflects the number of requirements that have been covered by T at a given point in time. Given a program under test Z, CE seeks to reorder a test suite T so that a set of requirements $r = \{r_1, r_2, \ldots, r_m\}$ are covered as early in the execution of T on Z as possible. In Figure 1, $time(n)$ corresponds to the length of time necessary for test cases 1 through n to fully execute. C is the function that takes a test suite T and a numeric *time* vaule, and returns the number of requirements covered by T at that point in time. R is the function that takes a single test case and returns a list of all of the requirements which that test case covers.

2.4 Genetic Algorithms

Attempting to leverage the power of natural evolutionary processes for numeric computation, genetic algorithms (GAs) are evolutionary computation techniques used to produce high-quality solutions to a wide range of optimization problems. The travelling salesman problem [14], finding the key to a simple substitution cipher [14], the 0/1 knapsack problem [6], designing parts for aircraft [8], and prioritizing a test suite are a few examples of problems to which genetic algorithms have thus far been successfully applied. Figure 2 illustrates the general structure of a genetic algorithm. The basic components of a genetic algorithm are listed below:

- A allele is the smallest indivisible unit of information present within the genetic algorithm. Allele representation varies widely depending on the problem being modeled. When prioritizing test suites, for example, a allele represents a single test case, and is typically implemented in memory as an integer. This is just one example, however; numerous allele encoding schemes exist.

- An individual represents a potential solution to the problem. Every point in the search space of candidate solutions is represented by a unique individual [13]. Each individual is composed of a set of alleles, known as its genome. In the problem of test suite prioritization, an individual corresponds to one specific ordering of the test suite. Its genome consists of an array of unique alleles, each representing a test case and its order within the test suite.

- A population is a collection of differing individuals. Initially, the genetic algorithm generates a random population of individuals, which it then "evolves," aiming to create new, more "fit" individuals.

- The fitness of individuals within a population is typically (but not always) represented by a single numeric value. Individuals with desirable traits are assigned high fitnesses, while individuals with less desirable traits are ranked as less fit. This fitness value allows individuals within the population to be hierarchically ordered, so that the traits of the most fit individuals are preserved and propagated while the traits of the least fit individuals are left to "die" out, as the least fit individuals are removed from the population. The fitness function must be fully defined; that is, it must be capable of evaluating every individual that it encounters [13]. One common approach to creating a fitness function for prioritization is to use a metric, such as coverage effectiveness, to evaluate and rank individuals. Such an approach is put forth and advocated in [7], and is particularly useful when evolving test suite prioritizations.

- A crossover operator takes two or more "parent" individuals and, by recombining the two genomes, produces one or more "child" individuals that share some of the properties of each parent. For instance, when recombining two individuals who represent solutions to a test suite prioritization problem, a crossover operator could first copy a substring from the first parent's genome into the child's genome, and then fill in the child's remaining allele positions in the order which the alleles appear in the second parent. This ensures that each allele appears exactly once in the child individual, which is a requirement of a valid test suite permutation. By creating new individuals from the genetic material of two fit individuals, a crossover operator intelligently selects new points in the search space that are likely to have high fitnesses themselves [13].

- A selection operator selects a set of parent individuals from a population with respect to their fitnesses. Most selection operators are probabilistic; the most fit individuals have a greater, but not guaranteed, probability to be selected over the other individuals of lesser fitness. The probabilistic nature of selection helps the GA to overcome local optima in the search space.

- A mutation operator makes small, random changes to the genome of

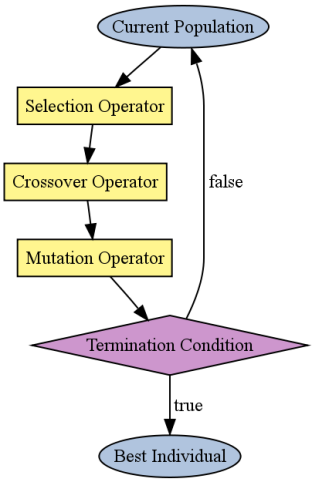

Figure 2: Basic structure of a genetic algorithm.

a small percentage of the children. In an individual representing a test suite permutation, for example, a mutation could appear as the swapping of the positions of two different alleles. Mutation also helps the genetic algorithm to overcome local optima, particularly when the GA is operating in a rugged fitness landscape [13].

- A fitness transformation operator is an optional part of a genetic algorithm that can be used to strengthen either the strongest or the weakest individuals. By emphasizing the strongest, convergence of the algorithm will occur more rapidly, while strengthening the weakest individuals enhances the GA's ability to overcome local optima.

- A termination condition is a necessary part of a genetic algorithm, since without this the GA would continue to evolve its population forever. A termination condition halts the evolution and causes the GA to return the most fit individual thus far generated. Possible criteria for termination include an execution time limitation, a target fitness, or stagnancy of the population (that is, the overall fitness of the population has ceased to improve).

Gelations Operators		
type	abbreviation	full name
Mutation Operators [14]	DM	Displacement mutation
	EM	Exchange mutation
	ISM	Insertion mutation
	IVM	Inversion mutation
	SIM	Simple inversion mutation
	SM	Scramble mutation
Crossover Operators [14]	CX	Cycle crossover
	MPX	Maximal preservative crossover
	OX1	Order crossover
	OX2	Order based crossover
	PMX	Partially-mapped crossover
	POS	Position based crossover
	VR	Voting recombination crossover
Selection Operators	ROU	Roulette-wheel selection
	TRU	Truncation selection
	RND	Random-based probabilistic selection
Transformation Operators	EXP	Exponential transformation
	UNT	No transformation
	WIN	Windowing transformation

Table 1: Operators implemented within Gelations.

3 Preliminary Work

3.1 Genetic Algorithm-Based Test Suite Prioritization System

This summer, I conducted an eight-week research project, under the supervision of Professor Kapfhammer, on the effectiveness and efficiency of genetic algorithms when applied to the problem of regression test suite prioritization. One product of this project is a working tool, written in Java, that prioritizes regression test suites through the use of genetic algorithms. The current name for this tool is GELATIONS (GEnetic aLgorithm-bAsed Test suIte priOritizatioN System). According to JavaNCSS [17], the prioritization system itself consists of 51 classes, 215 functions, and 1,806 non-commented source statements. The regression test suite for testing the prioritization system, written using the JUnit [9] unit testing framework, consists of 44 classes, 191 functions, and 3,441 non-commented source statements.

An illustration containing all of the prioritizer's configuration parameters is given in Figure 3. Through the use of interfaces and abstract classes, the system is able to easily interchange its components; for example, swapping out one mutation operator for another. Currently, seven different crossover operators, six mutation operators, three selection operators, and three fitness transformation operators have been implemented as a part of this system. When running the prioritization system, it is necessary to select a single mutation operator, crossover operator, selection operator, and transformation operator from each of the given categories. The abbreviations associated with each operator by the Gelations system are presented in Table 1 for future reference. All of the mutation and crossover operators are defined in [14].

Within this genetic prioritization system, I used the coverage effectiveness (CE) metric [10] to determine the fitnesses of all individuals comprising each population. The fitness of an individual (before any fitness transformations) is directly correlated to its coverage effectiveness value. However, because of the modular design of the fitness calculator, it would be easy to create a new fitness function that assigns individual fitness based on a different metric, such as average percentage of faults detected (APFD) [3] or average percentage of requirements covered (APRC).

In addition to the options listed above, a number of other configuration parameters exist for the Gelations prioritization system. The rate at which children are mutated, the percentage of each population which is composed of new children, and the number of individuals present within each population must also be specified. The final three configuration parameters control the genetic algorithm's termination. The program can terminate once a target fitness value has been met or exceeded, after a set amount of time has elapsed, or once a stagnancy threshold has been exceeded. By stagnancy threshold, what is meant is a set number of consecutive generations for which the algorithm has stagnated, or not improved the maximum fitness of the most fit individual within its population. All of these parameters are displayed in Figure 3.

3.2 Empirical Study

Using this tool, I conducted an extensive empirical study concerning the efficiency and effectiveness of the various configurations of my genetic prioritization system. The two metrics with which I was concerned were execution time of the prioritizer and the coverage effectiveness of the final ordering of the regression test suite under test. To test my prioritizer, I made use of the

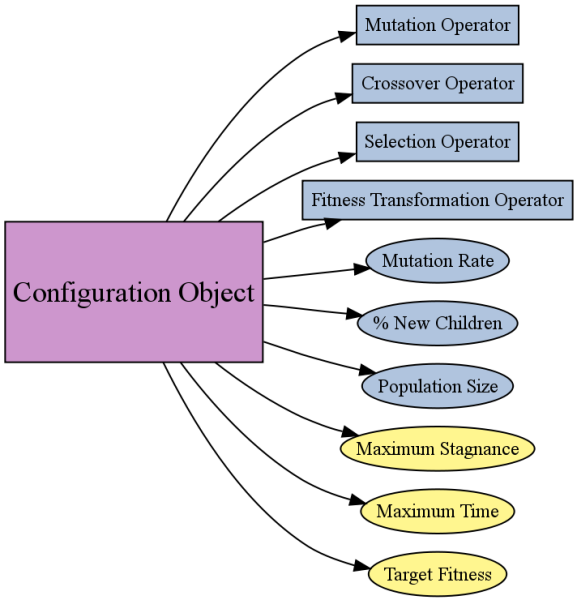

Figure 3: Configuration parameters accepted by Gelations.

JUnit test suites of nine real-world case study applications. Each of these test suites was instrumented by [25] in order to extract the coverage and execution time information associated with each of the test suites. By passing this data into a given configuration of my prioritizer, I was able to obtain numerous improved orderings of the test suite. As an experimental control, I implemented a simple random search to also produce permutations of the suite under test against which to compare the results of my genetic prioritizer. Random search simply takes a random subset of the search space of a specified size and examines each element within that subset by brute force.

Figure 4 displays the coverage effectiveness values of the solutions produced by my prioritizer with respect to the prioritizer's execution time. Each symbol in this graph corresponds to a single execution of the prioritizer. A symbol's position on the horizontal axis indicates the execution time of that execution of the prioritizer, while its position on the vertical axis indicates the coverage effectiveness of the prioritization produced by that execution. The data for each of the nine case study applications is graphed within a separate frame, and the data points are broken down with respect to the crossover operators used. This is not a comprehensive visualization of all of my data; rather, this image displays only data points generated using the ISM mutation operator, a mutation rate of 0.1, a child density of 1.0,

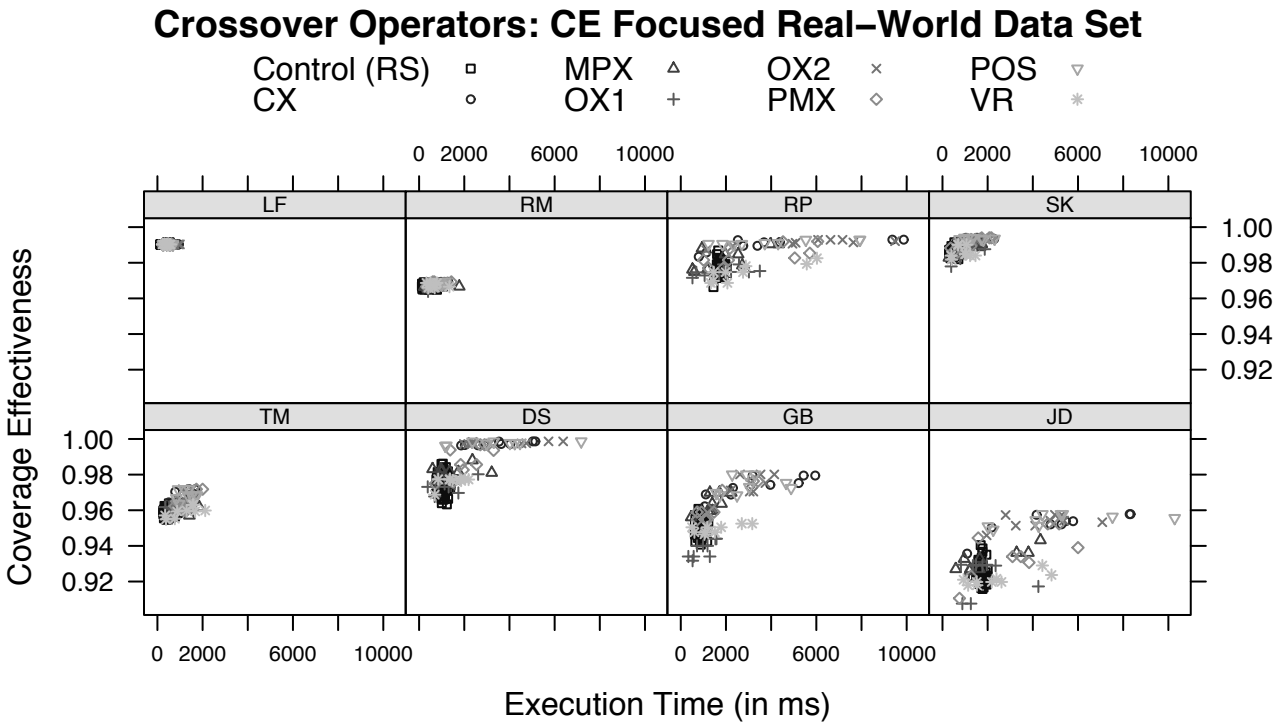

Figure 4: Subset of experimental results emphasizing greatest CE.

the WIN fitness transformation operator, and a population size of 150. The windowing transformation operator normalizes the differences between the fitness of each individual; for instance, in the case of ten individuals being compared, the least fit individual would be assigned a fitness of 0.1 regardless of its specific fitness value, the second least fit individual would be assigned a fitness of 0.2, and so on. This subset was chosen in order to place emphasis on the prioritizer's ability to produce orderings of extremely high coverage effectiveness. Figure 5 displays data corresponding to configurations of the prioritizer utilizing the ISM mutation operator, a mutation rate of 0.1, a child density of 0.6, the EXP fitness transformation operator, and a population size of 50, in order to emphasize the speed of my prioritizer.

As mentioned above, for this empirical study I used data extracted from the test suites of real-world case study applications. Eight different applications were used. These applications were JDepend (a Java package analyzer), Transaction Manager (a database-dependent ATM simulator), a

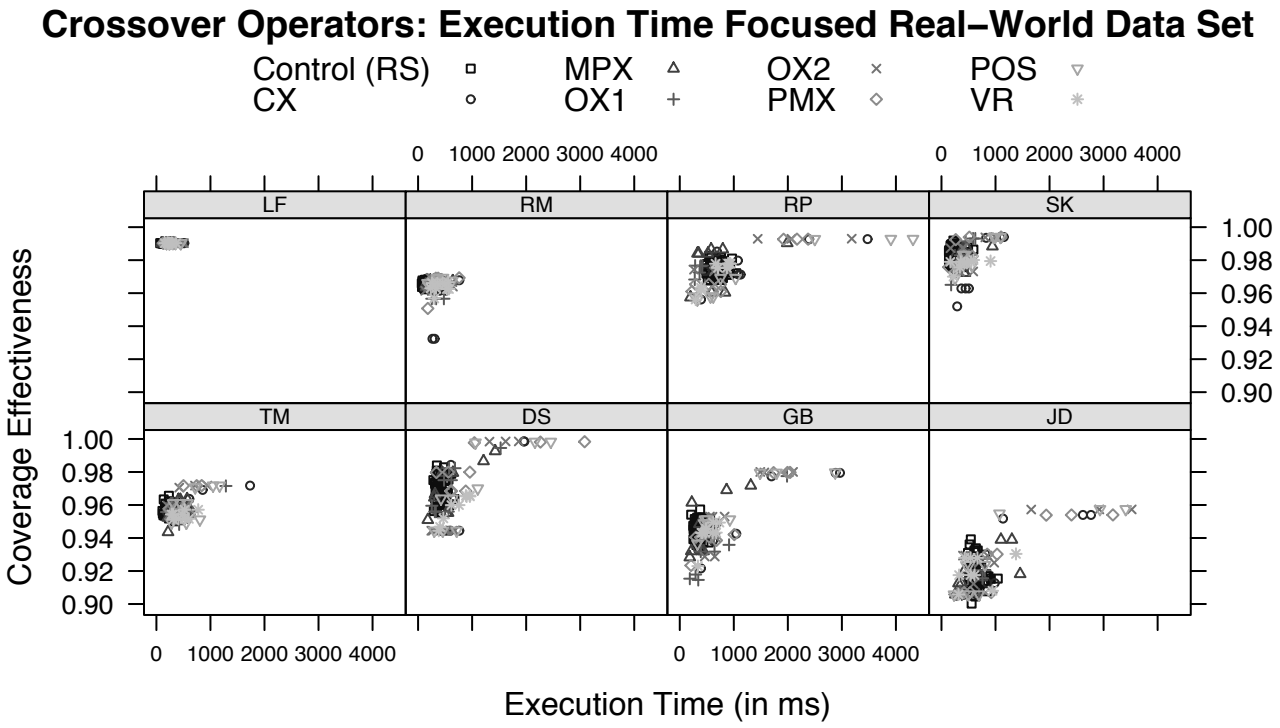

Figure 5: Subset of experimental results emphasizing shortest runtimes.

Sudoku puzzle solver, Data Structures (for testing different data structures), Reduce and Prioritize (a set of greedy algorithms for the reduction and prioritization of test suites), Loop Finder (for finding cycles within a graph), Reminder (a database-dependent application for storing and generating reminders), and GradeBook (for tracking and analyzing student's grades in a college course). Coverage and timing data was extracted from instrumented executions of the case study applications' test suites upon the applications themselves; from these executions, coverage trees were generated. The test coverage monitor instrumentation was implemented using Java 1.5 and AspectJ 1.5. More details concerning how the coverage information was acquired can be found in [11].

3.3 Statistical Analysis

In order to conduct my statistical analyses and to create my visualizations of the data generated by my empirical study, I used the R language for statistical computing [21]. I developed an array of functions to leverage the power of the gplots, tree, randomForest, and lattice packages, applying many of the visualizations and analyses made possible by each to my sets of prioritization data. Extensive functionality for creating box-and-whisker plots, conducting analysis of variance (ANOVA) tests, creating regression trees and random forests [1], and for creating complex scatter plot graphs through the lattice package is present within my current suite of R functions. These analyses are important to perform as they aid in uncovering different trends within the data. In addition to these functions, I also have facilities for reading and manipulating datasets and saving the resulting analyses and visualizations to disk in multiple formats.

4 Thesis

1. Through parameter tuning, optimization, refactoring, and extension, the genetic algorithm-based test suite prioritizer will produce prioritizations of higher quality and/or consume a shorter execution time than it currently does.

2. Prioritizations produced by the tuned genetic algorithm will consistently be better than those produced by existing greedy techniques in terms of coverage effectiveness.

5 Proposed Research

My proposed research can be roughly divided into three different categories.

5.1 Extension of the Gelations Prioritization Tool

In Section 3.1, I discussed the genetic algorithm-based test suite prioritization system that I constructed as a part of my summer research. I propose to improve and extend this system in a number of different ways. These improvements are listed in order of priority. The following improvements are of primary concern:

1. The preliminary empirical study was performed with a faulty version of the truncation selection operator. The fault within truncation involved a lack of elitism; sometimes non-elite elements would slip into the chosen set of parents. In addition, I plan to examine all of the other classes, particularly the operators and metric calculators, in great detail, refactoring the code as much as possible without compromising the program's modular design in order to produce the best performance of my system.

2. Tournament selection, a commonly used selection mechanism in genetic algorithms, is currently absent from my system. I will add this selection mechanism and incorporate this additional selection operator into my final empirical study. I plan to investigate both a deterministic version of tournament selection as well as a "noisy" non-deterministic version, as described in [20].

3. Now that I have a large set of data to examine, I judge that I can do a far better job of intelligently selecting parameters to configure my genetic algorithm than I did in my preliminary study. This intelligent selection will be accomplished through inspection of scatterplots, regression tree analysis, and random forest analysis.

4. To aid in investigation of the genetic prioritizer, I plan on implementing a comprehensive package of statistical analysis and visualization routines in the R programming language. This package will be sufficiently generalizable such that other research projects beyond my own will be able to leverage it for the analysis of regression test suite prioritization techniques.

The remaining improvements are not vital, but I judge them to be valuable and will pursue them if time permits:

1. By implementing additional modules for the calculation of APFD [24], APRC, and possibly other metrics as well (such as APBC, APDC, APSC [19], APFD_C [3], etc.), more future research projects may be able to benefit from the system. As the system is able to incorporate a wider variety of fault and coverage metrics, the pool of research projects that will be able to apply the system to their own prioritization research will grow.

2. In addition to refactoring of code, I also plan to explore alternative techniques for improving program performance. I will investigate al-

ternative forms of garbage collection such as incremental and generational garbage collection, and compiler optimizations such as mapping variables to machine registers and inlining static methods. These techniques, among others, are discussed in [12]. I will also investigate tools such as modern optimizing compilers and bytecode optimizers.

3. I would also like to explore implementing my genetic algorithm in a parallel manner. I will investigate simple global parallelization, in which fitness computation, selection, mutation, crossover, and/or transformation operators will themselves be parallelized in order to take advantage of multiple CPU cores through multithreading. This parallelization approach would not be terribly difficult to implement, and would speed up execution time considerably on multi-core machines. A second approach to parallelization, coarse-grained parallelization, involves partitioning the entire population into subpopulations and running a separate genetic algorithm on each population, while periodically exchanging individuals between populations through migration. These and other parallelization techniques for genetic algorithms are discussed in [22].

5.2 Study of Greedy vs. Genetic Algorithm-Based Test Suite Prioritizers

In Section 5.1, I listed a number of additions and modifications that I would like to make to my prioritization system. Once these modifications have been completed, I plan to conduct a new empirical study based upon the study conducted over the summer. The results of this experiment will be combined with the results obtained by Adam Smith in [25, 26] in order to compose a rigorous study highlighting the relative strengths and weaknesses of greedy and genetic prioritization methods. Because greedy techniques are currently the industry standard, it is important that the performance and quality of genetic prioritization techniques be compared to the performance and quality of greedy prioritization techniques. I judge that this is feasible, as we both analyze our prioritizations with regard to coverage effectiveness and execution time, and we both use eight of the same sets of test data extracted from real-world case study applications.

5.3 Comprehensive Framework for Statistical Analysis of Regression Testing Techniques

Not including scripts for automated generation of specific graphs, I currently have an R source file of 2,063 lines devoted to functions for the analysis and visualization of prioritization data. From my conversations with Professor Kapfhammer, I am convinced that the further development of the existing R code into a comprehensive, robust package would be both beneficial to myself as well as appreciated by others who are conducting similar types of research. By developing and releasing this package, others conducting prioritization experimentation will be relieved of the task of designing their own functions for the analysis of their data. Another practical benefit of such a package will be a standard format for data and standard set of statistical analyses for experimental results. By providing such a standard format, different prioritization techniques that each adhere to the format specified by this package will be simpler to compare than different prioritization techniques that do not adhere to a standard format. I aim to contribute to regression test suite prioritization research in a manner on top of which future projects can build.

In addition to refactoring and extending the existing functionality that I have already implemented, I would like to investigate the addition of new analyses and visualization techniques. Foremost among these are hierarchical clustering, association analysis, correlation analysis, and residual analysis, as presented in [15] and leveraged in [16]. I would also like to explore using a Kruskal-Wallis rank sum test and to experiment with using tree prediction machine learning techniques.

Hierarchical clustering will allow me to uncover similarities between multiple parameters, in terms of their impact on the experimental output. Association analysis allows me to quickly visualize relationships between parameter values and experimental results in order to apprehend specific trends. Correlation analysis offers a new way to analyze associations between parameters and experimental output through the use of the Spearman rank correlation coefficient[15]. Because preliminary analysis of my data leads me to believe that I may not be dealing with normal distributions, I judge that a Kruskal-Wallis one-way analysis of variance may be beneficial, as this technique does not require a normal distribution in order to perform an analysis of variance, unlike ANOVA. Through tree prediction, I hope to be able to predict the leaf values of a regression tree for a set of data without needing to first generate the tree.

6 Conclusion

Based on my preliminary work, I judge that search-based test suite prioritization techniques hold great promise as new methods for improving prioritizations. Although prioritization does not reduce the amount of time spent on executing regression test suites, it does allow resources to be leveraged in as efficient a manner as possible, detecting faults sooner so that they can be addressed and corrected. Genetic algorithms in particular show much potential as a tool to be leveraged towards this work. Through my research, I hope to determine the best ways in which to configure a genetic algorithm in order to produce test suite permutations of the highest quality within a reasonable execution time. This research will produce a tool that can be used by others to perform prioritizations of their own test suites, a detailed study comparing and contrasting my highly tuned genetic algorithms to current greedy techniques, and a comprehensive package of statistical analysis and visualization tools to help advance future research into test suite prioritization. I judge that these projects are quite feasible, and that to abandon this project at its current stage would be a waste of an excellent opportunity.

References

[1] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, 2001.

[2] Computer Science 290 Fall [redacted] Class. Search based regression test prioritization: An empirical study. Allegheny College, Meadville, Pennsylvania, December [redacted].

[3] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 329–338, Washington, DC, USA, 2001. IEEE Computer Society.

[4] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. *SIGSOFT Software Notes*, 25(5):102–112, 2000.

[5] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Transactions on Software Engineering*, 28(2):159–182, February 2002.

[6] Zoheir Ezziane. Solving the 0/1 knapsack problem using an adaptive genetic algorithm. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing*, 16(1):23–30, 2002.

[7] Mark Harman and John Clark. Metrics are fitness functions too. In *In Proceedings of the 10th International Symposium on Software Metrics*, pages 1–12. IEEE Computer Society, 2004.

[8] Luigi Iuspa, Francesco Scaramuzzino, and Pietro Petrenga. Optimal design of an aircraft engine mount via bit-masking oriented genetic algorithms. *Advances in Engineering Software*, 34(11-12):707–720, 2003.

[9] Welcome to junit.org! — junit.org. Accessed 10 September 2008. Available at: `http://www.junit.org/`.

[10] Gregory M. Kapfhammer and Mary Lou Soffa. Using coverage effectiveness to evaluate test suite prioritizations. In *Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, pages 19–20, New York, NY, USA, 2007. ACM.

[11] Gregory M. Kapfhammer and Mary Lou Soffa. Database-aware test coverage monitoring. In *Proceedings of the 1st India Software Engineering Conference*, pages 77–86, New York, NY, USA, 2008. ACM.

[12] Iffat H. Kazi, Howard H. Chen, Berdenia Stanley, and David J. Lilja. Techniques for obtaining high performance in Java programs. *ACM Comput. Surv.*, 32(3):213–240, 2000.

[13] John R. Koza, Forrest H Bennett III, David Andre, and Martin A. Keane. *Genetic Programming III*, chapter Background on Genetic Programming and Evolutionary Computation. Morgan Kaufmann Publishers Inc., San Fransisco CA, 1999.

[14] P. Larranaga, C.M.H. Kuijpers, R.H. Murga, I. Inza, and S. Dizdarevic. Genetic algorithms for the travelling salesman problem: A review of representations and operators. *Artificial Intelligence Review*, 13:129–170, 1999.

[15] Benjamin C. Lee and David M. Brooks. Spatial sampling and regression strategies. *IEEE Micro*, 27(3):74–93, 2007.

[16] Benjamin C. Lee, David M. Brooks, Bronis R. de Supinski, Martin Schulz, Karan Singh, and Sally A. McKee. Methods of inference and learning for performance modeling of parallel applications. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 249–258, New York, NY, USA, 2007. ACM.

[17] Chr. Clemens Lee. Accessed 10 September 2008. Available at: `http://www.kclee.de/clemens/java/javancss/`.

[18] Yuejian Li and Nancy J. Wahl. An overview of regression testing. *ACM SIGSOFT*, 24(1):69–73, January 1999.

[19] Zheng Li, Mark Harman, and Robert M. Hierons. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering*, 33(4):225–237, 2007.

[20] Brad L. Miller and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. Technical report, University of Illinois at Urbana-Champaign, July 1995. IlliGAL Report No. 95006.

[21] The R project for statistical computing. Accessed 18 September 2008. Available at: `http://www.r-project.org/`.

[22] Wilson Rivera. Scalable parallel genetic algorithms. *Artificial Intelligence Review*, 16(2):153–168, 2001.

[23] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proceedings of the International Conference on Software Maintenance*, pages 34–43, 1998.

[24] Gregg Rothermel, Roland H. Untch, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[25] Adam M. Smith. Incorporating time into test suite reduction and prioritization: An empirical study. Technical report, Allegheny College, 2008. Report no. CS08-05.

[26] Adam M. Smith and Gregory M. Kapfhammer. An empirical study of incorporating cost into test suite reduction and prioritization. In *Proceedings of the 2009 Symposium on Applied Computing*. ACM, 2009.