Time-Aware Test Suite Prioritization

Kristen R. Walcott
Mary Lou Soffa
Department of Computer Science
University of Virginia
{walcott, soffa}@cs.virginia.edu

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

ABSTRACT

Regression test prioritization is often performed in a time constrained execution environment in which testing only occurs for a fixed time period. For example, many organizations rely upon nightly building and regression testing of their applications every time source code changes are committed to a version control repository. This paper presents a regression test prioritization technique that uses a genetic algorithm to reorder test suites in light of testing time constraints. Experiment results indicate that our prioritization approach frequently yields higher average percentage of faults detected (APFD) values, for two case study applications, when basic block level coverage is used instead of method level coverage. The experiments also reveal fundamental trade-offs in the performance of time-aware prioritization. This paper shows that our prioritization technique is appropriate for many regression testing environments and explains how the baseline approach can be extended to operate in additional time constrained testing circumstances.

Categories and Subject Descriptors

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

General Terms

Verification, Experimentation

Keywords

test prioritization, coverage testing, genetic algorithms

1. INTRODUCTION

After a software application experiences changes in the form of bug fixes or the addition of functionality, regression testing is used to ensure that changes to the program do not negatively impact its correctness. However, regression testing can be prohibitively expensive, particularly with respect to time [13], and thus accounts for as much as half the cost of software maintenance [14, 24]. In one example, an industrial collaborator reported that for one of its products of approximately 20,000 lines of code, the entire test suite required seven weeks to run [8]. Since there is usually a limited amount of time allowed for testing [19], prioritization techniques can be used to reorder the test suite to increase the rate of fault detection earlier in test execution [8, 24]. However, no existing approach to prioritization incorporates a testing time budget.

As frequent rebuilding and regression testing gain in popularity, the need for a time constraint aware prioritization technique grows. For example, popular software systems like `PlanetLab` [3] and `MonetDB` [2] use nightly builds, which include building, linking, and unit testing of the software [19]. New software development processes such as extreme programming also promote a short development and testing cycle and frequent execution of fast test cases [23]. Therefore, there is a clear need for a prioritization technique that has the potential to be more effective when a test suite's allowed execution time is known, particularly when that execution time is short. This paper shows that if the maximum time allotted for execution of the test cases is known in advance, a more effective prioritization can be produced.

The time constrained test case prioritization problem can be reduced to the NP-complete zero/one knapsack problem [10, 24], which can often be efficiently approximated with a genetic algorithm (GA) heuristic search technique [6]. Just as genetic algorithms have been effectively used in other software engineering and programming language problems such as test generation [22], program transformation [9], and software maintenance resource allocation [5], this paper demonstrates that they also prove to be effective in creating time constrained test prioritizations. We present a technique that prioritizes regression test suites so that the new ordering (i) will always run within a given time limit and (ii) will have the highest possible potential for defect detection based on derived coverage information. This paper also provides empirical evidence that the produced prioritizations on average have significantly higher fault detection rates than random or more simplistic prioritizations, like the initial order or a reverse reordering. In summary, the important contributions of this paper are as follows:

1. A technique that uses a genetic algorithm to prioritize a regression test suite that will be run within a time constrained execution environment (Section 2 and Section 3).
2. An empirical evaluation of the effectiveness of the resulting prioritizations in relation to (i) GA-produced prioritizations using different parameters, (ii) the ini-

ISSTA'06, July 17–20, 2006, Portland, Maine, USA.

tial test suite ordering, (iii) the reverse of the initial test suite ordering, (iv) random test suite prioritizations, and (v) fault-aware prioritizations, showing that the GA-produced prioritizations are superior to the other test suite reorderings (Section 4).

3. An empirical evaluation of the time and space overheads of our approach. This evaluation reveals that the technique is especially applicable when (i) there is a fixed set of time constraints, (ii) prioritization occurs infrequently, or (iii) the time constraint is particularly small (Section 4).
4. A discussion of enhancements to the baseline approach that reduce the time overhead required to perform prioritization and extend the technique's applicability to other time constrained testing situations (Section 5).

2. TIME-AWARE TEST PRIORITIZATION CHALLENGES

Test prioritization schemes typically create a single reordering of the test suite that can be executed after many subsequent changes to the program under test [7, 24]. Test case prioritization techniques reorder the execution of a test suite in an attempt to ensure that defects are revealed earlier in the test execution phase. If testing must be terminated early, a reordered test suite can also be more effective at finding faults than one that was not prioritized [24]. Problem 1 defines the time-aware test case prioritization problem. Intuitively, a test tuple σ earns a better fitness if it has a greater potential for fault detection and can execute within the user specified time budget.

Problem 1. (***Time-Aware Test Suite Prioritization***)
Given: (i) A test suite, T, (ii) the collection of all permutations of elements of the power set of permutations of T, $perms(2^T)$, (iii) the time budget, t_{max}, and (iv) two functions from $perms(2^T)$ to the real numbers, $time$ and fit.
Problem: Find the test tuple $\sigma_{max} \in perms(2^T)$ such that $time(\sigma_{max}) \leq t_{max}$ and $\forall \sigma' \in perms(2^T)$ where $\sigma_{max} \neq \sigma'$ and $time(\sigma') \leq t_{max}$, $fit(\sigma_{max}) > fit(\sigma')$.

In Problem 1, $perms(2^T)$ represents the set of all possible tuples and subtuples of T. When the function $time$ is applied to any of these tuples, it yields the execution time of that tuple. The function fit is applied to any such tuple and returns a fitness value for that ordering. Without loss of generality, we assume that an awarded higher fitness is preferable to a lower fitness. In this paper, the function fit quantifies a test tuple's incremental rate of fault detection. Our technique considers the potential for fault detection and the time overhead of each test case in order to evaluate whether the test suite achieves its potential at the fastest rate possible.

For example, suppose that regression test suite T contains six test cases with the initial ordering for T that contains $\langle T_1, T_2, T_3, T_4, T_5, T_6 \rangle$, as described in Figure 1. For the purposes of motivation, this example assumes a priori knowledge of the faults detected by T in the program P. As shown in Figure 1(a), test case T_1 can find seven faults, $\{\phi_1, \phi_2, \phi_4, \phi_5, \phi_6, \phi_7, \phi_8\}$, in nine minutes, T_2 finds one fault, $\{\phi_1\}$, in one minute, and T_3 isolates two faults, $\{\phi_1, \phi_5\}$, in three minutes. Test cases T_4, T_5, and T_6 each find three faults in four minutes, $\{\phi_2, \phi_3, \phi_7\}$, $\{\phi_4, \phi_6, \phi_8\}$, and $\{\phi_2, \phi_4, \phi_6\}$, respectively.

	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6	ϕ_7	ϕ_8
T_1	X	X		X	X	X	X	X
T_2	X							
T_3	X				X			
T_4		X	X				X	
T_5				X		X		X
T_6		X		X		X		

	# Faults	Time Cost(Mins)	Avg. Faults per Min.
T_1	7	9	0.778
T_2	1	1	1.0
T_3	2	3	0.667
T_4	3	4	0.75
T_5	3	4	0.75
T_6	3	4	0.75

(a)

	Time Limit: 12 minutes			
	Fault (σ_1)	Time (σ_2)	APFD (σ_3)	Intelligent (σ_4)
	T_1	T_2 T_3 T_4 T_5	T_2 T_1	T_5 T_4 T_3
Tot. Faults	7	8	7	8
Tot. Time	9	12	10	11

(b)

Figure 1: Comparison of Prioritizations.

Suppose that the time budget for regression testing is twelve minutes. Because we want to find as many faults as possible early on, it would seem intuitive to order the test cases by only considering the number of faults that they can detect. Without a time budget, the test tuple $\langle T_1, T_4, T_5, T_6, T_3, T_2 \rangle$ would execute. Out of this, only the test tuple $\sigma_1 = \langle T_1 \rangle$ would have time to run when under a twelve minute time constraint and would find only a total of seven faults, as noted in Figure 1(b). Since time is a principal concern, it may also seem intuitive to order the test cases with regard to their execution time. In the time constrained environment, a time-based prioritization $\sigma_2 = \langle T_2, T_3, T_4, T_5 \rangle$ could be executed and find eight defects, as described in Figure 1(b). Another option would be to consider the time budget and fault information together. To do this, we could order the test cases according to the average percent of faults that they can detect per minute. Under the time constraint, the tuple $\sigma_3 = \langle T_2, T_1 \rangle$ would be executed and find a total of seven faults.

If the time budget and the fault information are both considered intelligently, that is, in a way that accounts for overlapping fault detection, the test cases could be better prioritized and thus increase the overall number of faults found in the desired time period. In this example, the test cases would be intelligently reordered so that the tuple $\sigma_4 = \langle T_5, T_4, T_3 \rangle$ would run, revealing eight errors in less time than σ_2. Also, it is clear that σ_4 can reveal more defects than σ_1 and σ_3 in the specified testing time. Finally, it is important to note that the first two test cases of σ_2, T_2 and T_3, find a total of two faults in four minutes whereas the first test case in σ_4, T_5, detects three defects in the same time period. Therefore, the "intelligent" prioritization, σ_4, is favored over σ_2 because it is able to detect more faults earlier in the execution of the tests.

3. TIME-AWARE PRIORITIZATION

The presented prioritization technique uses both testing time and potential fault detection information to intelligently reorder a test suite that adheres to Definition 1. We require that each test in T be independent so that we can guarantee that for all $T_i \in \langle T_1, \ldots, T_n \rangle, \Delta_{i-1} = \Delta_0$, and thus there are no test execution ordering dependencies [14]. This requirement enables our prioritization algorithm to reorder the tests in any sequence that maximizes the suite's ability to isolate defects.

Definition 1. A *test suite* T is a triple $\langle \Delta_0, \langle T_1, \ldots, T_n \rangle, \langle \Delta_1, \ldots, \Delta_n \rangle \rangle$, consisting of an initial test state, Δ_0, a test case sequence $\langle T_1, \ldots, T_n \rangle$ for some initial state Δ_0, and expected test states $\langle \Delta_1, \ldots, \Delta_n \rangle$ where $\Delta_i = T_i(\Delta_{i-1})$ for $i = 1, \ldots, n$.

3.1 Overview

A genetic algorithm is used to solve Problem 1. First, the execution time of each test case is recorded. Because a time constraint could be very short, test case execution times must be exact in order to properly prioritize. Care is taken to ensure that only the execution time of the test case was included in the recorded time and not that of class loading. Timing information additionally includes any initialization and shutdown time required by a test. For example, if T_i requires an open a network connection, this is performed before test execution and is added into T_i's overall execution time. Similarly, the shutdown time of T_i could include a substantial amount of time to store Δ_i for subsequent analysis after testing. Inclusion of initialization and shutdown time is necessary because these operations can greatly increase the execution time required by the test case.

The program P and each $T_i \in \langle T_1, \ldots, T_n \rangle$ are input into the genetic algorithm, along with the following user specified parameters: (i) s, maximum number of candidate test tuples generated during an iteration, (ii) g_{max}, maximum number of iterations, (iii) p_t, percent of the execution time of T allowed by the time budget, (iv) p_c, crossover probability, (v) p_m, mutation probability, (vi) p_a, addition probability, (vii) p_d, the deletion probability, (viii) tc, the test adequacy criterion, and (ix) w, the program coverage weight. We require that all probabilities and percentages $p_t, p_c, p_m, p_a, p_d \in [0, 1]$. The genetic algorithm uses heuristic search to solve Problem 1 and to identify the test tuple $\sigma_{max} \in perms(2^T)$ that is likely to have the fastest rate of fault detection in the provided time limit. In general, any $\sigma_j \in perms(2^T)$ has the form $\sigma_j = \langle T_i, \ldots, T_u \rangle$ where $u \leq n$.

3.2 Genetic Algorithm

The GAPrioritize algorithm in Figure 2 performs test case prioritization on T based on a given time constraint p_t, as desired by Problem 1. On line 1, this algorithm calculates p_t percent of the total time of T, and stores the value in t_{max}, the maximum execution time for a tuple. In the loop beginning on line 3, the algorithm creates a set R_0 containing s random test tuples σ from $perms(2^T)$ that can be executed in t_{max} time. R_0 is the first generation of s potential solutions to Problem 1. Once a set of test tuples is created, coverage information, which is explained in Section 3.2.1, is used by the $CalcFitness(P, \sigma_j, p_t, tc, w)$ method on line 10. The $CalcFitness(P, \sigma_j, p_t, tc, w)$ method is discussed in Section 3.2.2, and it is used to determine the "goodness" of σ_j. To simplify the notation, we denote F_j the fitness value of σ_j, where $F_j = fit(P, \sigma_j, tc, w)$. We also use $F = \langle F_1, F_2, \ldots, F_s \rangle$ to denote the tuple of fitnesses for each $\sigma_j \in R_g, 0 \leq g \leq g_{max}$.

The $SelectTwoBest(R_g, F)$ method on line 11 chooses the two best test tuples in R_g to be elements in the next gener-

Algorithm GAPrioritize$(P, T, s, g_{max}, p_t, p_c, p_m, p_a, p_d, tc, w)$
Input: Program P
Test suite T
Number of tuples to be created per iteration s
Maximum iterations g_{max}
Percent of total test suite time p_t
Crossover probability p_c
Mutation probability p_m
Addition probability p_a
Deletion probability p_d
Test adequacy criteria tc
Program coverage weight w
Output: Maximum fitness tuple $F_{max} \in F$ in set σ_{max}

1. $t_{max} \leftarrow p_t \times \sum_{i=0}^{n} time(\langle T_i \rangle)$
2. $R_0 \leftarrow \emptyset$
3. **repeat**
4. $\quad R_0 \leftarrow R_0 \cup \{CreateRandomIndividual(T, p_t)\}$
5. **until** $|R_0| = s$
6. $g \leftarrow 0$;
7. **repeat**
8. $\quad F \leftarrow \emptyset$
9. $\quad$ **for** $\sigma_j \in R_g$
10. $\quad\quad F \leftarrow F \cup \{CalcFitness(P, \sigma_j, p_t, tc, w)\}$
11. $\quad R_{g+1} \leftarrow SelectTwoBest(R_g, F)$
12. $\quad$ **repeat**
13. $\quad\quad \sigma_k, \sigma_l \leftarrow SelectParents(R_g, F)$
14. $\quad\quad \sigma_q, \sigma_r \leftarrow ApplyCrossover(p_c, \sigma_k, \sigma_l)$
15. $\quad\quad \sigma_q \leftarrow ApplyMutation(p_m, \sigma_q)$
16. $\quad\quad \sigma_r \leftarrow ApplyMutation(p_m, \sigma_r)$
17. $\quad\quad \sigma_q \leftarrow AddAdditionalTests(T, p_a, \sigma_q)$
18. $\quad\quad \sigma_r \leftarrow AddAdditionalTests(T, p_a, \sigma_r)$
19. $\quad\quad \sigma_q \leftarrow DeleteATest(p_d, \sigma_q)$
20. $\quad\quad \sigma_r \leftarrow DeleteATest(p_d, \sigma_r)$
21. $\quad\quad R_{g+1} \leftarrow R_{g+1} \cup \{\sigma_q\} \cup \{\sigma_r\}$
22. $\quad$ **until** $|R_{g+1}| = s$
23. $\quad g \leftarrow g + 1$
24. **until** $g > g_{max}$
25. $\sigma_{max} \leftarrow FindMaxFitnessTuple(R_{g-1}, F)$
26. **return** σ_{max}

Figure 2: The GA Prioritization Algorithm.

ation R_{g+1} of test tuples. The two best tuples are chosen in order to guarantee that R_{g+1} has at least one "good" pair. It is important to carry these highly fit tuples into R_{g+1} as they are in R_g because they are most likely very close to exceeding t_{max}. Any slight change to these test tuples could cause them to require too much execution time, thus invalidating them. Since the GA is trying to identify one particular test tuple, this elitist selection technique ensures that the best tuple in R_g survives on to R_{g+1} [11].

On line 13, $SelectParents(R_g, F)$ identifies pairs of tuples $\{\sigma_k, \sigma_l\}$ from R_g through a roulette wheel selection technique based on a probability proportional to $|F|$. The fitness values are normalized in relation to the rest of the test tuple set by multiplying each $F_j \in F$ by a fixed number, so that the sum of all fitness values equals one [11]. The test tuples are then sorted by descending fitness values, and accumulated normalized fitness values are calculated. A random number $r \in [0, 1]$ is next generated, and the first individual whose accumulated normalized value is greater than or equal to r is selected. This selection method is repeated until enough tuples are selected to fill the set R_{g+1}. Candidate test tuples with higher fitnesses are therefore less likely to be eliminated, but a few with lower fitness have a chance to be used in the test tuple set as well [11].

The $ApplyCrossover(p_c, \sigma_k, \sigma_l)$ method on line 14 may merge the pair $\{\sigma_k, \sigma_l\}$ to create two potentially new tuples $\{\sigma_q, \sigma_r\}$ based on p_c, a user given crossover probability, as explained in Section 3.2.3. Each tuple in the pair $\{\sigma_q, \sigma_r\}$ may then be mutated based on p_m, a user provided mutation probability, as described in Section 3.2.4. Finally, Section 3.2.5 explains how a new test case may be added or deleted from σ_q or σ_r using the $AddAdditionalTests(T, p_a, \sigma_r)$ and $DeleteATest(p_d, \sigma_r)$ methods. The crossover operator exchanges subsequences of the test tuples, and the mutation operator only mutates single elements. Test case addition and deletion are needed because no other operator allows for a change in the number of test cases in a test tuple.

After each of these modifications have been made to the original pair, both tuples σ_q and σ_r are entered into R_{g+1}, as seen on line 21. The same transformations are applied to all pairs selected by the $SelectParents(R_g, F)$ method until R_{g+1} contains s test tuples. In total, g_{max} sets of s test tuples are iteratively created in this fashion as specified in Figure 2 on lines 7–24. When the final set $R_{g_{max}}$ has been created, the test tuple with the greatest fitness, σ_{max}, is determined on line 25. This tuple is guaranteed to be the tuple with the highest fitness out of all g sets of size s.

3.2.1 Test Coverage

Since it is very rare for a tester to know the location of all faults in P prior to testing, the prioritization technique must estimate how likely a test is to find defects, which factors into the function fit of Problem 1. Recall that the function fit yields the fitness of the tuple σ_j based on its potential for fault detection and its time consumption. As it is impossible to reveal a fault without executing the faulty code, the percent of code covered by a test suite is used to determine the suite's potential. In this paper, two forms of test adequacy criteria tc are considered: (i) method coverage and (ii) block coverage [7, 14, 25]. A method is covered when it has been entered. A basic block, a sequence of instructions without any jumps or jump targets, is covered when it is entered for the first time. Because several high level language source statements can be in the same basic block, it is sensible to keep track of basic blocks rather than individual statements at the time of execution [7, 25].

Our genetic algorithm accepts coverage information based on the code covered in an application by an entire test suite. As noted by Kessis et al., this is the form that many tools such as Clover [15], Jazz [20], and Emma [25] produce. The presented prioritization approach can reorder a test suite without requiring coverage information on a per-test basis. While the genetic algorithm handles the common case, its calculation of test tuple fitness could be enhanced to use coverage information on a per-test basis, similar to [7]. Information revealing the impact of a test case T_i's test coverage on any other test case's coverage would also further improve the performance of the fitness function. This and other enhancements are explained in Section 5.

3.2.2 Fitness Function

The $CalcFitness(P, \sigma_j, p_t, tc, w)$ method on line 10 uses $fit(P, \sigma_j, tc, w)$ to calculate fitness. The fitness function, represented by fit in Problem 1, assigns each test tuple a fitness based on (i) the percentage of code covered in P by that tuple and (ii) the time at which each test covers its associated code in P. It is then appropriate to divide fit into two parts such that $fit(P, \sigma_j, tc, w) = F_{pri}(P, \sigma_j, tc, w) + F_{sec}(P, \sigma_j, tc)$. The primary fitness F_{pri} is calculated by measuring the code coverage cc of the entire test tuple σ_j. Because the overall coverage of the test tuple is more important than the order in which the coverage is attained, F_{pri} is weighted by multiplying the percent of code covered by the program coverage weight, w. The selection of w's value should be sufficiently large so that when F_{pri} and $F_{sec} \in [0, 1]$ are added together, F_{pri} dominates the result. Formally, for some $\sigma_j \in perms(2^T)$,

$$F_{pri}(P, \sigma_j, tc, w) = cc(P, \sigma_j, tc) \times w \tag{1}$$

The second component F_{sec} considers the incremental code coverage of the tuple, giving precedence to test tuples whose earlier tests have greater coverage. F_{sec} is also calculated in two parts. First, $F_{s-actual}$ is computed by summing the products of the execution time $time(\langle T_i \rangle)$ and the code coverage cc of the subtuple $\sigma_{j\{1,i\}} = \langle T_1 \dots T_i \rangle$ for each test case $T_i \in \sigma_j$. Formally, for some $\sigma_j \in perms(2^T)$,

$$F_{s-actual}(P, \sigma_j, tc) = \sum_{i=1}^{|\sigma_j|} time(\langle T_i \rangle) \times cc(P, \sigma_{j\{1,i\}}, tc) \tag{2}$$

F_{s-max} represents the maximum value that $F_{s-actual}$ could take (i.e., the value of $F_{s-actual}$ if T_1 covered 100% of the code covered by T.) For a $\sigma_j \in perms(2^T)$,

$$F_{s-max}(P, \sigma_j, tc) = cc(P, \sigma_j, tc) \times \sum_{i=1}^{|\sigma_j|} time(\langle T_i \rangle) \tag{3}$$

Finally, $F_{s-actual}$ and F_{s-max} are used to calculate the secondary fitness F_{sec}. Specifically, for $\sigma_j \in perms(2^T)$,

$$F_{sec}(P, \sigma_j, tc) = \frac{F_{s-actual}(P, \sigma_j, tc)}{F_{s-max}(P, \sigma_j, tc)} \tag{4}$$

As an example of a fitness calculation, let the program coverage weight $w = 100$, P be a program, and tc be a test adequacy criterion (e.g., method or block coverage). Suppose $\sigma_j = \langle T_1, T_2, T_3 \rangle$. Also, assume we have execution times $time(\langle T_1 \rangle) = 5$, $time(\langle T_2 \rangle) = 3$, and $time(\langle T_3 \rangle) = 1$, and test tuple code coverage $cc(P, \sigma_j, tc) = 0.20$. Then,

$$\begin{aligned} F_{pri}(P, \sigma_j, tc, w) &= 0.2 \times 100 \\ &= 20 \end{aligned}$$

F_{sec} next gives preference to test tuples that have more code covered early in execution. To calculate F_{sec}, the code coverages of $\sigma_{j\{1,1\}} = \langle T_1 \rangle$, $\sigma_{j\{1,2\}} = \langle T_1, T_2 \rangle$, and $\sigma_{j\{1,3\}} = \langle T_1, T_2, T_3 \rangle$ must each be measured. Suppose for this example that $cc(P, \sigma_{j\{1,1\}}, tc) = 0.05$, $cc(P, \sigma_{j\{1,2\}}, tc) = 0.19$, and, as already known, $cc(P, \sigma_{j\{1,3\}}, tc) = cc(P, \sigma_j, tc) = 0.20$. F_{sec} is calculated as follows,

$$\begin{aligned} F_{s-actual}(P, \sigma_j, tc) &= (5 \times 0.05) + (3 \times 0.19) + (1 \times 0.20) \\ &= 1.02 \\ F_{s-max}(P, \sigma_j, tc) &= 0.2(5 + 3 + 1) \\ &= 1.8 \end{aligned}$$

$$\begin{aligned} F_{sec}(P, \sigma_j, tc) &= \frac{1.02}{1.8} \\ &= 0.567 \end{aligned}$$

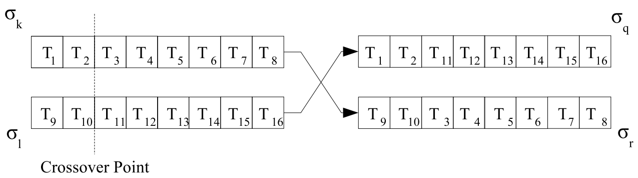

Figure 3: Crossover with Random Crossover Point.

Adding F_{pri} and F_{sec} gives the total fitness value F_j of σ_j. Therefore, in this example,

$$\begin{aligned} fit(P, \sigma_j, tc, w) &= F_{pri}(P, \sigma_j, tc, w) + F_{sec}(P, \sigma_j, tc) \\ &= 20 + 0.567 \\ &= 20.567 \end{aligned}$$

If a test tuple execution time $time(\sigma_j)$ is greater than the time budget t_{max}, F_j is automatically set to -1 by the $CalcFitness(P, \sigma_j, p_t, tc, w)$ method. Because such a tuple violates the execution time constraint, it cannot be a solution and thus receives the worst fitness possible. While a tuple σ_j with $F_j = -1$ could simply not be added to the next generation R_{g+1}, populations with individuals that have a fitness of -1 can actually be favorable. Since the "optimal" test tuple prioritization likely teeters on the edge of exceeding the designated time budget, any slight change to a σ_j with $F_j = -1$ could create a new valid test tuple. Therefore, some σ_j's with $F_j = -1$ are maintained in the next generation. If the test tuple execution time $time(\sigma_j) <= t_{max}$, Equations 1–4 are used.

3.2.3 Crossover

Crossover is used to vary test tuples from one test tuple set to the next through recombination. It is unlikely that the new test tuples after recombination will be identical to a particular parent tuple. As explained in the introduction to Section 3.2, pairs of test tuples $\{\sigma_k, \sigma_l\}$ are selected out of R_g. The $ApplyCrossover(p_c, \sigma_k, \sigma_l)$ method performs crossover to create two potentially new hybrid test tuples from $\{\sigma_k, \sigma_l\}$. First, a random number $r_1 \in [0, 1]$ is generated. If r_1 is less than the user provided value for p_c, the crossover operator is applied. Otherwise, the parent individuals are unchanged and await the next step, mutation. If crossover is to occur, the $ApplyCrossover(p_c, \sigma_k, \sigma_l)$ method on line 14 selects another random number $r_2 \in [0, min(|\sigma_k|, |\sigma_l|)]$ as the crossover point, where $|\sigma_k|$ and $|\sigma_l|$ are the number of test cases in σ_k and σ_l, respectively. The subsequences before and after the crossover point are then exchanged to produce two new offspring, as seen in Figure 3.

If crossover causes two of the same test cases to be in the same test tuple, another random test not in the current tuple is selected from T instead of including the duplicated test case. Although a test case may be run more than once in a test suite with all independent test cases, there is rarely a benefit to executing it again. We assume that multiple executions of a test case will produce the same results and that there is no benefit to multiple execution. However, the $ApplyCrossover$ method can be easily reconfigured to allow for duplicate test cases if there is a testing benefit. If the new tuple already includes all tests, no additions are made.

3.2.4 Mutation

The use of the $ApplyMutation(p_m, \sigma_j)$ method on lines 15 and 16 of Figure 2 also provides a way to add variation to a new population. The new test tuple is identical to the prior parent tuple except that one or more changes may be made to the new tuple's test cases. All test tuples that are selected on line 13 are first considered for crossover. Then they are subject to mutation at each test case position with a small user specified mutation probability p_m. If a random number $r_3 \in [0, 1]$ is generated such that r_3 is less than p_m for test case T_i, a new test not included in the current test tuple is randomly selected from T to replace T_i, as demonstrated for T_2 in Figure 4(a). Figure 4(b) also shows that if there are no unused tests in T when T_9 is chosen for mutation, the test tuple is still mutated. Instead of replacing the test with a random test, the test to be mutated is swapped with the test case that succeeds it.

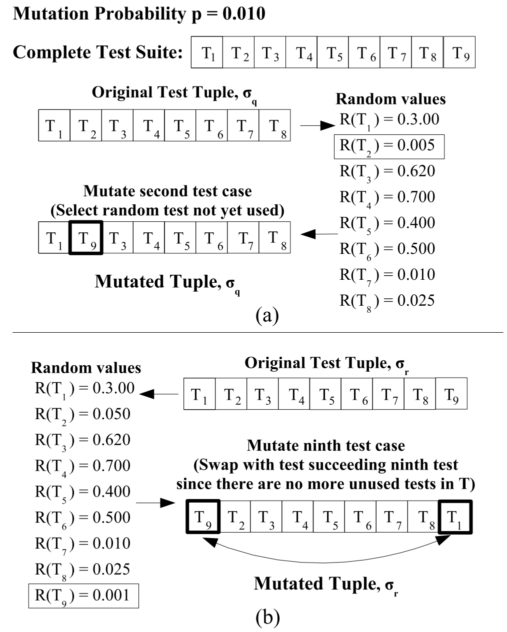

Figure 4: Mutation of a Test Tuple.

3.2.5 Addition and Deletion

Test cases can also be added to or deleted from the test tuples using the $AddAdditionalTests(T, p_a, \sigma_j)$ method on lines 17 and 18 or the $DeleteATest(p_d, \sigma_j)$ method on lines 19 and 20. As in messy genetic algorithms [12], the sets of tuples R_g must be allowed to grow beyond the initial set R_0. Addition and deletion ability permits such growth. While the crossover operator exchanges subsequences, it does not increase the number of test cases within an individual. Similarly, the mutation operator only mutates single elements at each index within the test tuple. Although addition and deletion operations are necessary, they should be performed infrequently so as to not violate the principle of the genetic algorithm. If a random number $r_4 \in [0, 1]$ is generated such that $r_4 < p_a$, a random test case is removed from the individual. If another random number $r_5 \in [0, 1]$ is generated and $r_5 < p_d$, a random test case not yet executed in the individual is added to the end of the test sequence.

4. EMPIRICAL EVALUATION

The primary goal of this paper's empirical study is to identify and evaluate the challenges that are associated with time-aware test suite prioritization. We implemented the

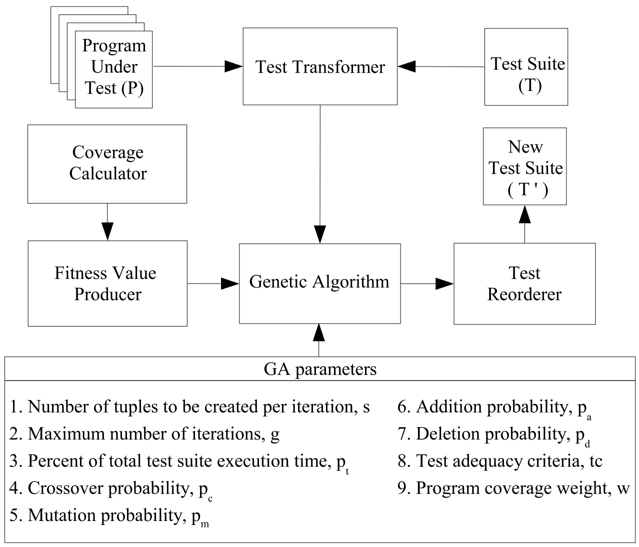

Figure 5: Overview of Prioritization Infrastructure.

approach described in Section 3 in order to measure its effectiveness and efficiency. The goals of the experiment are as follows:

1. Analyze trends in the average percent of faults detected by prioritizations generated using different GA parameter values.
2. Determine if the GA-produced prioritizations, on average, outperformed a selected set of other prioritizations according to the average percent of faults detected.
3. Identify the trade-offs between the configuration of the genetic algorithm and the time and space overheads associated with the creation of the prioritized test suite.

4.1 Experiment Design

All experiments were performed on GNU/Linux workstations with kernel 2.4.20-8, a 1.80 GHz Intel Pentium 4 processor[1] and 1 GB of main memory. The genetic algorithm was implemented in the Java programming language, and it prioritizes JUnit test suites. Figure 5 provides an overview of the test prioritization implementation with edges between interacting components. The test suite is first transformed into a set of test cases and test case execution times. JUnit's test execution framework provides `setUp` and `tearDown` methods that execute before and after a test case and can be used to clear application state, transforming Δ_{i-1} into Δ_0. The `tearDown` operation could also be used to store application state Δ_i prior to deletion. Thus, this paper's assumption of test independence in Section 3 is acceptable. To begin GA execution, the test cases and program information are input into the genetic algorithm along with the other nine parameters for the GA, as depicted in Figure 5.

4.1.1 Implementation

As the genetic algorithm executes, coverage information is gathered at most $|\sigma_j|$ times whenever the fitness of test tuple σ_j is calculated. Fitness is calculated before any test tuple is added to the next test tuple set R_g. Note that for the fitness function calculations, the program coverage weight w was set to 100 for all experiments because this would ensure that $fit(P, \sigma_j, tc, w) \in [0, 100]$. Emma, an open-source toolkit for reporting Java code coverage, is used to calculate test adequacy. Emma can instrument classes for method and block coverage, as described in Section 3.2.1. Coverage statistics are aggregated at method, class, package, and all classes levels for the application under test, and Emma, like most tools, only reports coverage for the entire test tuple. The overall runtime overhead of instrumentation added by Emma is small and the bytecode instrumentor itself is very fast, mostly limited by file input/output (I/O) speed [25].

Coverage calculation is expensive due to the number of times it must be gathered. In order to prevent redundant coverage calculations, memoization is performed [18]. This is especially useful in the calculation of the secondary fitness function F_{sec}, which requires the code coverage information for up to $|\sigma_j|$ subtuples of test cases for each $\sigma_j \in R_g$. Coverage information is used in the fitness function to calculate a fitness value $fit(P, \sigma_j, tc, w)$ for every $\sigma_j \in R_g$. Based on this value, the GA creates g_{max} sets of s test tuples. From the last generated test tuple set, the test tuple with the maximum fitness σ_{max} is returned. As seen in Figure 5, σ_{max} is then used in the new test suite T'.

	`Gradebook`	`JDepend`
Classes	5	22
Functions	73	305
NCSS	591	1808
Test Cases	28	53
Test Exec. Time	7.008 s	5.468 s

Figure 6: Case Study Applications.

4.1.2 Case Study Applications

Figure 6 reviews the two case study applications. `Gradebook` provides functions to perform typical grade book tasks including adding student homework grades, adding lab grades, and calculating curves. `JDepend` is used to traverse directories of Java class files and generate design quality metrics for Java packages. It allows the user to automatically measure the quality of a design in terms of its extensibility, reusability, and maintainability to manage package dependencies effectively [1]. The test cases of `Gradebook` differ from those in `JDepend` in that they are I/O-bound by their frequent interactions with a database. On average, `Gradebook`'s test cases take longer to run, while `JDepend`'s test cases have very short execution times.

4.1.3 Evaluation Metrics

In order to evaluate the effectiveness of a given tuple of test cases, prior knowledge of the faults within the program under test is assumed. A regression test suite prioritization can be empirically evaluated based on the weighted average of the percentage of faults detected over the life of the test suite, or the *APFD* [8, 14]. Preference is given to prioritization schemes that produce test suite tuples with high APFD values. Definition 2 shows how APFD can be calculated using notation introduced in [14].

Definition 2. Let σ_j be the test tuple under evaluation, Φ the set of faults contained in the program under test P, $|\sigma_j|$ the total number of test cases, and $reveal(\phi_f, \sigma_j)$ the position of the first test in σ_j that exposes fault $\phi_f \in \Phi$.

[1] "Intel" and "Pentium" are registered trademarks of Intel Corporation.

Faults	Test Cases						
	T_1	T_2	T_3	T_4	T_5	T_6	T_7
ϕ_1	X			X			
ϕ_2			X				
ϕ_3		X					X
ϕ_4				X			
ϕ_5		X				X	

Table 1: Faults Detected by $T = \langle T_1, \ldots, T_7 \rangle$.

Then APFD$(\sigma_j, P, \Phi) \in [\frac{-1}{2|\sigma_j|}, 1]$ can be defined as

$$\text{APFD}(\sigma_j, P, \Phi) = 1 - \frac{\sum_{\phi_f \in \Phi} reveal(\phi_f, \sigma_j)}{|\sigma_j||\Phi|} + \frac{1}{2|\sigma_j|}.$$

Since σ_j is a subtuple of T, it may contain fewer test cases than T. Moreover, σ_j may not be able to detect all defects. Therefore, we define $reveal(\phi_f, \sigma_j) = |\sigma_j| + 1$ if a fault ϕ_f was not found by any test case in σ_j. This would cause a prioritized test suite tuple that finds few faults to possibly have a negative APFD. Suites finding few faults are penalized in this way. For example, suppose that we have the test suite $T = \langle T_1, \ldots, T_7 \rangle$ and we know that the tests detect faults $\Phi = \{\phi_1, \ldots, \phi_5\}$ in P according to Table 1. Consider the two prioritized test tuples $\sigma_1 = \langle T_3, T_2, T_1, T_6, T_4 \rangle$ and $\sigma_2 = \langle T_1, T_5, T_2, T_4 \rangle$. Incorporating the data from Table 1 into the APFD equation yields

$$\text{APFD}(\sigma_1, P, \Phi) = 1 - \frac{3+1+2+5+2}{5 \times 5} + \frac{1}{2 \times 5} = 0.58$$

and

$$\text{APFD}(\sigma_2, P, \Phi) = 1 - \frac{1+5+3+4+3}{4 \times 5} + \frac{1}{2 \times 4} = 0.325$$

Note that σ_2 is penalized because it fails to find ϕ_2. According to the APFD metric, σ_1 with APFD$(\sigma_1, P, \Phi) = 0.58$ has a better percentage of fault detection than σ_2 with APFD$(\sigma_2, P, \Phi) = 0.325$ and is therefore more desirable.

To evaluate the efficiency of our approach, time and space overheads are analyzed by using a Linux process tracking tool. This tool supports the calculation of peak memory use and the total user and system time required to prioritize the test suite. The time overhead comprises user and system time overheads, where total time equals user time plus system time. User time includes the total time spent in user mode executing the process or its children, whereas system time incorporates the time that the operating system spends performing program services such as executing system calls.

4.2 Experiments and Results

Experiments were run in order to analyze (i) the effectiveness and the efficiency of the parameterized genetic algorithm and (ii) the effectiveness of the genetic algorithm in relation to random, initial ordering, reverse ordering, and fault-aware prioritizations. For all experiments, the resulting test tuples were run on programs that were seeded with faults using Jester [21]. Each source file in `JDepend` and `Gradebook` was seeded with faults multiple times as determined by a mutation configuration file, which contains value substitutions. For example, '`+`' is replaced by '`-`', '`>`' is replaced by '`<`', and so on [21]. 40 errors that could be found by at least one $T_i \in T$ were randomly selected for each application. 25, 50, and 75% of the 40 possible mutations were seeded into each program P, where the larger mutation sets were supersets of the smaller mutation sets.

GA parameters	
P	Gradebook, JDepend
$(g_{max},\ s)$	$(25, 60),\ (50, 30),\ (75, 15)$
p_t	$0.25, 0.50, 0.75$
p_c	0.7
p_m	0.1
p_a	0.02
p_d	0.02
tc	method, block
w	100

Table 2: Parameters used in GA Configurations.

	Block	Method
`Gradebook`	0.638993	0.573982
`JDepend`	0.715984	0.630298

Table 3: Gradebook and JDepend APFD Values.

4.2.1 GA Effectiveness and Efficiency

The first experiment compares the GA execution results and overheads when different GA parameter configurations are used. The genetic algorithms were run with the parameters described in Table 2. In order to run all possible configurations, 36 experiments were completed: 18 using `Gradebook` and 18 using `JDepend`. Thirty-six identical computers were used, each running one trial with one unique configuration. For example, one computer ran a genetic algorithm on the test suite T of the `Gradebook` application calculating $g_{max} = 25$ generations of tuple sets, each of which contained $s = 60$ test tuples. In this configuration, the prioritization was created to be run with $p_t = 0.25$, requiring solution test tuples to execute within 25% of the total execution time of T, and fitness was measured using method coverage.

Effectiveness. As observed in Table 3, on average, the prioritizations created with fitnesses based on block coverage outperformed those developed with fitnesses based on method coverage. In `Gradebook`, use of block coverage produced APFD values 11.32% greater than the use of method coverage, and in `JDepend`, block coverage APFD values increased by 13.59% over method coverage due to block coverage's finer level of granularity.

As the time budget is increased, the APFD values increase as well for both `Gradebook` and `JDepend`, although the amount of increase for a `JDepend` prioritization is less than that of the `Gradebook` prioritizations. This trend, which is due to the nature of the applications' test cases, can be observed in Figure 7(a). The `Gradebook` test cases that find the most faults take a significantly longer time to execute than the test cases of `JDepend`. A few other short tests exist in `Gradebook`'s test suite, but these are not the most crucial test cases in terms of defect detection. Thus, fewer influential `Gradebook` test cases can be executed within a shorter time budget of 25%. When p_t is increased to 50%, the majority of the test cases that find the most faults are able to be run within the time budget, which greatly increases test tuple APFD values. An increase to $p_t = 0.75$ allows for the inclusion of the shorter, less useful test cases.

Since these new test cases are unlikely to find many new faults, changes to the overall APFD of the test tuples are minor. `JDepend`'s test cases all have very short execution times, and many of them cover about the same amount of code. As in `Gradebook`, the longer running `JDepend` test cases generally detect more faults than the shorter tests. However, because the execution time difference between `JDepend` test

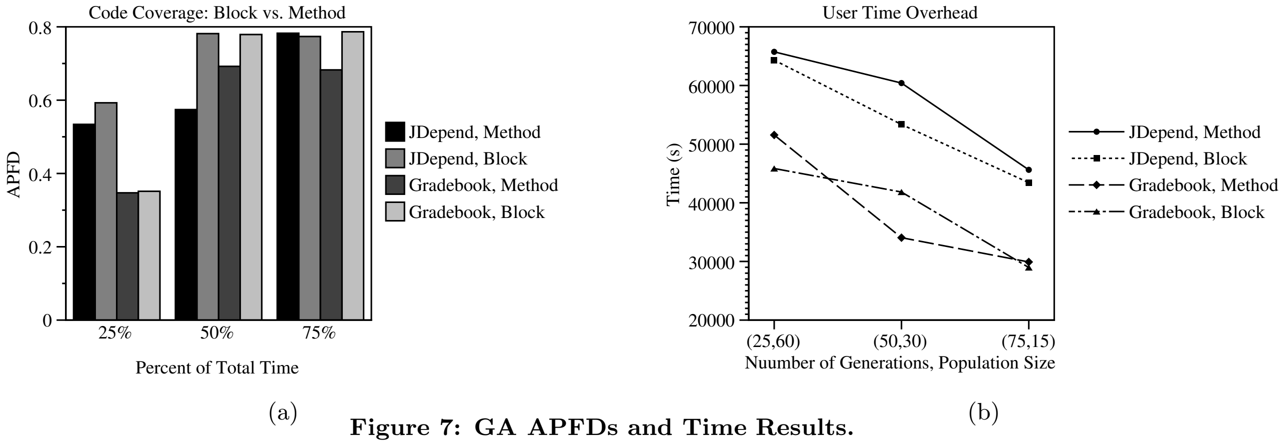

Figure 7: GA APFDs and Time Results.

cases is much smaller than that of `Gradebook` test cases, we observe a less drastic APFD increase in `JDepend`'s prioritizations as p_t grows. This can be seen in Figure 7(a), especially between $p_t = .25$ and $p_t = .50$.

Modification of the number of faults seeded and of (g_{max}, s) led to APFD values that were nearly constant in terms of block and method coverage in the test prioritizations for `Gradebook` and `JDepend`. This provides confidence in the results generated by the genetic algorithm because about the same percentage of defects can be found by any of the prioritizations in spite of how many faults there are or how the GA created the prioritizations. Just as in Table 3 and Figure 7(a), prioritizations based on block coverage slightly outperformed those using method coverage.

Efficiency. Space costs were insignificant, with the peak memory use of all experiments being less than 9344 KB. Most experiments ran with peak memory use of approximately 1344 KB. As is seen in Figure 7(b), the number of generations and the number of tuples per generation greatly impact the time overhead. For example, using block coverage, the genetic algorithm's prioritization of `Gradebook`'s test suite executed for 13.8 hours of user time on average when creating 25 generations of 60 test tuples. On the other hand, if 75 generations with 15 test tuples were created, the GA only required 8.3 hours of user time to execute. Due to memoization, many of the fitness values of test subtuples created in later GA iterations were already recorded from earlier iterations. Thus, the fitness of the subtuples did not need to be calculated again. In the experiments that created 25 generations of 60 test tuples, there is likely to be more genetic diversity. Thus, there are more subsequences that are likely to be found than when prioritization is performed with 75 generations of 15 test tuples. In this circumstance, Emma must be run many more times, and this increases the prioritization time overhead.

The same trend observed in Figure 7(b) occurs when the system time values for `Gradebook` and `JDepend` are compared. For example, a GA executing `Gradebook`'s test suite with 25 generations of 60 test tuples using block coverage requires 13.8 hours of user time and 0.78 hours of system time. However, a GA running `Gradebook`'s test suite with 75 generations of 15 individuals using block coverage required only 8.3 hours of user time and 0.44 hours of system time, a vast improvement over the $(25, 60)$ configuration. Time-aware prioritization of `JDepend` test suites consumed 18.0 hours of user time and 2.1 hours of system time when using the $(25, 60)$ configuration but only 12.5 hours of user time and 1.38 hours of system time using $(75, 15)$. A GA prioritizing the test suite of `JDepend` requires a longer execution time than a GA prioritizing the `Gradebook` test suite due to `JDepend`'s larger test suite. Since there are more possible subtuples that can be generated, on average, the fitness function had to be calculated more times.

As the percent of total test suite execution time was increased for both `Gradebook` and `JDepend`, the number of fitness function calculations also increased due to the fact that more test cases could be included in the prioritizations. Since the fitness function demonstrates itself to be the main bottleneck of the technique, when the genetic algorithm needs to run the fitness function calculator less frequently, less time is required overall to reach a result. This confirms the trend seen in Figure 7(b) as well. We also note that no significant difference was observed between the time overheads of test suite prioritization using block versus method coverage.

Discussion. Results indicate that the APFD values for `Gradebook` were similar irregardless of the value that was used for (g_{max}, s). However, Figure 7(b) reveals that a change in (g_{max}, s) had a significant impact on the time overhead of time-aware test suite prioritization. It is also clear from Figures 7(a) and (b) that on average block coverage outperformed method coverage in relation to APFD while not increasing the time overhead of test suite prioritization. Based on our empirical data, a configuration of GAPrioritize that uses $(g_{max}, s) = (75, 15)$ and $tc = block$ would yield the best results in the shortest amount of time.

Even though the time required to perform test suite prioritization is greater than the execution time of the test suite itself, a given prioritization can be re-used each time a software application is changed. In this way, the cost of the initial prioritization is amortized over the period of time during which the prioritized test suite is used. Furthermore, the experiment results clearly indicate that the prioritized tests will detect more faults earlier than a non-prioritized test suite that was executed over the same extended time period. Even in light of the time required for prioritization, the experiment results suggest that it might be advantageous to use the presented technique when there is a fixed set of short testing time constraints.

4.2.2 Alternative Prioritization Comparisons

Random Prioritizations. According to Do et al., randomly ordered test cases are useful because they redistribute fault-revealing test cases more evenly than origi-

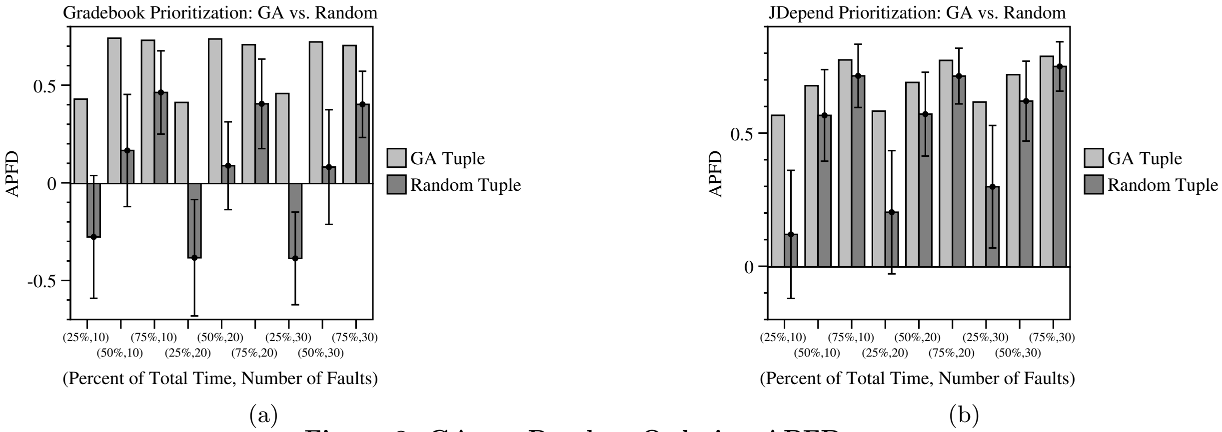

Figure 8: GA vs. Random Ordering APFDs.

nal, untreated test case orderings [7]. Using 18 computers, 10,000 prioritizations were randomly created on each machine. Three elements were varied to create the 18 configurations: (i) the percent of total test suite execution time p_t, (ii) number of faults $|\Phi|$, and (iii) the application P. A building approach was used to create the test tuples. For each prioritization, a test case would be added that was not previously used in the tuple. Randomly selected test cases were incrementally added until the next tuple to be added caused the test tuple to exceed t_{max}. Each of the generated prioritizations therefore would nearly fill the time limit but would not go over that limit.

Success of the genetic algorithm prioritizations is measured by comparing the prioritized test suites' APFD values to the APFD values of the other reorderings. A comparison between the APFD values, percent of total test suite execution time, and the number of faults seeded can be observed for GA-produced prioritizations in relation to randomly produced permutations in Figure 8. The comparison is described for the `Gradebook` application in Figure 8(a) and for the `JDepend` application in Figure 8(b). Each bar in the graphs represents the average of the APFD values of 10,000 random prioritizations, and the error bar represents the standard deviation from the mean APFD.

In the case of `Gradebook`, the GA-produced prioritizations performed extremely well in comparison to the randomly produced prioritizations. All APFD values from prioritizations based on the `Gradebook` application fell more than one standard deviation above the mean of the randomly produced prioritizations. Because the tests that detect the most faults in `Gradebook` are longer in execution time and fewer in number with regard to the other test cases, there was a greater probability of creating weak test tuples using random prioritization. As depicted in Figure 8(a), the test tuples executing with $p_t = 0.25$ had negative APFD values on average because they were only able to find a few of the seeded faults. Thus, there is a clear benefit to using intelligently prioritized tests instead of random prioritizations.

In the case of `JDepend`, the GA-produced prioritizations on average did not perform as well as the prioritizations of the test suite for `Gradebook`. This was anticipated because of the nature of `JDepend`'s test cases, which are much more interchangeable with respect to fault detection potential than those of `Gradebook`. As can be seen in Figure 8(b), on average, all GA-produced prioritizations that ran within 25% of the total test suite execution time had APFD values more than one standard deviation above the mean APFD value of the same set of randomly produced prioritizations. GA-produced prioritizations that ran within 50% and 75% of the total test suite execution time also had APFD values within one standard deviation above the mean of the randomly produced prioritizations.

Because the test cases of `JDepend` all have about the same adequacy and take around the same amount of time to execute, many different test subtuples have the same APFD. As observed in Figure 8(b), the average APFD for test tuples that are allowed to run in 75% of the total test suite execution time is likely to be closer to the best possible APFD value than that of test subtuples that are allowed to run in only 25% of the total test suite execution time. In other words, it is much easier for random prioritizations to have high APFD values when more of the original test suite can be run, particularly in the case of `JDepend`.

Overall, the GA-produced prioritizations performed extremely well in comparison to randomly generated prioritizations. Nearly all results were more than one standard deviation from the mean APFD values calculated for prioritizations that were produced randomly. All results had APFD values that were greater than the mean APFD values of random prioritizations. Note also from Figure 8(a) and Figure 8(b) that APFD values for the percent of total test suite execution time groups are all very similar. This again provides confidence in the results generated by the genetic algorithm because about the same percentage of faults can be found by any of the prioritizations in spite of how many defects there are or how the GA created the prioritizations.

Additional Prioritizations. Two simple forms of prioritization include those that execute test cases in the order in which they are written or the reverse of that order. Table 4 compares GA-produced prioritizations to initial and reverse ordering prioritizations. The genetic algorithm produced prioritizations that were up to a 120% improvement over initial orderings. For example, `Gradebook`'s initial tuple created using $p_t = 0.25$ and $|\Phi| = 30$ had APFD $= -0.892$ whereas the associated intelligently produced tuple had APFD $= 0.457$, as shown in Table 4. The time-aware prioritizations were also an improvement over all reverse ordering prioritizations in both `JDepend` and `Gradebook`.

Of course fault-aware prioritization cannot be performed in practice, but these reorderings are useful for comparison purposes. The fault-aware prioritizations were constructed by calculating the APFD for each test case and ordering the test cases accordingly until the addition of the next test would exceed the time limit. The `JDepend` GA-produced

p_t	$\|\Phi\|$	Initial Gradebook	Reverse Gradebook	Fault Aware Gradebook	GA Gradebook	Initial JDepend	Reverse JDepend	Fault-Aware JDepend	GA JDepend
0.25	10	-0.6	-0.233	0.66	0.428	0.525	-0.300	-0.05	0.567
0.25	20	-0.863	-0.208	0.72	0.412	0.478	-0.275	0.05	0.649
0.25	30	-0.892	-0.006	0.453	0.457	0.473	-0.133	0.083	0.617
0.50	10	-0.042	0.16	0.869	0.741	0.873	0	0.2	0.678
0.50	20	-0.192	0.167	0.873	0.737	0.819	0.013	0.175	0.690
0.50	30	-0.308	0.284	0.782	0.722	0.842	0.1	0.208	0.719
0.75	10	0.314	0.478	0.906	0.73	0.878	0.492	0.59	0.775
0.75	20	0.124	0.433	0.926	0.707	0.826	0.608	0.283	0.773
0.75	30	0.049	0.516	0.88	0.703	0.848	0.534	0.25	0.788

Table 4: Initial, Reverse, Fault-Aware, and Genetic Algorithm Prioritization APFDs.

test tuples performed much better than the fault-aware prioritizations described in Table 4. This is likely because most of the test cases in `JDepend` cover the same code segments. While the genetic algorithm identifies the overlap in test code coverage (and thus the fault detection potential), the fault-aware prioritization does not. Thus, the GA produced markedly better results for `JDepend`.

On the other hand, the prioritizations produced by the GA for `Gradebook` were not quite as good at finding defects quickly when compared to the fault-aware prioritizations for `Gradebook`, as noted in Table 4. This is because `Gradebook`'s test cases have little coverage overlap, causing few test cases to detect the same faults. Because the fault-aware prioritization technique has actual knowledge of all faults, it could specifically organize the test cases to best find the known faults without concern for fault detection overlap. Although the genetic algorithm's results did not have as high of APFD values in this case, its prioritizations are more general because they are not based on specific faults. Thus, they have the potential to perform well no matter where the defects in the code may exist.

4.3 Threats to Validity

Internal Validity. Threats to internal validity concern the factors that might have impacted the measured variables that were described in Section 4.1.3. The first threat to internal validity is related to potential faults within our test prioritization infrastructure. We controlled this threat by using a test coverage monitoring tool, Emma, and a test suite execution framework, JUnit, that have been extensively tested and are frequently used by software testing practitioners. We have a confidence in the correctness of these tools and we judge that they did not negatively impact the validity of our empirical study. We also tested each component of the genetic algorithm (e.g., the fitness function and the mutation and crossover operators) in isolation and we monitored the execution of the GA over an extended period of time in order to further verify that these components produced the correct output. The final threat to internal validity is related to the fact that we seeded a relatively small number of application faults using a mutation testing tool. However, it is important to observe that the experiment results in Section 4.2 indicate that the measured variables are not overly sensitive to the number of seeded faults.

External Validity. Threats to external validity would limit the generalization of the experiment results to new case study applications. External validity threats include (i) the size of the selected case study applications and (ii) the number of case study applications used in the empirical study. Yet, if we compare our case study applications to those that were used in other studies by Tonella [27] (mean size of 607.5), McMinn and Holcombe [17] (mean size of 119.2), and the Siemens application suite [24] (mean size 354.4), the average size of our programs and test suites (mean size 860.5) is greater than the size of the other applications. Since both Tonella and McMinn and Holcombe report a lines of code (LOC) size metric instead of NCSS, it is possible that the difference in case study application size is even greater.

We did not incorporate additional applications into the empirical study because `Gradebook` and `JDepend` served to identify the relevant challenges associated with time-aware test prioritization. Of course, future experiments with more large-scale case study applications will serve to confirm the results in Section 4.2. Finally, the use of a mutation testing tool to automatically generate the seeded faults is an additional threat to external validity. Even though this threat can be controlled by conducting additional case studies with a greater number of both seeded and real-world faults, it is interesting to remark that preliminary empirical studies suggest that mutation faults do correspond to the faults found in real-world applications [4].

Construct Validity. Threats to construct validity concern whether the experiment metrics accurately reflect the variables that the experiments were designed to measure. As discussed in Section 4.1.3, the APFD metric is our primary measure of the effectiveness of a test prioritization. Our formulation of APFD is limited because it does not incorporate the severity of the faults that are isolated by a test case. Also, the APFD metric does not measure whether a test can effectively support other important software maintenance activities such as debugging and automatic fault localization. However, it is important to note that recent empirical studies of test prioritization techniques also use the APFD metric [7, 13, 24]. Finally, it is unlikely that our metrics for time and space overhead introduce threats to construct validity since they directly measure values that would be useful to software testing practitioners.

5. ENHANCEMENTS

The time overhead results discussed in Section 4.2.1 should be evaluated in the context of other techniques that use genetic algorithms to approximate NP-complete problems. For example, Kulkarni et al. report that the use of a standard GA to identify compiler optimization sequences required on average, across six programs, a total of 26.68 hours [16]. The empirical study in this paper reveals that the calculation of $F_{s-actual}$ and the consideration of the incremental coverage of each test sub-tuple improves the effectiveness of the prioritized test suite and increases the time overhead of prioritization. The calculation of $F_{s-actual}$ is necessary because it accommodates the coverage overlap between the individual tests. If T was initially reduced in order to remove the majority of test coverage overlap and coverage was reported on a per-test basis, T_{reduce} could be prioritized using the presented approach and the modified fitness function $\mathcal{F}it$ described in Equation 5. The fitness of test tuple σ_j could be calculated with $\mathcal{F}it$ whenever $time(\sigma_j) \leq t_{max}$ and if σ_j exceeds the testing time limit we require that $\mathcal{F}it(P, \sigma_j, tc) = 0$.

$$\mathcal{F}it(P, \sigma_j, tc) = \sum_{i=1}^{|\sigma_j|} \frac{cc(P, T_i, tc)}{time(\langle T_i \rangle)} \tag{5}$$

Since $\mathcal{F}it$ does not need to consider the incremental code coverage of the σ_j that was derived from T_{reduce}, we judge that it will significantly decrease the time overhead of GAPrioritize. The performance of the baseline fitness function fit can also be improved if test suite reduction is not desirable because of concerns about diminishing the suite's potential to reveal faults. For example, during the calculation of $F_{s-actual}$, each computation of $time(\langle T_i \rangle) \times cc(P, \sigma_{j\{1,i\}}, tc)$ could be performed on a separate computer. If test execution histories are available, fit could be modified to focus on tests that have recently revealed faults. When test prioritization has been previously performed, the fitness of test tuple σ_{max} can be recorded and all subsequent executions of GAPrioritize could be terminated when the algorithm generates a tuple that has fitness greater than the prior σ_{max}.

6. RELATED WORK

There are many existing approaches to regression test prioritization that focus on the coverage of or modifications to the structural entities within the program under test [7, 8, 24, 26]. Yet, none of these prioritization schemes explicitly consider the testing time budget like the time-aware technique presented in this paper. Similar to our approach, Elbaum et al. and Rothermel et al. focus on general regression test prioritization and the identification of a single test case reordering that will increase the effectiveness of regression testing over many subsequent changes to the program [8, 24]. Do et al. present an empirical study of the effectiveness of test prioritization in a testing environment that uses JUnit [7]. This paper is also related to our work because Do et al.'s prioritization technique uses coverage information at the method and block levels. Recent research by Memon et al. implicitly assumes that testing is constrained by the amount of time available in an evening [19]. While their infrastructure is highly automated, it does not directly consider the time constraint and thus cannot guarantee that testing will complete in the allotted time.

7. CONCLUSIONS AND FUTURE WORK

This paper describes a time-aware test suite prioritization technique. Experimental analysis demonstrates that our approach can create time-aware prioritizations that significantly outperform other prioritization techniques. For one case study application, our technique created prioritizations that, on average, had up to a 120% improvement in APFD over other prioritizations. This paper identifies and evaluates the challenges associated with time-aware prioritization. We also explain ways to reduce the time overhead of prioritization. In future work, we intend to examine the enhancements to our approach that are discussed in Section 5. These improvements to our algorithm have the potential to increase the applicability of the presented technique.

8. REFERENCES

[1] `http://www.clarkware.com/software/JDepend.html`.

[2] `http://monetdb.cwi.nl/`.

[3] `http://www.planet-lab.org/`.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proc. of 27th ICSE*, pages 402–411, 2005.

[5] G. Antoniol, M. D. Penta, and M. Harman. Search-based techniques applied to optimization of project planning for a massive maintenance project. In *Proc. of the 21st ICSM*, pages 240–249, Washington, DC, USA, 2005.

[6] P. Chu and J. Beasley. A genetic algorithm for the multidimensional knapsack problem. *Journal of Heuristics*, 4(1):63–86, 1998.

[7] H. Do, G. Rothermel, and A. Kinneer. Empirical studies of test case prioritization in a JUnit testing environment. In *Proc. of 15th ISSRE*, pages 113–124, 2004.

[8] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, 2002.

[9] D. Fatiregun, M. Harman, and R. M. Hierons. Evolving transformation sequences using genetic algorithms. In *Proc. of 4th SCAM*, pages 66–75, 2004.

[10] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[11] D. E. Goldberg. *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Addison-Wesley, Reading, MA, 2002.

[12] D. E. Goldberg, B. Korb, and K. Deb. Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3(5):493–530, 1989.

[13] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. *ACM Trans. on Softw. Eng. and Meth.*, 10(2), 2001.

[14] G. M. Kapfhammer. Software testing. In *The Computer Science Handbook*, chapter 105. CRC Press, Boca Raton, FL, second edition, 2004.

[15] M. Kessis, Y. Ledru, and G. Vandome. Experiences in coverage testing of a Java middleware. In *Proc. of 5th SEM*, pages 39–45, 2005.

[16] P. A. Kulkarni, S. R. Hines, D. B. Whalley, J. D. Hiser, J. W. Davidson, and D. L. Jones. Fast and efficient searches for effective optimization-phase sequences. *ACM Trans. Archit. Code Optim.*, 2(2):165–198, 2005.

[17] P. McMinn and M. Holcombe. Evolutionary testing of state-based programs. In *Proc. of GECCO*, pages 1013–1020, 2005.

[18] P. McNamee and M. Hall. Developing a tool for memoizing functions in C++. *ACM SIGPLAN Not.*, 33(8):17–22, 1998.

[19] A. Memon, I. Banerjee, N. Hashmi, and A. Nagarajan. DART: A framework for regression testing "nightly/daily builds" of GUI applications. In *Proc. of ICSM*, 2003.

[20] J. Misurda, J. Clause, J. L. Reed, P. Gandra, B. R. Childers, and M. L. Soffa. Jazz: A tool for demand-driven structural testing. In *Proc. of 14th CC*, 2005.

[21] I. Moore. Jester- a JUnit test tester. In *Proc. of 2nd XP*, pages 84–87, 2001.

[22] R. P. Pargas, M. J. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Soft. Testing, Verif. and Rel.*, 9(4):263–282, 1999.

[23] C. Poole and J. W. Huisman. Using extreme programming in a maintenance environment. *IEEE Softw.*, 18(6):42–50, 2001.

[24] G. Rothermel, R. J. Untch, and C. Chu. Prioritizing test cases for regression testing. *IEEE Trans. on Softw. Eng.*, 27(10):929–948, 2001.

[25] V. Roubtsov. Emma: a free java code coverage tool. `http://emma.sourceforge.net/index.html`, March 2005.

[26] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *Proc. of ISSTA*, pages 97–106, 2002.

[27] P. Tonella. Evolutionary testing of classes. In *Proc. of ISSTA*, pages 119–128, 2004.