Efficient Time-Aware Prioritization with Knapsack Solvers

Sara Alspaugh†, Kristen R. Walcott†, Michael Belanich‡,
Gregory M. Kapfhammer‡ and Mary Lou Soffa†

†Department of Computer Science
University of Virginia
{alspaugh, walcott, soffa}@cs.virginia.edu

‡Department of Computer Science
Allegheny College
{belanim, gkapfham}@allegheny.edu

ABSTRACT

Regression testing is frequently performed in a time constrained environment. This paper explains how 0/1 knapsack solvers (e.g., greedy, dynamic programming, and the core algorithm) can identify a test suite reordering that rapidly covers the test requirements and always terminates within a specified testing time limit. We conducted experiments that reveal fundamental trade-offs in the (i) time and space costs that are associated with creating a reordered test suite and (ii) quality of the resulting prioritization. We find knapsack-based prioritizers that ignore the overlap in test case coverage incur a low time overhead and a moderate to high space overhead while creating prioritizations exhibiting a minor to modest decrease in effectiveness. We also find that the most sophisticated 0/1 knapsack solvers do not always identify the most effective prioritization, suggesting that overlap-aware prioritizers with a higher time overhead are useful in certain testing contexts.

Categories and Subject Descriptors

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

General Terms

Experimentation, Algorithms, Verification

Keywords

test prioritization, coverage testing, knapsack problem

1. INTRODUCTION

As a software system is developed and maintained, it is selectively retested to ensure that the addition of functionality and correction of faults does not introduce new errors. This regression testing process is necessary, yet it often incurs high time overhead, since large test suites can take days or weeks to run [9]. In order to reduce this cost, testers can either choose to execute a subset of tests or order the test cases according to some priority measure. However, these approaches often ignore the time constraints of the testing environment. For an example of these time constraints, many developers choose to do nightly builds of the application [7], and this limits the amount of time in which a test suite has to run. With this in mind, if test cases are prioritized according to greatest fault-detection capacity per unit of execution time, then more faults will be detected at the beginning of the test suite's execution [12]. Thus, whenever testing time constraints are known, this type of prioritization will detect more faults than would otherwise be possible.

For example, suppose that test suite $T = \langle T_1, T_2, T_3, T_4 \rangle$ and each test case detects four, one, two, and six faults, respectively (i.e., the test suite isolates a total of thirteen defects). If each test consumes an equal amount of execution time and the testing time constraint only allows half of the tests to run, then it is better to use the reordered test suite $T' = \langle T_4, T_1, T_3, T_2 \rangle$. This is due to the fact that T results in the detection of five faults while T' can detect ten faults in the same period of time. Thus, as this example demonstrates, test prioritization can achieve a compromise between detecting as many faults as possible during testing and executing the test suite in less time.

This paper empirically evaluates the efficiency and effectiveness of techniques that construct a time-aware test suite prioritization. In lieu of directly measuring fault detection effectiveness, we use surrogate metrics such as code coverage, coverage preservation, and order-aware code coverage, as discussed in Section 3. We selected these metrics because experiments by Hutchins et al. demonstrate that high coverage tests suites are good at detecting faults [5]. As such, we measure the *code coverage* of both the test suite and the time-aware prioritization. Code coverage is the percentage of the structural elements within the program (e.g., basic blocks or methods) that are executed during testing, with a high value indicating strong effectiveness. *Coverage preservation* is the proportion of code covered by a prioritization to the code covered by the original test suite. *Order-aware coverage*, as defined in [12], gives preference to prioritizations that cover a greater amount of code earlier on in the execution phase of that prioritization.

As long as the overlap in code coverage is not considered, the problem of constructing the best time constrained test suite is equivalent to the 0/1 knapsack problem, which can be described in the following manner: given a knapsack with fixed capacity and a set of distinct items each with its own value and weight, find the maximum cumulative value of items that can fit in the knapsack such that the sum of

WEASELTech'07, November 5, 2007, Atlanta, Georgia, USA.

the item weights in the knapsack does not exceed the knapsack's capacity [6]. An extended version of the 0/1 knapsack problem occurs if code coverage overlap is considered, as in Walcott et al. [12]. In summary, the important contributions of this paper are:

1. A description of an approach to time-aware test suite prioritization that uses traditional 0/1 knapsack solvers (Section 2).
2. An empirical study that measures the efficiency and effectiveness of test suite prioritization algorithms (Sections 3 and 4) according to:
   (a) The time and space overheads associated with prioritization techniques that ignore the overlap in test coverage versus those that include it.
   (b) The effectiveness of the prioritization schemes as measured by the code coverage, coverage preservation, and order-aware coverage metrics.

2. TIME-AWARE PRIORITIZATION

To address the problem of constructing a time-constrained test suite prioritization, this paper uses traditional techniques to solve the 0/1 knapsack problem. We define a test suite T as a tuple of test cases T_i from $i = 1$ to $i = n$ as $\langle T_1, T_2, \ldots, T_n \rangle$. The prioritization of T is denoted T'. In the context of the 0/1 knapsack problem, the maximum amount of time within which a prioritized test suite must run is the maximum capacity of the knapsack, the test cases are the knapsack items, each test case's execution time is its weight, and its percentage of code coverage is its value. When these values are passed into a 0/1 knapsack algorithm, the output is a final solved knapsack, namely, a prioritization that fits within the desired time limit.

After obtaining prioritization results, we next compare each prioritization to the results obtained by an algorithm that considers an extended 0/1 knapsack problem. This algorithm takes into account the fact that as test cases are added to a prioritization, their total value, or the percentage of program code covered by these test cases, is not cumulative. Rather, code coverage accumulates as test cases that cover code not already covered by test cases in the prioritization are added [12].

As defined by Kellerer et al., the 0/1 knapsack problem can be defined formally in terms of test suite prioritization in the following manner [6]:

$$\begin{array}{ll} \text{Maximize:} & \sum_{i=1}^{n} c_i x_i \\ \text{Subject to:} & \sum_{i=1}^{n} t_i x_i \leq t_{max},\ x_i = 0 \text{ or } 1, \end{array}$$

where c_i is the code coverage, t_i is the execution time of test case T_i, and t_{max} is the maximum time allowed for the execution of the prioritization.

The extended version of the 0/1 knapsack problem occurs if we take into account the fact that test cases may cover the same portion of code. Executing two test cases that cover the same requirements does not increase the overall coverage of the test suite under examination. Thus, we modify the knapsack problem statement such that the value of a test case is dependent upon the current contents of the knapsack. The value of each remaining test case must be adjusted accordingly. For example, if a test case that covers method M is placed in a prioritization, then other test cases that also cover M would add less value to the test case ordering if they were added. The overall value of a prioritization takes into account both the test case order and the test case code coverage overlap.

ta	T_1	T_2	T_3	*copt*
0	0	0	0	0
1	0	0	0	0
2	0	0	0	0
3	$2{+}copt_0$	0	0	2
4	$2{+}copt_1$	$1{+}copt_0$	0	2
5	$2{+}copt_2$	$1{+}copt_1$	$3{+}copt_0$	3

Figure 1: Generalized Tabular Example.

2.1 Knapsack Solvers as Prioritizers

The 0/1 knapsack problem is an NP-complete problem [3]. There are a number of algorithms that approximate the optimal solution to this problem, which vary in complexity and optimality. Seven knapsack algorithms are used in this paper and are described in terms of the test suite prioritization problem as follows:

Random: While the total execution time of the prioritization is less than or equal to the maximum allowed time limit, select a test case T_i randomly from the set of unused test cases and add it to the prioritization. If the addition of a test case causes the maximum time limit t_{max} to be exceeded, remove that test case and return the remaining test case ordering.

Greedy by Ratio: For each test case T_i, calculate the code-coverage-to-execution-time ratio, $\frac{c_i}{t_i}$. Sort the test cases in descending order according to this ratio, then successively place test cases from this ordering into the prioritization until the addition of the next test would cause the maximum time limit to be exceeded. **Greedy by Value** and **Greedy by Weight** are performed similarly, except code coverage and test execution time are used in place of the code-coverage-to-execution-time ratio, respectively.

Dynamic Programming: Divide the problem into subproblems, and solve each piece separately, storing the answers so as to avoid repeatedly solving the same problem [6]. The total code coverage of the prioritization for i test cases and t_{max} execution time is zero if there are no test cases in the solution or if t_{max} is zero. Otherwise, the solution for i test cases and t_{max} time either includes the ith test case or does not. In the first case, the total code coverage of the prioritization for i test cases and t_{max} time is equal to the total code coverage of the prioritization for $i-1$ test cases and $t_{max} - t_i$ time plus the code coverage of the ith test case, c_i. In the second case, the total code coverage is equal to the code coverage of the prioritization for $i-1$ test cases and t_{max} time. The best solution is selected.

Generalized Tabular: Like dynamic programming, solve subproblems of the main problem using a large table [4]. The table has $t_{max} + 1$ rows and $n + 1$ columns, where n is the number of test cases. Each row i represents a problem with maximum time limit ta_i for values 0 to t_{max}. The last item in each row is the optimal coverage solution, denoted $copt_{ta_i}$, for that problem. An example with test cases $\langle T_1, T_2, T_3 \rangle$ having coverages 2, 1, 3, and times 3, 4, 5, respectively, and $t_{max} = 5$ is shown in Figure 1. As seen in the example, the optimal coverage for each time limit is stored in the last column of the table, while the rows are numbered by the time limits. The other columns each correspond to a test case T_j. Element a_{ta_i, T_j}, as shown in Figure 1, is equal to $c_j + copt_{ta_i - t_j}$ if T_i can be added within the time limit

Test Case	T_1	T_2	T_3	T_4	T_5	T_6
coverage	4	5	2	6	8	1
time	105	60	60	95	225	32
c/t ratio	0.0381	0.0833	0.0333	0.0632	0.0356	0.0313

Figure 2: Example Test Cases.

Comparison	Inequality	Action
Compare T_2 and T_4	$5 \times \lfloor \frac{445}{60} \rfloor \geq 6 \times (\frac{445}{95})$ $35 \geq 28.1053$	Add T_2 to T' Time left = 385
Compare T_4 and T_1	$6 \times \lfloor \frac{385}{95} \rfloor \geq 4 \times (\frac{385}{105})$ $24 \geq 14.6667$	Add T_4 to T' Time left = 290
Compare T_1 and T_5	$4 \times \lfloor \frac{290}{105} \rfloor \geq 8 \times (\frac{290}{225})$ $8 \not\geq 10.3331$	Not conclusive Time left = 290

Figure 3: Scaling Heuristic Example.

ta_i, and 0 otherwise. After the table is complete, the full solution is recovered by working backward and retracing the steps taken to compute the elements of the table.

Core: Create a "core" solution using a subset of test cases, then use this core to find a solution to the overall prioritization problem [8]. First, find a good solution (the core solution) using the greedy by weight algorithm. Then, using the dynamic programming algorithm, try to find a better solution by replacing each test case in the core solution with another unused test case.

In addition to examining the efficiency and effectiveness of these seven techniques, each algorithm is examined in conjunction with the use of *scaling*. When scaling, the problem is reduced by means of a theorem described by Gossett [4]. The version specific to the prioritization problem addressed in this paper follows.

Suppose that for a prioritization with maximum allowed execution time t_{max}, there are n test cases in the test suite. Denote the code coverage values of the test cases by $c_1, c_2, \ldots, c_n$ and the execution time of the test cases by $t_1, t_2, ..., t_n$. Assume the test cases have already been ordered such that $\frac{c_1}{t_1} \geq \frac{c_2}{c_2} \geq \ldots \geq \frac{c_n}{t_n}$. If $c_1 \times \left\lfloor \frac{t_{max}}{t_1} \right\rfloor \geq c_2 \times \left(\frac{t_{max}}{t_2} \right)$, then it is possible to find an optimal knapsack solution that includes T_1.

To perform scaling, order all test cases by their code-coverage-to-execution-time ratios, as indicated by the theorem. Check if the inequality $c_1 \times \left\lfloor \frac{t_{max}}{t_1} \right\rfloor \geq c_2 \times \left(\frac{t_{max}}{t_2} \right)$ holds. If it does, put T_1 in the prioritization and subtract the execution time t_1 of T_1 from the maximum execution time t_{max}. Now consider the prioritization with maximum execution time $t_{max} - t_1$ for the remaining list of test cases, with T_2 now occurring first in the list, and so on. Continue down the list in this manner until the inequality ceases to hold—let us say that this occurs at T_i—and then stop. The $i - 1$ test cases placed in the knapsack through this process are guaranteed to be part of an optimal solution for the prioritization with maximum allowed execution time t_{max}. Finish by using any of the aforementioned techniques on the remaining unselected test cases. These test cases have maximum allowed execution time $t_{max} - \sum_{j=1}^{i-1} t_j$. The test cases that will be in the final solution for the prioritization with maximum allowed execution time t_{max} will be those in the solution for the prioritization with maximum allowed execution time $t_{max} - \sum_{j=1}^{i-1} t_j$ plus those determined to be part of the optimal solution by the scaling heuristic.

For example, suppose there are six test cases, with code coverages and execution times as shown in Figure 2, and a maximum execution time of 445 units. First, the test cases are ordered according to their code-coverage-to-execution-time ratio to yield $\langle T_2, T_4, T_1, T_5, T_3, T_6 \rangle$. Then comparisons are performed, as shown in Figure 3. After the last comparison fails, the heuristic no longer yields any information, so the rest of the problem is solved using one of the seven algorithms described in this section.

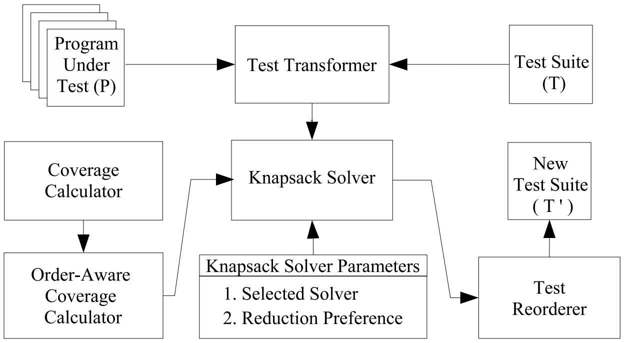

Figure 4: Overview of Prioritization Infrastructure.

3. EXPERIMENT GOALS AND DESIGN

The goals of this experiment are:

1. Measure empirically, using two case studies, the efficiency of seven knapsack algorithms used in prioritization, each with and without the use of a scaling heuristic, in terms of time and memory overhead.

2. Record, graph and analyze the effectiveness of each of these algorithms with and without scaling in terms of three coverage-based metrics: code coverage, coverage preservation, and order-aware coverage.

All of the algorithms described in this paper were implemented in Java and were used to prioritize JUnit test cases from two case study applications, described below. The prioritizations were performed on a dual-core AMD Opteron Processor, each core being 1.8 GHz, running the Fedora Core 3 GNU/Linux operating system with 2 GB of main memory and 2048 MB maximum heap size. To perform a prioritization, first the execution time and code coverage information of each test case in the test suite is recorded. From this information, a set of knapsack items is created. Next these items are used as input to the knapsack algorithms. Each algorithm returns a list of test cases representing the final test suite prioritization, as depicted in Figure 4. As the algorithms run, time overhead and memory information is gathered. Afterwards, code coverage, coverage preservation, and order-aware coverage information are calculated for each prioritization. The time, memory, and coverage information is used to compare the algorithms and examine the key trade-offs.

In order to measure the effectiveness of these algorithms in test suite prioritization, the test suites of two case study programs were used: `JDepend` and `GradeBook`. `JDepend` is a tool for creating design quality metrics for Java packages in terms of extensibility, reusability, and maintainability. `GradeBook` is a program that provides functions to perform tasks associated with creating and maintaining a grade book system for a course. Figure 5 gives information regarding each application and their test suites. The longer average execution

	`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 5: Case Study Applications.

time for the `GradeBook` test cases is due to the fact that its test cases involve frequent I/O interactions with a database.

3.1 Evaluation Metrics

In order to measure the effectiveness of these algorithms, three metrics were used: code coverage, coverage preservation, and order-aware coverage. These were used despite the fact that ideally, the effectiveness of a test suite prioritization would be based on the average percentage of faults it detected given a time constraint. However, since the nature and location of faults are unknown and unique to each program, it is not possible to calculate this metric, known as APFD [1], unless faults are artificially seeded into a program. While this can be a useful way of empirically judging the effectiveness of a prioritization, it runs the risk of not being representative of the type and number of faults that occur in real-world applications. Therefore, coverage information, which has been shown to be highly predictive of fault-detection potential, is used [12].

Code coverage, denoted $cc(P, T)$, where P is the program being tested, is a measure of the percentage of program source statements that are executed when the prioritized test suite is run. There are several different levels of granularity at which code coverage can be measured; this paper uses block coverage, where a block is defined as a sequence of instructions without any jumps or jump targets. A block is considered as covered when it is entered.

Coverage preservation, denoted $cp(P, T, T')$, is a proportional measure of the amount of code covered by the time-aware prioritization versus the amount of code covered by the entire test suite. In other words,

$$cp(P, T, T') = \frac{cc(P, T')}{cc(P, T)} \tag{1}$$

Order-aware coverage, as defined in Walcott et al. [12], takes into account not only the percentage of code covered by test cases in a prioritization, but also the order in which the test cases in the prioritization execute. This provides a way to measure the amount of code covered in conjunction with the time during the execution phase at which that code was covered. This is important because, as explained previously, it is desirable to have the test cases with the highest fault-detecting potential occur earlier in the prioritized test suite execution phase. Order-aware coverage is calculated in two parts, primary and secondary, which are summed. Let T' be a possible prioritization of T, and let w be a weighting factor. The primary value $\mathcal{C}_{pri}$ is obtained by measuring the code coverage of the entire prioritization and weighting that measurement by a value large enough to cause the primary component to dominate the result of the final order-aware coverage value. For simplicity in this study, we set w to 100. Then the primary overlap-aware coverage of T' is

$$\mathcal{C}_{pri}(P, T', w) = cc(P, T') \times w \tag{2}$$

Next, the secondary component $\mathcal{C}_{sec}$ considers the incremental code coverage of the prioritization and is calculated in two parts. The secondary-actual value $\mathcal{C}_{s-actual}$, the first calculation, is computed by summing the products of the execution time $time\left(\langle T_i \rangle\right)$ and the code coverage cc of $T'_{\{1,i\}} = \langle T_1 \ldots T_i \rangle$ for each test case $T_i \in T'$. In other words, the first value in the summation is the product of the execution time and the code coverage of T_1, the second value is the product of the execution time and code coverage of T_1 and T_2, and the ith value in the summation is the product of the execution time and code coverage of T_1 through T_i. Then for T',

$$\mathcal{C}_{s-actual}(P, T') = \sum_{i=1}^{|T'|} time\left(\langle T_i \rangle\right) \times cc\left(P, T'_{\{1,i\}}\right) \tag{3}$$

The second part, secondary-max $\mathcal{C}_{s-max}$, represents the maximum value that secondary-actual function could take, that is, what the value of $\mathcal{C}_{s-max}$ would be if the first test case covered 100% of the code covered by the entire prioritization. The secondary-max value for T' is

$$\mathcal{C}_{s-max}(P, T') = cc(P, T') \times \sum_{i=1}^{|T'|} time\left(\langle T_i \rangle\right) \tag{4}$$

The secondary value $\mathcal{C}_{sec}$, then, is the ratio of the secondary-actual and the secondary-max values. For T',

$$\mathcal{C}_{sec}(P, T') = \frac{\mathcal{C}_{s-actual}(P, T')}{\mathcal{C}_{s-max}(P, T')} \tag{5}$$

All of the coverage information is obtained using Emma, an open source Java code coverage tool that reports code coverage statistics at method, class, package, and all-classes levels [10]. The results reported in this paper are based on block level coverage, because the use of block level coverage has been shown to give better results than levels of a coarser grain, such as method level [12]. As this paper also examines the trade-offs between the effectiveness of a prioritization and the time and space overhead incurred in performing the prioritization, execution time and memory statistics were also obtained. To do so, a Linux process tool, which calculates the peak memory use and total user and system time required by a program, was used.

4. EXPERIMENTS AND RESULTS

Experiments were run in order to analyze the effectiveness and efficiency of the seven test suite prioritizers described in Section 2.1 and the overlap-aware solver described by Walcott et al. [12]. The solvers prioritized the test suites of `Gradebook` and `JDepend` so that resulting test tuples would execute within 25, 50, and 75% of the total execution time of the initial test suites.

Prioritizer Effectiveness. First, we examine the overall coverage, order-aware coverage, and coverage preservation of each of the resulting prioritizations. These can be seen in Figures 6(a)- 6(d). As would be expected, the coverage overlap-aware solver achieves the highest overall coverage for each testing time constraint. Greedy by value, solver 3, also performs very well for `Gradebook`, while greedy by ratio and greedy by weight, solvers 2 and 4, create good prioritizations for `JDepend`.

The success of these solvers is understandable in light of the nature of the test suites. In the `Gradebook` test suite, there is only a little coverage overlap between test cases, so a greedy by value approach is likely to add worthwhile test cases to the prioritization at each iteration, which is shown in Figure 6(c). JDepend's test cases have very short execution times, and many of them cover about the same amount of code. Thus, a solver that orders the test cases so that the shortest tests run first does well. For such a test suite, a greedy algorithm prioritizing based on the ratio of code coverage to execution time performs equally well, as seen in Figure 6(d). Note that because the execution time difference between JDepend's test cases is much smaller than that of Gradebook's test cases, we observe a less drastic coverage difference over the JDepend prioritizations and as the time limit increases. Figures 6(a) and 6(b) show a similar trend with regard to coverage preservation.

One might think that the core algorithm would produce best results among the non-overlap-aware solvers. However, in Figures 6(a)- 6(d), we observe that this is not true for either JDepend or Gradebook. While the core algorithm achieves a higher utility result than other solvers, there is no guarantee that the total coverage will also be high once overlapping coverage is considered.

Prioritizer Efficiency. Next we evaluated the time and space overheads incurred by each prioritizer, which are displayed in Figures 6(e)- 6(h). Among the traditional knapsack solvers, the time and memory costs were insignificant in all but the dynamic programming, generalized tabular, and core algorithms. In Figure 6(g), we see that the memory requirements of the generalized tabular solver were especially prohibitive, reaching over 1039MB at peak usage.

The overlap-aware algorithm compared favorably in terms of memory, requiring only 9.1MB of memory [12]. However, the algorithm is time intensive and needs multiple hours to execute. For the seven algorithms described in this paper, the scaling technique successfully reduced the prioritization execution time. In one case in Gradebook, the time was decreased by 330%, as seen in Figure 6(e). However, as described in Figure 6(h), scaling does not always improve the time overhead. It also occasionally had a negative impact on memory overhead, particularly for JDepend in Figure 6(g).

Discussion. Results indicate that a trade-off must be made between efficiency and final coverage. The design of the test suite is also of great importance. As shown in Figure 6(a), if there is little overlap between the test cases, a cheaper prioritizer can be used with favorable results, comparable to those of an overlap-aware solver. However, if there is a large amount of overlap between test cases, the added expense of an overlap-aware solver would be worthwhile. While more sophisticated solvers such as dynamic programming, generalized tabular, and core are likely to obtain higher utility than simple solvers, neither group can make any guarantee regarding final cumulative coverage of the result. Thus, if correctness of the program is of highest importance, resources should be better spent using a solver that takes test case overlapping coverage into account.

5. RELATED WORK

We differentiate this paper from prior work because it presents time-aware prioritization algorithms that guarantee the termination of a test suite within a specified time limit. Moreover, the empirical assessment in this paper incorporates metrics that are related to both the (i) time and space overhead of the prioritizers and (ii) effectiveness of the resulting prioritizations. As an example of related research, Srivastava and Thiagarajan report on a testing tool that prioritizes a test suite according to the coverage of program changes at the basic block level [11]. Even though their *Echelon* tool can consider the running time of each test, the experimental analysis does not evaluate this configuration of their testing framework. Elbaum et al. also present prioritization algorithms that incorporate both the cost and the criticality of a test case [2]. However, their method does not solve instances of the 0/1 knapsack problem to prioritize test suites for time constrained execution. Finally, Elbaum et al. empirically study prioritization effectiveness while reserving an experimental study of efficiency for later work.

6. CONCLUSIONS AND FUTURE WORK

This paper describes how knapsack solvers can efficiently perform time-aware test suite prioritization. When provided with a testing time budget, these prioritization techniques create a reordered test suite that quickly covers the test requirements and always stops execution within a specified time limit. The experimental results demonstrate that it is sensible to ignore the overlap in test case coverage when (i) the prioritization algorithm must incur a minimal time overhead and (ii) a modest decrease in coverage preservation is acceptable. However, we also find that the most sophisticated 0/1 knapsack solvers do not always create the most effective ordering of the test suite, suggesting that overlap-aware prioritizers with a higher time overhead are appropriate in contexts where correctness is the highest priority. In future research, we will experimentally evaluate how other knapsack solvers (e.g., branch and bound) trade-off the efficiency of prioritization with the effectiveness of the reordered tests. Future empirical studies will incorporate additional case study applications that are larger and written in other programming languages. We also intend to calculate other measures of test suite effectiveness (e.g., APFD).

7. REFERENCES

[1] 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.

[2] S. G. Elbaum, A. G. Malishevsky, and G. Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proc. of 23rd ICSE*, pages 329–338, 2001.

[3] 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.

[4] E. Gossett. *Discrete Mathematics with Proof.* Pearson Education, Inc., Upper Saddle River, New Jersey, 2003.

[5] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. of ICSE*, pages 191–200, 1994.

[6] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems.* Springer-Verlag, Berlin, Germany, 2004.

[7] 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.

[8] D. Pisinger. Core problems in knapsack algorithms. *Operations Research*, 47(4):570–575, 1999.

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

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

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

[12] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos. Time-aware test suite prioritization. In *Proc. of ISSTA*, pages 1–12, New York, NY, USA, 2006.

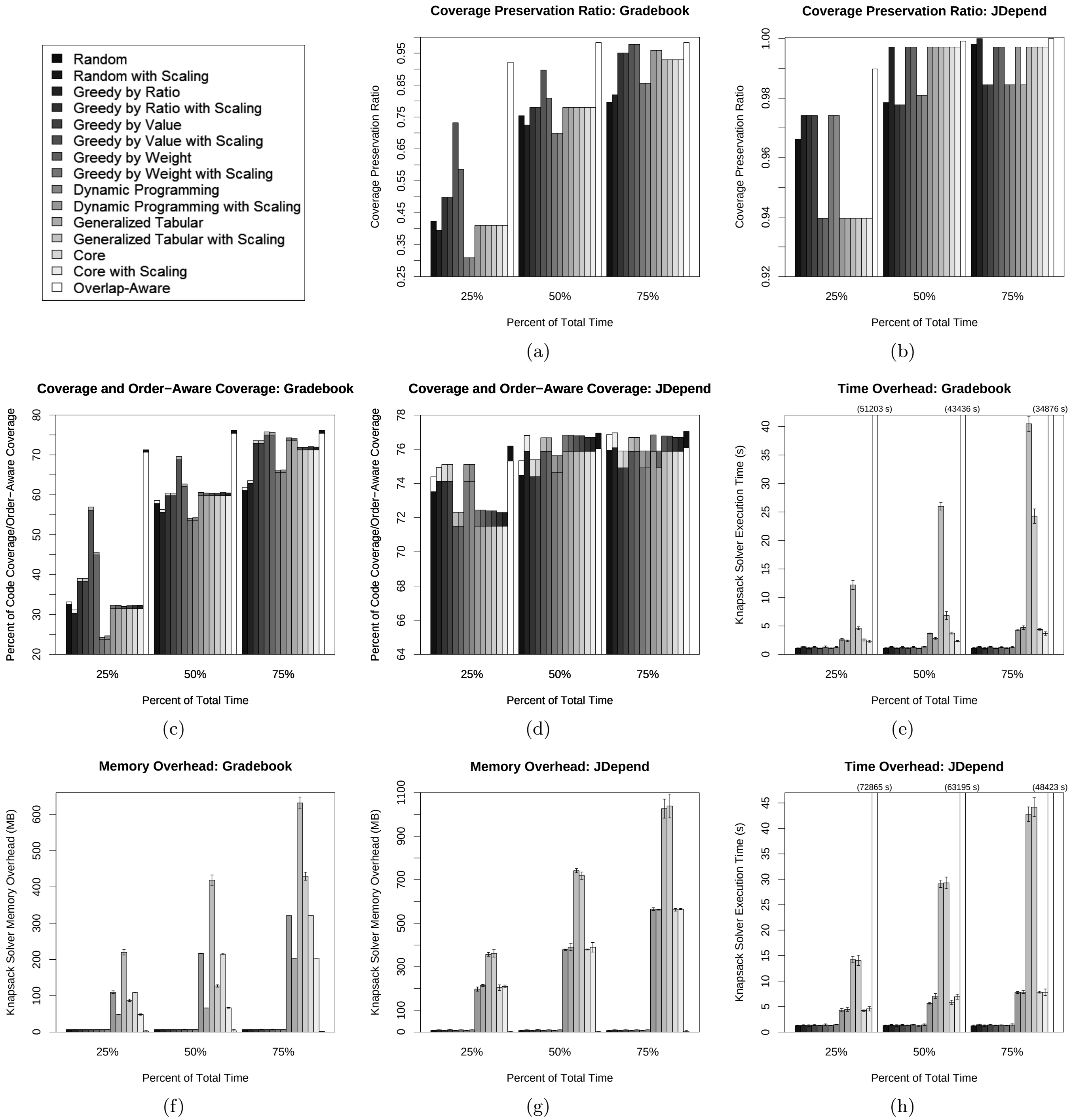

Figure 6: Efficiency and Effectiveness of Test Suite Prioritization.