Using Dependency Structures for Prioritization of Functional Test Suites

Shifa-e-Zehra Haidry and Tim Miller

Abstract—Test case prioritization is the process of ordering the execution of test cases to achieve a certain goal, such as increasing the rate of fault detection. Increasing the rate of fault detection can provide earlier feedback to system developers, improving fault fixing activity and, ultimately, software delivery. Many existing test case prioritization techniques consider that tests can be run in any order. However, due to functional dependencies that may exist between some test cases—that is, one test case must be executed before another—this is often not the case. In this paper, we present a family of test case prioritization techniques that use the dependency information from a test suite to prioritize that test suite. The nature of the techniques preserves the dependencies in the test ordering. The hypothesis of this work is that dependencies between tests are representative of interactions in the system under test, and executing complex interactions earlier is likely to increase the fault detection rate, compared to arbitrary test orderings. Empirical evaluations on six systems built toward industry use demonstrate that these techniques increase the rate of fault detection compared to the rates achieved by the untreated order, random orders, and test suites ordered using existing "coarse-grained" techniques based on function coverage.

Index Terms—Software engineering, testing and debugging, test execution

1 Introduction

A large portion of most software engineering projects typically involves testing. To increase the effectiveness of testing effort within limited resources, test case prioritization can be performed. Test case prioritization is the process of organizing test cases in a sequence to achieve a certain goal. One goal of test case prioritization is to increase the rate of fault detection; that is, to find most defects and as early as possible. Finding defects earlier can increase early defect fixing and ultimately lead to earlier delivery. Prioritization has typically been applied to test suites that take days or weeks to run; however, with agile development processes becoming more prevalent in industry, the potential for prioritization techniques to have an impact is increasing.

Previous work on test case prioritization [6], [9], [12], [15], [23], [26], [28] demonstrates that prioritization techniques are effective for improving rate of fault detection. However, these approaches do not consider test suites that contain functional dependencies between tests. Functional dependencies are the interactions and relationships among system functionality determining their run sequence; for example, a function G can only be executed if some precondition holds and function F enables this precondition. As test cases mirror this functionality, they also inherit these dependencies; therefore, executing some test cases requires executing other test cases first.

A common strategy for handling test case dependencies is to group these fine-grained tests with dependencies into coarse-grained tests. For example, a test suite containing three tests, A, B, and C, with dependencies $A \rightarrow B$ and $A \rightarrow C$ would be repackaged into two separate tests: one containing the sequence $A; B$, and one containing the sequence $A; C$. However, this has the undesirable side effect of executing A twice, which can be avoided if one of the sequences $A; B; C$ or $A; C; B$ is executed instead, assuming compositionality. For many test suites, executing redundant test cases would be inconsequential; however, for large test suites in which prioritization is important, it is likely that testers would want to avoid running redundant test cases.

While many existing techniques could be applicable to test suites containing functional dependencies, applying them requires new algorithms for computing the sequencing of tests to preserve the dependencies. For example, when using the metric proposed by Rothermel et al. [23] that schedules test cases based on the amount of code coverage achieved, the test case that achieves the highest coverage is assigned the highest priority. However, when dependencies exist, additional test cases may need to be run first. If this test sequence achieves low coverage, then running an alternative scenario with the same number of test cases may achieve a higher coverage.

In this paper, we propose a set of new techniques for functional test case prioritization based on the inherent structure of dependencies between tests, which we call *dependency structure prioritization*. Given that these dependencies reflect the dependencies of the system itself, it is proposed that ordering test executions based on the complexity of interactions between tests can increase the fault detection rate as compared with arbitrary test orderings. Our hypothesis is that, as a result, faults will be revealed earlier because scenarios containing more relationships are more complex and more fault prone.

The techniques are divided into two categories: one based on *open* dependency structures, in which a functional dependency between test cases t_1 and t_2 indicates that t_1

• *The authors are with the Department of Computing and Information Systems, University of Melbourne, Victoria 3010, Australia. E-mail: s.haidry2@pgrad.unimelb.edu.au, tmiller@unimelb.edu.au.*

Manuscript received 1 Mar. 2011; revised 12 Mar. 2012; accepted 19 Mar. 2012; published online 18 Apr. 2012.
Recommended for acceptance by G. Rothermel.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2011-03-0061.
Digital Object Identifier no.10.1109/TSE.2012.26.

0098-5589/13/$31.00 © 2013 IEEE Published by the IEEE Computer Society

must be executed *at some point before* t_2; and one based on *closed* dependency structures, in which a dependency between test cases t_1 and t_2 indicates that t_1 must be executed *immediately before* t_2.

We identify two strengths to these techniques. First, they are model-based—they use only the information contained in the dependency structure—which means that calculating the test order does not require statistics from previous test runs, such as the amount of code each test covers. This allows prioritization on the first test run, as well as allowing up-to-date prioritization for an iteration before the tests from the previous iteration have finished executing. As agile processes lead to shorter development iterations, this is becoming more important; that is, for some systems, the test execution time may be longer than the time allocated for a single iteration.

Second, by maintaining fine-grained test suites, but executing entire scenarios, we achieve a good balance between the problem of fine-grained versus coarse-grained tests. Fine-grained tests lead to more flexible test suites, and lead to less problems with cascading test failures [1], but coarse-grained tests have a higher level of fault detection, due to the interaction between the "inner" tests [22].

We present experimental results on six software systems built toward industry use, and compare our techniques with the untreated test ordering defined by the test engineer, random prioritization, two existing coarse-grained techniques based on function coverage [4], and greedy prioritization, which uses information about known faults, and is therefore not a valid prioritization technique (defined in Section 5.2). Results indicate that our techniques outperform test engineers, random prioritization, and the coarse-grained techniques, but not greedy prioritization. This indicates that dependency structure prioritization is a promising approach to improve the rate of fault detection.

2 Test Case Prioritization

In many software projects, it is necessary to execute test suites in order of priority to utilize limited resources and time effectively, and to increase the possibility of finding defects early. Rothermel et al. [23] report of an industry collaborator with a test suite that takes approximately seven weeks to run. In such cases, prioritization is important.

Definition 2.1 (Test Case Prioritization). *Test case prioritization is the process of scheduling test cases to be executed in a particular order so that test cases with a higher priority are executed earlier in the test sequence. The priority is defined relative to some test criterion; for example, the number of code statements covered.*

Definition 2.2 (Rate of Fault Detection). *Jeffrey and Gupta [6] define the rate of fault detection as the number of defects found in allocated time.*

Rothermel et al. [23] discuss several goals for prioritization. Our focus in this paper is to "to increase the rate of fault detection for a test suite." That is, we want to increase the possibility of detecting high number of faults earlier when running a test suite. Rothermel et al. [23] define the test case prioritization problem as follows.

Definition 2.3 (The Test Case Prioritization Problem). *Given a test suite T, the set of permutations of T, PT, and function f that assigns an* award value *to an ordering of T, find $T' \in PT$ such that:*

$$\forall T'' \in PT \bullet f(T') \geq f(T''). \quad (1)$$

The function f is the quality measure of T. Therefore, the above predicate reads: Find the test sequence T' such that no other test sequence is of higher quality. Ideally, the function f determines the actual rate of fault detection of a test suite; however, this information cannot be known until the tests have been run. Therefore, test case prioritization techniques must approximate this function.

There are several techniques available that estimate f, for example, prioritizing tests based on the amount of code coverage they achieve [23] and prioritizing tests based on the relative importance of the requirements they test [9], [26]. In this paper, we introduce several test case prioritization techniques based on the dependency structure of test cases in the test suite. It is our hypothesis that these techniques provide a reliable approximation of the function f.

3 Dependency Structures

In this section, we provide the necessary background on dependency structures to understand this paper.

Definition 3.1 (Functional Dependency). *In software engineering, sequences of interactions between subsystems, or between systems and users, are known as* scenarios. *The sequencing in scenarios represents the order in which the interactions take place, enforcing dependencies between interactions. Put simply, some interactions cannot occur until and unless some other interactions occur first. When an interaction I_1 is required to be executed before an interaction I_2, we say that I_2 is* dependent upon I_1.

Take the example of a cash withdrawal scenario through an automatic teller machine. To withdraw cash, the user must first insert their card, then enter their pin number, select the transaction type (withdraw), enter the required amount, and finally, collect the cash. The steps in this scenario are related; for example, without pin confirmation, the user cannot enter the amount they wish to withdraw. There can be many test cases developed to test such scenarios. For example, to test the "enter amount" functionality, a test engineer can develop test cases to check a valid/invalid amount, etc. However, the valid pin confirmation test case must be executed before any of these, and therefore, the valid/invalid amount tests are dependent on the valid pin test.

Functional dependencies in software are thus inherently contained in the requirements or design. As a result, the test cases inherit these dependencies as well: If a particular requirement R_1 is dependent on another requirement R_2, then the test(s) for R_1 is dependent on the test(s) for R_2.

Definition 3.2 (Dependency Structure). *A dependency structure is a directed acyclic graph (DAG), $G = (V, E)$, in which V is a set of nodes, and E is a set of arcs between these nodes. In this work, the set V defines a set of test cases. The set E defines the functional dependencies between the tests.*

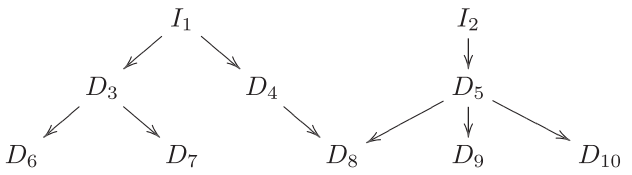


Fig. 1. An example dependency structure.

Fig. 1 depicts an example dependency structure. In this structure, I_1 and I_2 have no dependencies, while nodes D_3 to D_{10} are dependent on other nodes; for example, D_3 is dependent on I_1. Note the difference between *direct* and *indirect* dependencies: D_6 is directly dependent upon D_3 and indirectly dependent upon I_1.

Definition 3.3 (Open and Closed Dependency Structures). *An* open *dependency structure is one in which a dependency between two test cases t_1 and t_2 specifies that t_1 must be executed at some point before t_2, but not necessarily immediately before t_2. In other words, once t_1 has been executed, the dependent node remains open for execution, irrelevant of any other nodes being executed. For example, in Fig. 1, if node I_1 is executed, then nodes D_3 and D_4 are available. If I_2 is then executed, nodes D_3, D_4, and D_5 are all available to be executed.*

A closed *dependency structure is one that is not open; that is, a dependency between two test cases t_1 and t_2 specifies that t_1 must be executed immediately before t_2. For example, if all dependencies in Fig. 1 are closed dependencies and nodes I_1 and I_2 are executed in order, then to execute D_3 or D_4, node I_1 would need to be executed again.*

Some dependency structures may contain a mix of both open and closed dependencies. Such structures would be considered closed dependency structures. However, sequences of closed dependencies can be regrouped into single tests, resulting in an open dependency structure.

Definition 3.4 (Independent and Dependent Test Cases). *An independent test case is defined as a test case whose execution is not dependent on the execution of any other test case. A dependent test case is one whose execution is dependent upon execution of one or more test cases.*

Ryser and Glinz [24] describe one method for identifying dependencies in software, which can then be mapped to the test cases. They split dependencies into three categories:

1. *abstraction dependencies*, which are dependencies based on hierarchical decomposition in a system model, such as aggregation and composition;
2. *temporal dependencies*, which are dependencies based on time—that is, one scenario must be executed before the other temporally; and
3. *causal dependencies*, which are dependencies based on data or resources—that is, some resource that is produced by scenario A is used by scenario B.

In Ryser and Glinz's notation for dependency charts, temporal dependencies map to *strict dependencies*, which are similar to our closed dependencies, and causal dependencies map to *loose dependencies*, which are similar to our open dependencies.

To generalize our work, our approach to dependency-based prioritization is independent of the method used to define the dependencies. We do not focus on how to define dependencies among test cases, but focus on prioritizing the test cases on the basis of an inherent dependency structure.

Example 3.1. To provide an example of test dependencies, we discuss one of the systems used as part of our evaluation in Section 5: the GSM 11.11 system. This is a file system with security permissions on a *Subscriber Identity Module* (SIM) for mobile devices.

Part of the functionality of the GSM system is to be able to select a file, and to read/write data from/to that file. There are several types of file, including binary files. The $READ\_BINARY$ and $UPDATE\_BINARY$ operations each require that the currently selected file is a binary file, and that the necessary security permissions are enabled.

There are several tests related to these two operations; however, we consider only five for illustration:

1. select a binary file,
2. select a record file (a nonbinary file),
3. attempt to read a binary file where the selected file is not a binary file,
4. attempt to update a binary file where the selected file is not a binary file, and
5. attempt to update a binary where the selected file is a binary file and is successfully read.

These tests can be written to be independent, and therefore could be run in any order. However, each of them requires some setup to be run. For example, tests C and D both require that a nonbinary file is selected. While the tests can be written to bring the system into this state, the code required to select a binary file is also a test in its own right: test B.

As a result, a test engineer will generally note that test A selects a binary file, leaving the system in a state that allows test E to run. Test E is the just assertions related to its test specification (checking that the file cannot be read), and can be reduced to a smaller and more efficient test, but under the condition that A is executed before E. This creates the dependency $A \rightarrow E$. There are two additional dependencies in these tests: $B \rightarrow C$ and $B \rightarrow D$.

These three dependencies are all closed dependencies, so only their immediate dependents can be executed once one has been run. This is because the nondependent tests, A and B, change the state of the file system by modifying which file is currently selected. As a result, some other tests in the dependency structure that previously could have been scheduled now become invalid.

Any dependencies in which C, D, and E are the nondependent tests will be open dependencies. This is because none of these tests change the state of the file system, so all previously enabled operations remain enabled, and all effects of applying these will be the same as if $C/D/E$ have not been executed.

We record and maintain these dependencies in a matrix, and using this, we can automatically schedule either test A or test B first. If we schedule test A, we can now schedule either test B or E. However, if we schedule test B, we must schedule test A again before we schedule test E due to the closed dependency.

The extraction of test dependencies requires some understanding of the requirements or design of the system

under test; however, the extraction must be done from the tests themselves. It is important to note that dependencies between tests are not *defined* by our work. That is, we are not defining them for the purpose of prioritization. The dependency $A \rightarrow E$ is a necessity; that is, to run test E, test A must be run first, due to some underlying dependency in the system. A test engineer defines the dependency as a matter of efficiency, and we use the information to construct a dependency structure.

While test engineers can create different dependencies based on the tests they write, in our experience extracting dependency structures is a deterministic process. The approach to extraction is dependent on the application domain, the system requirements, and the tests themselves.

4 Prioritizing Test Cases Based on Dependency Structure

It is well argued that much complexity in software systems results from the coupling and interactions between the parts that make up these systems [11], [18]. As such, it is reasonable to claim that testing the parts of the system with more interactions earlier in the testing cycle may increase the fault detection rate.

Our hypothesis is that the dependency structure between test cases is closely related to the interactions between the parts of systems. Therefore, by assigning a higher priority to tests that contain dependencies, we increase the likelihood of finding errors early in the test run. This is initially supported Rothermel et al.'s empirical study [22], in which they found that concatenating tests together increases the fault detection rate of the tests due to the interactions that occur between the tests. Our work aims to add further support to the hypothesis.

In this section, we present several related techniques for functional test case prioritization based on dependency structures. We will call this family of prioritization techniques *Dependency Structure Prioritization* (DSP).

These techniques assign priority based on a *graph coverage value*. The graph coverage value of a test case is a measurement of the complexity of the dependents of that test case. Open dependency structures and closed dependency structures must be treated differently due to their inherent differences. For example, in a closed dependency structure that forms a tree, to execute all test cases we must execute all paths in the structure, which may require executing some test cases more than once. For an open dependency structure, this is not necessary.

For open dependency structures, we define two ways to measure the graph coverage value of a test case based on a dependency structure:

1. the total number of dependents of the test case, and
2. the longest path of direct and indirect dependents of the test case.

Using these values, the priority of tests is calculated using a weighted depth-first search algorithm in which the next child selected in the search is defined by its graph coverage value.

A depth-first search increases the likelihood that parts of the system with many interactions will be selected for testing first. The parts with the most interrelationships and dependencies are likely to be more complex and thus more prone to having faults; thus executing respective test cases first should increase the fault detected rate.

The first coverage measure increases the likelihood of closely related interactions being executed earlier. The second coverage measure increases the likelihood of linearly related scenarios being executed earlier, and therefore increases the likelihood that the most complex scenarios are executed first.

For closed dependency structures, we define three ways to measure the graph coverage value. These measure the coverage of *paths* throughout the dependency structure. The three coverage measures for paths are:

1. the number of nonexecuted test cases in the path (recall that a test case may be executed more than once if they have more than one dependent);
2. the ratio of nonexecuted to executed test cases in the path, with a higher weighting for those test cases toward the end of the path, thereby giving higher priority to longer paths; and
3. a combination of the above two: the number of nonexecuted test cases divided by the height of the path.

The rationale for these techniques is the same as for the longest path measure for open dependency structures: Each path is a linear scenario. Each of these ranks the weight of the paths based on how likely a new path is to contain nonexecuted test cases while recognizing the complexity of interactions.

4.1 Prioritization for Open Dependency Structures

We define two graph coverage measures based on open dependency structures: *DSP volume* and *DSP height*.

4.1.1 DSP Volume

The DSP volume coverage measure gives a higher weight to those test cases that have more dependents. The reasoning behind such a measure is that those test cases that are close to each other in a dependency structure will likely be closely related in the system itself and will interact. As such, the complexity at this part of the system is higher.

To calculate the DSP volume of a test case, one needs to calculate all direct and indirect dependents of that test case. This can be done using a straightforward transitive closure algorithm on the graph. An algorithm for calculating this for all test cases is shown in Algorithm 1, which is based on Warshall's algorithm for transitive closure calculation of a Boolean matrix [29]. In this algorithm, if a direct edge exists between two nodes, i and j, then replacing row i with the Boolean OR of rows i and j will result in the direct dependents of j becoming dependents on i.

Algorithm 1. DSP_VOLUME for transitive closure calculation of all test cases based on the dependency structure.
Input: G: an $n \times n$ Boolean adjacency matrix representing direct dependencies between test cases.
Output: D: an $n \times n$ Boolean adjacency matrix representing indirect dependencies between test cases.
1: $D := copy\ of\ G$
2: **for** $i \in 1..n$ **do**

```
3:   for j ∈ 1..n do
4:     if D[i, j] = 1 then
5:       for k ∈ 1..n do
6:         D[i, k] := D[i, k] ∨ D[j, k]
7:       end for
8:     end if
9:   end for
10: end for
11: return D
```

To calculate the weight, $w(t)$, for a test case, t, one simply counts the number of indirect dependents by counting the number of 1s in row t.

4.1.2 DSP Height

The DSP height coverage measure gives a higher weight to those test cases that have the deepest dependents. The reasoning behind such a measure is that those test cases that have deeper dependents are contained in longer scenarios. Like closely grouped test cases, the scenarios are likely to contain more interactions, and therefore have a higher complexity.

To calculate the DSP height of a test case, one needs to calculate the height of all paths from that test case, and take the length of the longest path as the weight. This can be done using a straightforward depth-first search algorithm on the graph. An algorithm for calculating this for all test cases is shown in Algorithm 2, which is based on the Floyd-Warshall shortest-path algorithm [5]. The difference between Algorithm 2 and the Floyd-Warshall algorithm is at line 5, in which we take the maximum of the two path lengths instead of the minimum.

Algorithm 2. DSP_HEIGHT for calculation of the maximum length path from a test case based on the dependency structure.

Input: G: an $n \times n$ Boolean adjacency matrix representing direct dependencies between test cases.

Output: D: an $n \times n$ integer adjacency matrix representing the length of indirect dependencies between test cases.

```
1: D := copy of G
2: for i ∈ 1..n do
3:   for j ∈ 1..n do
4:     for k ∈ 1..n do
5:       D[i, j] := max(D[i, j], D[i, k] + D[k, j])
6:     end for
7:   end for
8: end for
9: return D
```

To calculate the weight, $w(t)$, of a test case, t, one simply calculates the maximum height path from t by finding the maximum value in row t.

4.1.3 Calculating Test Case Ordering

Calculating test case ordering is done using a weighted depth-first search algorithm in which the weights are defined by the graph coverage values defined in Sections 4.1.1 and 4.1.2.

Assuming that the function $w(t)$ returns the weight of a test case using either of the graph coverage values, the prioritization of a test suite is calculated using Algorithms 3 and 4.

Algorithm 3. WEIGHTED_DFS for test case prioritization based on dependency structure.

Input: G: an $n \times n$ Boolean adjacency matrix representing direct dependencies between test cases.

Input: w: a weight function mapping test cases to their graph coverage values.

Output: T: a test suite prioritized by the graph coverage value.

```
1: T := ⟨⟩
2: I := get_independent_tests(G)
3: while I ≠ ∅ do
4:   i ∈ {t : I | (∀t' ∈ I | w(t) ≥ w(t'))}
5:   T := WEIGHTED_DFS_VISIT (i, G, w, T)
6:   I := I \ {i}
7: end while
8: return T
```

Algorithm 4. WEIGHTED_DFS_VISIT for visiting vertices in the dependency structure.

Input: G: an $n \times n$ Boolean adjacency matrix representing direct dependencies between test cases.

Input: v: the vertex from which to start the search

Input: w: a weight function mapping test cases to their graph coverage values.

Input: T: the tests that have been prioritized so far.

Output: T: a test suite prioritized by the graph coverage value.

```
1: T := T ⌢ ⟨v⟩
2: C := get_children(G, v)
3: while C ≠ ∅ do
4:   c ∈ {t : C | (∀t' ∈ C | w(t) ≥ w(t'))}
5:   if c ∉ T then
6:     T := WEIGHTED_DFS_VISIT (c, G, w, T)
7:   end if
8:   C := C \ {c}
9: end while
10: return T
```

Summarizing the algorithms, the independent test case with the highest value will be run, followed by its dependent test case with the highest graph coverage value. For more than one test case having the same number of dependent test cases, any arbitrary child is selected. When all dependents (both direct and indirect) have been executed, the independent test with the second highest coverage value is executed.

As an example, consider the dependency structure in Fig. 1. If we use the DSP volume coverage measure, test I_1 has the highest priority, as it contains more children than I_2. Test case D_3 has the second-highest priority as it has two children while its sibling has only one. When all dependents of I_1 have been executed, test case I_2 will be executed, followed by its dependents. The final ordering is I_1-D_3-D_6-D_7-D_4-D_8-I_2-D_5-D_9-D_{10}. Note that line 5 of Algorithm 4 prevents test case D_8 from executing a second time.

4.1.4 Complexity Analysis

For both open DSP techniques, the worst-case performance for calculating the DSP orderings is $O(|V|^3 + |V| + |E|)$. $|V|^3$ is the time required to complete the transitive closure for calculating the indirect dependencies or longest path for

each test case, and $|V| + |E|$ is the time taken to perform the weighted depth-first search.

4.2 Prioritization for Closed Dependency Structures

We define three graph coverage measures based on closed dependency structures: *DSP sum*, *DSP ratio*, and *DSP sum/ratio*. These three prioritization techniques provide weights to *paths* in the dependency structure, rather than individual test cases, in which a path is a complete traversal from a root node to a leaf node. The reasoning for this is straightforward. To execute a prefix of a path and later execute the remainder of the path would again require us to execute the prefix again, which will find no new faults in a deterministic system. In a nondeterministic system, executing previously executed tests is less likely to find faults than executing unexecuted tests. While the prefix may also be executed as part of another path, executing prefixes will always result in more test cases being executed. Therefore, it is assumed that for a closed dependency structure, executing the entire path will almost always be more beneficial than reexecuting some prefixes.

Before weights are given to paths, one must first calculate the paths to be executed in the dependency structure. Our aim is to execute all tests at least once, so executing all paths is not necessary as some paths may execute a sequence of tests that have been executed by other paths. Instead, we execute a set of *linearly independent paths* of the dependency structure. A set of paths is linearly independent if all paths contain at least one node that is not present in any other path in the set. To generate a set of linearly independent paths, we use the modified depth-first search algorithm proposed by Poole [19].

4.2.1 DSP Sum

The DSP sum coverage measure gives a higher weight to paths that have more nonexecuted test cases. The reasoning behind this measure is to find a balance between longer paths, as in the DSP height coverage measure, but to not execute long paths with few nonexecuted tests cases, as these are less likely to find new faults.

To calculate the DSP sum of a path, one simply counts the number of nonexecuted test cases in that path.

Assuming the existence of a relation $seen(t_i)$, which is true if and only if the test case t_i has been previously scheduled in the current test sequence, the DSP sum of a path p is defined as follows:

$$DSP\_sum(p) \;=\; \#\{i \in 1..\#p \mid \neg seen(t_i)\}, \tag{2}$$

in which the # operator returns the size of a list or set.

4.2.2 DSP Ratio

The DSP ratio coverage measure gives a higher weight to paths that have a higher ratio of nonexecuted tests to executed tests, while also giving weight to longer paths. The reasoning behind this is to minimize the number of executed test cases, which are not able to find any new faults.

To calculate the DSP ratio of a path, one first calculates the weighted sum of the path in which the weight of a test case is its index in the path if it has not been executed, or 0 otherwise, and then divides this by the height of the path. Formally,

$$DSP\_ratio(p) \;=\; \frac{\sum_1^{\#p} w(t_i)}{\#p}, \tag{3}$$

in which

$$w(t_i) \;=\; \begin{cases} i, & \text{if } \neg seen(t_i), \\ 0, & \text{otherwise.} \end{cases}$$

In this equation, the depth of a test case in the tree (the index of its location in the path) means that longer paths get a higher weighting, while executed tests reduce the score, because they have a weight of 0, but contribute to the length of the path.

4.2.3 DSP Sum/Ratio

The DSP sum/ratio coverage is simply the number of nonexecuted test cases divided by the height of the path. Therefore, this is similar to the DSP ratio, except it does not weight the depth of test cases. Formally,

$$DSP\_sum/ratio(p) \;=\; \frac{\#\{i \in 1..\#p \mid \neg seen(t_i)\}}{\#p}. \tag{4}$$

4.2.4 Calculating Test Case Priorities

The priority of a test case is dependent on the priority of the path in which it falls. To prioritize an entire test suite, we use a greedy algorithm that selects the next path as the path with the highest priority.

It must be noted that the priority value of all remaining paths must be recalculated after each path is selected. If we choose the path with the highest priority given one of our DSP metrics, by executing that path we change the value of some of the other paths in the dependency structure because some of these paths will contain some of the executed tests. Thus, to prioritize the test suite we must iterate over the process of weighting all remaining paths, then selecting the highest priority path.

4.2.5 Complexity Analysis

For all three DSP techniques, the worst-case complexity for calculating the DSP orderings for the three closed techniques is $O(|V| + |E| + 2^{|2V|}/2)$. $|V| + |E|$ is the time required to calculate a set of linearly independent paths in the graph, while $2^{|2V|/2}$ is the worst-case complexity of calculating the priority of paths. There is a worst case of $2^{|V|}$ paths and, after each path is selected, all unselected paths must be reevaluated because the coverage value of these paths may have changed.

5 Evaluation

In this section, we outline an experimental evaluation of our DSP techniques using six systems built toward industry use which are all currently in use as far as the authors are aware. All six systems are used to evaluate both open and closed dependency structures. The goal of this evaluation is to establish whether dependency structure prioritization is able to increase the rate of fault detection relative to existing coarse-grained techniques and relative to random fine-grained techniques.

In the first set of experiments, we compare the two open dependency structure prioritization techniques (DSP height

TABLE 1
Metrics for the Systems Being Tested

Artifact	Type	Lines of code	Functions	Faults	Tests	Depend-encies	Graph density	Unconnected tests	Maximum depth
Elite_1	component	2487	54	72	64	64	0.03175	3	17
Elite_2	component	2487	54	64	64	64	0.03175	3	17
GSM_1	unit	385	14	15	51	65	0.05098	0	6
GSM_2	unit	975	60	14	51	65	0.05098	0	6
CRM_1	component	1875	33	50	30	30	0.06897	0	6
CRM_2	component	1875	33	30	30	30	0.06897	0	6
MET	system	10,674	270	37	81	80	0.02469	0	6
CZT	component	27,246	756	27	548	314	0.00210	161	5
Bash ($\times 6$)	system	$\approx$ 59,800	$\approx$ 1051	3-8	1061	461	0.00094	460	12

and DSP volume) outlined in Section 4.1, with three coarse-grained test prioritization, total function coverage, additional function coverage, and optimal, and with the untreated prioritization applied by a test engineer, a breadth-first search prioritization technique, random prioritization, and "greedy" prioritization.

In the second set of experiments, we compare the three closed dependency structure prioritization techniques (DSP sum, DSP ratio, and DSP sum/ratio) outlined in Section 4.2, with total function coverage, additional function coverage, random prioritization, and "greedy" prioritization.

5.1 Objects of Analysis

We performed the evaluation on six systems, which are described in this section. These systems represent a wide range of possible systems, from single units, to components, to command-line systems, and web sites. We obtained test suites for each of these systems, which were all manually generated by the developers on the projects,[1] with the exception of the GSM test suite, which was generated by the authors for the purpose of the evaluation (see details in Section 5.1.2). In some cases, we also obtained dependency information for the test suites, but in others the dependency structures were derived by the authors.

Table 1 contains information about the systems that summarizes their size, the number of tests, number of dependencies (edges in the dependency structure), the number of unconnected tests (no dependents or dependencies), the depth of the graph, and the density of the dependency structure, which is defined as the number of edges divided by the maximum number of possible edges:

$$\frac{|E|}{|V|(|V|-1)/2}.$$

Note that this table includes information for six versions of the Bash system, so the lines of code, number of functions, and number of faults vary between versions.

For all systems except the GSM system, all dependencies structures are open dependency structures. Our techniques for closed dependency structures are applicable to open dependency structures as well because all dependencies are preserved. The GSM dependency structure contains a mixture of open and closed dependencies. For completeness, we have used this to evaluate our techniques for open dependency structures as well as closed dependency structures.

5.1.1 Elite

The first system is called *Elite*. Elite is an Interactive Voice Response system that deals with automatic voice recording for incoming calls. The purpose of Elite is to direct callers to their destinations according to the options selected by them. We used the first internal version release of this system. Defect reports were compiled for us by the developers on the project. These reports specified which tests detect which faults. The test suite already contained a predefined dependency structure, which was calculated by a test engineer.

Two different defect reports were used for the Elite system due to a statistical outlier. According to the defect reports, test case number 24 detects 27 faults out of 72. The next highest number of faults detected by a single test case is seven, occurring for two test cases. It is unclear whether there is an error in the defect reports for test case 24 or whether this test case is remarkably effective.

The mean and standard deviation of the number of defects found by all test case are 2.57 and 3.65, respectively. Applying Peirce's outlier test [17], based on a dataset of size 60 [21] (there are 64 test cases), the maximum "allowable" deviation is 9.72. The actual deviation of test case 24 is 23.35, which is significantly higher than 9.72. This indicates that such a high value is unusual, if not an error in the defect reports, so this test case was removed. However, it is important to note that both the original and the modified defect reports were assessed.

To distinguish the two different versions in the study, we will refer to the systems as Elite_1 (containing the outlier) and Elite_2 (not containing the outlier). In the second case, the defect reports were modified such that test case 24 revealed no faults.

5.1.2 GSM

The second system used is the GSM 11.11 system, a file system with security permissions on a *Subscriber Identity Module* (SIM) for mobile devices. The access permissions to files can be modified if the correct codes are presented to the system, and files can be read/updated. Miller and Strooper describe the GSM 11.11 system in detail [16]. We have access both separate implementations described by Miller and Strooper: one that is part of the Sun's Javacard

1. The second author is a developer on the CZT project and, previous to this study, defined many of the tests used.

Framework and one written by Miller and Strooper for demonstration purposes that is not used in the field. We use the four versions of the implementations described by Miller and Strooper: two containing real faults and two containing faults seeded by a developer.

Both GSM systems share the same interface and functionality, therefore, only one test suite and one dependency structure were required. One of the authors derived a test suite using the Test Template Framework [27], and prepared an untreated test suite that was ordered in a manner that was consistent with the layout of the test templates.

5.1.3 CRM

The third system, CRM, is a system that connects a customer relationship management system to a database back-end that contains a search engine. CRM extracts the data from the database based on specified search parameters and imports the data into the customer management system. We obtained data from an internally released version of the system. The defect reports from the testing were acquired.

As with the Elite system, we used two different defect reports due to statistical outliers. Two test cases revealed 18 and 13 faults, respectively, with the next highest count being only 5. The mean and standard deviation of the number of faults found per test case for the CRM system are 2.33 and 3.88, respectively. Applying Peirce's outlier test [17], based on a dataset of size 29 [21] (there are 29 test cases), the maximum "allowable" deviation is 8.08. The actual deviations of the two test cases are 15.67 and 10.67, respectively, so these two data values are possible outliers. As with the Elite case study, both defect reports are considered.

To distinguish the two different versions in the study, we will refer to the systems as CRM_1 (containing the outlier) and CRM_2 (not containing the outlier). As with the Elite system, the defect reports were modified such that the rejected test cases revealed no faults.

5.1.4 MET

The MET system is a meteorological website built for a government department. We obtained data from an internally released version of the software, and the defect reports from this release were used to generate the defect matrix. The test suite already contained a predefined dependency structure, which was calculated by a test engineer.

5.1.5 CZT

The CZT system is the parser and typechecker for the Community Z tools[2] package, a set of tools for the Z and Object-Z specification languages. Existing test cases from the code repository were available; however, all dependent test cases were grouped together as coarse-grained scenarios. The authors separated these into fine-grained tests so that individual test cases were not executed multiple times. The dependency structure was extracted manually by one of the authors by defining a dependency if and only if there is a declaration made in one test and referenced in a subsequent test (a data dependency between tests). Faults were extracted by analyzing log messages in the project repository that had occurred before version 1.5 of the system.

5.1.6 Bash

The final case study uses the well-known Unix shell the *Bourne-again shell* (Bash), obtained from the Software-artifact Infrastructure Repository[3] (SIR). The package contains six versions of the Bash system, each containing just under 60,000 lines of code and approximately 1,050 functions. The number of faults ranged from three to eight, depending on the version, and the test suite was the same for all six versions.

There was no dependency information for the tests in the Bash test suite. To create a dependency structure, the authors used information from two test suites in the package. One test suite is coarse grained, with each test script containing a sequence of possibly dependent tests collated into one sequence. The other test suite is fine grained, with each test separate, and all variable declarations, which cause dependencies, pulled out into start-up scripts. The authors used the fine-grained test suite, but defined the dependencies based on the coarse-grained test suite. A dependency was defined when a variable was defined in one test and used in a subsequent test.

5.2 Independent Variables

The independent variables in both sets of experiments were the techniques used to prioritize the test cases. For the open dependency structure case studies, the techniques considered were:

1. *Coarse-grained total function coverage [cg-fn-total]*: This is a greedy prioritization algorithm proposed by Elbaum et al. [4] that schedules the next test based on the number of source-code functions it covers. There is currently no technique that proposes code-based prioritization for tests with dependencies, so this technique packages the tests into coarse-grained tests by taking an entire path through the dependency structure as a single test. The set of tests is a set of linearly independent paths through the dependency structure.
2. *Coarse-grained additional function coverage [cg-fn-addtl]*: This is similar to coarse-grained total function coverage, except that each test is scheduled based on the number of *new* functions that it covers; that is, the functions that are not covered by any of the higher priority tests. This requires the greedy algorithm to recalculate function coverage for all tests after each greedy selection. This technique and the coarse-grained total function coverage technique can be considered the state of the art in prioritization as they are two of the best performing techniques proposed to date.
3. *Coarse-grained optimal [cg-optimal]*: Proposed by Rothermel et al. [23]. Our empirical study uses systems containing known faults. As a result, it enables us to prioritize tests based on their actual fault-finding ability. This technique is similar to coarse-grained additional function coverage, except

2. See http://czt.sourceforge.net/.

3. See http://sir.unl.edu/.

that instead of using function coverage information, it uses the actual fault-finding ability of tests. This technique greedily chooses the next coarse-grained test that finds the most *new* faults. This is only possible for programs that contain known faults, so it is only of interest as an upper bound for the function coverage techniques.

4. *Untreated ordering [untreated]*: This is the order in which the original test suite was run when we obtained the systems. The ordering is based on the test engineer's acquired knowledge about the system, taking into consideration the dependencies among test cases. We do not know how these tests were prioritized, if at all. This serves as a baseline in the experiment.
5. *Breadth-first search (BFS) [bfs]*: A BFS was considered out of interest to demonstrate the effectiveness of DSP in general. If our hypothesis that executing entire scenarios is more likely to increase the fault detection rate is correct, then BFS should perform worse than the DSP and random techniques.
6. *Random ordering [random]*: While the untreated test case is considered a baseline, it may depend on the layout of the test suite. As a result, we measure the performance of 1,000 randomly ordered test suites. These are generated using an algorithm for traversing the dependency structure so as to preserve the dependency orderings. To generate these, the algorithm randomly chooses any test case that has not been previously executed, provided that all of its dependencies have been executed.
7. *Greedy prioritization [greedy]*: The greedy prioritization technique chooses the test case that finds the most new faults, provided that all of its dependencies have been executed previously. This is only possible for programs that contain known faults, so it is only of interest for comparison against our techniques.
8. *DSP height prioritization [dsp-height].*
9. *DSP volume prioritization [dsp-volume].*

For the closed dependency structure case studies, the orderings considered were:

1. *Coarse-grained total function coverage [cg-fn-total]:* as above.
2. *Coarse-grained additional function coverage [cg-fn-addtl]:* as above.
3. *Random ordering [random]:* 1,000 random orderings were again considered. In this case, all linearly independent paths in the dependency structure were calculated and were randomly concatenated to form test orderings.
4. *Greedy prioritization/coarse-grained optimal [greedy]:* This technique greedily prioritizes the set of linearly independent paths based on their fault-finding ability. This is equivalent to coarse-grained optimal prioritization, and represents an upper bound for our DSP techniques.
5. *DSP sum prioritization [dsp-sum].*
6. *DSP ratio prioritization [dsp-ratio].*
7. *DSP sum/ratio prioritization [dsp-sum-ratio].*

TABLE 2
An Example Test Defect Matrix

Test case	Faults				
	1	2	3	4	5
$t1$					x
$t2$		x	x		
$t3$		x		x	
$t4$	x	x		x	

An "x" represents that the test case uncovered the corresponding fault.

Untreated test suites were not available for the closed dependency structures, so these were not considered. In addition, BFS was not investigated for closed dependency structures. To execute a BFS, one would have to execute all level 1 tests, then execute them again to execute level 2 tests, then execute all level 1 and 2 tests to execute level 3 tests, and so on. Such test suites would achieve such low fault detection rates that they would offer no interesting insight.

We do not have access to the source code of the Elite, CRM, or MET systems. As such, we cannot report on the results of total function coverage and additional function coverage for these systems. However, we have fault reports, so we can report on the coarse-grained optimal value for these systems, which provides us with an upper bound on the value for the function coverage metrics.

Studies on prioritization techniques consider the use of *optimal* test orderings. This is similar to our greedy orderings in that they are only possible if all faults are known in advance. For a test suite with no dependencies, an optimal test suite will order test cases by their fault-finding value; that is, the test that finds the most faults is executed first. However, for open dependency structures, calculating the optimal value is NP-hard. This is discussed further in Section 7.2.4.

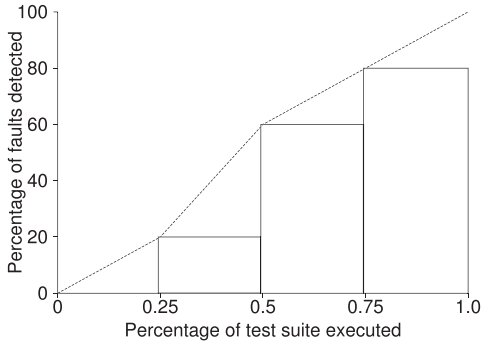
5.3 Measures

5.3.1 Measure 1: Average Rate of Fault Detection

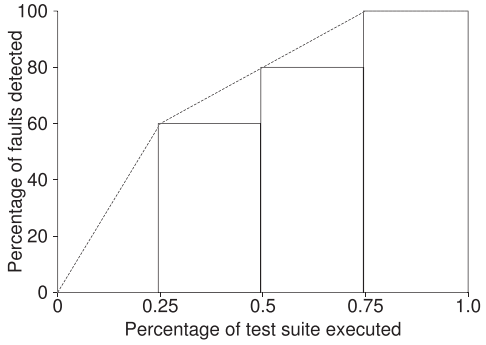
The first dependent variable of the experiments is the average rate of fault detection for each of the test suite orderings. This measures how quickly a test suite detects faults.

Definition 5.1 (APFD). *To measure the rate of fault detection, we use the* Average Percentage of Faults Detected *(APFD) metric, defined by Rothermel et al. [23]. APFD is measured as a percentage, with higher values implying a faster rate of fault detection.*

As an example, consider the test defect matrix in Table 2, in which an "x" in a cell indicates the test case (row) revealed the fault (column). If we take the "default" order $t1$-$t2$-$t3$-$t4$, then we can plot the percentage of faults detected against the percentage of the test cases that have been executed as a histogram, as in Fig. 2a. After executing $t1$, 20 percent of faults have been detected. After $t2$, 60 percent have been detected, and so on. The area under the dotted line in Fig. 2a measures the rate of fault detection. If we consider a more efficient ordering, $t4$-$t1$-$t2$-$t3$, after executing $t4$, which is now the first test case, 60 percent of faults have been detected. The area under the dotted line in Fig. 2b is larger than in Fig. 2a, which implies that the second test ordering detects faults at a faster rate.



(a) Test ordering $t1$-$t2$-$t3$-$t4$



(b) Test ordering $t4$-$t1$-$t2$-$t3$

Fig. 2. Average rate of fault detection for two test suites, each with its tests drawn for the defect matrix in Table 2.

Rothermel et al. [23] define APFD as follows:

> Let T be an ordered test suite containing n test cases and let F be a set of m faults revealed by T. Let TF_i be the first test of T which reveals fault i. The APFD for test suite T is given by the equation:

$$APFD = 1 - \frac{TF_1 + \cdots + TF_m}{nm} + \frac{1}{2n}. \quad (5)$$

For the test suite with order $t1$-$t2$-$t3$-$t4$, the first test to detect fault 1 is test $t4$, whose test order is 4, because it is the fourth test case in the order. The first tests to detect faults 2-5 are tests $t2$, $t2$, $t3$, and $t1$, respectively. Filling the values (5) will reveal the APFD to be 52.5 percent:

$$APFD = 1 - \frac{4+2+2+3+1}{4 \times 5} + \frac{1}{2 \times 4} = 52.5\%.$$

Alternatively, the test suite can be executed with ordering $t4$-$t1$-$t2$-$t3$. Note that test case 4 now has test order 1 because it is the first test case executed. This has an effect on the APFD, which is now 72.5 percent:

$$APFD = 1 - \frac{1+1+3+1+2}{4 \times 5} + \frac{1}{2 \times 4} = 72.5\%.$$

From this, one can see that a test case that detects more faults will increase the rate of fault detection if it is executed earlier.

However, APFD is only useful for comparing test suites that contain the same number of tests. This is not the case in our experiments because the coarse-grained test suites execute redundant fine-grained tests, which affects the APFD measure. Take the following example. If we run the test order $t1$-$t2$-$t3$-$t4$ from Fig. 2a, the APFD is 52.5 percent. However, if we run the order twice in succession, that is, $t1$-$t2$-$t3$-$t4$-$t1$-$t2$-$t3$-$t4$, the APFD is 86.25 percent. This is due to the fact that we execute redundant test cases after having found all of the faults and, because executing a redundant test will not uncover a new fault, more faults are detected earlier in the test order (as a percentage). Clearly, the first order is preferred over the second order, as it is half the length but finds the same number of faults.

To reduce the impact of this, the value of n, the number of tests, in (5) is the length of the longest test order defined by any of the prioritization techniques for a given system. Thus, we consider a uniform cost for executing each test suite.

5.3.2 Measure 2: Number of Test Cases Required to Find All Faults

The second measure attempts to mitigate the problem of assuming a uniform cost for executing each test suite in the APFD metric. The cost of executing a coarse-grained test suite is more than that of a fine-grained test suite due to the redundant tests, and the APFD measure does not take this into account. For example, in the GSM system, the coarse-grained test suites are almost four times the size of the fine-grained test suites. While a cost-cognizant APFD measure has been proposed [3], this assumes test suites of the same size containing individual tests with varying costs. Our evaluations consider different size test suites with uniform costs for individual tests.

To consider this execution cost element, the second measure that we take is the number of test cases that are executed before all known faults are found, which we call the *all faults* metric. As with APFD, this is expressed as a percentage of the entire test suite, and is adjusted based on the length of the test suites. If T is a test suite containing n tests and F is a set of m faults revealed by T, the *all faults* measure for T is defined as

$$AF = \frac{TF_m}{n},$$

in which TF_m is the test that finds the mth (final) fault.

5.4 Setup of Case Studies

First, we used the defect reports to identify which defects were detected by which test cases for each system. From the defect reports, we gathered the data and identified defects for each test case and collated this information in the form of a test defect matrix.

Second, we created a test dependency matrix to identify the dependencies for each test case for each system.

Third, we developed scripts that use these two sources of information to automatically calculate the ordering for the techniques being applied and to automatically calculate the APFD for these.

Finally, for each system, we generated 1,000 random subgraphs of the dependency structure and measured the APFD of all techniques over these 1,000 random samples.

6 Results

In this section, we present the results of experiments outlined in Section 5.

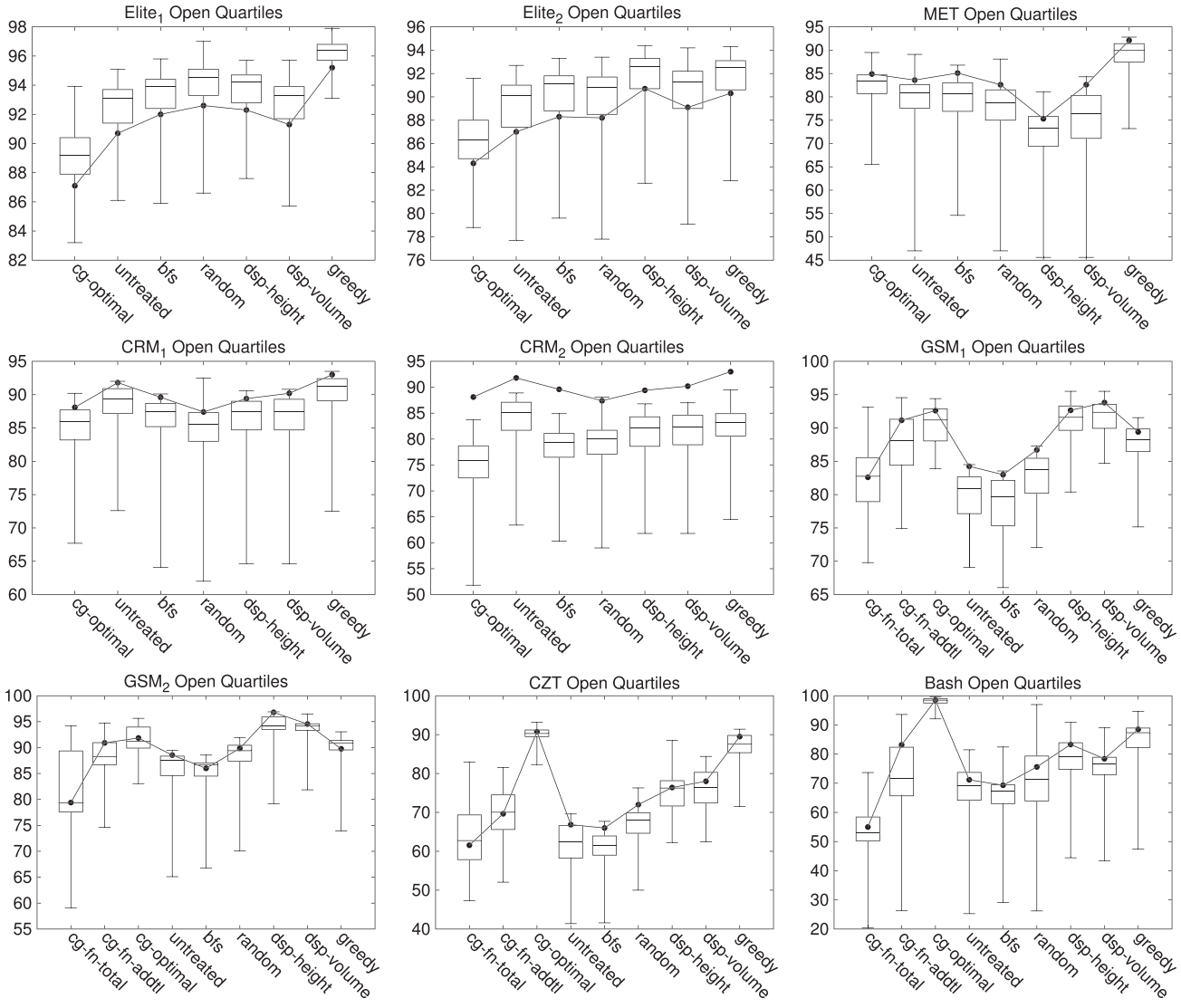


Fig. 3. APFD box plots of all systems for 1,000 random subgraphs of the open dependency graph. Line plots show the APFD for the complete test suites.

6.1 Open Dependency Structures

The APFD performances of the prioritization techniques are shown in Fig. 3. This figure presents a box plot of the minimum, maximum, median, and first/third quartile APFDs for each technique on each system for the random selection of dependency graphs. The line points show the APFDs for the complete test suites.

From these results, one can see that the DSP prioritization methods achieved a higher APFD than the coarse-grained optimal prioritization technique in six of the nine experiments, and achieved notably higher APFD than total function coverage in all four experiments for which function coverage information was available, and outperforms additional function coverage in all four as well. We note that, while we did not have function coverage information for five of the experiments, the coarse-grained optimal technique is the upper bound of function coverage prioritization, and in four of the experiments, the DSP techniques achieved a higher APFD than optimal. This provides support to our hypothesis that using fine-grained test suites is effective in prioritization.

DSP prioritization methods achieved a higher APFD than untreated and random for all experiments except Elite_1 and MET. For MET, DSP height performs markedly worse than all other methods. For Elite_2, GSM_1, and GSM_2, at least one DSP method outperforms the greedy ordering. Optimal and greedy coverage both achieved much higher APFD than other techniques for the CZT and Bash systems, which is due to the small number of faults. Neither DSP technique performs markedly better than the other.

As expected, the BFS method performed worse than random, providing evidence for our hypothesis that executing entire scenarios is more likely to increase the rate of fault detection. From the results of the untreated prioritization, it appears that untreated test suites can achieve a high rate of fault detection, but not consistently.

The DSP techniques performed better than random for all experiments except Elite_1 and MET. Recall from Section 5 that the defect reports for Elite_1 contained an outlier test case that uncovered a high number of test cases. Removing only this data from the experiment produced a noticeable change in the results, as demonstrated by the results for

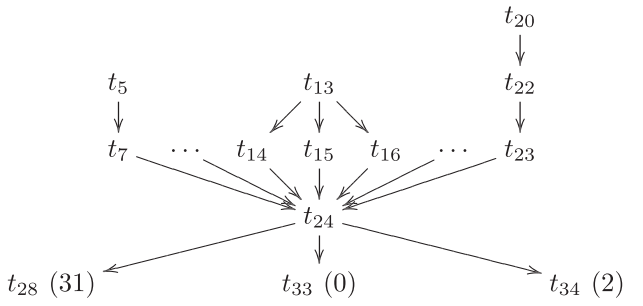


Fig. 4. An excerpt of the test case dependency structure for the *Elite* system.

Elite_2. Further investigation uncovered why the change was so remarkable.

Fig. 4 shows an excerpt of the dependency structure for the Elite system, in which the numbers in brackets on the leaf nodes specify the number of direct and indirect dependents of that node. In this structure, test case 24 (the outlier), which reportedly uncovered 27 faults, is present in almost every scenario (some unconnected parts of the graph are omitted, but these constitute only six test cases). Several independent test cases are omitted from Fig. 4, indicated using the ellipses (...). These are all direct parents of test case 24.

For any random ordering, test case 24 is likely to be executed as the second or third test case in almost every test order except the scenario that starts with test case 20. For the DSP techniques, the independent test case with the highest coverage for both DSP coverage values is test case 20 because this is both the "highest" test case as well as the test case with the most children. As such, this will be the first test executed, and test case 24 will be the fourth. As a result, the APFD of the DSP orderings is likely to be lower than for all test orderings in which test case 20 is not the first test.

The second measure taken in the experiment was the *all faults* measure. Fig. 5 shows the resulting box plots for these. Recall that a higher value implies a less efficient test suite.

From this, one can see that the DSP techniques outperform the untreated, BFS, and random orderings, and greedy performs the best. However, a more important results is regarding the coarse-grained test suites. As with APFD, our DSP methods outperformed both function coverage orderings by a notable amount, and outperformed the coarse-grained optimal metric in seven of the nine experiments.

Taking both metrics into account, the DSP test suites find faults at a faster rate than the coarse-grained test suites, and at a notably lower cost.

6.2 Closed Dependency Structures

The APFD performance of each of the prioritization techniques can be determined from Fig. 6, which presents the minimum, maximum, median, and first/third quartile APFDs for each technique on each experiment for the random selection of test cases, and the APFD for each technique applied over the complete test suite. The line points show the APFDs for the complete test suites.

From these results, one can see that the three DSP methods outperform random prioritization for all experiments except CZT. The difference between the three DSP techniques is minimal.

A more encouraging result is with the function coverage techniques. All three DSP prioritization techniques outperform total function coverage techniques in the four experiments with function coverage information, and were comparable to additional function coverage.

This is a particularly pleasing result. Although the improvement the DSP techniques show over coarse-grained techniques is larger in the open dependency experiments, that result is expected due to the function coverage test suites executing redundant tests. However, for the closed dependency suites, this is not the case: The function coverage and DSP methods execute the same paths in the dependency structure as each other, just in a different order. Therefore, they contain the same number of tests.

Results for the *all faults* measures are shown in Fig. 7.

As with the APFD, we see that DSP techniques perform better than random, although in this case, the difference is not as notable. Our DSP techniques perform comparably with additional function coverage, and marginally outperform total function coverage, which again is a pleasing result.

6.3 Threats to Validity

The first threat to validity in our evaluation is that the two metrics used to measure effectiveness consider only fault-finding ability over the number of tests and ignore other factors such as fault severity and the cost of running individual tests.

The main threat to the external validity of this experiment relates to size. The experiment was performed on only six systems (although with seven implementations in total, as we used two different implementations of the GSM system). Despite this, we believe that the variety of systems allows us to draw solid conclusions.

Finally, the test suites themselves are subject to human bias, and test suites derived by other test engineers could have produced different results.

6.4 Discussion

Despite the limitations of the study, the results of both experiments provide clear evidence that the DSP methods have potential to be used to prioritize test suites. Their performances for both APFD and the all faults metric were better than the untreated ordering and the average of the random orderings, which were used as baselines.

Our DSP methods outperformed two existing metrics, total function coverage and additional function coverage, for both open and closed dependency structures. For open dependency structures, this is expected due to the function coverage test suites executing redundant tests; however, for the closed tests suites, this was not the case: Both the function coverage and DSP methods treat the dependency structure as coarse grained and so contain the same number of tests.

The results of the experiments also add further weight to the argument that test case prioritization can improve the rate of fault detection.

One aspect of this work to consider is the cost effectiveness of DSP prioritization, considering that dependencies must be extracted and maintained. Manual effort was not recorded as part of our experiments; however, we

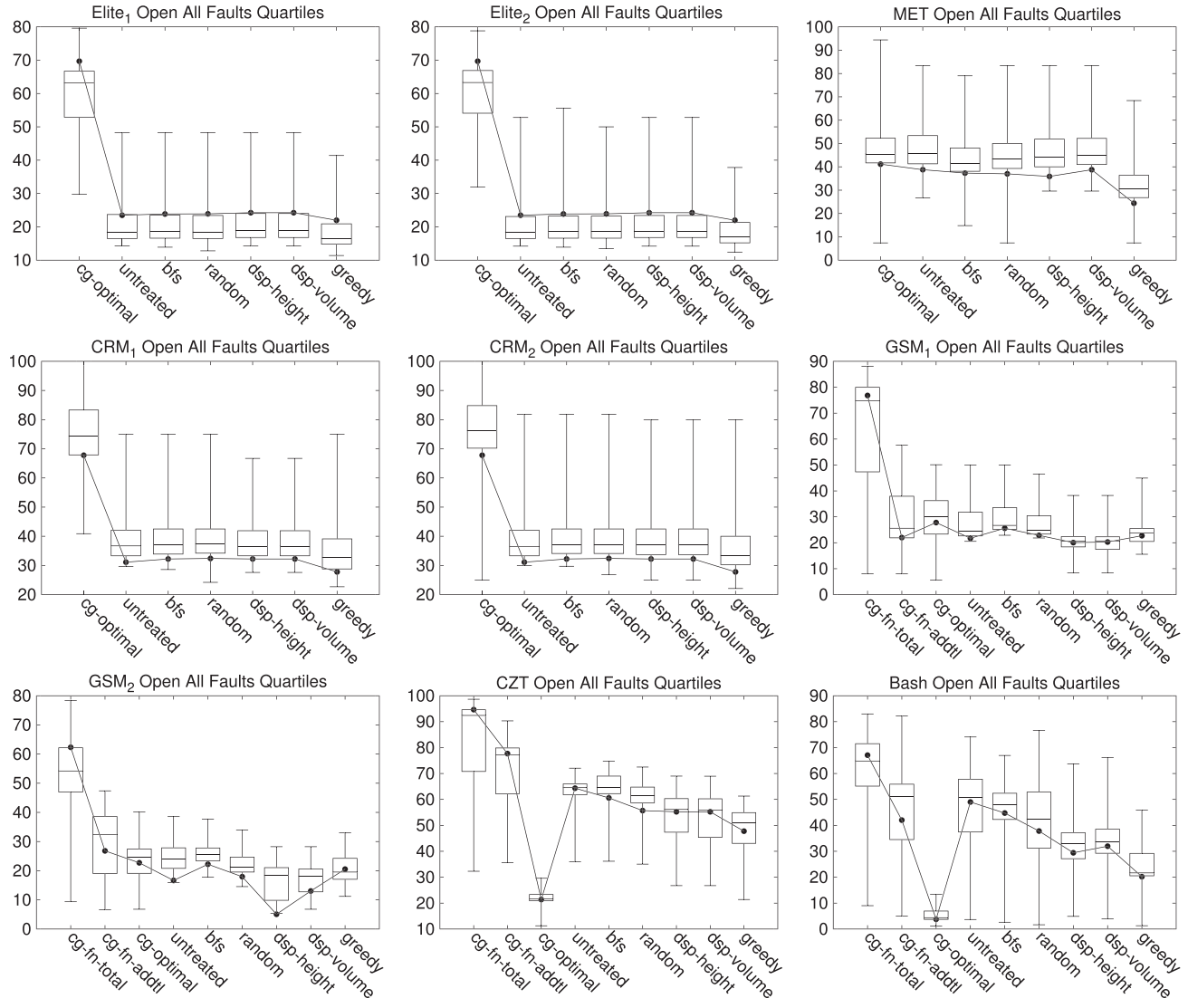


Fig. 5. Box plots of the *all faults* metric over all experiments for 1,000 random subgraphs of the open dependency graph. Line plots show the values for the complete test suites.

know from performing the case studies that the Bash system proposed the largest challenge in extracting dependencies, due to its size (1,061 tests). Despite this, the extraction took less than one day. For the Bash and CZT systems, dependency extraction could be automated in a straightforward manner by simply recording the variable definitions and uses on which the dependencies are based. We believe that the results demonstrate a cost-effective technique. Furthermore, we believe that dependency structures could be extracted automatically from the test suites themselves. If a testing framework can determine dependencies to run the tests in an order that preserves these dependencies, then the dependency structure can be extracted using the same method. Extracting fine-grained tests with dependencies from coarse-grained test suites presents a much greater challenge.

7 Related Work

Several researchers have provided solutions for test case prioritization for increasing the fault detection rate, and others have looked at the use of dependencies in testing. However, none have looked at the combination of the two. In this section, we outline the work most relevant to ours.

7.1 Test Dependencies

We are not the first authors to consider using the dependencies to improve testing efficiency. Ryser and Glinz [24] introduce *dependency charts* to represent dependencies between scenarios, and use these to drive testing. First, a natural language representation is used to map scenario descriptions to system requirements. To eliminate any ambiguities, scenarios are refined to state charts with pre and postconditions, data ranges with data values, thus clarifying the dependency structures among scenarios through graphical chart representations. On the basis of these state charts, test cases are extracted by traversing all the internal paths of the dependency charts.

Kim and Bae [8] argue that there are mainly two types of features in a system: essential and accidental. Accidental features are dependent on the essential features, thus creating levels within a system. They define a systematic approach to align the features as essential and accidental

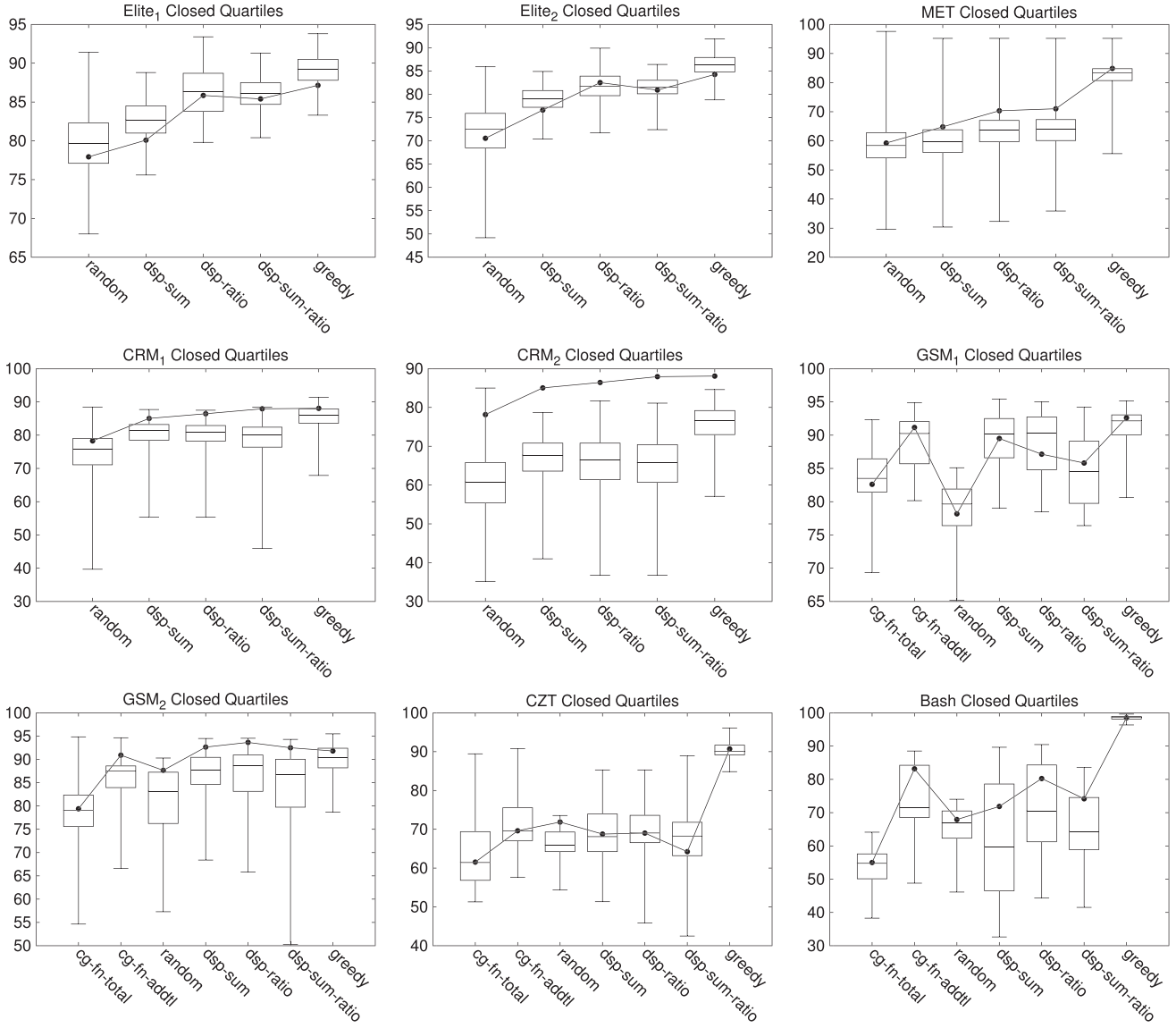


Fig. 6. APFD box plots of all experiments for 1,000 random subgraphs of the closed dependency graph. Line plots show the APFD for the complete test suites.

according to their dependencies. Accidental features change more often than the relatively static essential features; therefore, Kim and Bae claim that by properly managing the changes in accidental features effectively, systems testing effort can be reduced, ensuring trustworthiness.

Test case prioritization is not considered in either of the above approaches.

7.2 Test Case Prioritization

In this section, we outline several prioritization techniques that we consider closely related to our approach. We divide these into three categories:

- *history-based*: these techniques use information about the test suite from previous execution cycles to determine test priorities;
- *knowledge-based*: these techniques use human knowledge to determine test priorities; and
- *model-based*: these techniques use a model of the system to determine test priorities.

7.2.1 History-Based Prioritization

For improving the rate of fault detection during regression testing, Rothermel et al. [23] emphasize using execution information acquired in previous test runs to define test case priority. On the basis of this information, they define multiple techniques to prioritize test cases, including prioritizing on the basis of code coverage, prioritizing on the basis of probability of finding faults, and prioritizing on the basis of code not tested before, thereby prioritizing untested code. A greedy algorithm selects the test with the highest coverage (called *total coverage*), or the test with the highest coverage on code not covered by all tests that have already been prioritized (called *additional coverage*), until all tests have been scheduled. These techniques are shown to be effective at increasing the APFD of test suites.

We hypothesize that prioritizing on the basis of code or function coverage is similar to our dependency structure-based approach because by executing parts of the software that interacts with other parts, we are likely to be increasing

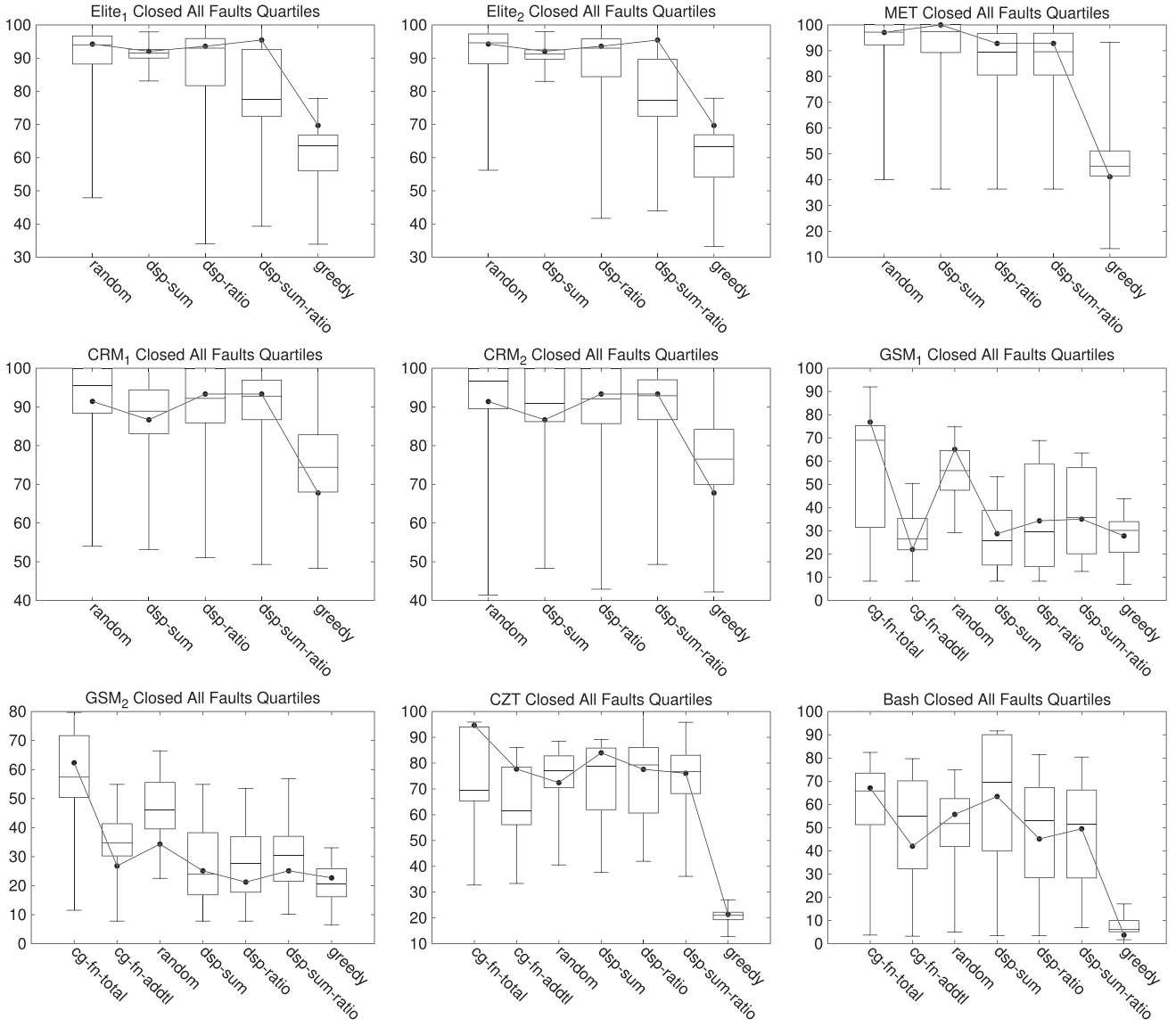


Fig. 7. Box plots of the *all faults* metric over all experiments for 1,000 random subgraphs of the closed dependency graph. Line plots show the values for the complete test suites.

code coverage. However, dependency structures contain no information about the code or functions that are executed. Future work will determine how closely related these are.

Li et al. [14] present several nongreedy algorithms for prioritizing tests, including hill climbing algorithms and genetic algorithms. Their research is in response to Rothermel et al.'s note that greedy algorithms do not prioritize tests optimally [23]. Li et al. compare these nongreedy algorithms to Rothermel et al.'s greedy algorithms, noting that while the greedy algorithms achieve the best fault detection rate, genetic algorithms also achieve a high fault detection rate, although at an increased computational cost.

Jeffrey and Gupta [6] base an approach on the number of paths covered during test runs. They called these affecting paths *relevant slices*. The higher the number of relevant slices covered, the higher the priority of that test case. The relevant slices can only be achieved through test runs.

Wong et al. [30] present a test suite minimization and prioritization technique for regression testing based on changes in the source code from previous runs. By locating the statements in that source code that have changed and using execution traces from the previous test runs, Wong et al. rank test cases based on the number of change statements that are executed. Their results indicate that the approach is more cost-effective than not using minimization or prioritization.

Li [12] presents a code-coverage-based prioritization technique. Li uses an extended form of *dominator analysis* to rank the priority of blocks of program code. A block B_1 dominates another block B_2 if a test execution cannot go through block B_2 without going through B_1. From the relationships defined between code blocks, a dominator graph is derived. Each node is then ranked based on the number of blocks that are executed if that block is executed, and test cases are prioritized based on this value. Although coverage information is required, Li uses symbolic execution to determine which tests execute which blocks. In later work [13], Li et al. use this technique to automatically generate test suites, including priority.

7.2.2 Knowledge-Based Prioritization

Ma and Zhao [15] introduce a prioritization technique based on program structure analysis to improve the effectiveness of finding severe defects. They first rank modules on the basis of a prioritization index called *testing importance*. *Testing Importance of Module* (TIM) is a combination of fault proneness and the importance of a module. After each module has been ranked, all respective test cases are then prioritized on the basis of their TIM ranking. Their experimental results show an improvement in "severe" fault detection as compared to faults detected by untreated and random prioritization.

Krishnamoorthi and Sahaaya [9] present a method for test case prioritization based on software requirements. Test cases are ranked using six factors: customer priority, requirements change, implementation complexity, completeness, traceability, and fault impact. Through this technique, system level test cases can be prioritized. Experimental results show improvement in severe fault detection as compared to random prioritization.

Srikanth et al. [26] introduce a technique similar to that of Krishnamoorthi and Sahaaya, but with different factors. They rank test cases based on the requirements volatility, customer priority, implementation complexity, and fault proneness of the requirements. Experiments showed that their technique increases the probability of detecting severe faults, and that customer priority contributes most toward this.

Tonella et al. [28] present the *case-based ranking* (CBR) system, which is a machine-learning-based approach to prioritization. It exploits already existing user knowledge regarding the program and test cases to determine test case priority by comparing two test cases. Thus, relative priority is achieved. User knowledge gathered over time is embedded in the prioritization indexes iteratively to continuously refine the CBR system.

A problem with Tonella et al.'s approach is that a large number of pairwise comparisons need to be made. For even a small test suite, the number of comparisons is high, which requires effort on behalf of the tester. To mitigate this, Yoo et al. [31] present a clustering method for test case prioritization that reduces the amount pairwise comparisons required. Clusters of tests are created based on their similarity—for example, tests that cover similar statements of the program are clustered together—and the tester is requested to do pairwise combination on a single test from each cluster, with the assumption that the single tests are representative of the entire cluster. Yoo et al. also rank the priority of tests *within* clusters using statement coverage, and *interleave* tests between clusters to arrive at a prioritized order. Their empirical results demonstrate that their method can outperform standard statement-based coverage.

Qu et al. [20] define a prioritization technique based on combinatorial interaction testing (CIT). CIT is a systematic technique for choosing subsets of test case combinations to control the combinatorial explosion problem. To prioritize tests, the test engineer first weights the *choice* (uncombined tests) in each category using some knowledge of how the tests related to the system, such as complexity or code coverage. For each choice, the weight of combining it with each other choice (the combinations) provides an *interaction benefit*. Tests are then prioritized based on this interaction benefit. Results from a simple pairwise strategy increase fault detection rate in regression testing compared to an all combination, and a standard branch-based technique achieved the best rates of fault detection overall.

Basanier et al. [2] describe an approach for prioritizing test sequences generated from UML models. Their method generates *main trees*, in which each level of a tree represents a different integration level, and model the hierarchical dependencies between these levels. A test engineer ranks the *weight* of nodes in trees with weights in the range $[0, 1]$, where the weight represents the importance of the test. The weight of all children of a node must total to 1. At each integration level the priority of a test is the product of all nodes from the root node to the node representing that test.

Jiang et al. [7] present an adaptive random testing (ART) technique for test prioritization. We have categorized this as knowledge based because knowledge about the input domain is required; however, it is automatable. Jiang et al. use a greedy algorithm for prioritization which schedules the next test that is furthest away from all tests that have already been prioritized. The distance measure is based on ART techniques, which aim to spread the selection of random tests as evenly as possible over the input domain. The results demonstrate that ART prioritization achieves a higher fault detection rate than random prioritization and total coverage prioritization, but that additional coverage prioritization achieves a higher fault detection rate than ART prioritization. However, ART prioritization is markedly less computationally expensive than additional coverage techniques.

7.2.3 Model-Based Prioritization

Recent work by Kundu et al. [10] has produced a test case generation and prioritization technique for object-oriented programs that is perhaps the most closely related to our work. This technique takes UML sequence diagrams as system models and translates them into *sequence graphs*, which are graphs that model all interactions between the system and the user. Test cases are generated by taking all basic paths in the sequence graph, and generating a test case for each path. Each basic path represents a single scenario in the system.

Each edge in the sequence graph is given two weights: 1) a *message weight*, which is the number of messages between the edges nodes, and 2) an *edge weight*, which is the number of paths in the sequence diagram that pass through this edge. Using these weights, three prioritization metrics for paths are defined: 1) the sum the message weights of all edges in a path, 2) the weighted average edge weight of all edges in a path, and 3) the average of the message weight multiplied by the edge weight for all edges in a path. Kundu et al. do not take measures such as APFD in their evaluation; however, results indicate that their techniques can increase the rate of code coverage, which is likely to increase APFD.

The result of Kundu et al.'s technique is similar to ours in that interactions between different system components are given a higher priority in the system. However, our technique can be applied to any set of test cases with dependencies, so it is not linked to any particular type of system modeling tools or languages. In addition, we

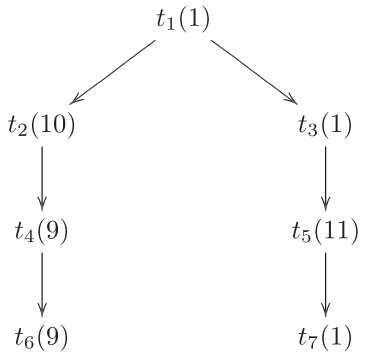


Fig. 8. An example dependency structure with test case weights.

maintain fine-grained test suites, whereas Kundu prioritizes entire test sequences.

7.2.4 Discussion

From the related work, it can be contended that test case prioritization helps in improving all testing effort and not just regression testing. The results in this paper further strengthen this claim.

With the exception of Kundu et al. [10], these techniques do not consider functional dependencies between test cases. Furthermore, many of these methods require either information from previous test runs or input from the tester. In contrast, our techniques require only the test case dependencies. These dependencies need to be considered during testing whether our approach is used or not. As Ryser and Glinz [24] note, "*it is not possible to test the system thoroughly if the dependencies and relations are not considered appropriately*." The only overhead in our approach is in recording test dependencies in a format such that the dependency structure can be extracted.

Further to the above points, it is not immediately clear how straightforward it would be to apply many existing prioritization techniques to test cases with dependencies.

Take the example in Fig. 8. In this example, the numbers in parentheses represent test case quality (or rank), such as code coverage achieved or expected fault-finding ability. Existing prioritization techniques advocate that t_5 is the highest quality test case, so it should be run first. However, this violates the dependencies, because the sequence t_1-t_3 must be run before t_5. For a naive algorithm that chooses t_5 as the highest priority, running t_1-t_3-t_5, is not optimal. If we consider that the numbers in the parentheses represent perfect predictions of the amount of faults that each test will find, we have the following APFD scores:

$$APFD(t_1\text{-}t_2\text{-}t_4\text{-}t_6\text{-}t_3\text{-}t_5\text{-}t_7) = 62\%,$$
$$APFD(t_1\text{-}t_3\text{-}t_5\text{-}t_2\text{-}t_4\text{-}t_6\text{-}t_7) = 56\%.$$

Therefore, if one runs the left branch followed by the right branch, the score is higher. The problem is further complicated if we consider that there may be more tests that depend on t_6 and t_7. To calculate the optimal score, one is required to know the weights of these other test cases.

Such a scheduling problem is equivalent to scheduling jobs with precedence constraints on a single machine, which is known to be NP-hard [25]. If we consider cases such as additional function coverage, the problem becomes even more difficult because the coverage achieved by a test changes, dependent on the tests that are run before it. Despite these difficulties, approximation algorithms could be used for coverage-based prioritization methods to handle dependencies.

As a final point, we consider the combination of dependency-based prioritization with other prioritization techniques. Even though our techniques perform well, the results show that for systems containing many unconnected tests, such as in the CZT and Bash systems, the performance is lower compared to systems with fewer unconnected tests. We attribute this to the fact that a set of unconnected tests will be arbitrarily prioritized by our techniques. To improve our methods, other prioritization methods could be used to prioritize unconnected tests. This requires information other than dependency information. We see the value in an approach that combines dependency information and other information, such as coverage, to improve test ordering.

8 Conclusions and Future Work

In this paper, we have defined a new set of techniques for prioritizing test suites that contain dependencies between test cases. The techniques prioritize tests based on the dependency structure of the test suite itself. Six systems being developed toward use in industry are used to empirically assess the strength of these new techniques, measured by the average fault detection rate, in comparison to randomly generated test suites, greedily generated test suites, and the untreated test suite used by test engineers, where available. The results indicate that test suites prioritized by our techniques outperform the random and untreated test suites, but are not as efficient as the greedy test suites. In addition, for open dependency graphs, our techniques achieved better APFDs for most experiments than the state-of-the-art coarse-grained function coverage techniques. For open dependency structures, this improvement was at a greatly lower execution cost. For closed dependency graphs, our techniques achieved better APFDs than total function coverage, and was comparable to additional function coverage.

The results indicate that our techniques offer a solution to the prioritization problem in the presence of test cases with dependencies. There are two significant strengths to our approach. First, information from previous test runs is not needed to calculate the priorities, so our techniques can be used on first versions of systems. Furthermore, it can be used even if previous test runs have not completed, which is useful in development processes containing short iterations. Second, maintaining fine-grained test suites and prioritizing these based on dependencies preserves the flexibility of fine-grained test suites, while also enabling larger test scenarios to be uncovered, thus increasing the probability of each test finding a fault.

The results of the empirical study demonstrate the potential of our techniques for prioritization and justify further investigation in this area. In particular, we are interested in automatically extracting dependency structures from test suites. A further investigation will assess who to extract dependency structures from models in model-based testing, and how to use additional information from the

models to increase the effectiveness of the techniques. For example, the weight of a test case may be lower if it and its children execute model operations that have been previously executed, in comparison to a node where new operations are executed, thereby increasing function coverage. Combinations of dependency structure prioritization and existing prioritization techniques could prove useful for test suites that contain many tests with no dependents or dependencies, such as the CZT and Bash systems outlined in Section 5.

In addition to assessing the techniques on more systems, we also plan to compare our techniques with other types of prioritization techniques, such as other code-coverage-based techniques [4]. First, however, we must modify those techniques to consider test case dependencies, as these techniques assume no dependency relationship between test cases.

REFERENCES

[1] J. Bach, "Useful Features of a Test Automation System (Part iii)," *Testing Techniques Newsletter,* Oct. 1996.

[2] F. Basanieri, A. Bertolino, and E. Marchetti, "The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects," *Proc. Fifth Int'l Conf. Unified Modeling Language,* pp. 275-303, 2002.

[3] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization," *Proc. 23rd Int'l Conf. Software Eng.,* pp. 329-338, 2001.

[4] S. Elbaum, A.G. Malishevsky, and G. Rothermel, "Test Case Prioritization: A Family of Empirical Studies," *IEEE Trans. Software Eng.,* vol. 28, no. 2, pp. 159-182, Feb. 2002.

[5] R.W. Floyd, "Algorithm 97: Shortest Path," *Comm. ACM,* vol. 5, no. 6, p. 345, June 1962.

[6] D. Jeffrey and N. Gupta, "Experiments with Test Case Prioritization Using Relevant Slices," *J. Systems and Software,* vol. 81, no. 2, pp. 196-221, 2008.

[7] B. Jiang, Z. Zhang, W. Chan, and T. Tse, "Adaptive Random Test Case Prioritization," *Proc. IEEE/ACM Int'l Conf. Automated Software Eng.,* pp. 233-244, 2009.

[8] J. Kim and D. Bae, "An Approach to Feature Based Modelling by Dependency Alignment for the Maintenance of the Trustworthy System," *Proc. 28th Ann. Int'l Computer Software and Applications Conf.,* pp. 416-423, 2004.

[9] R. Krishnamoorthi and S.A. Sahaaya Arul Mary, "Factor Oriented Requirement Coverage Based System Test Case Prioritization of New and Regression Test Cases," *Information and Software Technology,* vol. 51, no. 4, pp. 799-808, 2009.

[10] D. Kundu, M. Sarma, D. Samanta, and R. Mall, "System Testing for Object-Oriented Systems with Test Case Prioritization," *Software Testing, Verification, and Reliability,* vol. 19, no. 4, pp. 97-333, 2009.

[11] K.S. Lew, T.S. Dillon, and K.E. Forward, "Software Complexity and Its Impact on Software Reliability," *IEEE Trans. Software Eng.,* vol. 14, no. 11, pp. 1645-1655, Nov. 1988.

[12] J. Li, "Prioritize Code for Testing to Improve Code Coverage of Complex Software," *Proc. 16th IEEE Int'l Symp. Software Reliability Eng.,* pp. 75-84, 2005.

[13] J. Li, D. Weiss, and H. Yee, "Code-Coverage Guided Prioritized Test Generation," *J. Information and Software Technology,* vol. 48, no. 12, pp. 1187-1198, 2006.

[14] Z. Li, M. Harman, and R. Hierons, "Search Algorithms for Regression Test Case Prioritization," *IEEE Trans. Software Eng.,* vol. 33, no. 4, pp. 225-237, Apr. 2007.

[15] Z. Ma and J. Zhao, "Test Case Prioritization Based on Analysis of Program Structure," *Proc. 15th Asia-Pacific Software Eng. Conf.,* pp. 471-478, 2008.

[16] T. Miller and P. Strooper, "A Case Study in Model-Based Testing of Specifications and Implementations," *Software Testing, Verification, and Reliability,* vol. 22, no. 1, pp. 33-63, 2012.

[17] B. Peirce, "Criterion for the Rejection of Doubtful Observations," *Astronomical J. II,* vol. 45, pp. 161-163, 1852.

[18] C. Perrow, *Normal Accidents: Living with High-Risk Technologies.* Princeton Univ Press, 1999.

[19] J. Poole, "A Method to Determine a Basis Set of Paths to Perform Program Testing," Nat'l Inst. of Standards and Technology, Nov. 1995.

[20] X. Qu, M.B. Cohen, and K.M. Woolf, "Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization," *Proc. IEEE Int'l Conf. Software Maintenance,* pp. 255-264, 2007.

[21] S.M. Ross, "Peirce's Criterion for the Elimination of Suspect Experimental Data," *J. Eng. Technology,* vol. 20, no. 2, pp. 38-41, 2003.

[22] G. Rothermel, S. Elbaum, A. Malishevsky, P. Kallakuri, and B. Davia, "The Impact of Test Suite Granularity on the Cost-Effectiveness of Regression Testing," *Proc. 24th Int'l Conf. Software Eng.,* p. 130, 2002.

[23] G. Rothermel, R.H. Untch, C. Chu, and M.J. Harrold, "Prioritizing Test Cases for Regression Testing," *IEEE Trans. Software Eng.,* vol. 27, no. 10, pp. 929-948, Sept. 2001.

[24] J. Ryser and M. Glinz, "Using Dependency Charts to Improve Scenario-Based Testing," *Proc. 17th Int'l Conf. Testing Computer Software,* 2000.

[25] J.B. Sidney, "Decomposition Algorithms for Single-Machine Sequencing with Precedence Relations and Deferral Costs," *Operations Research,* vol. 23, no. 2, pp. 283-298, 1975.

[26] H. Srikanth, L. Williams, and J. Osborne, "System Test Case Prioritization of New and Regression Test Cases," *Proc. Fourth Int'l Symp. Empirical Software Eng.,* pp. 62-71, 2005.

[27] P. Stocks and D. Carrington, "A Framework for Specification-Based Testing," *IEEE Trans. Software Eng.,* vol. 22, no. 11, pp. 777-793, Nov. 1996.

[28] P. Tonella, P. Avesani, and A. Susi, "Using the Case-Based Ranking Methodology for Test Case Prioritization," *Proc. 22nd IEEE Int'l Conf. Software Maintenance,* pp. 123-133, 2006.

[29] S. Warshall, "A Theorem on Boolean Matrices," *J. ACM,* vol. 9, no. 1, pp. 11-12, Jan. 1962.

[30] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal, "A Study of Effective Regression Testing in Practice," *Proc. Eighth Int'l Symp. Software Reliability Eng.,* pp. 230-238, 1997.

[31] S. Yoo, M. Harman, P. Tonella, and A. Susi, "Clustering Test Cases to Achieve Effective and Scalable Prioritisation Incorporating Expert Knowledge," *Proc. 18th Int'l Symp. Software Testing and Analysis,* pp. 201-212, 2009.

Shifa-e-Zehra Haidry received the master's degree in software systems engineering from the University of Melbourne in 2009. She has more than 7 years of experience in the field of software quality assurance and testing. Currently, she is working as a senior software quality engineer at an IT development company. Her main research interests include software quality and software verification and validation, specifically software quality improvement through process engineering and testing tools and techniques.

Tim Miller received the PhD degree in 2005 from the University of Queensland. He is a lecturer in the Department of Computing and Information Systems at The University of Melbourne. He spent 4 years at the University of Liverpool, United Kingdom, as a postdoctoral researcher associate in the Agent ART group. In 2008, he moved to Melbourne to take up his current post. His primary research interests include the area of software engineering for intelligent systems.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/publications/dlib.