PHALANX: A Graph-Theoretic Framework for Test Case Prioritization

Murali Krishna Ramanathan
Dept. of Computer Science
Purdue University
rmk@cs.purdue.edu

Mehmet Koyuturk
Dept. of Electrical Engineering & Computer Science
Case Western Reserve University
koyuturk@eecs.case.edu

Ananth Grama, Suresh Jagannathan
Dept. of Computer Science
Purdue University
{ayg, suresh}@cs.purdue.edu

ABSTRACT

Test case prioritization for regression testing can be performed using different metrics (e.g., statement coverage, path coverage) depending on the application context. Employing different metrics requires different prioritization schemes (e.g., maximum coverage, dissimilar paths covered). This results in significant algorithmic and implementation complexity in the testing process associated with various metrics and prioritization schemes. In this paper, we present a novel approach to the test case prioritization problem that addresses this limitation. We devise a framework, PHALANX, that identifies two distinct aspects of the problem. The first relates to metrics that define ordering relations among test cases; the second defines mechanisms that implement these metrics on test suites. We abstract the information into a test-case dissimilarity graph – a weighted graph in which nodes specify test cases and weighted edges specify user-defined proximity measures between test cases. We argue that a declustered linearization of nodes in the graph results in a desirable prioritization of test cases, since it ensures that dissimilar test cases are applied first. We explore two mechanisms for declustering the test case dissimilarity graph – *Fiedler (spectral) ordering* and a *greedy* approach. We implement these orderings in PHALANX, a highly flexible and customizable testbed, and demonstrate excellent performance for test-case prioritization. Our experiments on test suites available from the Subject Infrastructure Repository (SIR) show that a variety of user-defined metrics can be easily incorporated in PHALANX.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging — Testing tools; D.2.5 [**Software Engineering**]: Testing and Debugging — Tracing; D.2.9 [**Software Engineering**]: Management – Software quality assurance

SAC' 08 March 16-20, 2008, Fortaleza, Ceará, Brazil.

Keywords

regression testing, test prioritization, program analysis

1. INTRODUCTION

Regression testing is an important part of the software development cycle [14, 27, 33]. It ensures that software is backward compatible, i.e., desired properties in the newer version of a program are consistent with older versions, or that a change in a program property with respect to the older version is intentional. One of the critical concerns with this process is the time taken to execute all test suites on a newer version [33]. Consequently, techniques to prioritize test cases [8, 32, 33] or eliminate seemingly redundant tests [13, 19, 34] assume significant importance. While eliminating redundant test case can reduce the time taken for regression testing, there remains the possibility that bugs in the newer version can go undetected [34]. Consequently, prioritization remains an attractive, albeit conservative, alternative; its main goal is to detect as many bugs in the shortest time possible, so that bugs can be patched early, rather than eliminating test cases that are redundant with respect to previously encountered tests. The test case prioritization problem is defined as follows by Rothermel *et al.* [32]:

DEFINITION 1. **Test Case Prioritization.** *Let T be a test suite,* i.e.*, a set of test cases. Let Π be the set of permutations of T. Given prioritization function $f : \Pi \rightarrow \mathbb{R}$, find $\pi' \in \Pi$ such that for all $\pi \in \Pi$, $f(\pi') \geq f(\pi)$.*

Here, f is a measure of the quality of a permutation of the test cases. A larger value of $f(\pi)$ corresponds to a more desirable prioritization, characterized by π. The matter of choosing a good f is an important design criterion in test case prioritization. A commonly used class of prioritization functions relates to the *coverage* of certain program properties. Many coverage based test prioritization approaches have been proposed [8, 13, 32, 33]. In these approaches, a test case is selected first if it provides *maximum coverage* of some program property. For certain program properties (e.g., path coverage [18, 19, 21]), the notion of maximum coverage may be ill-defined. For example, program behavior with respect to path coverage is characterized by the *sequence* of actions it performs or the *sequence* of statements it executes. If test case prioritization is to be performed based on such program properties, it is beneficial to consider the

dissimilarity a test case has with other tests in the test suite. By executing dissimilar test cases upfront, different aspects of the program are tested earlier.

The underlying theme of our work is that a broad range of program-centric properties can be quickly tested by executing dissimilar tests one after another. Motivated by this observation, we separate issues related to prioritization into two groups. The first one is concerned with metrics that abstract test case functionality based on specific program-centric properties, and which defines an ordering specification based on this abstraction. We conceptualize this specification as a mapping from test cases to points in a metric space where the distance between two points is a measure of their dissimilarity. For example, a possible choice for this metric is one that abstracts test case behavior with respect to the difference in the statements, paths, functions, sequence of functions, branches, faults, etc. that are executed. According to this abstraction, test cases that generate mostly the same elements (e.g., statements, paths, etc.) reside closer in the corresponding metric space, compared to test cases that have less similarity with respect to the generated elements. The second layer is concerned with algorithmic issues necessary to efficiently construct a priority ordering from these points.

Contributions

The contributions of this paper are three-fold:

1. We present a two-tiered architecture, PHALANX , that separates the issue of a specific user-defined prioritization metric from the algorithmic aspects of identifying dissimilar test cases based on the ordering induced by the metric.

2. An examination of many user-defined metrics for test case prioritization. For notions of dissimilarity between paths, we employ techniques that compute longest common subsequences [16] (LCS) to obtain an edit distance between paths; this edit distance is then used to define edge weights between test cases.

3. A case study of our approach on a number of real-world benchmarks and test-suites obtained from Subject Infrastructure Repository [7]. Our results indicate that PHALANX is effective for regression testing over a range of different user-defined metrics.

2. MOTIVATING EXAMPLE

Consider the code fragment shown below, which takes two arguments and characterizes the first argument as odd(`s2`) or even(`s3`) and positive(`s5`) or negative(`s6`) and simply prints(`s4`) the second argument after storing(`s1`) it in a local variable `str`:

```
void main(int argc, char *argv[]) {
  /* assume argv[0] = "prog" */
   int x;
   char *str = NULL;
   if(argc > 2) {
        str = (char *)malloc(strlen(argv[2])+1);
        strcpy(str, argv[2]);  /* s1 */
   }
   x = atoi(argv[1]);
   if(x%2 != 0)
        write(1, "odd input",9); /* s2 */
   else
        write(1, "even input",10); /* s3 */
   /* assume in older version
   write(1,argv[0],strlen(argv[0])) */
   write(1, str, strlen(str)); /* s4 */
   if(x > 0)
        write(1, "positive",8); /* s5 */
   else
        write(1, "negative",8); /* s6 */
}
```

	arg1	arg2
t1	7	abc
t2	8	abc
t3	-7	abc
t4	-8	abc
t5	8	-

Table 1: Arguments for test cases.

	s1	s2	s3	s4	s5	s6
t1	x	x		x	x	
t2	x		x	x	x	
t3	x	x		x		x
t4	x		x	x		x
t5			x	x	x	

Table 2: Statement coverage by test cases.

Assume that an older version of the above fragment executes the statement `s4` with `argv[0]` instead of `str`. In the newer version, the above fragment is obviously erroneous (at statement `s4`), if there is exactly one argument provided. Consider five test inputs on the older version as shown in Table 1.

The coverage table using statement coverage as the testing metric is given in Table 2. The method presented in Rothermel *et al.* [32] on uncovered statement coverage can be used in the above context. The obtained ordering of test cases is `t1, t4, t2, t3` and `t5`. This is because test case `t1` covers statements `s2` and `s4`, while test case `t4` covers statements `s3 and s6` apart from commonly covering `s1` and `s4`. Similar reasoning applies for test cases `t2` and `t3`. Another potential ordering is `t2, t3, t1, t4` and `t5`. While other equally valuable orderings are possible, in a majority of such orderings, test case `t5` is placed at the end. Notice that `s4` appears in all the tests and therefore it becomes harder to differentiate between test cases. Such application contexts motivate the exploration of different prioritization techniques.

In the above case, the correctness of `s4` is dependent on whether `s1` is executed or not. Therefore, a user-defined metric that uses path information can effectively identify the error. By applying PHALANX with path coverage as the user defined metric, we obtained the ordering as `t2, t3, t5, t1` and `t4`. For some other application, prioritization based on branch-coverage may outperform prioritization based on paths. In fact, the effectiveness of a prioritization scheme is significantly dependent on the property of the program under consideration, the test suite [8], the required level of efficiency in regression testing, and the time available for constructing an ordering. The focus of this paper is to alleviate the test designer from issues peripheral to the testing process (e.g., algorithm design for prioritization) by designing a framework, which will enumerate a list of prioritized test cases given the quantified dissimilarities among every pair of test cases.

3. PHALANX: THE FRAMEWORK

3.1 Test Case Dissimilarity and Minimum Similarity Ordering

For a given test suite, a program, and a user-defined measure of dissimilarity, we compute a dissimilarity function $\delta_P : T \times T \rightarrow \mathbb{R}$ as follows. For an existing version of a program, each test case is executed. The program is instrumented to obtain the memory operation sequence trace for each test case. A trace is a sequence of <*Operation, Value*> tuples, where *Operation* is either a read or write to memory and *Value* is the value read from or written into memory. The trace is analogous to a string and the tuple is analogous to a letter in an alphabet. Comparing two test cases is equivalent to comparing the corresponding trace sequences. Based on a user-defined cost function, the Levenstein [16] distance between any pair of trace sequences is calculated using dynamic programming [4]. (The Levenstein distance between two strings is defined as the shortest sequence of edit operations that lead from one string to the other.) If the Levenstein distance between the two test cases is zero, then we regard the two test cases as being identical.

Once a dissimilarity function is specified, we define an optimization problem with an objective function that associates the dissimilarity between test cases with their temporal proximity in the resulting ordering. This approach is based on the idea that dissimilar test cases should be executed close to each other.

DEFINITION 2. **Minimum Similarity Ordering Problem (MSO).** *Given test suite T, program P, and a test case dissimilarity function δ_P, let $\pi = \{i_1, i_2, ..., i_{|T|}\}$ define a permutation of the test cases in T. Among all permutations of T, find a permutation π that minimizes the weighted aggregate distance induced by π, which is defined as*

$$\Delta(\pi) = \sum_{j=1}^{|T|-1} \sum_{k=j+1}^{|T|} \delta_P(t_i, t_j)|i_j - i_k|. \quad (1)$$

Observe that $\Delta(\pi)$ is a prioritization function specified by Definition 1. Therefore, the framework that is characterized by the notion of dissimilarity function and the minimum similarity ordering problem defines a class of test case prioritization problems. The Minimum similarity ordering problem is NP-hard, since the *optimal linear arrangement* problem, a special case of MSO where δ_P is a binary function, is known to be NP-hard [17]. However, these and other related intractable graph problems have been well-studied, and many effective heuristics exist in the literature. To take advantage of this, we propose a graph model for test case prioritization to study the problem in a graph-theoretic framework.

3.2 Test Case Dissimilarity Graph

From the instrumented data, we construct a *test case dissimilarity graph* defined as follows:

DEFINITION 3. **Test Case Dissimilarity Graph.** *Given test suite T and program P, and a test case dissimilarity function δ_P, the corresponding test case dissimilarity graph $G(V, E, w)$ is a weighted undirected graph with vertex set V, edge set E, and weight function $w : E \rightarrow \mathbb{R}$, constructed as follows. For all $t_i \in T$, there exists $v_i \in V$. For all pair of test cases, $t_i, t_j \in T$, there exists $v_i v_j \in E$, such that $w(v_i v_j) = \delta_P(t_i, t_j)$.*

This model transforms the minimum similarity ordering problem into a graph optimization problem. The goal is to find an optimal ordering of vertices and the distance between any pair of vertices in the ordering is penalized by a factor that is equal to the weight of the edge between these two vertices.

We illustrate the process of generating a test case dissimilarity graph for the example given in Section 2. By instrumenting the older version of the program (recall that statement `s4` is executed with `argv[0]` instead of `str`) and executing the test cases t_1, t_2, t_3, t_4, and t_5, we obtain the following sequences:

```
t1: <W, abc> <W, odd input> <W, prog> <W, positive>
t2: <W, abc> <W, even input> <W, prog> <W, positive>
t3: <W, abc> <W, odd input> <W, prog> <W, negative>
t4: <W, abc> <W, even input> <W, prog> <W, negative>
t5: <W, even input> <W, prog> <W, positive>
```

As suggested earlier, the dissimilarity between any two test cases is the edit distance corresponding to the trace sequences. For example, the edit distance between test cases t_3 and t_5 is computed as follows:

```
t3: <W,abc> <W, odd input> - <W,prog> <W,negative> -
t5: - - <W, even input> <W,prog> - <W,positive>
```

Counting the number of gaps(`-`), we find that the edit distance is five. Computing the edit distance between all possible pairs of sequences, we obtain the following adjacency matrix A_G of the test case dissimilarity graph G:

$$A_G = \begin{bmatrix} 0 & 2 & 2 & 4 & 3 \\ 2 & 0 & 4 & 2 & 1 \\ 2 & 4 & 0 & 2 & 5 \\ 4 & 2 & 2 & 0 & 3 \\ 3 & 1 & 5 & 3 & 0 \end{bmatrix} \quad (2)$$

This metric, which we refer to as *Memory Value Sequence* (MVSQ), is one of four metrics that we use in this paper. The other metrics used in the paper are enumerated below:

1. *Function Sequence* (FSQ): Test case differentiation is based on function call sequences.
2. *Branch Sequence* (BSQ): Test case differentiation is based on branch operation sequences.
3. *Memory Statement Sequence* (MSSQ): Differentiation is based on sequence of memory statements without taking the values read or written into consideration.

3.3 Algorithms for Prioritizing Test Cases

The test case dissimilarity graph G is used to generate a minimum similarity ordering of the nodes. In this paper, we present two approaches to solve this problem; the first uses a simple greedy algorithm, and the second uses Fiedler ordering.

3.3.1 Greedy Algorithm

The greedy algorithm orders the nodes of $G(V, E, w)$ by growing an ordered list ℓ of nodes and locally minimizing the optimization function at each step by appending the node that is most heavily connected (dissimilar) to the nodes in the list. The output to the algorithm is the ordered list ℓ.

1. Select edge $uv \in E$ such that has the $w(uv)$ is maximum among all edges. Let $\ell = [u, v]$. The test-case that corresponds to u is the head b of list ℓ and v is the tail e.

2. While not all $v \in V$ are present in ℓ, do the following:

   - Among all edges $ev \in E$, where $v \notin \ell$ pick the edge with maximum weight $w(ev)$.
   - Add v to ℓ as the new tail e.

3. List ℓ gives the ordered list of test cases.

In the example illustrated above, an execution of the greedy approach orders the test cases as t_3, t_5, t_1, t_2 and t_4. Upon careful observation, it is evident that this ordering is not necessarily unique when equal edge weights are present in the test case graph. Furthermore, the ordering obtained using such a mechanism is not always globally optimal. In some cases, after choosing the edge with maximum weight in the graph, subsequent edges chosen may have smaller weight than the remaining edges in the graph (discussed in Section 3.3.2). Therefore, we apply a better approximation for optimal linear ordering problems [25] and evaluate the use of Fiedler vectors for ordering test cases.

3.3.2 Spectral ordering Algorithm

Due to space limitations, we are unable to present a brief overview of spectral graph theory and refer the reader to [17, 10, 15, 25]. Once we generate the test case similarity graph G, we use Fiedler(spectral) ordering to find a desirable test case prioritization through the following procedure:

1. Compute Laplacian L_G of G.
2. Compute the second smallest eigenvalue of L_G, $\lambda_2(L_G)$.
3. Extract the eigenvector x_2 that corresponds to $\lambda_2(L_G)$.
4. Sort the entries of x_2 and reorder the indices (that represent the vertices in G) accordingly.
5. The order of vertices that correspond to the reordered list of indices forms the desired ordering of test cases.

Recall the motivating example in Section 2. We compute the Fiedler vector for the graph G given in Equation 2 as follows. The eigenvalues of the Laplacian of these graph are 0, 0.4801, 0.5440, 0.6774, and 0.7761. The eigenvector corresponding to the second smallest eigenvalue $\lambda_2 = 0.4801$ is

$$x_2 = [0.3805, -0.7486, -0.2801, 0.3805, 0.2676]^T, \qquad (3)$$

whose entries are ordered as $x_2(2) < x_2(3) < x_2(5) < x_2(1) < x_2(4)$. Reordering the test cases accordingly, we obtain the following Fiedler ordering for test case prioritization: $\{t_2, t_3, t_5, t_1, t_4\}$. Note that although t_5 is scheduled earlier by the greedy algorithm, this need not always be the case. A greedy approach optimizes locally whereas the Fiedler ordering provides an approximation to the global optimum. Also observe that, using the prioritization given by the Fiedler ordering, all possible paths in the program are explored within the first three tests.

4. EXPERIMENTAL RESULTS

4.1 Implementation

The implementation consists of three components for performing the following tasks: (i) instrumentation, (ii) generating a test case dissimilarity graph, and (iii) computing the priority of test cases. We use PIN [23], a dynamic binary instrumentation tool, for instrumentation purposes. A test case dissimilarity graph is a connected graph(see Definition 3). The only information that is needed for generating a test case graph corresponds to the edge weights. These edge weights are calculated based on the prioritization metric. Since the prioritization metrics explored in this paper are based on paths, edge weights are given by the edit distance between the corresponding paths. Dynamic programming takes quadratic time in the total length of the sequence and is not scalable to long sequences. To alleviate this scalability problem, we discuss an approach using Rabin fingerprinting [28] on generated sequences. We use existing implementations [29] to compute Rabin fingerprint and the hashes. For computing the priority of test cases, we use MATLAB [24] which provides basic library functions to calculate eigenvalues and eigenvectors. Efficient techniques are available for calculating the second eigenvalue and the corresponding eigenvector based on the sparsity of the matrix. In general, for the application of test case prioritization, the total time taken to construct a prioritization is $O(n^2)$, where n is the number of test cases under consideration. The greedy heuristic has identical complexity.

We explore four different types of hitherto unexplored path-sensitive prioritization mechanisms that are based on program paths to show the efficacy of PHALANX. We specifically consider four such prioritization metrics – differentiation based on (a) function sequences (FSQ), (b) branch operation sequences (BSQ), (c) sequence of memory statements without values of reads/writes (MSSQ) and (d) sequence of memory statements with values of reads/writes (MVSQ).

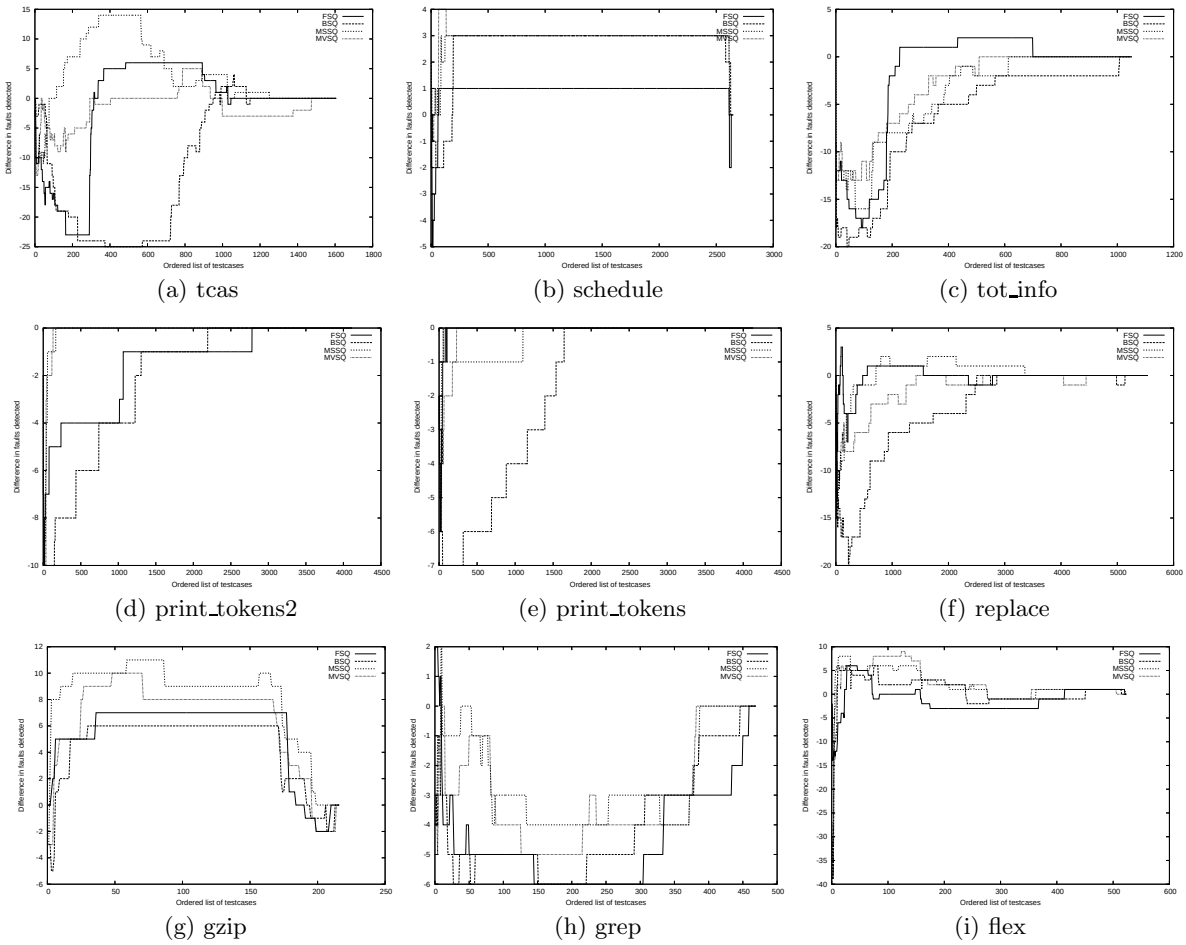
4.2 Subject Infrastructure Repository (SIR)

We perform our experiments on a Linux 2.6.11.10 (Gentoo release 3.3.4-r1) system running on a Intel(R) Pentium(R) 4 CPU 3.00GHz with 1GB memory. The version of the PIN [23] tool used was a special release 1819 (2005-04-15) for Gentoo Linux. The sources were compiled using GCC version 3.3.4.

For our experiments, we use the benchmarks (subjects) from the Subject Infrastructure Repository (SIR) [7]. The repository provides C programs for use in experimentation with testing and analysis techniques. The C programs vary across a range of parameters: the number of lines of code, the number of versions, the number of faults and the size of the test pool used for testing. Table 3 presents the subjects used. The number of lines of code varies from 100 to 11K. The test pool size varies from 217 to 5542.

4.3 Experimental Results

We execute the base version of each program with all the test cases present in the test-suite and collect traces. Subsequently, we apply the techniques specified in earlier sections of the paper to obtain a test case graph. From the test case graph, we obtain two possible orderings of test cases for each of the metrics used. The first ordering is based on the greedy approach and the second ordering is based on Fiedler ordering. Based on the fault matrix available with each subject, we obtain the list of faults and the list of test



(a) tcas (b) schedule (c) tot_info

(d) print_tokens2 (e) print_tokens (f) replace

(g) gzip (h) grep (i) flex

Figure 1: Normalized effectiveness of prioritization under Fiedler ordering compared to a greedy heuristic.

cases that are used in detecting each fault.

We evaluate the orderings using two different methods: (i) the *APFD* metric, which measures the weighted average of the percentage of faults detected over the life of the test suite, used in [8], and (ii) the realized difference between the number of faults detected by PHALANX and greedy approaches after executing a specific number of test cases.

More specifically, let T be a test suite containing n test cases, F be a set of m faults, TF_i be the first test case in ordering T' of T that reveals fault i. The APFD is calculated as follows:

$$APFD = 1 - \frac{TF_1 + ... + TF_m}{nm} + \frac{1}{2n}$$

For a more elaborate description of this metric, we refer the reader to [8]. A larger value of APFD denotes a better ordering of test cases because the first few test cases reveal a significant number of faults. Table 3 presents the APFD values for the subject programs for the four metrics described in Section 3. A majority of the values are greater than 80%, signifying the detection of faults early in the regression testing process.

On average, the performances of Fiedler and greedy ordering approach are comparable. However, when worst case behavior is compared, Fiedler ordering performs significantly better. For example, the APFD values for `gzip` under different metrics is significantly better with Fiedler ordering as compared to the greedy approach. The improved worst-case behavior can be attributed to the optimization function used in Fiedler ordering . For the greedy approach, the disadvantage of using local maximum as a criteria for ordering becomes obvious in benchmarks such as *gzip* and *schedule*. We also observe that MVSQ seems to be a very useful metric as it not only specifies the values written into, or read from memory, but also defines the paths taken by a program, does not incur significantly greater cost and displays better performance compared to the other three metrics.

While APFD is a useful metric for measuring the efficacy of an ordering, we observe from our experiments that in some scenarios, a few faults are detected early and the number of faults detected subsequently drops. Interestingly, this may result in a comparable APFD value of another ordering in which cumulatively more faults are detected, albeit after executing a few more test cases than the former. In Figure 1, we present the behavior of various subjects and the corresponding test suites by calculating the difference in the number of faults detected by Fiedler and greedy ordering approaches. Each graph represents the normalized number

	Fiedler Ordering				Greedy ordering			
	FSQ	BSQ	MSSQ	MVSQ	FSQ	BSQ	MSSQ	MVSQ
tcas	81.65	56.26	88.22	81.33	84.97	82.49	77.95	85.24
schedule	76.78	96.30	98.17	98.57	66.96	66.86	66.43	66.26
tot_info	84.38	72.21	78.37	82.05	92.84	99.08	95.33	94.83
print_tokens2	84.49	79.09	98.72	98.65	99.95	99.92	99.94	99.93
print_tokens	98.74	73.62	95.51	97.81	99.52	99.70	99.65	99.71
replace	96.04	82.17	96.02	90.56	96.52	94.31	95.59	95.02
gzip	77.77	71.57	91.87	81.03	43.09	42.90	38.36	39.40
grep	68.19	71.41	73.36	72.46	95.85	96.06	88.58	91.18
flex	92.30	93.58	93.53	94.54	94.01	92.35	89.75	90.44

Table 3: Comparison of APFD (given in percent) obtained with Fiedler and Greedy ordering approaches

of faults found using Fiedler ordering with respect to the greedy approach for the four prioritization metrics. For example, whenever the value along the *y-axis* is greater than zero, it indicates more faults detected by Fiedler ordering . For most of the subjects, a majority of the faults are detected after executing the first 20% of the test cases present in the test pool, irrespective of the ordering scheme used. Furthermore, we also find it intuitive that for a majority of the subjects (except `grep`), the difference either decreases or increases quite rapidly. This behavior is easily observable in Fig 1(a), where within execution of a small number of test cases from some point in the ordering, a large number of faults are detected as shown by the spikes. We hypothesize that this behavior is an artifact of placing dissimilar test cases in close proximity.

Figures 1(g) and 1(h) present a contrasting study of Fiedler and greedy ordering schemes. In Figure 1(g), a large number of faults are detected quite early in the regression testing process. For this subject, greedy ordering finds a local maxima and some of the faults are not detected until many more test cases are executed. In contrast, Figure 1(h) shows the improved performance of greedy ordering. It becomes clear from the figure that a few faults (detected by exactly one test case in the pool) are detected early using the greedy ordering. The list of faults that are detected later reflect the small APFD values using Fiedler ordering.

5. RELATED WORK

Rothermel *et al.* [32] present a number of techniques that use test execution information to prioritize test cases for regression testing. These techniques are classified broadly into three categories: (i) order test cases based on total coverage of code components, (ii) order test cases based on coverage of code components previously uncovered, and (iii) order test cases based on estimated ability to reveal faults. The elaborate experiments that accompany this work show that each of the prioritization techniques improved the rate of fault detection.

In subsequent work, Elbaum *et al.* [8] address three important questions: (i) are prioritization techniques more effective when made specific to modified versions?; (ii) does granularity (e.g., statement *vs* function level) of coverage matter?; and (iii) do inclusion of measures of likelihood of faults provide any improvements? New techniques are presented and detailed experiments suggest that version-specific prioritization improves test case ordering for the specified version; granularity and likelihood of faults do not significantly improve prioritization ordering.

Srivastava and Thiagarajan [33] present *Echelon*, a test-prioritization system that runs under a Windows environment. Test cases are prioritized based on the changes made to a program. Echelon uses a binary matching system that can accurately compute the differences at the basic block level between two versions of the program in binary form. Test cases are ordered to maximize coverage of affected regions using a fast, simple and intuitive heuristic. Echelon has been found to be quite effective on large binaries.

The framework presented in this paper is complementary to these techniques. Prioritization based on the above metrics can be easily coded into our framework. Furthermore, novel metrics based on program paths have been presented to show the effectiveness of our framework.

Harrold *et al.* [13] present a technique for selecting a representative set of test cases from a test-suite, which provides similar coverage. In other words, the technique is used to eliminate redundant and obsolete test cases. In [9], Rothermel *et al.* show through experiments that a potential drawback of redundant test elimination techniques is that in minimizing a test suite, the ability of that test suite to reveal faults in the software may be reduced. Jeffrey and Gupta [19] present another technique to minimize this drawback. Apart from the commonly shared goal of improving the process of regression testing, the problem addressed in this paper is orthogonal to the redundant test-case elimination problem.

In [6], Dickinson *et al.* find failures by cluster analysis of execution profiles. For observation-based testing, the hypothesis is that executions that exhibit unusual behaviors are good candidates for testing. Cluster analysis, a multivariate analysis method for finding groups in a population of objects, is performed; subsequently these clusters are filtered by selecting one or more executions from each cluster. This approach is shown to perform better than simple random sampling techniques. The techniques used in [6] and our work are based on the hypothesis that testing a program with dissimilar test inputs is useful. Specifically, we use Fiedler ordering, a declustering technique useful for flow-sensitive metrics to place dissimilar test-cases closeby (in a global sense) to quickly detect failures.

Bryce and Colbourn [2] present a case for test case prioritization for interaction testing, a mechanism for testing systems deployed on a variety of hardware and software configurations. They adapt a greedy method to generate a set of tests to identify all pairwise interactions when the ordered set of test cases is run to completion; otherwise the more important test cases are run if the testing is terminated without

completion. Bryce *et al.* perform a detailed study of greedy algorithms for the construction of software interaction test suites in [3]. We consider regression testing in our paper and present a greedy method to order test cases for flow sensitive metrics. The methods presented in [2] can be potentially used for regression testing. We plan to explore the application of pairwise interactions among program objects (instead of software configurations) to prioritize test cases for regression testing.

There has been substantial work in the area of impact analysis [1, 20, 26, 31, 30]. In many of these approaches, functions that follow a modified function in some execution path are added to the impact set. One of the reasons for using dynamic impact analysis is to reduce the parts of program that need to be retested while performing regression testing. For example, in [31], Ren *et al.* detect a set of changes responsible for a modified test's behavior and the set of tests that are affected by a modification are identified. The information obtained through impact analysis can help in designing new methods for prioritization. As long as these new methods can be specified in terms of a relation between test cases, a test case dissimilarity graph can be built and subsequently our approach for ordering test cases can be applied.

Many interesting techniques have been devised for bug detection in software systems [11, 5, 12, 22]. For example, in [12], Godefroid *et al.* present a technique to automatically generate test cases so that the coverage of the program is increased. Insofar as these efforts aim to generate useful test cases, they serve an important, albeit complementary role to our work.

6. REFERENCES

[1] T. Apiwattanapong, A. Orso, and M. Harrold. Efficient and precise dynamic impact analysis using execute-after sequences. In *ICSE '05*, pages 432–441, 2005.

[2] R. Bryce and C. Colbourn. Test prioritization for pairwise interaction coverage. In *A-MOST '05*, pages 1–7, 2005.

[3] R. Bryce, C. Colbourn, and M. Cohen. A framework of greedy methods for constructing interaction test suites. In *ICSE '05*, pages 146–155, 2005.

[4] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to algorithms*. MIT Press and McGraw-Hill Book Company, 2nd edition, 2001.

[5] C. Csallner and Y. Smaragdakis. Dsd-crasher: a hybrid analysis tool for bug finding. In *ISSTA '06*, pages 245–254, Portland, Maine, USA, 2006.

[6] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *ICSE '01*, pages 339–348, Washington, DC, USA, 2001.

[7] H. Do, S.G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

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

[9] J. Ostrin G. Rothermel, M.J. Harrold and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *ICSM*, 1998.

[10] A. George and A. Pothen. An analysis of spectral envelope reduction via quadratic assignment problems. *Siam J. Matrix Anal. Appl.*, 18(3):706–732, July 1997.

[11] P. Godefroid. Compositional dynamic test generation. In *POPL '07: Proceedings of the 34th Annual ACM Symposium on Principles of Programming Languages*, Jan 2007.

[12] P. Godefroid, N. Klarslund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pages 213–223, Chicago, Il, 2005.

[13] M.J. Harrold, R. Gupta, and M.L. Soffa. A methodology for controlling the size of a test suite. *ACM Trans. Softw. Eng. Methodol.*, 2(3):270–285, 1993.

[14] M.J. Harrold, D. Rosenblum, G. Rothermel, and E. Weyuker. Empirical studies of a prediction model for regression test selection. *IEEE Trans. Softw. Eng.*, 27(3):248–263, 2001.

[15] Bruce Hendrickson and Robert Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Sci. Comput.*, 16(2):452–469, 1995.

[16] D. Hirschberg. Algorithms for the longest common subsequence problem. *Journal of ACM, 24(4)*, pages 664–675, 1977.

[17] S. B. Horton. *The Optimal Linear Arrangement Problem: Algorithms and Approximation*. PhD thesis, Georgia Institute of Technology, 1997.

[18] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *ICSE '94*, pages 191–200, 1994.

[19] D. Jeffrey and N. Gupta. Test suite reduction with selective redundancy. In *ICSM*, pages 549–558, 2005.

[20] J. Law and G. Rothermel. Whole program path-based dynamic impact analysis. In *ICSE '03*, pages 308–318, 2003.

[21] B. Liblit, M. Naik, A. Zheng, A. Aiken, and M. Jordan. Scalable statistical bug isolation. In *PLDI*, pages 15–26, Chicago, Illinois, 2005.

[22] B. Livshits and T. Zimmermann. Dynamine: a framework for finding common bugs by mining software revision histories. In *Proceedings of the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC-FSE)*, Sep, 2005.

[23] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[24] http://www.matlab.com.

[25] B. Mohar. Laplace eigenvalues of graphs – a survey. *Disc. Math.*, 109:171–183, 1992.

[26] A. Orso, T. Apiwattanapong, and M. Harrold. Leveraging field data for impact analysis and regression testing. In *ESEC-FSE*, pages 128–137, 2003.

[27] A. Orso, N. Shi, and M.J. Harrold. Scaling regression testing to large software systems. In *FSE*, pages 241–252, Newport Beach, CA, USA, November 2004.

[28] Michael Rabin. Fingerprinting by Random Polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[29] http://www.cs.cmu.edu/ hakim/software/.

[30] M.K. Ramanathan, A. Grama, and S. Jagannathan. Sieve: A tool for automatically detecting variations across program versions. In *Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 241–252, Tokyo, Japan, 2006.

[31] X. Ren, F. Shah, F. Tip, B. Ryder, and O. Chesley. Chianti: A tool for change impact analysis of Java programs. In *OOPSLA '04*, pages 432–448, Vancouver, BC, Canada, 2004.

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

[33] A. Srivastava and J. Thiagarajan. Effectively prioritizing tests in development environment. In *ISSTA '02*, pages 97–106, 2002.

[34] W.E. Wong, J.R. Horgan, S. London, and A.P. Mathur. Effect of test set minimization on fault detection effectiveness. In *ICSE '95*, pages 41–50, 1995.