2018 IEEE International Symposium on Software Reliability Engineering Workshops

Graphite: A Greedy Graph-Based Technique for Regression Test Case Prioritization

Maral Azizi
Departemant of Computer Sceince and Engineering
University of North Texas
maralazizi@my.unt.edu

Hyunsook Do
Departemant of Computer Sceince and Engineering
University of North Texas
hyunsook.do@unt.edu

***Abstract*—To date, various test prioritization techniques have been developed, but the majority of these techniques consider a single objective that could limit the applicability of prioritization techniques by ignoring practical constraints imposed on regression testing. Multi-objective prioritization techniques try to reorder test cases so that they can optimize multiple goals that testers want to achieve. In this paper, we introduced a novel graph-based framework that maps the prioritization task to a graph traversal algorithm. To evaluate our approach, we performed an empirical study using 20 versions of four open source applications. Our results indicate that the use of the graph-based technique can improve the effectiveness and efficiency of test case prioritization technique.**

I. Introduction

Test case prioritization is a technique that improves the effectiveness of regression testing by exercising more important test cases earlier [33]. Increasing the fault detection rate has been the common objective of the majority of test case prioritization techniques [39], but, in practice, depending on an organization's circumstances, testers might need to achieve multiple goals such as finding more critical faults [29] or optimizing resource usage while running test cases [36]. Considering only a single objective could limit the applicability of prioritization techniques because such an approach ignores the practical constraints imposed on regression testing. For example, testers often face time constraints and resource/budget limitations during testing; thus, they need techniques that can offer various choices that address such limitations.

Recently, some researchers have started working on test prioritization techniques that achieve multiple objectives [18], [12], [20], [21], [38], [28]. For instance, Wang et al. in [36] proposed a resource-aware multi-objective optimization solution with a fitness function defined based on four cost-effectiveness measures. Their objectives were to find more faults and to maximize resource usage. The fundamental notion of multi-objective formulations is achieving multiple goals by optimizing different metrics. Finding a solution for such a problem is a complicated task that requires multiple evaluations and analyses with various factors. Evolutionary algorithms such as genetic algorithms have been successfully applied to this type of problem. The majority of multi-objective techniques have been implemented using evolutionary algorithms [18], [36], [24] because of their capability of solving difficult problems by heuristically searching complex spaces.

However, one problem with applying evolutionary algorithm is the time it takes to converge. Obtaining desired results using genetic algorithms usually requires a large population and several generations. With such a substantial simulation, obtaining a solution might take several days. For instance, using a genetic algorithm in the context of multi-objective regression testing for *mySQL* took up to nine days [18]. One way to address this drawback is to use a visual model, which can significantly decrease the analysis effort. Graphs make it easy to describe complex structures to model objects and properties in a direct and intuitive way. In particular, they provide a simple and powerful solution to a variety of problems that are typical in software engineering [11]. However, the applications of graph transformation have not been investigated by researchers in testing areas as thoroughly as other software engineering domains.

In this paper, we propose a novel Graph-based framework (*Graphite*) that accommodates test case prioritization techniques that attempt to achieve two goals simultaneously. The proposed approach uses three sources of information: code coverage, test dissimilarity, and test execution time. Code coverage is the most widely used metric for implementing test case prioritization techniques. The underlying hypothesis of this technique is that tests cases with higher coverage tend to have a higher chance of finding faults [13], [14], [17], [30]. We also used test dissimilarity, because empirical evidence indicates that diverse tests can produce a high fault detection rate [10], [22], [23], [34]. Further, test execution time was considered because one of our goals was to reduce the execution time of the reordered tests; thus, when we implement our prioritization techniques, we need to incorporate this information. Using these three inputs, our graph framework first maps the test cases into graph nodes, and links them based on their similarities, and then the traversal algorithm finds the most important test cases to be executed first.

Our empirical results indicate that the proposed approach can improve both the effectiveness and efficiency of test prioritization techniques. The results also show that the use of *Graphite* can reduce the algorithm execution time while being more effective than other prioritization algorithms in terms of the fault detection rate. The main contributions of our research

978-1-5386-9443-5/18/$31.00 ©2018 IEEE
DOI 10.1109/ISSREW.2018.00014

are as follows:

- Our research proposes a novel graph-based framework for test case prioritization that represents the test cases in a graph model and links them to incorporate three aspects of software metrics: code coverage, dissimilarities, and test execution time.
- Our proposed approach provides an efficient search algorithm to find more important test cases through the generated graph to achieve two goals of regression testing.
- Our study also empirically evaluates the proposed approach by comparing it with three test case prioritization techniques, which are commonly used for regression testing.

The rest of the paper is structured as follows. Section II explains the graph-based approach for multi-objective test case prioritization. Section III outlines the research questions and the details of the experimental design. Section IV presents the results of our study, and Section V discusses the threats to validity. Section VI presents related work, and Section VII discusses conclusions and future work.

II. Graph-Based Approach (Graphite)

The goal of Graphite is to improve the effectiveness of test prioritization and reduce the testing cost at the same time. More formally, for a given a test suite T and time budget t_{max}, the problem is to find the test order $\{T_1, T_2, ..., T_n\}$ that optimizes the average fault detection rate by maximizing the code coverage and diversity and does not exceed t_{max}.

To solve this problem, we propose a graph-based framework (*Graphite*) that utilizes three sources of information (code coverage, test diversity, and test execution time). *Graphite* maps the test case prioritization problem to a graph search problem, in which nodes of the graph are test cases and the edges between nodes are relationships between test cases. In the following subsections, we describe Graphite in detail including graph generation and traversal algorithms with a walk-through example.

A. Graph Generation

To build our graph, first we define the graph properties. Two major elements of a graph are nodes and edges. In a graph database, nodes and edges can have extra features such as labels, type, direction, and properties. In order to map a test case prioritization task to a graph, we need to determine which feature of the test case has to be extracted and how we can map this feature to a graph model. In this subsection, we explain the extracted feature and the mapping function.

1) Node Values: In this approach, we instantiate two metrics formulations with code coverage as a measure of test adequacy and execution time as a measure of cost. Thus, code coverage becomes one of the two metrics, and it should be maximized for a given cost. Time is the other metric, and it should be minimized for a given code coverage. Therefore, we define the graph nodes with the following properties: (1) node label: test ID; and (2) node value: code coverage divided by test execution time. We then normalized the node value, which ranges from 0 to 1. The node value identifies the potential benefit of the test case in terms of code coverage per time unit.

2) Edge Values: After creating graph nodes, we need to build edges among them. In a graph, edges determine the relationship between nodes. Therefore, to create links between nodes, we need to define a function to measure similarity between test cases. For example, assume that we have two test cases, 1 and 2. If test 1 exercises method A and B and test 2 exercises method B and D, and method are the elements in the test script, then we can conclude that method B is a common entity between the two tests 1 and 2, which can then link these two test cases. In our approach, we measured similarity among test cases using Jaccard Index, which is one of the widely used similarity functions [22], [23]. In this formula, similarity between two sample sets A and B can be measured by the size of the intersection divided by the size of the union of the sample sets ($Jaccard\ Index(A, B) = \frac{|A \cap B|}{|A \cup B|}$).

Algorithm 1 shows how to generate the graph using the above mentioned metrics. Initially, the algorithm receives a set of test cases as an input, then it assigns a value to each node by $(Code\ Coverage/Execution\ time)$ (line 10), and then it goes through the entire set of test cases and calculates the distance from the current node to the other nodes, inserting corresponding distance values to the data base (lines 12–16). After we assign a distance and node value to the first node, this node will add to the output list $WeightedGraph$ (line 15). Algorithm 1 reiterates the above mentioned steps for every remaining test case in each test suite. The output of the algorithm would be a weighted, acyclic, undirected graph in which both nodes and edges are labeled by values.

B. Graph Traversing Algorithm

After building the graph, we need to determine the goal of the graph search. Here, our goal is to traverse the graph of test cases to find more important test cases that have

Algorithm 1 Graph Generation

```
1:  Inputs: TestSet  The entire set of test cases.
2:  Outputs: WeightedGraph  A weighted graph of the entire set of test cases with node and edge value.
3:  Declare: node  A node represents test cases.
4:  Declare: edge  An edge represents relation between two nodes.
5:  procedure GraphGeneration(TestSet)
6:      cn ← ∅;   // current node
7:      for each test t ∈ TestSet[] do
8:          cn ← TestSet[t];
9:          node_id ← cn.Label();
10:         node_value ← coverage/executionTime(cn);
11:         CalcDist(cn, TestSet[] − cn);
12:         if cn.hasLink(TestSet[]) then
13:             cn.edge.add(all linked node);
14:             cn.edge_value ← distance of all linked nodes;
15:             WeightedGraph.Add(cn);
16:         end if
17:     end for
18:     return WeightedGraph;
19: end procedure
```

greater potential to find faults earlier with minimum execution time. To support this goal, our algorithm should perform the following tasks: (1) find a path that traverses the nodes with higher gain value earlier, and (2) traverse all nodes in the graph. (Hence, we are applying this algorithm for test case prioritization and need to run all test cases; in other techniques such as test case selection we can eliminate some nodes based on the requirements.)

Algorithm 2 shows the pseudo-code of the graph traversal algorithm. The algorithm begins by analyzing all available nodes in the entire set, and then it selects a node that has the highest value (line 13). Then, it visits the entire remaining nodes that have a link with the current selected node and selects the next node by considering its gain value. The gain value of each node is calculated by adding the node value and the edge value (line 16). The next node to be selected is the node with the highest gain value from the current node (lines 15–22). After finding the second candidate node, the algorithm removes the previous node from the node set to avoid duplicate calculations (line 14). The algorithm is recursively applied to the remaining node until all nodes are selected.

Algorithm 2 Graph Traverser

```
1:  input: WeightedGraph[]   A weighted graph from algorithm 1.
2:  Output: OrderList[]   A list of the reordered index of test cases.
3:  procedure GraphTraverser(WeightedGraph, cn)
4:      Q ← the set of all nodes in WeightedGraph
5:      gain ← ∅;
6:      maxGain ← ∅;
7:      for each node v ∈ WeightedGraph do
8:          distance[v] ← ∞;
9:          previous[v] ← ∅;
10:         distance[cn] ← ∅;
11:     end for
12:     while Q != empty() do
13:         u ← node in Q with largest node value[u];
14:         remove u from Q;
15:         for each neibore v of u do
16:             gain ← dist(u, v) + v.value;
17:             if gain > maxGain[v] then
18:                 maxGain[v] ← gain;
19:                 previous[v] ← u;
20:                 OrderList.Add(v);
21:             end if
22:         end for
23:     end while
24:     return OrderList[];
25: end procedure
```

C. An Example of Graph Generation and Traversal Procedure

Figure 1 shows the graph that was built using Algorithm 1 for this example. The five nodes of the graph are representing the five test cases that are labeled by the test case identities. Each node value has been calculated from its code coverage divided by its execution time. For instance, T_2 covers two statement and its execution time is 3 minutes therefore, the value of this node is equal to 0.66 (2/3). We assigned the edge values by calculating the Jaccard Distance for each node. For example, the Jaccard Index between T_1 and T_2 is equal to 0.2 (1/5). Then, the Jaccard Distance of T_1 and T_2 is equal to the $1 - Jaccard\ Index(T_1, T_2)$ that returns 0.8 (1 - 0.2).

As we mentioned earlier, Algorithm 2 first selects the node with maximum value. In this example, node T_3 has highest value (0.75), therefore, the traversal procedure will start from this node. Then, the traversal algorithm calculates the gain value of each node from the current node. The gain value of node T_5 has the highest value compared to all other nodes (0.5 + 1 = 1.5). Therefore, the second candidate is node T_5 (Figure 1.B). At this time T_3 and all its connected edges will be eliminated from the database. In this step, the $CurrentNode$ is T_5 and this procedure will be repeated until the traverser algorithm crosses all remaining nodes. Finally, the traversed path using Algorithm 2 would be $\{T_3, T_5, T_2, T_4, T_1\}$.

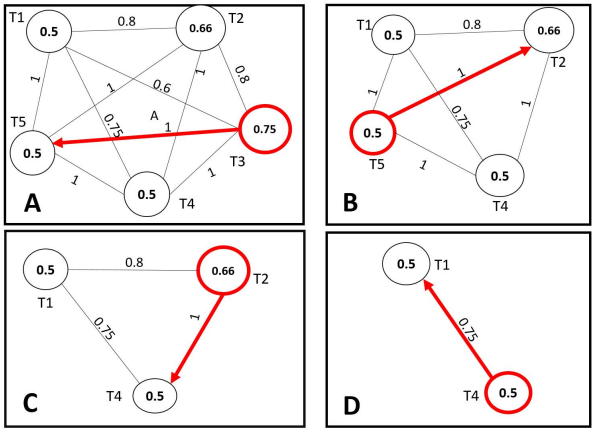


Fig. 1: An Example of Graph Traversal Algorithm

III. Empirical Study

In this paper, we investigate whether the use of Graphite can improve the effectiveness of test case prioritization techniques. To assess our proposed technique, we performed an empirical study. The following subsections present our objects of analysis, data collection, study setup, and threats to validity. In particular, we address the following research question:

RQ: Can Graphite improve the effectiveness of test case prioritization?

A. Objects of Analysis

In this study, we used four open source programs. Table I lists the applications and their associated data: "LOC" (the number of lines of code), "Faults" (the number of faults), "Version" (the number of versions), and Test Cases (the number of test cases). *nopCommerce* is an open source e-commerce shopping cart web application built on the ASP.Net platform [1]. *Umbraco* is a large-scale open-source content management system, which has been written in $C\#$ language [2]. Two other applications were taken from the Software-Artifact Infrastructure Repository (SIR) [3]: *jmeter* and *jtopas*, which are written in Java.

The faults used in *nopCommerce* and *Umbraco* are real defects that have been reported by users [1], and the faults used

[1]The reported faults for *nopCommerce* are available in the *GitHub* repository [4], and the bug history for *Umbraco* is accessible through their website [5].

TABLE I: Subject Applications Properties

Object	LOC	# Test Cases	# Faults	# Versions
nopCommerce	226,354	543	40	4
Umbraco-CMS	128,037	3,381	30	4
jmeter	43,400	78	9	7
jtopas	5,400	128	5	5

in *jmeter* and *jtopas* are seeded faults that came with the programs from the SIR repository. All applications have multiple consecutive versions and test cases. In total, 20 versions of four programs were used to create the data required for evaluating the proposed technique with other control techniques.

B. Variables and Measures

1) Independent Variables: To investigate our research questions, we manipulated two independent variables: prioritization technique and time constraint.

Variable 1: prioritization technique. We considered eight different test case prioritization techniques, which we classified into two groups: control and heuristic. Table II summarizes these groups and techniques. The second column shows prioritization techniques for each group, and the third column is a short description of each prioritization technique. For our heuristic techniques, we used the approach explained in Section II, so, here, we only explain the control techniques.

1) Total Statement (T): This technique prioritizes test cases based on the total number of statements they cover. If multiple tests cover the same number of statements, they are ordered randomly.
2) Additional Statement (A): This technique prioritizes test cases based on the number of additional statements they cover. If multiple test cases cover the same number of statements not yet covered, they are ordered randomly.
3) Test Dissimilarity (D): This technique prioritizes test cases using the dissimilarity among test cases. To measure the dissimilarity, we used the Jaccard Index.

Variable 2: time constraints. Due to deadline and budgetary constraints, it is common for software companies to cut back on testing activities. To consider this practical situation and to assess the effects of time constraints on test case prioritization techniques, we consider time constraints as our second independent variable. We utilize three time constraint levels: 25%, 50%, and 75%. Time constraints 25%, 50%, and 75% represent situations in which time constraints reduce the amount of testing that can be done by 25, 50, and 75%, respectively.

Because the test cases used in this study have different execution times, to implement time constraints, we used the total execution time of the test suite. For instance, in the case of 25%, for each version of the program and for each prioritized test suite we paused the execution of the test cases as soon as 25 percent of total execution time for that particular application was reached.

2) Dependent Variable and Measures: Our dependent variables for research question are test execution time, percentage of test suite run to reveal 100% of the faults, and average percentage of fault detection (APFD) that measures the average percentage of faults detected during the test suite execution. APFD values range from 0 to 100; higher values indicate faster fault detection rates. Given T as a test suite with n test cases and m number of faults, F as a collection of detected faults by T, and TF_i as the first test case that catches the fault i, we calculate APFD [32] as follows:

$$APFD = 1 - \frac{TF_1 + TF_2 + ... + TF_m}{nm} + \frac{1}{2n}$$

C. Data Collection and Experimental Setup

To perform our study, we needed to collect code coverage information, test execution time, and dissimilarity. To collect the code coverage data (statement coverage), we used the code coverage analysis tool provided by Microsoft Visual Studio for *nopCommerce* and *Umbraco*, and the unit tests code coverage plug-in provided by NetBeans IDE [6] for *jmeter* and *jtopas*. We collected test execution time using a PC with CPU Core i7, memory 16 GB, and operating system Windows 10. Also, we applied one dissimilarity function that we explained in Section II-A2 to calculate the dissimilarity score of test cases.

IV. Data and Analysis

In this section, we present the results considering our research question, which investigates whether *Graphite* can be effective in improving test case prioritization by minimizing test execution time and achieving a higher fault detection rate at the same time. Figure 2 represent the average percentage of the test suite run and the average test execution time needed to achieve 100 percent of fault detection for each application. The horizontal axis of each figure shows the prioritization techniques, the left vertical axis shows the percentage of the test suite run, and the right vertical axis shows the total execution time needed to reveal 100 percent of the faults for each corresponding technique. Figure 2 indicates that, overall, our heuristic technique outperformed the three control techniques except in few cases.

However, the results showed some variation across applications. In the case of *nopCommerce*, Graphite outperformed all other techniques. Between all examined control techniques, overall, the multi-objective technique showed better results, but the diversity-based approach (D) was slightly more effective than the coverage based approaches (T and A). In the case of *Umbraco*, the trend is similar to that of *nopCommerce*, but the difference between control and heuristic techniques was more noticeable. In particular, Graphite needed to run less than 30% of the test cases to catch all faults, while the total coverage technique (T) required more than 70% of the test cases. Further, Graphite took less than 1,400 seconds to detect all the faults, while the total coverage technique (T) took more than 3,500 seconds. Among the control techniques, the diversity based technique (D) that showed the best results,

TABLE II: Test Case Prioritization Techniques

Group	Technique	Description
Control	Total Statement (T)	Reorders test cases based on the total number of statements they cover.
	Additional Statement (A)	Reorders test cases based on the number of additional statements they cover.
	Dissimilarity (D)	Test case prioritization based on their Jaccard distance.
Heuristic	Graphite (G)	Prioritization using graph-based technique by applying Jaccard Index for weighting the edges.

requiring about 39% of tests and 2,000 seconds to catch all the faults.

The results from *jmeter* and *jtopas*, which are small applications and have a small number of tests with a short execution time, showed different trends from what we observed with *nopCommerce* and *Umbraco*. For both applications, except for the total coverage technique (T), the results in all other techniques did not show much difference. For example, in *jmeter*, the heuristic show slightly better results than the control techniques, but their differences are not noticeable. Graphite needed 65% of the tests and 63 seconds to detect all the faults, while the worst of the control techniques (T) needed about 87% of the tests and 89 seconds. As for *jtopas*, the heuristic was no better than the additional coverage technique (A). Similar to *jmeter*, the test execution time needed to catch all the faults was not substantially different among the techniques except for the total coverage technique (which ranged from 145 to 180 seconds).

To understand how our heuristic performs under time constraints, we calculated APFD values for techniques considering three different time constraint levels (25%, 50%, and 75%). Applying 25% time constraints means only 25% of tests would be executed. Table III shows the results of each application version for four different prioritization techniques. When we applied 25% of time constraints, Graphite yielded higher APFD values than control techniques for 14 out 20 cases. When we applied 50% of time constraints, Graphite outperformed 15 of 20 cases, and under 75% of time constraints, Graphite outperformed 18 out of 20 cases.

We also compared our approach with the best performed control technique for each application (e.g, additional-statement for *jtopas*). In the case of nopCommerce, Graphite's improvement rates over A were 46%, 29%, and 8% for 25%, 50%, and 75% of time constraints, respectively. For *Umbraco*, the improvement rates were slightly better than nopCommerce: 172%, 33%, and 29% for 25%, 50%, and 75% of time constraints, respectively. However, for for two small applications, (*jmeter* and *jtopas*), Graphite (G) did not perform well. For those small applications, APFD values of G were very close to the additional-statement technique (A), and for some cases, A even performed better than G (v1 and v2 in *jtopas*). One plausible reason for these results is that the characteristics of software artifacts for these two groups (small and large) are quite different from each other. Small applications have a small number of tests and faults, execution times of all tests for a given application are almost equivalent and relatively short, and the faults are hand seeded. However, large applications have a large number of tests and faults, the execution time of each test is different, and the faults are real faults that were reported by users.

Considering these factors, our heuristic might have less impact on these small applications because one of our prioritization criteria is test execution time, which is uniform for small applications. Also, the types of faults could have affected the outcome of our results, but we need further investigation to see whether this factor indeed affected our outcome. Based on the result of our experiment, we can conclude that Graphite works well for large scale applications with a large number of tests and faults, but it might not be suitable for small applications.

V. Threats to Validity

The main threat to validity in this study is the choice of distance function. Although we used the most commonly applied algorithm, there is no guarantee that this algorithm (Jaccard Index) is the best choice, and the results can be affected by what algorithm is selected. This limitation can be addressed through additional empirical studies with different algorithms.

VI. Related Work

Test case prioritization techniques reorder test cases to maximize some objective functions, such as detecting defects as early as possible. One of the most commonly used prioritization techniques is the greedy technique [37], [9], [15], [7], [39], [16], [33]. In this technique, the capability of a test case is measured by the amount of code it covers. The basic assumption is that test cases with higher code coverage have a better chance to reveal faults than test cases with lower code coverage. Although this approach is naïve and easy to implement, many empirical studies have shown that it can be effective [13], [14], [17], [30], [25].

More recently, several prioritization techniques have been developed using other types of information. One of these techniques is prioritizing test cases based on their dissimilarity. Empirical studies have shown that measuring test case dissimilarity could improve the effectiveness of regression testing noticeably [26], [19], [35], [8], [27], [36]. The diversity between test cases can be measured by utilizing a distance function. Common distance functions used in the regression testing area include the Hamming distance, Jaccard Index, and edit distance. For example, Hemmati et al. investigated possible similarity functions to support similarity-based test selection in the context of model-based testing, using genetic algorithms to an industrial software system [22].

Another study done by Ramanathan et al. [31] is closely related to our research. They proposed a graph-based algorithm for test case prioritization, in which each node of the graph represents a test case and the edges are weighted by the dissimilarity between the corresponding pairs of test cases.

TABLE III: Descriptive statistics of APFD results with three different time constraints. For each application version, the indicators are presented for the additional statement greedy algorithms based on, statement coverage (A), the total statement greedy algorithms based on, statement coverage (T), and the graph-based technique (G)

		Percent of Total Time					Percent of Total Time					Percent of Total Time					Percent of Total Time		
	Alg	25%	50%	75%		Alg	25%	50%	75%		Alg	25%	50%	75%		Alg	25%	50%	75%
nop v4.00	T	0.20	0.39	0.39	nop v3.90	T	0.14	0.56	0.63	nop v3.80	T	0.28	0.32	0.47	nop v3.70	T	0.11	0.48	0.55
	A	0.34	0.53	0.69		A	0.41	0.55	0.71		A	0.35	0.49	0.68		A	0.27	0.64	0.64
	D	0.42	0.61	0.61		D	0.21	0.69	0.69		D	0.47	0.54	0.65		D	0.33	0.59	0.67
	G	**0.46**	**0.70**	**0.70**		G	**0.43**	**0.67**	**0.79**		G	**0.55**	**0.73**	**0.73**		G	**0.51**	**0.74**	**0.74**
Um v7.7.4	T	0.07	0.35	0.35	Um v7.5.4	T	0.1	0.26	0.57	Um v7.4.3	T	0.13	0.25	0.43	Um v7.3.1	T	0.11	0.32	0.53
	A	0.26	0.56	0.56		A	0.29	0.59	0.64		A	0.32	0.66	0.66		A	0.24	0.57	0.60
	D	0.28	0.58	0.58		D	0.37	0.71	0.71		D	0.42	0.64	0.64		D	0.49	0.61	0.61
	G	**0.77**	**0.82**	**0.82**		G	**0.79**	**0.79**	**0.79**		G	**0.68**	**0.76**	**0.76**		G	**0.74**	**0.80**	**0.80**
jmeter v1	T	0.21	0.3	0.34	jmeter v2	T	0.06	0.13	0.19	jmeter v3	T	0.08	0.16	0.27	jmeter v4	T	0.1	0.15	0.19
	A	0.45	0.6	0.75		A	**0.58**	0.63	0.76		A	0.45	0.59	0.76		A	0.27	0.32	0.69
	D	0.45	0.66	0.81		D	0.52	0.57	0.64		D	0.48	0.62	0.77		D	0.34	0.7	0.82
	G	**0.49**	**0.74**	**0.89**		G	0.54	**0.66**	**0.85**		G	**0.58**	**0.72**	**0.86**		G	**0.38**	**0.78**	**0.90**
jmeter v5	T	0.11	0.16	0.28	jmeter v6	T	0.34	0.55	0.68	jmeter v7	T	0.09	0.13	0.21	jtopas v1	T	0.17	0.31	0.59
	A	**0.55**	0.67	0.71		A	0.39	0.53	0.68		A	**0.58**	0.7	0.84		A	**0.73**	**0.89**	**0.92**
	D	0.43	0.61	0.74		D	0.21	0.57	0.68		D	0.34	**0.75**	0.89		D	0.47	0.6	0.73
	G	0.49	**0.74**	**0.89**		G	**0.54**	**0.89**	**0.94**		G	**0.58**	0.72	**0.93**		G	0.60	0.83	0.9
jtopas v2	T	0.16	0.25	0.59	jtopas v3	T	0.24	0.39	0.67	jtopas v4	T	0.12	0.21	0.34	jtopas v5	T	0.12	0.28	0.61
	A	**0.78**	**0.87**	**0.95**		A	0.61	0.77	0.91		A	**0.76**	**0.89**	0.89		A	**0.7**	**0.88**	0.95
	D	0.61	0.74	0.83		D	0.65	0.89	0.89		D	0.55	0.68	0.75		D	0.5	0.75	0.83
	G	0.49	0.62	0.78		G	**0.84**	**0.90**	**1**		G	0.58	0.84	**0.98**		G	0.66	0.83	**0.96**

Spectral ordering has been applied on their implemented graph to find the optimized order of the test cases. This approach focuses on optimizing the test ordering by only measuring the test case dissimilarity, whereas our approach focuses on optimizing two objectives, which are improving the effectiveness and the efficiency of regression testing. In our work, we utilized three metrics: code coverage, test cases dissimilarity, and test cost, to build our graph, and we provide a traversal algorithm to search for the maximized values of code coverage and test dissimilarity, and minimum test cost.

VII. Conclusions and Future Work

In this research, we have introduced a novel graph-based prioritization approach (*Graphite*) to improve the effectiveness as well as to achieve two goals simultaneously. We evaluated our approach using four open-source applications with three widely used techniques, and our empirical results indicate that the proposed approach was able to improve the effectiveness and efficiency of prioritization. The performance of our approach was particularly outstanding when we had a limited time budget.

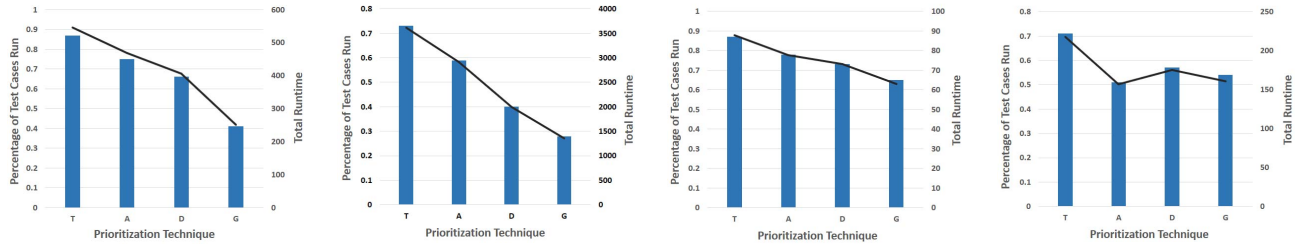
For future work, we wish to continue to expand on this research as more data becomes available across a wider range of applications and different metrics. We also want to investigate how this research applies to different areas of regression testing such as test selection and reduction.

Acknowledgment

This work was supported, in part, by NSF CAREER Award CCF-1564238 to University of North Texas.

References

[1] hhttp://www.nopcommerce.com/. [Accessed: Jan. 26, 2017].
[2] https://umbraco.com/. [Accessed: Oct. 06, 2017].
[3] http://sir.unl.edu//. [Accessed: Oct. 06, 2017].
[4] https://github.com/nopSolutions/nopCommerce/issues//. [Accessed: Oct. 06, 2017].
[5] http://issues.umbraco.org/issues//. [Accessed: Oct. 06, 2017].
[6] http://plugins.netbeans.org/plugin/5620/unit-tests-code-coverage-plugin//. [Accessed: Oct. 06, 2017].
[7] K. K. Aggrawal, Y. Singh, and A. Kaur. Code coverage based technique for prioritizing test cases for regression testing. In *ACM SIGSOFT Software Engineering Notes*. ACM, 2004.
[8] M. Al-Hajjaji, T. Thm, J. Meinicke, M. Lochau, and G. Saake. Similarity-based prioritization in software product-line testing. In *Proceedings of the International Software Product Line Conference* , 2014.



nopCommerce Umbraco jmeter jtopas

Fig. 2: Percentage of Test Suite Run vs Total Test Execution Time for 100% Fault Detection

[9] R. Carlson, H. Do, , and A. Denton. A clustering approach to improving test case prioritization: An industrial case study. In *ICSM '11 Proceedings of the 2011 27th IEEE International Conference on Software Maintenanc*, pages 382–391. IEEE-ACM, 2011.

[10] E. G. Cartaxo, P. D. L. Machado, and F. G. O. Neto. On the use of a similarity function for test case selection in the context of model-based testing. In *Proceedings of the International Conference of Software Testing, Verification and Reliability*. IEEE-ACM, 2009.

[11] A. Corradini, H. Ehrig, H. J. Kreowsky, and G. Rozenbreg. *Graph Transformation*. Springer, 2002 edition, 2002.

[12] J. T. de Souza, C. L. Maia, F. G. de Freitas, and D. P. Coutinho. The human competitiveness of search based software engineering. In *In Proceedings of 2nd International Symposium on Search based Software Engineering (SSBSE)*, page 143152. IEEE, 2010.

[13] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. An empirical study of the effect of time constraints on the cost-benefits of regression testing. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*. IEEE-ACM, 2008.

[14] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel. The effects of time constraints on test case prioritization: A series of controlled experiments. 36(5), 2010.

[15] S. Elbaum, A. G. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. Technical Report 00-60-03, Oregon State University, February 2000.

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

[17] S. Elhaum, A. G. Malishevsky, , and G. Rothermel. Test case prioritization: A family of empirical studies. 28(2):159–182, 2002.

[18] M. G. Epitropakis, Sh. Yoo, M. Harman, and E. K. Burke. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In *Proceedings of the International Symposium on Software Testing and Analysis(ISSTA)*, pages 234–245, 2015.

[19] C. Fang, Z. Chen, K. Wu, and Z. Zhao. Similarity-based test case prioritization using ordered sequences of program entities. *Journal of Software Quality.*, 22(2):65–95, 2014.

[20] Q. Gu, B. Tang, and D. Chen. Optimal regression testing based on selective coverage of test requirements. In *In International Symposium on Parallel and Distributed Processing with Applications (ISPA 10)*, page 419 426. IEEE, 2010.

[21] M. Harman. Making the case for morto: Multi objective regression test optimization. In *In Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, ICSTW*, page 111114. IEEE, 2011.

[22] H. Hemmati, A. Arcuri, and L. C. Briand. Empirical investigation of the effects of test suite properties on similarity based test case selection. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, 2011.

[23] H. Hemmati, A. Arcuri, and L. C. Briand. Achieving scalable model based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, (1):143, 2013.

[24] Md. M. Islam, A. Marchetto, A. Susi, and G. Scanniello. A multi-objective technique to prioritize test cases based on latent semantic indexing. In *Proceedings of the European Conference on Software Maintenance and Reengineering*, pages 234–245, 2012.

[25] M. Khatibsyarbini, M. Adham Isa, D. N.A.Jawawi, and R. Tumeng. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology.*, 93(10):74–93, 2018.

[26] Y. Ledru, A. Petrenko, and S. Boroday. Using string distances for test case prioritisation. In *Proceedings of the International Conference on Automated Software Engineering* , 2009.

[27] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran. Prioritizing test cases with string distances. *Journal of Automated Software Engineering.*, (1):6595, 2012.

[28] Z. Li, Y. Bian, R. Zhao, and J. Cheng. A fine-grained parallel multi-objective test case prioritization on gpu. In *In G. Ruhe and Y. Zhang, editors, Search Based Software Engineering, volume 8084 of Lecture Notes in Computer Science*, page 111125. Springer, 2013.

[29] A. Marchetto, Md. Mahfuzul Islam, W. Asghar, Angelo Susi, and Giuseppe Scanniello. A multi-objective technique to prioritize test cases. *IEEE Transaction on Software Engineering.*, 42(10), 2016.

[30] X. Qu, M.B. Cohen, and G.Rothermel. Configuration-aware regression testing: an empirical study of sampling and prioritization. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 75–85. IEEE-ACM, 2008.

[31] M. K. Ramanathan, M. Koyuturk, A. Grama, and S. Jagannathan. Phalanx: A graph-theoretic framework for test case prioritization. In *Proceedings of the ACM symposium on Applied computing (SAC)*, pages 16–20, 2008.

[32] G. Rothermel, R. Untch, C. Chu, , and M. J. Harrold. Test case prioritization: An empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 179–188. IEEE-ACM, 1999.

[33] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering*, 27(10):929–948, October 2001.

[34] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein. Static test case prioritization using topic. In *Empirical Software Engineering*, pages 1–31. IEEE-ACM, 2012.

[35] R. Wang, , S. Jiang, and D. Chen. Similarity-based regression test case prioritization. In *Proceedings of the Software Reliability Engineering (ISSRE)*, 2015.

[36] S. Wang, S. Ali, T. Yue, O. Bakkeli, and M. Liaaen. Enhancing test case prioritization in an industrial setting with resource awareness and multi-objective search. In *Proceedings of the International Conference of Software Engineering Companion*, 2016.

[37] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of the International Symposium on Software Reliability Engineering*, pages 230–238, November 1997.

[38] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *Proceedings of the International Conference on Software Testing and Analysis*, pages 140–150, July 2007.

[39] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.