Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Static Test Case Prioritization Using Topic Models

Stephen W. Thomas · Hadi Hemmati · Ahmed E. Hassan · Dorothea Blostein

Received: date / Accepted: date

Abstract Software development teams use test suites to test changes to their source code. In many situations, the test suites are so large that executing every test for every source code change is infeasible, due to time and resource constraints. Development teams need to prioritize their test suite so that as many distinct faults as possible are detected early in the execution of the test suite. We consider the problem of static black-box test case prioritization (TCP), where test suites are prioritized without the availability of the source code of the system under test (SUT). We propose a new static black-box TCP technique that represents test cases using a previously unused data source in the test suite: the *linguistic data* of the test cases, i.e., their identifier names, comments, and string literals. Our technique applies a text analysis algorithm called topic modeling to the linguistic data to approximate the functionality of each test case, allowing our technique to give high priority to test cases that test different functionalities of the SUT. We compare our proposed technique with existing static black-box TCP techniques in a case study of multiple real-world open source systems: several versions of Apache Ant and Apache Derby. We find that our static black-box TCP technique outperforms existing static black-box TCP techniques, and has comparable or better performance than two existing execution-based TCP techniques. Static black-box TCP methods are widely applicable because the only input they require is the source code of the test cases themselves. This contrasts with other TCP techniques which require access to the SUT runtime behavior, to the SUT specification models, or to the SUT source code.

Keywords Testing and debugging · Test case prioritization · Topic models

1 Introduction and Motivation

Software development teams typically create large test suites to test their source code (Ali et al, 2009). These test suites can grow so large that it is cost prohibitive

S. W. Thomas · H. Hemmati · A. E. Hassan · D. Blostein
School of Computing, Queen's University, Canada
E-mail: {sthomas, hemmati, ahmed, blostein}@cs.queensu.ca

to execute every test case for every new source code change (Rothermel et al, 2001). For example, at Google, developers make more than 20 source code changes per minute, resulting in 100 million test cases executed per day (Kumar, 2010). In these situations, developers need to *prioritize* the test suite so that test cases that are more likely to detect undetected faults are executed first. To address this challenge, researchers have proposed many automated test case prioritization (TCP) techniques (Elbaum et al, 2002; Rothermel et al, 2001). Most of these TCP techniques use the *execution* information of the system under test (SUT): the dynamic run-time behavior, such as statement coverage, of each test case (Chen et al, 2011; Simao et al, 2006). While execution information is a rich information source for the TCP technique, execution information may be unavailable for several reasons (Zhang et al, 2009):

– Collecting execution information can be cost prohibitive, both in terms of time and resources.
– For large systems, execution information will be quite large, making storage and maintenance costly.
– Execution information must be continuously updated as source code and test cases evolve.

To address situations in which the execution information is not available, researchers have proposed TCP techniques based on *specification models* of the tests (Hemmati et al, 2012; Korel et al, 2007). These models describe the expected behavior of the SUT and test suite. However, specification models are also sometimes not available, for similar reasons: the models may be cost prohibitive to manually generate, or the maintenance of the models as source code and test cases evolve may be cost prohibitive.

To address situations when *neither* the execution information nor specification models are available, researchers have recently developed *static* TCP techniques. In particular, Zhang et al. (2009) and Mei et al. (2011) propose techniques based on the static call graph of the test cases. Additionally, Ledru et al. (2011) treat each test case as a string of characters, and prioritize test cases by using a simple string edit distance to determine the similarity between test cases. In these techniques, the goal is to give high priority to test cases that are highly dissimilar (e.g., because they invoke different methods, or have high string distances), thereby maximizing test case diversity and casting a wide net for detecting unique faults (Hemmati et al, 2010b, 2011). While static TCP techniques do not have as much information to work with as those based on execution information or specification models, static techniques are less expensive and are lighter weight, making them applicable in many practical situations.

However, existing static TCP techniques make little or no use of an important data source embedded within test cases: their *linguistic data*, i.e., the identifier names, comments, and string literals that help to determine the functionality of the test cases (Kuhn et al, 2007). In this paper, we propose a new static TCP technique that uses the linguistic data of test cases to help differentiate their functionality. Our technique uses a text analysis algorithm, called topic modeling, to create *topics* from the linguistic data, and prioritizes test cases that contain different topics. The main advantage of our technique, compared to existing black-box static TCP techniques, is that topics abstractly represent test cases' functionality, which

is robust to trivial differences in the test's source code and can capture more information than call graphs alone.

We compare our proposed technique to two existing black-box static TCP techniques in a detailed case study on several real-world systems. We find that our topic-based technique increases the average fault detect rate over existing techniques by as much 31%, showing improved performance across all studied systems.

In summary, this paper makes the following contributions:

- We introduce a novel black-box static TCP technique, based on *topic models* of the *linguistic data* in test cases (Section 3).
- We compare our proposed TCP technique to two existing static TCP techniques in a detailed case study. We find that our proposed technique achieves the same or better performance than existing static techniques (Sections 4 and 5).

We make our datasets and results publicly available (Thomas, 2012a) and encourage researchers to replicate our studies on additional datasets and systems.

2 Test Case Prioritization

Test case prioritization (TCP) is the problem of ordering the test cases within the test suite of a system under test (SUT), with the goal of maximizing some criteria, such as the fault detection rate of the test cases (Wong et al, 1997). Should the execution of the test suite be interrupted or stopped for any reason, the more important test cases (with respect to the criteria) have been executed first. More formally, Rothermel et al. (2001) define the TCP problem as follows.

Definition 1 (Test case prioritization) Given: T, a test suite; PT, the set of permutations of T; and f, a performance function from PT to the real numbers. Find: $T' \in PT$ s.t. $(\forall T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$.

In this definition, PT is the set of all possible prioritizations of T and f is any function that determines the performance of a given prioritization. The definition of performance can vary, as developers will have different goals at different times (Rothermel et al, 2001). Developers may first wish to find as many faults as possible; they may later wish to achieve maximal code coverage. In these scenarios, the problem definition of TCP is the same, but the performance function f being optimized changes. Researchers have proposed and evaluated many techniques to solve the TCP problem, based on a range of data sources and prioritization algorithms; Yoo and Harman (2010) have conducted a thorough survey of such techniques. In general, each TCP technique tries to achieve optimal prioritization based on the following steps (Hemmati et al, 2010a).

1. Encode each test case by what it *covers*, i.e., what it does, based on the elements in the available dataset. For example, the elements may be the program statements of the SUT that the test case covers.
2. Prioritize test cases using a distance maximization algorithm, based on maximizing either coverage or diversity.

(a) *Maximize coverage.* Prioritize test cases so that the maximum number of elements are covered. The intuition behind this strategy is that by maximizing element coverage, the technique tests as much of the SUT as possible and increasing its chances of fault detection.
(b) *Maximize diversity.* First, determine the *similarity* (equivalently *distance*) between test cases, for some definition of similarity (e.g., intersection of statements covered). Then, prioritize test cases by maximizing their dissimilarity. The intuition for this strategy is that pairs of test cases that are similar will likely detect the same faults, and therefore only one needs to be executed.

Researchers have proposed many distance maximization algorithms, including greedy algorithms, clustering, and genetic algorithms (Hemmati et al, 2010a).

In this paper, we categorize TCP techniques primarily based on step 1 above, i.e., how the technique encodes the test cases into elements, resulting in the following five categories.

Definition 2 (White-box execution-based prioritization) Prioritization based on the dynamic execution behavior of the test cases, with access to the source code of the SUT. Test cases are encoded by which source code elements they execute. Requires the source code of the SUT and test cases to be instrumented, compiled, and executed.

Definition 3 (Black-box execution-based prioritization) Prioritization based on the dynamic execution behavior of the test cases, using the execution logs of the source code. Test cases are encoded using the contents of their execution logs. Does not require the actual source code of the SUT, but instead assumes that execution log files are available for each test case.

Definition 4 (Grey-box Model-based prioritization) Prioritization based on specification models (e.g., state diagrams) of the SUT and test cases. Test cases are encoded by which paths they take on the specification model. Does not require execution information. Requires specification models to be created and maintained.

Definition 5 (White-box static prioritization) Prioritization based on the source code of the SUT and test cases. Test cases are encoded based on some aspect of the static snapshot of the source code of the SUT and test cases. Does not require execution information or specification models. Requires access to the source code of the SUT and test cases.

Definition 6 (Black-box static prioritization) Prioritization based on the source code of the test cases. Test cases are encoded based on some aspect of the source code of the test cases. Does not require execution information, specification models, or source code of the SUT. Only requires source code of the test cases.

Table 1 categorizes related work, which we now summarize.

2.1 White-box Execution-based Prioritization

Most existing TCP techniques are white-box execution-based (also called *dynamic coverage-based*): they use the dynamic execution information of the test cases to prioritize them (Yoo and Harman, 2010). Many of these technique aim to maximize source code coverage. For example, Wong et al. (1997) present a coverage-based technique to prioritize test cases in the context of regression testing, taking into account the changes in the source code between versions and giving high priority to those test cases that likely test the changed portions of the source code. Rothermel et al. (2001) present a family of coverage-based techniques, all based on statement-level execution information, and define the Average Percentage of Fault-Detection (APFD) metric that is widely used today for evaluating the effectiveness of TCP techniques. Many subsequent studies also use statement-level execution information in the source code as the basis for prioritization cases (Feldt et al, 2008; Jiang et al, 2009; Jones and Harrold, 2003; Leon and Podgurski, 2003; Masri et al, 2007; McMaster and Memon, 2006), differing mainly in coverage metric definition or maximization algorithm.

Other work prioritizes test cases by maximizing test case diversity. For example, Yoo et al. (2009) use a clustering algorithm to differentiate test case's execution profiles. Simao et al. (2006) build a feature vector for each test case (which can include any desired aspect of the test case), and then use a neural network to identify test cases with the most dissimilar features. Ramanathan et al. (2008) build a graph of test cases similarity and experiment with different graph orders for test case prioritization. In building the graph, the authors consider standard execution information as well as additional heuristics, such as memory operations.

2.2 Black-box Execution-based Prioritization

It is possible for execution-based TCP techniques to not require access to the source code of the SUT. For example, Sampath et al. (2008) focus on prioritizing test cases for web applications, where logs of user behavior (i.e., data requests sent

Table 1 A two dimensional classification of TCP techniques, based on the data available (execution information, specification models, or static source code) and maximization strategy (coverage-based or diversity-based).

Max. strategy	Execution-based		Model-based	Static	
	White-box	Black-box	Grey-box	White-box	Black-box
Coverage	e.g., (Elbaum et al, 2002; Feldt et al, 2008; Jones and Harrold, 2003)	(Sampath et al, 2008)	(Korel et al, 2007)	(Zhang et al, 2009)	this paper
Diversity	(Ramanathan et al, 2008; Simao et al, 2006; Yoo et al, 2009)	–	(Hemmati et al, 2010a,b, 2012)	–	(Ledru et al, 2009, 2011; Mei et al, 2011), this paper

to the web application server) are available to the prioritization algorithm. The user data in the logs contains the dynamic execution scenarios for the user, which captures the execution information of the SUT without actually requiring access to its source code. Using the data in the logs, Sampath et al. consider a number of strategies to prioritize test cases based, most of which try to maximize the method coverage of the SUT.

To the best of our knowledge, there have been no proposed black-box execution-based techniques based on maximizing the diversity of test cases. However, this could be achieved by, for example, clustering the elements in the user data logs of a web application, and then selecting test cases from different clusters.

2.3 Grey-box Model-based Prioritization

For situations in which the execution behavior of the SUT is not available, due to time, budget, or resource constraints, researchers have proposed techniques based on the specification models of the source code (Hemmati et al, 2010a,b, 2012; Korel et al, 2007). Specification models represent the expected behavior of the source code (via e.g., a UML state diagram) and test cases (e.g., paths on the state diagram). Specification models relate each test case to an execution path in the state model. By definition, all model-based TCP techniques are gray-box, in that they require knowledge of the internal data structures and architecture of the SUT, but do not explicitly require the source code of the SUT during the prioritization of the test suite.

For example, Korel et al. (2007) compute the difference between two versions of the specification model of a SUT, and give high priority to those test cases whose models intersect with the difference. In effect, the authors are maximizing the coverage of the model in the new version. The authors also propose a set of heuristics to enhance this basic technique.

Hemmati et al. (2010a; 2010b) explore a number of diversity-based algorithms which operate on the similarity between test cases' paths in the state model, and give high priority to those test cases whose paths are most dissimilar. The authors consider a wide range of maximization algorithms including genetic algorithms, clustering algorithms, and greedy algorithms.

In these model-based TCP techniques, the specification models of the SUT and test cases are required. Developers must create and maintain these specification models of the source code and test cases, which may not always be feasible due to the time and labor required.

2.4 White-box Static Prioritization

If both the execution behavior and specification models of the SUT are unavailable, then the TCP technique must rely only on static snapshots of the source code of the SUT and/or the test cases themselves (Ledru et al, 2011). We call this *static* TCP, which is our focus in this paper. Among static techniques, some require access to the source code of the SUT (white-box) and others do not (black-box).

An example of a white-box static TCP technique is the *call graph-based* technique, proposed by Zhang et al. (2009) and later extended by Mei et al. (2011).

The call graph-based technique uses the static call graph of test cases to approximate their dynamic source code coverage. After statically extracting the call graph of each test case, the authors define a Testing Ability (TA) metric that measures the number of source code methods that a test invokes, taking into account the methods invoked by the already-prioritized test cases. (In their work, Zhang et al. refer to this technique as "JuptaA", while Mei et al. refer to it as "JUPTA".) The authors define the TA metric by

$$\mathrm{TA}(T_i, PS) = \sum_{m \in \mathrm{Rel}(T_i, PS)} \mathrm{FCI}(m)$$

where T_i is a test case under consideration, PS is the set of already-prioritized test cases, $\mathrm{Rel}(T_i, PS)$ is a function that returns the set of methods relevant to T_i but not covered by the already-prioritized test cases PS, and the function $\mathrm{FCI}(m)$ represents the probability that method m contains a fault. Zhang et al. and Mei et al. treat all methods equally, and therefore $\mathrm{FCI}(m)$=1 for all m.

The call graph-based technique is coverage-based and prioritizes test cases using a greedy algorithm that iteratively computes the TA metric on each unprioritized test case. Initially, the test case with the highest TA metric is added to PS. Then, at each iteration, the greedy algorithm adds to PS the test case with the highest TA value. In this way, the greedy algorithm attempts to maximize the number of source code methods that are covered by the prioritized test suite.

In the above white-box static TCP technique, the maximization strategy is based on maximizing code coverage. To the best of our knowledge, no white-box static TCP technique has been proposed that maximizes based on test case diversity.

2.5 Black-box Static Prioritization

An example of a black-box static TCP technique is the *string-based* technique, proposed by Ledru et al. (2009; 2011). The string-based technique treats test cases as single, continuous strings of text. The technique uses common string distance metrics, such as the Levenshtein edit distance, between test cases to determine their similarity. The intuition is that if two test cases are textually similar, they will likely test the same portion of source code and therefore detect the same faults.

To measure the distance between two strings (i.e., test cases), Ledru et al. consider several distance metrics, including Euclidean, Manhattan, Levenshtein, and Hamming. The authors find that the Manhattan distance has the best average performance for fault detection.

To maximize diversity between strings, Ledru et al. use a greedy algorithm, similar to that of the call graph-based technique, that always prioritizes the test case which is furthest from the set of already-prioritized test cases. To do so, Ledru et al. define a distance measure between a single test case and a set of test cases. For a test case T_i, the set of already-prioritized test cases PS, and a distance function $d(T_i, T_j)$ which returns the distance between T_i and T_j, the authors define the distance between T_i and PS to be:

$$\mathrm{AllDistance}(T_i, PS, d) = \min\{d(T_i, T_j) \mid T_j \in PS, j \neq i\}. \quad (1)$$

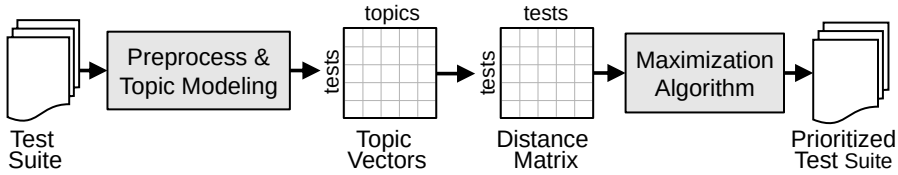


Fig. 1 Overview of our proposed topic-based TCP technique.

The authors choose the min operator because it assigns high distance values to test cases which are most different from all other test cases.

Similar to the greedy algorithm in the call graph-based technique, the greedy algorithm in the string-based technique iteratively computes the AllDistance metric for each unprioritized test case, giving high priority to the test with the highest AllDistance value at each iteration.

Despite its simplicity, this basic approach shows encouraging results that inspire us to consider the textual information to help differentiate test cases.

In this paper, during the evaluation of our proposed technique (which we present next), we also implement a black-box static TCP technique that maximizes test cases by maximizing coverage, rather than diversity. Our implementation is a modification of the white-box call graph-based technique, and is described further in Section 4.1.

3 Topic-based Prioritization

We propose a new black-box static TCP technique that takes advantage of an additional data source in the test scripts: the developer knowledge contained in the identifier names, comments, and string literals (Kuhn et al, 2007), collectively called *linguistic data*. We apply a text analysis technique called topic modeling to the test case linguistic data to abstract test cases into *topics*, which we use to compute the similarity between pairs of test cases. The motivation for using topic modeling stems from recent research that found that the topics discovered from linguistic data are good approximations of the underlying *business concerns*, or functionality, in the source code (Kuhn et al, 2007; Maskeri et al, 2008). We hypothesize that when two test cases contain the same topics, the test cases are similar in functionality and will detect the same faults.

Topic modeling is a general technique to abstract textual data that can be applied to any corpus of documents, such as newspaper articles, books, and recently source code (Linstead et al, 2008; Maskeri et al, 2008; Thomas et al, 2010). In addition to approximating the business concerns in source code, topic modeling is attractive for several reasons. First, topic modeling requires no training data or predefined taxonomies—it uses the statistics of word co-occurrences only. Second, topic modeling can be applied to source code with relative ease and still extract meaningful topics (Maskeri et al, 2008). Finally, topics modeling is fast and scales to millions of documents in real time (Porteous et al, 2008).

Our proposed TCP technique performs the following steps (see Fig. 1).

	Top Topic Words
z_6	stmt (0.052), pass (0.049), fail (0.049), count (0.047), ...
z_8	file (0.052), stream (0.029), url (0.024), ...
z_9	procedur (0.046), paramet (0.039), call (0.039), type (0.039), ...
z_{11}	stream (0.041), stmt (0.040), length (0.037), conn (0.037), ...
z_{13}	meta (0.046), column (0.043), met (0.033), odbc (0.033), ...

(a) A selection of the created topics.

	z_6	z_8	z_9	z_{11}	z_{13}
`metadataMultiConn.java`	0.00	0.00	0.00	0.08	0.79
`metadata.java`	0.01	0.00	0.13	0.00	0.84
`backupRestore1.java`	0.00	0.39	0.02	0.49	0.00
`executeUpdate.java`	0.75	0.08	0.00	0.00	0.00

(b) The topic vectors for a selection of test cases.

Fig. 2 An example topic model showing five topics and four documents. Given the raw documents (not shown), the topic modeling technique automatically creates the topics. (a) The topics (z_i) are defined by a probability distribution over words; the highest probable words are shown. (b) The documents are represented by a vector indicating which topics they contain, and to what amount. For example, `metadataMultiConn.java` contains two topics: z_{11} and z_{13}. The distance between documents can now be computed using these topic vectors.

1. Preprocess the test suite to extract the linguistic data (i.e., identifier names, comments, and string literals) for each test case.
2. Apply topic modeling to the preprocessed test suite, resulting in a vector of topic memberships for each original test case (i.e., a vector describing the probability that the test case is assigned to each topic; see Fig. 2).
3. Define the distance between pairs of test cases based on their topic membership vectors. Many standard distance metrics are applicable (Blei and Lafferty, 2009), including Manhattan distance and Kullback-Leibler (Kullback and Leibler, 1951) distance.
4. Finally, prioritize test cases by maximizing their average distances to already-prioritized test cases, using any of several distance maximization algorithms (e.g., greedy, clustering, or genetic algorithms).

3.1 Background Information on Topic Models

Given a corpus of text documents, the goal of topic modeling is to extract a set of *topics*, which can then be used for some analysis task (in our case, to calculate the similarity between documents). Topic modeling works in two steps. (See Fig. 2 for an example taken from our case study.) The first step analyzes the statistical co-occurrences of the words in the documents and creates a set of *topics*. Each topic is defined as a probability distribution over the unique words in the corpus. Words with high probability in a given topic tend to co-occur frequently. For example, in Fig. 2(a), the highest-probable words in topic z_{13} are "meta", "column", and "met", indicating that these words often occur together in many of the individual documents. Note that the distance between any two words in the document is not important, only whether they both occur in a given document. As is often the

case, the highest probable words in a topic share a semantic theme, in this case the theme being *metadata columns*.

The second step of topic modeling assigns topic membership vectors to each original document. A topic membership vector indicates the proportion of words in a document that come from each topic. For example, in Fig. 2(b), 79% of the words in `metadataMultiConn.java` come from the metadata-related topic (z_{13}). The utility of these two steps is this: by grouping all test cases that contain the metadata-related topic, we approximate the group of all test cases that test the metadata, independent of the idiosyncrasies of the text in each test case, and without the need for manually-created labels on each document.

The inferred topics are defined as probability distributions over the corpus vocabulary, and documents (i.e., test cases) are defined as probability distributions over topics. The number of topics to be created, K, is a parameter to the model. Two other parameters, α and β, help control the per-document topic smoothing and per-topic word smoothing, respectively. Given a dataset (i.e., words in documents) and values for K, α, and β, topic modeling uses a machine learning algorithm, such as Gibbs sampling or simulated annealing, to infer the topics and the topic vectors of each document (Blei and Lafferty, 2009; Blei et al, 2003).

3.2 Advantages over Existing Static Techniques

The key benefit of using topic modeling is that it uses a test script's linguistic data to provide an approximation of the business concerns of the test script. Business concerns often cannot be deduced from a test script's call graph alone. For example, consider how the call graph-based technique treats the following two example test scripts.

```
/* Read in the default color data for the menu bar, pull-down menus,
 * mouse hovers, and button down indicators. */
String defaultColors = readFileContents("default_color_options.dat");
...
```

```
/* Read in the terrain map data for Tucson, AZ. The file contains elevation
 * points of the terrain in a 10x10 meter grid. */
String terrain = readFileContents("tucson_map.dat");
...
```

Despite their obvious differences in functionality (which can be seen from the comments and identifier names), the call graph-based technique will consider these two test snippets to be identical, since they both make a single method call to the same method (i.e., `readFileContents()`). In this case, the linguistic data serves to help identify that the test snippets perform different functions, even if their call graphs are similar.

Additionally, while the string-based approach does use the comments and identifier names during the calculation of string distances, it does so in an overly rigid manner. Consider the following three example test scripts, which illustrate the effect of identifier names on the similarity measure computed by the string-based technique:

```
/* Test 1 */ int printerID=getPrinter();
/* Test 2 */ int printerID; int x; x=getPrinter(); printerID=x;
/* Test 3 */ int compiler=getCompiler();
```

Here, tests 1 and 2 use similar identifier names and are identical in terms of execution semantics, but they are considered relatively dissimilar due to the number of edits required to change one string into the other, compared to the strings' lengths. On the other hand, tests 1 and 3 have a higher string similarity, even though they have no common identifier names nor functional similarities.

4 Case Study Design

We are interested in the following research question.

Is our proposed topic-based TCP technique more effective than existing black-box static TCP techniques?

In this research question, we relate the term *effectiveness* to the standard *APFD* evaluation measure (Rothermel et al, 2001). To answer this question, we perform a detailed case study on multiple real-world software systems. We now describe the details and design decisions of our study.

4.1 Techniques Under Test

We implement and test the following static black-box TCP techniques.

- **Black-box call graph-based**. To provide a fair comparison with our proposed topic-based technique, we have implemented a black-box version of the call graph-based technique. Our implementation uses TXL (Cordy, 2006) to extract the static call graph of the test cases without relying on the source code. Essentially, our implementation extracts a list of method names invoked by each test case, but does not continue down the call graph further into the source code, as we assume the source code is unavailable. We then prioritize the test cases using the same greedy algorithm used by the white-box call graph-based technique described in Section 2.4.
- **String-based**. We have implemented the black-box static technique proposed by Ledru et al. (2011), which uses the string edit distance between test cases to maximize test case diversity.
- **Topic-based**. This technique, proposed in Section 3, uses an abstracted representation of test cases, based on topic modeling, to maximize test case diversity.

As a baseline technique, we implement and test the random technique.

- **Random**. This technique randomizes the order of the test cases.

4.2 Systems Under Test

We obtain data from the Software-artifact Infrastructure Repository (SIR), a public repository containing fault-injected software systems of various sizes, domains, and implementation languages (Do et al, 2005). The SIR is a superset of the popular Siemens test suite. The dataset for each system includes the fault-injected source code, test cases, and fault matrices (i.e., a description of which faults are

detected by which test cases) for several versions of the system. The faults were manually injected by humans in an attempt to provide realistic and representative errors made by programmers. We choose this repository because it contains many real-world, open-source systems and many previous TCP studies have used it, providing an opportunity for fair comparison.

Table 2 summarizes the five systems under test that we use from SIR. The *failure rate* of each system indicates the portion of test cases that detect at least one fault. We selected these five systems based on the following criteria.

- The test cases are written in a high-level programming language (e.g., unit tests).
- The fault matrix contains at least five faults.
- The fault matrix contains at least ten test cases.

The first criterion is required since we use static TCP techniques (which are based on analysis of the source code of the test cases); see Section 5.2.8 for a discussion. The last two criteria ensure that the datasets are large enough to draw reasonable conclusions from our results.

Both Ant and Derby are maintained by the Apache Foundation (Apache Foundation, 2012b), and both are written in the Java programming language. Ant is a command line tool that helps developers manage the build system of their software (Apache Foundation, 2012a). Derby is a lightweigth relational database management system (Apache Foundation, 2012c). We consider the versions of Derby as different systems under test because the injected faults are not propagated between versions. Each version is injected with new faults in new locations in the source code. Additionally, test cases are added and removed between versions. (An exception is Derby v4, which is very similar to Derby v3, so we only include Derby v3 in our case study.) Thus, for the purposes of our case study, we treat each version of Derby as an independent system.

These datasets provide us with test oracles on which to evaluate the techniques under test. Specifically, the datasets include a *fault matrix* which indicates which faults each test is able to detect. As the fault matrix is unknown in practice, we only use the fault matrix to evaluate the relative performance of each TCP technique; the fault matrices are not accessible to any of the TCP techniques during the prioritization process.

Table 2 Systems under test (from the Software-artifact Infrastructure Repository (Do et al, 2005)).

System	Version	Release date	SLOC (K)	No. of faults	No. of tests	Failure rate
Ant v7	1.5.3	4/2003	90	6	105	17%
Derby v1	10.1.2.1	11/2005	420	7	98	21%
Derby v2	10.1.3.1	6/2006	416	9	106	34%
Derby v3	10.2.1.6	10/2006	517	16	120	22%
Derby v5	10.3.1.4	8/2007	571	26	53	64%

4.3 Data Preprocessing

We preprocess the text of the test cases for the topic-based techniques in the standard way when applying topic modeling techniques to source code (Marcus et al, 2004). We did not preprocess the test cases in the string-based technique, to be consistent with the approach of Ledru et al. (2011).

First, we remove special characters (e.g., "&", "!", "+"), numbers, and programming language keywords. This step leaves us with identifier names, comments, and string literals. Next, we split identifier names based on camel case and underscore naming schemes, for example turning "identifierName" into the words "identifier" and "name". Next, we stem each word into its base form, for example turning both "names" and "naming" into "nam". Finally, we remove common English-language stop words, such as "the", "it", and "is". These steps help reduce the size of the test case vocabulary, allowing the topic modeling techniques to operate on a cleaner dataset than raw, unprocessed source code.

4.4 Distance Metrics and Maximization Algorithms

For our case study, we fix the distance metric to be used by both string-based and topic-based techniques. We use the Manhattan distance metric, because it is applicable to both TCP techniques and Ledru et al. found it to be optimal for string-based TCP (Ledru et al, 2011).

For distance maximization, we implement the greedy algorithm used by previous research (Elbaum et al, 2002). Namely, our implementation first uses the AllDistance metric (Equation 1) to determine the test case that is most dissimilar from the set of all test cases, and adds it to the (initially empty) set of already-prioritized test cases, PS. Then, our implementation iteratively finds the test case that is most dissimilar (again, based on the AllDistance metric) to all the test cases in PS, and adds it to PS. If two or more test cases have the same dissimilar value, the tie is broken randomly. The implementation continues in this way until all test cases have been prioritized.

4.5 Topic Modeling Technique

Our proposed topic-based TCP technique can use any of several underlying topic modeling techniques. In this case study, we use the well-known latent Dirichlet allocation (LDA) (Blei et al, 2003) model. We choose LDA because LDA is a generative statistical model, which helps to alleviate model overfitting, compared to other topic models like Probabilistic LSI (Hofmann, 1999). Also, LDA has been shown to be effective for a variety of software engineering purposes. For example, researchers have used LDA to analyze source code evolution (Linstead et al, 2008; Thomas et al, 2011), calculate source code metrics (Gethers and Poshyvanyk, 2010; Liu et al, 2009), localize features and concerns (Wang et al, 2011), program comprehension (Savage et al, 2010), mine aspects (Baldi et al, 2008), refactor code (Oliveto et al, 2011), and recover traceability links between source code and requirements documents (Asuncion et al, 2010; Gethers et al, 2011). Finally, LDA

is fast and scalable to millions of documents, keeping the TCP overhead to a minimum.

We use the `lda` package (Chang, 2011) of the R programming environment (Ihaka and Gentleman, 1996) as our LDA implementation. As LDA is a generative statistical model that uses machine learning algorithm to approximate the ideal set of topics, some randomness is involved. Given a different random number seed, LDA may produce a slightly different set of topics (Blei et al, 2003). The `lda` package uses collapsed Gibbs sampling (Porteous et al, 2008) as the machine learning algorithm to approximate topics.

To use LDA, we must specify four parameters: the number of topics, K; the document-topic smoothing parameter, α; the topic-word smoothing parameter, β; and the number of Gibbs sampling iterations to execute, II. For a given corpus, there is no provably optimal choice for K (Wallach et al, 2009). The choice is a trade-off between coarser-grained topics (smaller K) and finer-grained topics (larger K). Setting K to extremely small values results in topics that approximate multiple concerns (imagine only a single topic, which will contain all of the concerns in the corpus!), while setting K to extremely large values results in topics that are too fine and nuanced to be meaningful. Common values in the software engineering literature range from 5 to 500 topics, depending on the number of documents, granularity desired, and task to perform (Griffiths et al, 2007; Thomas, 2012b). We seek topics of medium granularity, and thus use $K{=}N/2.5$ topics per system (rounded to the nearest integer), where N is the number of documents in the SUT. Thus, we use $K = 42$, 39, 42, 48, and 21 for the five systems listed in Table 2, respectively. For α, β, and the number of iterations II, we use the defaults in the `lda` package (which are $\alpha{=}0.1$, $\beta{=}0.1$, and $II{=}200$) across all five systems, as these values are not dependent on the number of documents in the SUT. (We investigate parameter sensitivity in Section 5.2.5.)

4.6 Random Samples of Performance

To mitigate the effect of randomization and to capture the variation in the performance of each TCP technique, we run each technique several times with different random number seeds. Doing so allows us to quantify the expected behavior of each TCP technique, as well as calculate statistics on the variability of each technique.

In particular, for the topic-based technique, we first run the topic modeling technique 30 times, each with the same parameters (i.e., K, α, β, and number of iterations) but a different initial random seed. This results in a set of 30 topic models (i.e., the topics themselves as well as the topic membership vectors for each test case), each showing a slight variation. We compute the distance matrices for each of the 30 topic models. We then execute the greedy maximization algorithm 30 times for each distance matrix, which causes ties to broken in different ways.

For string-based techniques, there is no randomization when computing the distance matrix, because only deterministic processes are being executed; the process of computing the distance between the raw strings is deterministic. Instead, we apply the greedy algorithm 900 times, randomly breaking ties in different ways, in order to collect an equal number of samples as the topic-based technique. Similarly, for the call-graph technique, we apply the greedy algorithm 900 times, ran-

domly breaking ties in different ways. For the random prioritization technique, we compute 900 random prioritizations.

4.7 Analysis and Evaluation Metrics

We use a standard metric to evaluate the performance of the different techniques: *average percentage of fault-detection* (APFD) (Rothermel et al, 2001). APFD captures the average of the percentage of faults detected by a prioritized test suite. APFD is given by

$$APFD = 100 * \Big(1 - \frac{TF_1 + TF_2 + \cdots + TF_m}{nm} + \frac{1}{2n}\Big), \tag{2}$$

where n denotes the number of test cases, m is the number of faults, and TF_i is the number of tests which must be executed before fault i is detected. As a TCP technique's effectiveness increases (i.e., more faults are detected with fewer test cases), the APFD metric approaches 100. (We note that APFD is bounded in the range of $[100 * (1 - (1/2n)), 100 * (1 - (3/(2n))]$.)

To compare and contrast the APFD values of the different TCP techniques, we use three metrics. The first, $\%\Delta$, is the percent difference between the mean APFD results of two techniques:

$$\%\Delta(\mu_1, \mu_2) = \frac{\mu_2 - \mu_1}{\mu_1}, \tag{3}$$

where μ_1 and μ_2 are the mean APFD results of two TCP techniques. Second, we use the Mann-Whitney U statistical test (also known as the Mann-Whitney-Wilcoxon test) to determine if the difference between the APFD results are statistically significant. The Mann-Whitney U test has the advantage that the sample populations need not be normally distributed (non-parametric). If the p-value of the test is below a significance threshold, say 0.01, then the difference between the two techniques is considered statistically significant.

While the Mann-Whitney U test tells us whether two techniques are different, it does not tell us how much one technique outperforms another, i.e., the *effect size*. To quantify the effect size between the techniques, our third metric is the Vargha-Delaney A measure (Vargha and Delaney, 2000), which is also robust to the shape of the distributions under comparison (Arcuri and Briand, 2011). The A measure indicates the probability that one technique will achieve better performance (i.e., higher APFD) than another technique. When the A measure is 0.5, the two techniques are equal. When the A measure is above or below 0.5, one of the techniques outperforms the other.

5 Case Study Results

We wish to evaluate the effectiveness of topic-based TCP with respect to other state-of-the-art static techniques. To do so, we compare the APFD results (Equation 2) of the topic-based technique (TOPIC) against three baselines: 1) the random TCP technique (RANDOM); 2) the call graph-based technique (CALLG); and 3) the string-based technique (STRG).

Table 3 Mean APFD results (μ) and comparisons for each technique: random (RNDM), call-graph (CALLG), string (STRG), and topic (TOPIC). Each technique is compared to TOPIC using percent change in mean (%Δ), the p-value of the Mann-Whitney U test (p-val), and the Vargha-Delaney A measure (A); see Section 4.7 for definitions of these metrics. We bold the best technique(s) for each SUT.

	RNDM	CALLG	STRG	TOPIC								
					vs. RNDM			vs. CALLG			vs. STRG	
	μ	μ	μ	μ	%Δ	p-val	A	%Δ	p-val	A	%Δ	p-val	A
Ant v7	70.6	**85.5**	77.2	**84.8**	20.1	<0.001	0.94	-0.9	0.515	0.49	9.8	<0.001	0.94
Derby v1	90.7	91.8	90.4	**93.7**	3.4	<0.001	0.64	2.1	<0.001	0.77	3.7	<0.001	0.86
Derby v2	91.5	87.6	90.3	**94.6**	3.3	<0.001	0.69	8.0	<0.001	0.97	4.8	<0.001	0.91
Derby v3	92.7	93.4	93.5	**94.0**	1.3	<0.001	0.55	0.6	<0.001	0.67	0.5	<0.001	0.67
Derby v5	89.1	73.8	82.1	**96.4**	8.1	<0.001	0.86	30.5	<0.001	1.00	17.4	<0.001	1.00

5.1 Results

Table 3 and Fig. 3 present and compare the APFD results for the five systems under test. Fig. 3 shows boxplots for each technique, which summarize the distribution of the 900 random samples that we collected for each technique. Table 3 shows the mean APFD values along with comparisons between techniques using the Mann-Whitney U test (to determine if the techniques are statistically different) and the Vargha-Delaney A measure (to quantify how different the techniques are). We use the figure and table to make the following observations.

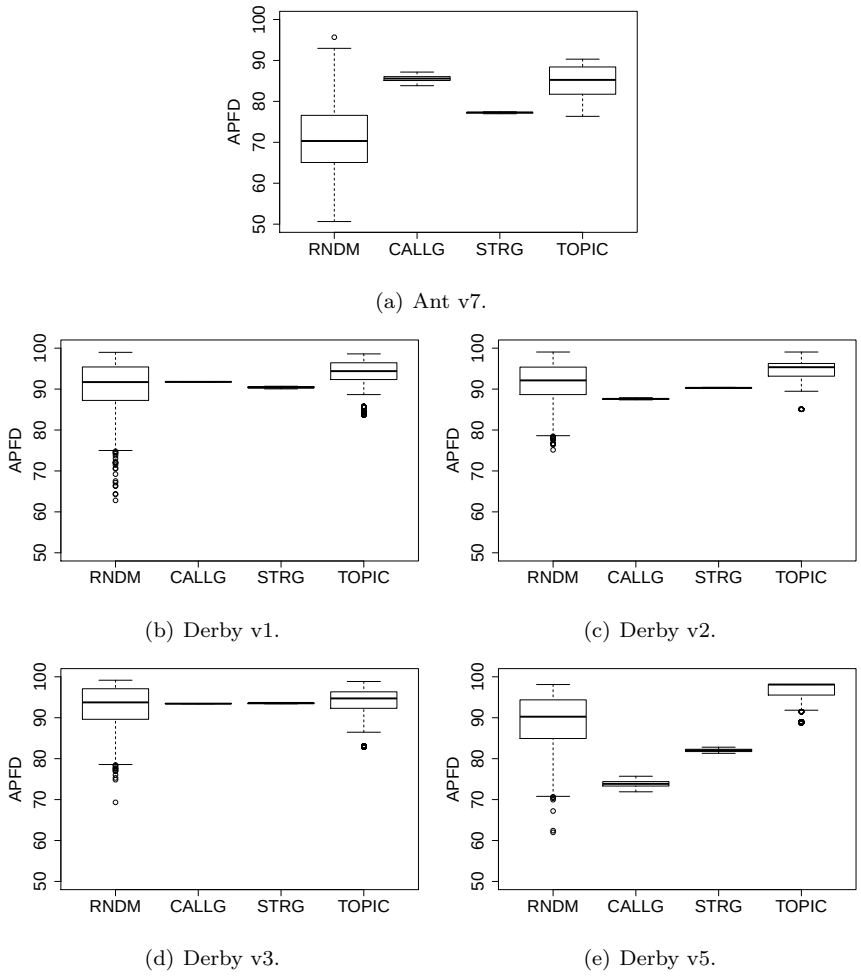
First, we find in Table 3 that the differences between techniques are almost always statistically significant according to the Mann-Whitney U test (i.e., the resulting p-values are <0.01), with one exception (Ant v7: TOPIC compared to CALLG). This indicates that the techniques capture different aspects of the tests and that the techniques result in significantly different prioritizations. In the exceptional case, TOPIC and CALLG exhibit equivalent effectiveness, as the p-value is $\geq$0.01.

Second, TOPIC always outperforms RNDM, as determined by higher mean APFD values and a effect sizes $\geq$ 0.5. In the best case, TOPIC outperforms RNDM by 20%; on average TOPIC outperforms RNDM by 7.2%.

Third, TOPIC outperforms CALLG in all but one system under test. In Ant v7, the performance of TOPIC is not statistically different from the performance of CALLG: the p-value is $\geq$0.01 and the effect size is very close to 0.5. However, in each of the other four systems under test, TOPIC outperforms CALLG by up to 31% and by 10.3% on average.

Finally, TOPIC always outperforms STRG. In the best case, TOPIC outperforms STRG by 17%; TOPIC outperforms STRG by 7.2% on average.

Our proposed topic-based TCP technique is more effective than state-of-the-art static techniques when applied to the studied systems.



(a) Ant v7.

(b) Derby v1.

(c) Derby v2.

(d) Derby v3.

(e) Derby v5.

Fig. 3 APFD results for each system under test (different subplots) and each technique (different boxes): random (RNDM), call graph (CALLG), string (STRNG), and topic (TOPIC). The boxplot show the distributions of 900 random repetitions of each technique (Section 4.6).

5.2 Discussion

We now discuss our results to better understand the characteristics of our proposed topic-based technique. We start by providing a detailed example from our case study. We then compare the topic-based technique against a well known, execution-based technique, leveraging the fact that both techniques were tested on similar datasets. We then examine how the characteristics of each SUT's fault matrix affect the results of the tested TCP techniques. We then examine how sensitive the results of the topic-based technique is to its parameters. We then investigate whether the string-based technique is sensitive to the preprocessing steps. We examine the run time of each technique, outline practical advantages and disadvantages of the

Table 4 The distances between four example test scripts in Derby: `metadataMultiConn.java` (a), `metadata.java` (b), `backupRestore1.java` (c), and `executeUpdate.java` (d). Fig. 4 contains snippets of these test scripts. We also show the rank of each distance, out of 98 total test scripts.

	Topic-based		String-based		Call graph-based	
	Dist.	Rank	Dist.	Rank	Dist.	Rank
(a) vs. (b)	0.401	1	293892	26	44	64
(a) vs. (c)	1.837	41	131250	1	37	44
(a) vs. (d)	1.952	57	400366	56	20	1

techniques, and finally enumerate potential threats to the validity of our case study.

5.2.1 Comparison of Similarity Measures: An Example

To understand our technique in more depth, we present an example from our case study. Fig. 4 contains an example from version v1 of the Derby DBMS. In this example, we compare one test script, `metadataMultiConn.java`, against all other test scripts in the suite using the similarity measures from the three static TCP techniques: topics, call graphs, and string distances. `metadataMultiConn.java` is a test script that tests the metadata columns of a DBMS using multiple connections. Our desire is to have a similarity measure that rates `metadataMultiConn.java` as highly similar to test scripts that also test metadata columns, such as `odbc_metadata.java`, `dbMetaDataJdbc30.java`, or `metadata.java`.

Fig. 4 contains a snippet of `metadataMultiConn.java` along with snippets of three other test scripts, the closest for each of the three static TCP techniques. Table 4 shows the actual distances between the test scripts.

According to the topic-based similarity measure, `metadataMultiConn.java` is closest to the `metadata.java` test script. As `metadata.java` also primarily exists to test metadata columns, the similarity with `metadataMultiConn.java` meets our desired similarity requirement. Fig. 2 confirms why these two test scripts are considered similar, namely they both have high membership in topic z_{13}.

However, with either the call-graph based or string-based similarity measures, `metadata.java` is only the 26th or 64th most similar to `metadataMultiConn.java`, respectively, indicating that these measures were unable to capture the similarity between these two scripts. Additionally, these measures did not rate any of our other desired metadata-related files as highly similar to `metadataMultiConn.java`. Using the string-based similarity measure, an unrelated test script named `backupRestore1.java` is considered the closest to `metadataMultiConn.java`, even though they share no obvious funtionalities. (`backupRestore1.java` creates a table in memory, backs it up on the hard disk, and restores it into memory.) The call graph-based similarity measure considers `executeUpdate.java` to be the most similar to `metadataMultiConn.java`, even though `executeUpdate.java` deals primarily with testing Derby's `Statement::executeUpate()` method, and has nothing to do with metadata columns nor multiple database connections. This example provides intuition as to how topic-based similarity may better determine the similarity between pairs of test cases than existing measures.

```
public class metadataMultiConn {
    ...
  Connection conn = ij.startJBMS();
  conn.setAutoCommit(autoCommit);
  Connection conn1 = getConnection(args, false);
  metadataCalls(conn1);
  Connection conn2= getConnection(args, false);
  metadataCalls(conn2);
  ...
  DatabaseMetaData dmd = conn.getMetaData();
  ...
  DatabaseMetaData dmd = conn1.getMetaData();
  ...
}
```

(a) `metadataMultiConn.java`

```
/** Test of database meta-data. This program simply calls each of the meta-data
  * methods, one by one, and prints the results. */
public class metadata extends metadata_test {
  ...
  ResultSetMetaData rsmd = s.getMetaData();
  ...
  for (int i=1; i<=numCols; i++) {
    System.out.print("[" + rsmd.getColumnTypeName(i) + "]");
  }
  ...
```

(b) `metadata.java`

```
public class backupRestore1{
  ...
  Connection conn = ij.startJBMS();
  conn.setAutoCommit( false);
  ...
  //just open to a file in existing backup, so that rename will fail on next backup
  rfs = new RandomAccessFile("extinout/mybackup/wombat/service.properties", "r");
  ...
  PreparedStatement insStmt = conn.prepareStatement(...);
  insStmt.setBinaryStream( 2, new ByteArrayInputStream( blob1), blob1.length);
  insStmt.setAsciiStream( 3, new ByteArrayInputStream( clob1), clob1.length);
  ...
}
```

(c) `backupRestore1.java`

```
class executeUpdate{
  ...
  Statement stmt = conn.createStatement();
  int rowCount = stmt.executeUpdate("create table exup(a int)");
  if (rowCount != 0)
    System.out.println("FAIL - non zero return count on create table");
  else
    System.out.println("PASS - create table");
  ...
}
```

(d) `executeUpdate.java`

Fig. 4 Four abbreviated test scripts from Derby v1. Full versions are online (Thomas, 2012a). Fig. 2 contains the topics for each test script, and Table 4 shows the static TCP similarities between them.

5.2.2 Comparison to Execution-based Results

While not the primary focus of our paper, we now wish to compare our results with those of popular white-box execution-based TCP techniques to gain an understanding of the performance differences. The most popular white-box execution-based techniques are based on basic block coverage, where a block can be defined as an individual statement or a method (Elbaum et al, 2002; Yoo and Harman, 2010). Specifically, we consider the results of Zhang et al. (2009), who have reported the

APFD results for the Ant system under test for two variations of their white-box execution-based technique, both based on method coverage information, which is consistent with our approach. Their first technique, called Method-total (MTT), orders tests cases based on the coverage of methods. Their second technique, called Method-additional (MAT), orders test cases based on the coverage of methods not yet covered.

We note that the mean APFD results reported for CALLG by Zhang et al. (87.8%) is quite similar to the results we report in this paper (85.5%), encouraging us that our implementation of their technique is accurate, and that the comparisons above are fair.

The studies performed by Zhang et al. and this paper are slightly different. Zhang et al. test 8 versions of Ant, and compute only one random sample per version. The results they report are the mean APFD value across the 8 versions. We, on the other hand, collect 900 samples from a single version of Ant, and report the mean of these 900 samples. (We only tested a single version Ant, because the remaining versions did not meet the size requirements we imposed in Section 4.2.) Even though this is not an exact comparison, it can still shed some light in the comparison of the techniques. Zhang et al. present the mean APFD value for MTT and MAT to be 64.7% and 87.5%, respectively. Our results for the mean APFD value for TOPIC is 84.1%. Thus, TOPIC outperforms one execution-based technique, and has similar performance to another. Given that the execution-based techniques have much more information available, we find these results encouraging.

5.2.3 Effects of the Characteristics of Fault Matrices on Results

A fault matrix represents which faults each test case can detect. Rows represent test cases and columns represent faults: if entry i, j is 1, then test case T_i detects fault f_j. Fault matrices are not known in advance by practitioners (hence the need for TCP techniques), but each system has a fault matrix nonetheless.

The success of any TCP technique depends on the characteristics of the system's underlying fault matrix (Rothermel et al, 2002). If, for example, all test cases can detect all faults, then developers can use any TCP technique (even random!) and achieve identical results. Similarly, if each test case can detect exactly one fault, and each fault is detected by exactly one test case (an unsorted diagonal fault matrix), again all possible TCP techniques will achieve identical results, assuming that faults have equal importance. Prioritization techniques become necessary when the fault matrix is between these two extremes.

The characteristics of the fault matrices in the systems under test vary widely, which helps explain the differences in APFD results. Table 5 quantifies various characteristics of each fault matrix: the number of faults and tests; the failure rate; the average number of faults detected by each test case; the average number of test cases that detect each fault; the percentage of tests that detect no faults; the percentage of tests that detect at least half of the faults; and the percentage of tests that detect all of the faults.

Ant v7 has the lowest failure rate, lowest average number of faults per test case, and lowest average number of test cases that detect each fault. Additionally in Ant v7, no test cases can detect more than 50% of the faults, and 83% of the tests do not detect any faults at all. Thus, detecting faults in Ant v7 is relatively difficult,

Table 5 Characteristics of the fault matrices of the systems under test.

	Faults	Tests	Failure rate (%)	Avg. faults per test	Avg. tests per fault	% tests detecting X% faults		
						X=0	X=50	X=100
Ant v7	6	105	17.1	0.21	3.67	82.9	0.0	0.0
Derby v1	7	98	21.4	0.69	9.62	78.6	9.2	5.1
Derby v2	9	106	34.0	0.92	10.89	66.0	7.5	3.8
Derby v3	16	120	21.7	1.66	12.47	78.3	10.0	6.7
Derby v5	26	53	64.2	4.37	8.90	35.8	13.2	7.5

and performance by all techniques is generally the worst for Ant v7, compared to the other systems under test. Additionally, RNDM has worse performance for Ant v7 than any other system under test, since random guessing is likely to choose tests that detect no faults.

Derby v5 has the largest percentage of tests that can detect all of faults, the smallest percentage of tests that detect no faults, and the largest average number of faults detected per test case. These three characteristics combined indicate that faults are relatively easy to detect, compared to other systems. Thus, TCP techniques become less important, as the probability of selecting a valuable test case at random is high. Indeed, RNDM has significantly better performance than the CALLG and STRG techniques in this system. However, TOPIC is still able to outperform RNDM, highlighting that TOPIC is valuable even in the extreme case of easy-to-detect faults.

In the other three systems under test (Derby v1, Derby v2, and Derby v3), the fault matrices have more middle-of-the-road characteristics: they lie somewhere between the extreme cases of Ant v7 (faults are harder to detect) and Derby v5 (faults are easier to detect). In these systems, we found that the performance differences between the techniques under test were the smallest. Even still, TOPIC consistently outperformed the other techniques, indicating that TOPIC is the best choice even for systems with average characteristics.

5.2.4 Run Time

To compare run times of the three static techniques, we ran a single instance of each technique on modest hardware (Ubuntu 9.10, 2.8 GHz CPU, 64GB RAM) and modest software (in R, the `lda` package for LDA, the `man` package for Manhattan distance, and our own implementation of the greedy algorithms) for our largest system under test, Derby v3. The topic-based technique ran in 23.1 seconds (22.9 to extract topics, <1 to compute the distance matrix, and <1 to run the greedy algorithm). The string-based technique ran in 29.6 seconds (29.15 to compute the distance matrix and <1 to run the greedy algorithm). Computing the distance matrix is longer for the string-based technique because the length of each string is much longer than the topic vectors. The call graph-based technique ran in 168.5 seconds (168.4 to extract the call graph of all tests using TXL and <1 to run the greedy algorithm).

The run times of all techniques, even with these unoptimized implementations, is trivial compared to the run time of most test cases (Hemmati et al, 2012; Rothermel et al, 2001). Further, the techniques scale: LDA can run on millions of

documents in real time (Porteous et al, 2008) and many parts of all three techniques can be parallelized, such as extracting call graphs and computing distance matrices.

5.2.5 Parameter Sensitivity of LDA

In our experiment, the topic-based TCP technique used the LDA topic model to automatically create topics that are used to compare test cases. As described in Section 4.5, LDA depends on four input parameters: the number of topics, K; document-topic and topic-word smoothing parameters, α and β; and the number of sampling iterations, II. In our experiment, we fixed these values based on the characteristics of the SUTs. In this section, we investigate how sensitive our results are to the chosen parameter values.

For each of the four parameters, we compare two values to the baseline value used in our case study: some value smaller than the baseline value; and some value larger than the baseline value. For K, we consider $K'/2$ and $K' * 2$, where K' is the original value of K used in our case study. For α, we consider $\alpha'/2$ and $\alpha' * 2$. For β, we consider $\beta'/2$ and $\beta' * 2$. Finally, for II, we consider $II'/10$ and $II' * 10$. We keep constant all other settings and design decisions from our original case study (see Section 4).

Table 6 summarizes the values tested for each parameter and shows the results. We find that changing the values of the four parameters sometimes results in a change in mean APFD, but never by a large magnitude. Some parameter changes result in an increased mean APFD, and some parameter changes result in a decreased mean AFPD. Some results are statistically significant, while others are not. We make two conclusions. First, although there is some variability in the results, and more work is needed to fully quantify the parameter space, we find that parameter values do not play a pivotal role in the results of the topic-based TCP technique. Second, the results of our original case study are not biased by showing only the results of the best possible parameter values.

5.2.6 Effects of Preprocessing on String-based TCP

In our experiment, we implemented the string-based technique exactly as described by Ledru et al. (2011) in an effort to provide a fair comparison. In particular, we computed the string edit distance between two test scripts using their raw, unprocessed text. In our topic-based technique, before we compute the distance between test scripts, we perform a text preprocessing step that removes programming language punctuation and keywords, splits compound identifiers, removes stopwords, and performs word stemming (see Section 4.3), as is common with topic models (Thomas, 2012b). To determine whether the difference in performance between the string-based technique and topic-based technique is due to the techniques themselves, and not the text preprocessing step, we perform a simple experiment. Namely, we execute the string-based technique using the preprocessed test scripts (i.e., those used by the topic-based technique), and compare the results to those of the string-based technique using the raw test scripts. We keep constant all other settings and design decisions from our original case study (see Section 4).

Table 6 Results of the parameter sensitivity analysis. For each SUT, we show the baseline mean APFD, which are a result of the parameter values used in the original case study. We then change the value of each parameter and report the resulting mean APFD, along with a comparison with the baseline mean APFD, using the p-value of the Mann-Whitney U test (p-val), and the Vargha-Delaney A measure (A); see Section 4.7 for definitions of these metrics.

Lower				Higher			
Value change	μ	p-val	A	Value change	μ	p-val	A
Ant v6 (Baseline: μ=84.8 using K=42, α=0.1, β=0.1, and II=200)							
K=21	83.6	<0.001	0.41	K=84	84.3	<0.001	0.45
α=0.05	83.0	<0.001	0.38	α=0.20	85.2	0.912	0.50
β=0.05	83.0	<0.001	0.36	β=0.20	83.8	<0.001	0.44
II=20	82.4	<0.001	0.32	II=2000	83.9	<0.001	0.45
Derby v1 (Baseline: μ=93.7 using K=39, α=0.1, β=0.1, and II=200)							
K=19	91.0	<0.001	0.30	K=78	94.0	0.774	0.50
α=0.05	92.0	<0.001	0.32	α=0.20	93.5	0.652	0.51
β=0.05	92.1	<0.001	0.41	β=0.20	92.3	<0.001	0.40
II=20	92.3	<0.001	0.39	II=2000	89.4	<0.001	0.17
Derby v2 (Baseline: μ=94.6 using K=42, α=0.1, β=0.1, and II=200)							
K=21	91.2	<0.001	0.23	K=84	95.3	<0.001	0.58
α=0.05	89.8	<0.001	0.11	α=0.20	94.5	0.008	0.54
β=0.05	94.1	0.353	0.51	β=0.20	93.3	<0.001	0.42
II=20	91.7	<0.001	0.29	II=2000	88.9	<0.001	0.09
Derby v3 (Baseline: μ=94.0 using K=48, α=0.1, β=0.1, and II=200)							
K=24	87.4	<0.001	0.16	K=96	92.1	<0.001	0.32
α=0.05	91.0	<0.001	0.21	α=0.20	94.7	<0.001	0.58
β=0.05	91.4	<0.001	0.33	β=0.20	93.2	0.222	0.48
II=20	91.5	<0.001	0.36	II=2000	87.7	<0.001	0.10
Derby v5 (Baseline: μ=96.4 using K=21, α=0.1, β=0.1, and II=200)							
K=10	93.1	<0.001	0.23	K=42	96.0	<0.001	0.44
α=0.05	90.4	<0.001	0.10	α=0.20	90.6	<0.001	0.09
β=0.05	95.8	<0.001	0.39	β=0.20	94.3	<0.001	0.32
II=20	95.2	<0.001	0.36	II=2000	94.7	<0.001	0.33

Table 7 shows the results. We use the Mann-Whitney U test and the Vargha-Delaney A measure (see Section 4.7) to compare the two versions of the string-based technique. In all five SUTs, the p-value of the Mann-Whitney U test is greater than 0.01, indicating that there is not a significant difference between the two versions of the string-based technique. In addition, the A measure is always close to 0.5, further indicating that the two versions of the technique are comparable. Thus, we conclude that the performance of the string-based TCP technique is not dependent on the text preprocessing step, and that the success of the topic-based technique is not due solely to its text preprocessing.

5.2.7 Practical Advantages of Static Techniques

Compared to existing execution-based techniques (based on code coverage information) and model-based techniques (based on specification models of the source code), static techniques enjoy several practical advantages.

Table 7 Mean APFD results for the string-based technique without and with text preprocessing. We compare the two using the p-value of the Mann-Whitney U test (p-val), and the Vargha-Delaney A measure (A); see Section 4.7 for definitions of these metrics.

	Without preprocessing	With preprocessing	p-val	A
Ant v6	77.22	77.21	0.264	0.486
Derby v1	90.38	90.38	0.549	0.508
Derby v2	90.30	90.30	0.392	0.512
Derby v3	93.52	93.51	0.031	0.472
Derby v5	82.05	82.06	0.516	0.509

First, there is no need to collect coverage information, a process that can be time-, money-, and resource-expensive due to the need to instrument and execute the entire test suite. Similarly, there is no need to create specification models of the SUT, also a process that can be time- and labor-intensive.

Second, there is no need to store coverage information on disk, which can become quite large and cumbersome for large systems with many tests. The data needed for static techniques are the test cases themselves, which are already being stored on disk.

Finally, with black-box static techniques, there is no need to maintain the coverage information or specification models as the source code and test cases evolve. Since black-box static techniques work directly on the test cases themselves, these techniques are independent of source code or model changes.

5.2.8 Scope of Static Techniques

Despite the advantages of static techniques over execution-based or model-based techniques, static techniques may not be appropriate for some TCP tasks.

First, static techniques work best on tests written in a high-level programming language. Static techniques will not perform well with short tests, or those written in command form. For example, the test cases of the Unix `bash` shell are specified as short scripts, such as `eval $1=\"\${$1:-$2}\"`. These types of tests do not have enough linguistic data or an extractable call graph for static techniques to gather data and diversify test cases.

Another consideration of the call graph-based and topic-based techniques (but not string-based) is that these techniques treat test cases as unordered bags of words. Thus, these techniques cannot capture sequences of operations in test cases. For example, if one test case opened a connection to a database, inserted rows, and then closed the connection, while another test case opened a connection to a database, closed the connection, and then inserted rows (intentionally creating an error), a bag of words model would not differentiate the two: in the case of the call graph technique, the same methods are being called and therefore the test cases would be identical; in the case of the topic-based technique, the same topics are present in the test cases, and the distance metric would not detect a difference.

5.3 Potential Threats to the Validity of the Empirical Study

Our case study provides an initial evaluation of a promising static TCP technique, and we are encouraged by the results. However, we note the following internal and external threats to the validity of our study.

Internal Validity. One potential threat to the validity of our results is our limited access to large, high-quality datasets. In particular, some of the SUTs exhibit characteristics that may not be representative of real-world systems. For example, for some SUTs, multiple test cases were able to detect all the faults in the system (Section 5.2.3). However, our results are still based on carefully selected systems from the widely-used SIR repository (Do et al, 2005).

The topic-based technique requires the topic modeling parameters—K, α, β, and the number of iterations—to be specified beforehand. However, there is currently no method for determining the optimal values of each parameter for any given project (Wallach et al, 2009), and empirically estimating the optimal values is not feasible without performing a large number of case studies. This problem is shared by topic modeling techniques in other domains, such as analyzing scientific literature (Griffiths and Steyvers, 2004) and bug localization (Lukins et al, 2010). Some research has proposed heuristics for determining the number of topics in source code (Grant and Cordy, 2010). In addition, as we found in Section 5.2.5, our results are not particularly sensitive to the exact values of the parameters. Still, further research is required to fully understand the parameter space.

Finally, the topic-based technique is based on a machine learning algorithm, which inherently involves randomness to infer the topics from the test scripts. As a result, different random number seeds may yield slightly different results. We mitigated this effect in our case study by executing 30 iterations of the topic-based technique, each with a different random number seed, and reporting the average of the results. Practitioners can mitigate the effects of randomness by using a sufficiently large number of Gibbs sampling iterations.

External Validity. Despite testing as many systems as possible from the publicly-available SIR repository, we still have only studied a limited set of systems. Our systems were all written in Java, were medium sized, and did not cover all possible application domains and testing paradigms. We therefore cannot quantify with any certainty how generalizable our results will be to other systems.

6 Conclusion and Future Work

Many test case prioritization (TCP) techniques require the execution behavior or specification model of each test case. In this paper, we considered the situation in which these information sources are not available, so-called *static* TCP. Further, we considered *black-box* static TCP, in which the source code of the system under test is not available. To this end, we proposed a new *topic-based* black-box static TCP technique, which uses topic modeling to abstract each test case into higher-level topics. We built the topics using the *linguistic data* in the test cases, i.e., the identifier names, comments, and string literals, a data source that has thus far been overlooked by existing TCP techniques, but we feel offers tremendous value to the TCP process. Using these topics, we calculated the dissimilarity between pairs of

test cases, and gave high priority to those test cases that were most dissimilar, thereby diversifying our prioritization and casting a wide net for detecting unique faults.

Given the linguistic data in the test cases, there is a spectrum of text analysis methods that can be used to differentiate test cases. On the shallow extreme, one could use the simplest text comparison possible i.e., string equivalence, to compare test cases. While this approach is fast and simple, it has the potential to miss important aspects of the test cases and be misled by trivialities in the test scripts. On the deep extreme of the spectrum, one could use full natural language processing, complete with grammars, parse trees, and parts-of-speech tagging. This approach is powerful, but requires training data, is inefficient, and is error-prone, making the automation of TCP difficult. Somewhere in the middle of the spectrum is topic modeling, which uses a bag-of-word model and word co-occurrences to approximate the functionality from each test case. We feel that topic modeling is a good trade-off for comparing test cases, because it is fast and unsupervised, yet still offers strong discrimination between test cases.

In a detailed case study of five real-world systems, we compared our topic-based TCP technique to three baseline TCP techniques: random prioritization; a black-box version of call graph-based prioritization, and string-based prioritization. We found that the proposed topic-based prioritization outperforms existing black-box static techniques by up to 31%, and is always at least as effective as the baseline techniques. These results indicate that making use of the linguistic data in test cases is an effective way to statically prioritize test cases. Further, our technique enjoys the advantage of being lightweight, in that it does not require the execution behavior or specification models of the system under test, and instead operates directly on the test scripts themselves.

Although our technique can stand on its own, our technique can also complement existing techniques. The heart of a TCP technique is the similarity measure that is used to assess similarity of test cases. More sources of information are better: in the future one could combine the strengths of existing TCP techniques by combining the various sources of information to make a more informed similarity measure. In this scenario, the contribution of this paper (i.e., a technique to extract useful information from the linguistic content of test cases) could be used to enhance *any* TCP technique, since even execution-based and model-based techniques always have the source code of test cases available.

In future work, we wish to fine tune our topic-based technique by investigating the effect of using different distance metrics between test cases, such as the Kullback-Leibler and Hellinger distance metrics. Likewise, we wish to consider additional distance maximization algorithms, such as hill climbing, genetic algorithms, and simulated annealing. We wish to consider other text analysis algorithms, such as the Vector Space Model and Latent Semantic Indexing. Finally, we plan to perform additional case studies, containing additional SUTs as well as execution-based and model-based TCP techniques, to further verify our results.

References

Ali S, Briand LC, Hemmati H, Panesar-Walawege RK (2009) A systematic review of the application and empirical investigation of search-based test case genera-

tion. IEEE Transactions on Software Engineering 36(6):742–762
Apache Foundation (2012a) Ant. URL http://ant.apache.org
Apache Foundation (2012b) Apache. URL http://www.apache.org
Apache Foundation (2012c) Derby. URL http://db.apache.org/derby
Arcuri A, Briand L (2011) A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd International Conference on Software Engineering, pp 1–10
Asuncion HU, Asuncion AU, Taylor RN (2010) Software traceability with topic modeling. In: Proceedings of the 32nd International Conference on Software Engineering, pp 95–104
Baldi PF, Lopes CV, Linstead EJ, Bajracharya SK (2008) A theory of aspects as latent topics. ACM SIGPLAN Notices 43(10):543–562
Blei DM, Lafferty JD (2009) Topic models. In: Text Mining: Classification, Clustering, and Applications, Chapman & Hall, London, UK, pp 71–94
Blei DM, Ng AY, Jordan MI (2003) Latent Dirichlet allocation. Journal of Machine Learning Research 3:993–1022
Chang J (2011) lda: Collapsed Gibbs sampling methods for topic models. URL http://cran.r-project.org/web/packages/lda
Chen S, Chen Z, Zhao Z, Xu B, Feng Y (2011) Using semi-supervised clustering to improve regression test selection techniques. In: Proceedings of the 4th International Conference on Software Testing, Verification and Validation, pp 1–10
Cordy JR (2006) The TXL source transformation language. Science of Computer Programming 61(3):190–210
Do H, Elbaum S, Rothermel G (2005) Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Engineering 10(4):405–435
Elbaum S, Malishevsky A, Rothermel G (2002) Test case prioritization: A family of empirical studies. IEEE Transactions on Software Engineering 28(2):159–182
Feldt R, Torkar R, Gorschek T, Afzal W (2008) Searching for cognitively diverse tests: Towards universal test diversity metrics. In: Proceedings of the International Conference on Software Testing Verification and Validation Workshop, pp 178–186
Gethers M, Poshyvanyk D (2010) Using relational topic models to capture coupling among classes in object-oriented software systems. In: Proceedings of the 26th International Conference on Software Maintenance, pp 1–10
Gethers M, Oliveto R, Poshyvanyk D, Lucia A (2011) On integrating orthogonal information retrieval methods to improve traceability recovery. In: Proceedings of the 27th International Conference on Software Maintenance, pp 133–142
Grant S, Cordy JR (2010) Estimating the optimal number of latent concepts in source code analysis. In: Proceedings of the 10th International Working Conference on Source Code Analysis and Manipulation, pp 65–74
Griffiths TL, Steyvers M (2004) Finding scientific topics. Proceedings of the National Academy of Sciences 101:5228–5235
Griffiths TL, Steyvers M, Tenenbaum JB (2007) Topics in semantic representation. Psychological Review 114(2):211–244
Hemmati H, Arcuri A, Briand L (2010a) Reducing the cost of model-based testing through test case diversity. In: Proceedings of the 22nd International Conference on Testing Software and Systems, pp 63–78

Hemmati H, Briand L, Arcuri A, Ali S (2010b) An enhanced test case selection approach for model-based testing: An industrial case study. In: Proceedings of the 18th International Symposium on Foundations of Software Engineering, pp 267–276

Hemmati H, Arcuri A, Briand L (2011) Empirical investigation of the effects of test suite properties on similarity-based test case selection. In: Proceedings of the 4th International Conference on Software Testing, Verification and Validation, pp 327–336

Hemmati H, Briand L, Arcuri A (2012) Achieving scalable model-based testing through test case diversity. ACM Transactions on Software Engineering and Methodology, to appear in 22(1)

Hofmann T (1999) Probabilistic Latent Semantic Indexing. In: Proceedings of the 22nd International Conference on Research and Development in Information Retrieval, pp 50–57

Ihaka R, Gentleman R (1996) R: A language for data analysis and graphics. Journal of Computational and Graphical Statistics pp 299–314

Jiang B, Zhang Z, Chan W, Tse T (2009) Adaptive random test case prioritization. In: Proceedings of the 24th International Conference on Automated Software Engineering, pp 233–244

Jones J, Harrold M (2003) Test-suite reduction and prioritization for modified condition/decision coverage. IEEE Transactions on Software Engineering 29(3):195–209

Korel B, Koutsogiannakis G, Tahat L (2007) Model-based test prioritization heuristic methods and their evaluation. In: Proceedings of the 3rd International Workshop on Advances in Model-based Testing, pp 34–43

Kuhn A, Ducasse S, Girba T (2007) Semantic clustering: Identifying topics in source code. Information and Software Technology 49(3):230–243

Kullback S, Leibler R (1951) On information and sufficiency. The Annals of Mathematical Statistics 22(1):79–86

Kumar A (2010) Development at the speed and scale of Google. Presented at QCon 2010, San Francisco, CA, USA

Ledru Y, Petrenko A, Boroday S (2009) Using string distances for test case prioritisation. In: Proceedings of the 24th International Conference on Automated Software Engineering, pp 510–514

Ledru Y, Petrenko A, Boroday S, Mandran N (2011) Prioritizing test cases with string distances. Automated Software Engineering 19(1):65–95

Leon D, Podgurski A (2003) A comparison of coverage-based and distribution-based techniques for filtering and prioritizing test cases. In: Proceedings of the International Symposium on Software Reliability Engineering, pp 442–456

Linstead E, Lopes C, Baldi P (2008) An application of latent Dirichlet allocation to analyzing software evolution. In: Proceedings of the 7th International Conference on Machine Learning and Applications, pp 813–818

Liu Y, Poshyvanyk D, Ferenc R, Gyimothy T, Chrisochoides N (2009) Modeling class cohesion as mixtures of latent topics. In: Proceedings of the 25th International Conference on Software Maintenance, pp 233–242

Lukins SK, Kraft NA, Etzkorn LH (2010) Bug localization using latent Dirichlet allocation. Information and Software Technology 52(9):972–990

Marcus A, Sergeyev A, Rajlich V, Maletic JI (2004) An information retrieval approach to concept location in source code. In: Proceedings of the 11th Working

Conference on Reverse Engineering, pp 214–223
Maskeri G, Sarkar S, Heafield K (2008) Mining business topics in source code using latent Dirichlet allocation. In: Proceedings of the 1st conference on India software engineering conference, pp 113–120
Masri W, Podgurski A, Leon D (2007) An empirical study of test case filtering techniques based on exercising information flows. IEEE Transactions on Software Engineering 33(7):454–477
McMaster S, Memon A (2006) Call stack coverage for GUI test-suite reduction. IEEE Transactions on Software Engineering 34(1):99–115
Mei H, Hao D, Zhang L, Zhang L, Zhou J, Rothermel G (2011) A static approach to prioritizing JUnit test cases. IEEE Transactions on Software Engineering
Oliveto R, Gethers M, Bavota G, Poshyvanyk D, De Lucia A (2011) Identifying method friendships to remove the feature envy bad smell. In: Proceeding of the 33rd International Conference on Software Engineering (NIER Track), pp 820–823
Porteous I, Newman D, Ihler A, Asuncion A, Smyth P, Welling M (2008) Fast collapsed Gibbs sampling for latent Dirichlet allocation. In: Proceeding of the 14th International Conference on Knowledge Discovery and Data Mining, pp 569–577
Ramanathan MK, Koyuturk M, Grama A, Jagannathan S (2008) PHALANX: A graph-theoretic framework for test case prioritization. In: Proceedings of the 23rd ACM Symposium on Applied Computing, pp 667–673
Rothermel G, Untch R, Chu C, Harrold M (2001) Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering 27(10):929–948
Rothermel G, Harrold M, Von Ronne J, Hong C (2002) Empirical studies of test-suite reduction. Software Testing, Verification and Reliability 12(4):219–249
Sampath S, Bryce RC, Viswanath G, Kandimalla V, Koru AG (2008) Prioritizing user-session-based test cases for web applications testing. In: Proceedings of the 1st International Conference on Software Testing, Verification, and Validation, pp 141–150
Savage T, Dit B, Gethers M, Poshyvanyk D (2010) TopicXP: Exploring topics in source code using latent Dirichlet allocation. In: Proceedings of the 26th International Conference on Software Maintenance, pp 1–6
Simao A, de Mello RF, Senger LJ (2006) A technique to reduce the test case suites for regression testing based on a self-organizing neural network architecture. In: Proceedings of the 30th Annual International Computer Software and Applications Conference, pp 93–96
Thomas SW (2012a) URL http://research.cs.queensu.ca/˜sthomas/
Thomas SW (2012b) Mining software repositories with topic models. Tech. Rep. 2012-586, School of Computing, Queen's University
Thomas SW, Adams B, Hassan AE, Blostein D (2010) Validating the use of topic models for software evolution. In: Proceedings of the 10th International Working Conference on Source Code Analysis and Manipulation, pp 55–64
Thomas SW, Adams B, Hassan AE, Blostein D (2011) Modeling the evolution of topics in source code histories. In: Proceedings of the 8th Working Conference on Mining Software Repositories, pp 173–182
Vargha A, Delaney HD (2000) A critique and improvement of the CL common language effect size statistics of McGraw and Wong. Journal of Educational and Behavioral Statistics 25(2):101–132

Wallach HM, Murray I, Salakhutdinov R, Mimno D (2009) Evaluation methods for topic models. In: Proceedings of the 26th International Conference on Machine Learning, pp 1105–1112

Wang S, Lo D, Xing Z, Jiang L (2011) Concern localization using information retrieval: An empirical study on Linux kernel. In: Proceedings of the 18th Working Conference on Reverse Engineering, pp 92–96

Wong W, Horgan J, London S, Agrawal H (1997) A study of effective regression testing in practice. In: Proceedings of the 8th International Symposium On Software Reliability Engineering, pp 264–274

Yoo S, Harman M (2010) Regression testing minimization, selection and prioritization: A survey. Software Testing, Verification and Reliability 22(2):67–120

Yoo S, Harman M, Tonella P, Susi A (2009) Clustering test cases to achieve effective and scalable prioritisation incorporating expert knowledge. In: Proceedings of the 18th International Symposium on Software Testing and Analysis, pp 201–212

Zhang L, Zhou J, Hao D, Zhang L, Mei H (2009) Prioritizing JUnit test cases in absence of coverage information. In: Proceedings of the 25th International Conference on Software Maintenance, pp 19–28