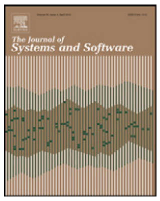
The Journal of Systems and Software xxx (2011) xxx–xxx

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss



Test coverage optimization for large code problems

Ying-Dar Lin[a,*], Chi-Heng Chou[a], Yuan-Cheng Lai[b], Tse-Yau Huang[c], Simon Chung[d], Jui-Tsun Hung[a], Frank C. Lin[e]

[a] Department of Computer Science, National Chiao Tung University, Taiwan
[b] Department of Information Management, National Taiwan University of Science and Technology, Taiwan
[c] Department of Communications Engineering, National Chiao Tung University, Taiwan
[d] Cisco Systems Inc., USA
[e] San Jose State University, USA

ARTICLE INFO

Article history:
Received 1 May 2010
Received in revised form 12 May 2011
Accepted 12 May 2011
Available online xxx

Keywords:
Regression testing
Test case selection
Test coverage
Test intensity
Software maintenance

ABSTRACT

Software developers frequently conduct regression testing on a series of major, minor, or bug-fix software or firmware releases. However, retesting all test cases for each release is time-consuming. For example, it takes about 36 test-bed-days to thoroughly exercise a test suite made up of 2320 test cases for the MPLS testing area that contains 57,758 functions in Cisco IOS. The cost is infeasible for a series of regression testing on the MPLS area. Thus, the test suite needs to be reduced intelligently, not just randomly, and its fault detection capability must be kept as much as possible. The mode of *safe* regression test selection approach is adopted for seeking a subset of modification-traversing test cases to substitute for fault-revealing test cases. The algorithms, CW-NumMin, CW-CostMin, and CW-CostCov-B, apply the *safe-mode* approach in selecting test cases for achieving full-modified function coverage. It is assumed that modified functions are *fault-prone*, and the fault distribution of the testing area is *Pareto-like*. Moreover, we also assume that once a subject program is getting more mature, its fault concentration will become stronger. Only function coverage criterion is adopted because of the scalability of a software system with large code. The metrics of *test's function reachability* and *function's test intensity* are defined in this study for algorithms. Both CW-CovMax and CW-CostMin algorithms are not safe-mode, but the approaches they use still attempt to obtain a test suite with a maximal amount of function coverage under certain constraints, i.e. the effective-confidence level and time restriction. We conclude that the most effective algorithm in this study can significantly reduce the cost (time) of regression testing on the MPLS testing area to 1.10%, on the average. Approaches proposed here can be effectively and efficiently applied to the regression testing on bug-fix releases of a software system with large code, especially to the releases having very few modified functions with low test intensities.

1. Introduction

Over the lifetime of a large software product, the number of test cases could drastically increase as new versions of software are released. Because the cost of repeatedly retesting all test cases may be too high, software testers tend to remove redundant or trivial test cases to construct a reduced test suite for regression testing at a reasonable cost. Generally, test cases are selected under certain criteria such as coverage criteria, resources constraints, or fault detection capability. In literature, researchers have developed an array of selection algorithms for regression testing, based on a variety of models of code coverage or fault detection capability. However, many algorithms demand a long execution time, and a huge number of test cases exist for a large body of code.

Bearing in mind factors of scalability and practicability, code coverage information in this study is investigated at the *function-level* granularity, rather than the *statement-level* one, e.g. condition/decision or branches. In particular, *test cases* and *functions* form an *interlaced net*; therefore, test case selection can depend on function's attributes, and vice versa. The interlaced net correlation leads to two metrics – function's *test intensity* and test's *function reachability*. The former indicates the percentage of test cases covering a function, while the latter denotes the percentage of functions reached by a test case.

Leung and White (1989) indicated that the test suite reduction problem has two subproblems – *test selection problem* and *test plan update problem*. Solutions to the former problem focus on how to select test cases to construct a reduced test suite, which can still effectively reveal faults. Yet solutions to the latter problem must cope with the management of test plans for a software system that

* Corresponding author.
E-mail addresses: ydlin@cs.nctu.edu.tw (Y.-D. Lin), payton.chou@gmail.com (C.-H. Chou), laiyc@cs.ntust.edu.tw (Y.-C. Lai), jason.hung.sb@gmail.com (J.-T. Hung).

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.05.021

had experienced several releases of modifications. Cisco did not provide test plan updates information; hence, only test selection problems can be dealt with when the automated regression test system is applied.

Recently Yoo and Harman (2010) showed a survey of regression testing on three problems – *test suite minimization, regression test-case selection (RTS), and test case prioritization*. All share a common thread of optimization when a test suite reduction is exercised from an existing pool of test cases. In this survey, regression testing is described as "*Regression testing is performed when changes are made to existing software; the purpose of regression testing is to provide confidence that the newly introduced changes do not obstruct the behaviors of the existing, unchanged part of the software*." These problems are restated as follows.

1.1. Test suite minimization problem

Given: A test suite of test cases where a set of testing requirements must be satisfied to provide the desired test coverage of the program, and subsets of the test suite where each subset is associated with *one* of the testing requirements such that any test case in the subset satisfies the testing requirement.

Problem: Find a minimal representative subset of the test suite that satisfies all the testing *requirements*.

Test suite minimization problem is well known as the minimal hitting-set problem, or the minimal set-cover problem (Garey and Johnson, 1979). Approaches to this problem typically emphasize on how to identify redundant test cases to be removed, so that a minimal test suite can be constructed. Because this problem is NP-complete, heuristics methods (Wong et al., 1998; Leung and White, 1989) are encouraged. In literature, the greedy methods, (Harrold et al., 1993; Jeffrey and Gupta, 2005, 2007; Chen and Lau, 1998a), genetic methods (Whitten, 1998; Ma et al., 2005; Mansour and El-Fakih, 1999), and linear programming methods (Black et al., 2004) are commonly applied. In addition, the hitting set algorithm (Harrold et al., 1993) categorizes test cases according to the degree of "essentialness," and selects test cases in order from the most "essential" to the least "essential." The heuristic G/GE/GRE algorithms (Chen and Lau, 1998a) are developed depending on the essential, the 1-to-1 redundant, and the greedy strategies (G: greedy strategy, E: essential strategy, and R: 1-to-1 redundant strategy).

Other approaches include modeling the cost-benefits for regression testing (Malishevsky et al., 2002), measuring the impact of test case reduction on fault detection capability (Wong et al., 1998, 1999; Rothermel et al., 1998, 2002), and analyzing fault detection capability, especially with the branch coverage technique (Harrold et al., 1993; Jeffrey and Gupta, 2005, 2007). The performance of several test suite reduction techniques are examined by experiments or simulations (Zhong et al., 2006; Chen and Lau, 1998b). Because the algorithms in (Chen and Lau, 1998a,b) do not exactly meet our requirements, G algorithms is revised and applied to test case selection, as shown in section 3.

1.2. Test case selection problem

Given: A subject program with a corresponding test suite, and a modified version of this subject program.

Problem: Find a reduced test suite for the modified version of the subject program.

In literature (Yoo and Harman, 2010), approaches to test case selection problems (Rothermel and Harrold, 1996) and to test suite minimization problems are different in how they use *changes*, or modified code, while selecting test cases. Approaches to test suite minimization problems are based on a single release of a subject program while those to regression test case selection problem are based on changes between a previous and the current version of a subject program. Hence, the approaches to test case selection problems are modification-aware (Yoo and Harman, 2010) for emphasizing the coverage of *code changes*. Moreover, Rothermel and Harrold introduced the concept of *modification-revealing* test case in (Rothermel and Harrold, 1994a) and assumed that identifying *fault-revealing* test cases for new release program is possible through the modification-revealing test cases between the original and new release of a subject program. Rothermel also adopted a weaker criterion that selects all *the modification-traversing* test cases. A test case is modification-traversing if and only if it executes new or modified code in the new release of a program, or it executes former code yet deleted in the new release. This led to a premise that selecting a subset of modification-traversing test cases and remove test cases that are guaranteed not to reveal faults in the new release of a program is possible. Thus, an approach to *safe* regression test selection problem was introduced in Rothermel and Harrold (1997), though it is still not safe for detecting all possible faults, but providing a safe sense of always selecting modification-traversing test cases into a reduced test suite. In Section 3, an algorithm that selects test cases for a test suite with full-modified function coverage is of *safe-mode* and considered a safe regression test selection. For instance, CW-NumMin, CW-CostMin, and CW-CostCov-B algorithms are of safe mode while CW-CovMax and CW-CostMin-C algorithms are not because these two algorithms do not intend to achieve full modified function coverage.

Other approaches to test case selection problems employ distinct techniques such as data flow analysis (Harrold and Soffa, 1989), the graph-walk approach (Rothermel and Harrold, 1993, 1997, 1994b), the modification-based technique (Chen et al., 1994), the firewall approach (Leung and White, 1990; White and Leung, 1992; Zheng et al., 2007) and so on. Strengths and weaknesses of these approaches can be found in (Yoo and Harman, 2010).

1.3. Test case prioritization problem

Given: A test suite and a set of permutations of the test suite.

Problem: Find a test suite where test cases are exercised in order, and a specified maximal gain is achieved under certain constraints.

The approach to this problem was first proposed by Wong et al. (1998), and extended by Harrold (1999). Empirical Studies can be found in Rothermel et al. (1999, 2001). The CW-CovMax and CW-CostMin-C algorithms in Section 3 are the variants of approaches to test case prioritization problems, except that these algorithms merely emphasize on selecting test cases, instead of exercising test cases in order.

In this work, six algorithms are implemented by a database-driven method to reduce the size of test suites, and experiments are conducted by an automated production system for regression testing on the MPLS test area of Cisco IOS. The automated system provides information on code coverage traces and execution time for each test case, while a source control system imports a history of code modification for analyzing faults detected from the newly modified code.

Faults detected in regression testing are real, compared to the faults that are hand-seeded in small subject programs when examining the test suite minimization problem. They are probably fixed if detected in a series of releases of a subject program during regression testing. Thus, unless retesting all test cases, we cannot acquire the total number of faults that can be detected. Even nobody can know the total number of real faults that exist in a subject program. Therefore, it is infeasible to calculate the fault detection effectiveness for regression testing on an industrial software system.

PDF-SA algorithm applies the function's test intensity as a metric in reducing the function space by removing infrastructure functions from an original function pool. CW-NumMin algorithm selects test cases referring to modified function information; by contrast, CW-CostMin and CW-CostCov-B algorithm refer to modification-aware and cost-aware information. Either CW-CovMax algorithm or CW-CostMin-C algorithm applies the cost-aware approach with specified constraints while selecting test cases.

This paper is organized as follows: Section 2 discusses the problems in how to select test cases. In Section 3, approaches to the problems are analyzed and implemented by six algorithms. Section 4 briefly addresses the implementation of database-driven test case selection services, but abundantly reveals the experimental results. Conclusions and future work are learned in Section 5.

2. Test suite reduction problems

We applied empirical analysis to the regression testing in the MPLS testing area of the Cisco Internetwork Operating System (IOS). The area is composed of 57,758 functions and a test suite for this area contains 2320 test cases. A single test case takes a running time from about 10 min to 100 min when a sequence of configuration and testing steps is performed. If all 2320 test cases were exercised, it would take about five weeks. Hence, a smaller subset of the test suite is required and the fault detection capability of the reduced test suite must still or almost be kept.

Referring to the problem of safe regression test selection, we are more interested in full-modified function coverage. In addition, we are also concerned with how to achieve a maximal amount of function coverage under certain constraints. Therefore, test suite reduction problems have become to how to select test cases from a test suite to construct a smaller one for regression testing. Can we obtain the minimal number of test cases or minimal cost of test cases by the information on modified functions since the last regression round? Can we balance cost and test coverage? Given a limited testing time, how can we obtain a maximal amount of modified function coverage? Given a required level of coverage, how can we acquire minimal cost of a test suite? How can we reduce the running time of the selection algorithms? In Table 1, six problems with corresponding names and descriptions are posed.

To solve the NumMin or CostMin problem, we need to find a minimal number of test cases or minimal cost of a test suite with full-modified function coverage. In the CostCov-B problem, we attempt to balance a cost-driven strategy against a coverage-driven strategy. A restriction time is given in the CovMax problem for removing test cases with higher executing time than the restriction time, and then the test case with a maximal function execution rate is selected into the reduced test suite. In the CostMin-C problem, an effective-confidence level is used for determining whether function coverage is adequate under a coverage-driven strategy. Only *reachable* functions are taken into account here, and they must be registered in the RFC database, a database for the Regression Function Coverage Architecture (RFCA), as shown in Fig. 1.

In the SA problem, various test intensity thresholds are used for identifying *infrastructure functions*. For example, if a test intensity threshold is 100% and a function has a test intensity of 100%, the function is touched or covered by *every* test case. It acts as an infrastructure function, that can be removed from the function space of MPLS testing area. The removal of infrastructure functions can significantly reduce the test case selection time and the test case execution time.

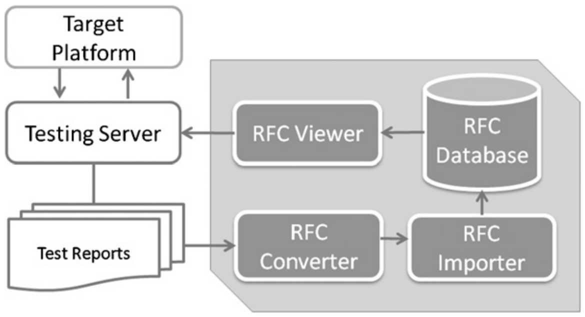


Fig. 1. Regression Function Coverage Architecture.

3. Designing six corresponding algorithms

Six algorithms in Table 2 are designed for solving the problems mentioned in Section 2. Most algorithms here are developed for the test case selection problem, rather than the test suite minimization problem or the test case prioritization problem (Yoo and Harman, 2010). Algorithms are classified into two categories – PDF-SA (Probability Density Function - Selection Acceleration) and CW (characteristics weight). Typically, the PDF-SA algorithm is performed prior to CW algorithms. Therefore, the process of regression testing is separated into two phases. First, the PDF-SA algorithm applies for removing infrastructure functions under a specified test intensity threshold. Next, one of CW-algorithms performs to acquire the reduced test suite for regression testing. For instance, the CW-NumMin algorithm, a modification-aware algorithm, can be used to select a minimum number of test cases with full-modified function coverage. We also use the greedy approach while selecting test cases based on updated function coverage for achieving a maximal amount of modified function coverage or reachability.

The variable and function notation systems are summarized in Tables 3 and 4.

3.1. PDF-SA algorithm

The PDF-SA algorithm employs a test intensity threshold η, ranging from 0% to 100% (default value $\eta = 100\%$), as a minimum boundary value for removing infrastructure functions. Now a function f_i having a test intensity η_{f_j}, more than a preset threshold η, is considered an *infrastructure function*. Function's *test intensity* η_{f_j} represents the percentage of test cases that touch function f_j, or $\eta_{f_j} = |T_{f_j}|/|T_{all}| \times 100\%$. The value of $|T_{f_j}|$ is the number of test cases that touch f_j, compared to the value of $|T_{all}|$ whose test cases are in the original test suite. If the threshold η is set with 100%, the same

Table 1
A list of six test suite reduction problems.

	Problem	Problem name	Description
1	The Number-Minimization problem	NumMin	Given a set of modified functions, obtain the minimal number of test cases
2	The Cost-Minimization problem	CostMin	Given a set of modified functions, obtain the minimal cost of test cases
3	The Cost-and-Coverage Balance problem	CostCov-B	Given modified functions, obtain test cases that can balance cost and the coverage of non-modified functions
4	The Coverage-Maximization problem	CovMax	Given a restriction time, obtain the maximal coverage of modified functions
5	The Cost-Minimization with Confidence Level problem	CostMin-C	Given an effective-confidence level, obtain the minimal cost of a set of test cases
6	The Selection Acceleration problem	SA	Find the infrastructure functions to reduce the function space

Table 2
Test coverage optimization problems and strategies.

	Algorithm	Objective	Constraint	Strategy
1	PDF-SA	Remove infrastructure functions	A test intensity threshold	Remove infrastructure functions to speed up CW-algorithms
2	CW-NumMin	Minimize the number of test cases	A set of modified functions	Decrease testing time
3	CW-CostMin	Minimize the cost of test cases	A set of modified functions	Decrease testing time
4	CW-CostCov-B	Balance the total cost of test cases against the non-modified function coverage	A set of modified functions, cost factor, coverage factor	Provide cost-driven and coverage-driven tests
5	CW-CovMax	Maximize function coverage	An amount of restriction time (min)	Provide cost-driven tests and increase function coverage
6	CW-CostMin-C	Minimize the cost of test cases until the function execution rate is greater than a specified effective-confidence level	An effective-confidence level	Provide coverage-driven tests and decrease testing time

Table 3
Variables.

Variable	Description
f_j	Function f_j in RFC database, where $j = 1, \ldots, m$
t_i	Test case t_i in a test suite, where $i = 1, \ldots, n$
T_{all}	The test suite including all test cases in the RFC database
T_{sel}	A set of selected test cases, i.e. a subset of T_{all}
T_{f_j}	Test coverage of function f_j
F_{all}, $F_{T_{all}}$	Universal function coverage of all functions in the RFC database
F_{sel}, $F_{T_{sel}}$	Function coverage of T_{sel}
F_{t_i}	Function coverage of test case t_i
F_{mod}	Function coverage of all modified functions
λ_{cost}	Cost factor
λ_{cov}	Coverage factor
τ	Restriction time, in minutes
η	Test intensity threshold
μ	Effective-confidence level
γ_{t_i}	Function reachability of t_i
η_{f_j}	Test intensity of f_j

Table 4
Functions.

Function	Description
$C(t_i)$	Function to calculate the cost, or running time, of test case t_i in minutes
$N_{mod}(F_{t_i})$	Function to calculate the number of modified functions in test case t_i
$\bar{N}_{mod}(F_{t_i})$	Function to calculate the number of non-modified functions in test case t_i
$N_{mod}(F_{T_{all}})$, or $N_{mod}(F_{all})$	Function to calculate the number of modified functions in all test cases, i.e. $\lvert F_{mod}\rvert$
$N_{mod}(F_{T_{sel}})$, or $N_{mod}(F_{sel})$	Function to calculate the number of modified functions of all the selected test cases
$W_{mod}(F_{t_i})$	The modified function weight of test case t_i
$w_{mod}(F_{t_i})$	The normalized modified function weight of test case t_i
$\bar{n}_{mod}(F_{t_i})$	The normalized non-modified function coverage of test case t_i
$f_{cv}(F_{t_i}, F_{mod}, \lambda_{cost}, \lambda_{cov})$	The comprehensive function for test case t_i

as the default value, functions covered by every test case will be removed from the function space F_{all}.

Algorithm 1. PDF-SA algorithm.

```
1  Input $\eta$, $F_{all}$, $T_{all}$
2  Output $F_{sel}$
3  Begin
4    for $\forall f_i$, $0 \le j \le |F_{all}|$, where $f_i \in F_{all}$
5      $\eta_{f_j} = |T_{f_j}|/|T_{all}| \times 100\%$;
6      if $\eta_{f_j} \ge \eta$ then $F_{sel} = F_{sel} + f_i$; // function whose test intensity is
   greater than or equal to $\eta$ is selected
7    end-for
8    return $F_{sel}$;
9  End
```

Steps of the PDF-SA algorithm are summarized as follows: Line 1: Given a test intensity threshold η, a function space F_{all}, and a test case pool T_{all}. Line 4: Check each function whether the function is an infrastructure function. Line 5: Calculate the test intensity η_{f_j} for function f_j's by $|T_{f_j}|/|T_{all}|$. Line 6: If f_j's test intensity is greater than or equal to the threshold η, function f_j is selected as an infrastructure function into F_{sel}. Line 8: Return F_{sel}, a set of infrastructure functions.

We assume that infrastructure functions provide only a little or no fault detection capability, and their existence may considerably impair the effectiveness of regression testing. Thus, it is better to remove infrastructure functions before performing test case selection algorithms. Once the size of function space F_{all} in the database is reduced, the effectiveness of CW-algorithms can be further improved.

3.2. CW-NumMin, CW-CostMin, and CW-CostCov-B algorithms

The goal of algorithms in this section is to construct test suites covering all modified functions; in other words, all changes in the subsequent releases of a software system must be re-tested to ensure the quality and reliability of the new released program. Three algorithms designed here by different test case selection strategies are to achieve a full-modified function coverage. Each is implemented with one of the characteristics weight (*CW*) – CW-NumMin (Algorithm 2), CW-CostMin (Algorithm 3), and CW-CostCov-B (Algorithm 4). These algorithms share the same structure except their own *CW* functions. Here, Algorithms 3 and 4 show only statements that are different from those in Algorithm 2.

3.2.1. Algorithm CW-NumMin: $N_{mod}(F_{t_i})$

Function $N_{mod}(F_{t_i})$ is used as the characteristic weight function in the CW-NumMin algorithm for selecting the test case with a maximal amount of modified function coverage. The value of $N_{mod}(F_{t_i})$ is obtained by the cardinality of the intersection F_{mod} and F_{t_i}, i.e., $|(F_{t_i} \cap F_{mod})|$. F_{t_i} is t_i's function coverage, and F_{mod} is the set of all modified functions.

Algorithm 2. CW-NumMin algorithm.

```
1  Input $F_{mod}$, $F_{all}$, $T_{all}$
2  Output $T_{sel}$
3  Declare $t_{sel}$: //the selected test case
4  $UpdateT()$:
5    for $\forall t_i$, $0 \le i \le |T_{all}|$, where $t_i \in T_{all}$
6      $F_{t_i} = F_{t_i} - F_{t_{sel}}$;
7      if $F_{t_i} = \varnothing$ then $T_{all} = T_{all} - t_i$;
8    end-for
9  Begin
10     while ($T_{all} \ne \phi \wedge N_{mod}(F_{t_i}) \ne 0$)
11       $t_{sel} = \mathrm{argmax}_{t_i \in T_{all}} N_{mod}(F_{t_i})$; //select the test case having a
   maximal amount of modified function coverage.
12       $T_{sel} = T_{sel} + t_{sel}$;
13       $T_{all} = T_{all} - t_{sel}$;
14       $UpdateT()$;
15     end-while
16     return $T_{sel}$;
17 End
```

Steps of the CW-NumMin algorithm are briefly described as follows: Line 1: Given a modified function space F_{mod}, a function space F_{all}, and an original test suite T_{all}. Lines 11 and 12: With a maximal amount of modified function coverage, the test case t_i is selected into T_{sel}. Line 13: Remove the selected test case t_{sel} from the original test suite T_{all}. Line 14: *UpdateT*() is exercised to remove every function that occurs in the test case t_{sel} from every test case, t_i, in the updated test suite T_{all}. Note empty test cases are removed from the updated T_{all}. Line 16: Return a test suite T_{sel} that covers all modified functions.

In the algorithm, the selection for each loop will select a test case having a maximal number of modified functions. In other words, the modification-aware greedy approach to the test case selection problems applies to acquire a minimum size of test suite that covers all modified functions. Because this test suite covers all changes in the subject program, this algorithm is a safe-mode algorithm (Rothermel and Harrold, 1993, 1997).

3.2.2. Algorithm CW-CostMin: $W_{mod}(F_{t_i})$

This CW-CostMin algorithm is for constructing a test suite that can retest all the modified functions as soon as possible. To achieve this goal, we apply a heuristic greedy approach, modification-aware and time-aware, to the test case selection problem. The *characteristics weight* function $W_{mod}(F_{t_i})$ is for test case t_i to calculate the average modified function execution rate, i.e. the number of modified functions tested per minute by t_i. The weight function $W_{mod}(F_{t_i})$ is defined as $N_{mod}(F_{t_i})/C(t_i)$. The value of$C(t_i)$ is the execution time of test case t_i containing modified and non-modified functions. In Algorithm 3, line 11 indicates that a test case is selected when it has a maximal modified function execution rate. This guarantees all modified functions can be exercised at a minimum cost or at a highest rate, but it does not guarantee the cost or size of the selected test suite is minimal.

Algorithm 3. CW-CostMin algorithm (only statements different from those in Algorithm 2 are shown).

```
10    while ($T_{all} \neq \phi \wedge W_{mod}(F_{t_i}) \neq 0$)
11        $t_{sel} = \text{argmax}_{t_i \in T_{all}} W_{mod}(F_{t_i})$; //select a test case having a maximal
      modified function execution rate.
```

3.2.3. Algorithm CW-CostCov-B: $f_{cv}(F_{t_i}, F_{mod}, \lambda_{cost}, \lambda_{cov})$

Unlike algorithms in (Black et al., 2004) using bi-criteria in constructing reduced test suites, this algorithm apply two criteria while selecting test cases. The criteria are implemented with a comprehensive function $f_{cv}(F_{t_i}, F_{mod}, \lambda_{cost}, \lambda_{cov})$, a linear combination of normalized modified function execution rate and normalized non-modified function coverage, or $w_{mod}(F_{t_i}) \times \lambda_{cost} + \bar{n}_{mod}(F_{t_i}) \times \lambda_{cov}$. The function $w_{mod}(F_{t_i})$ is a *normalized* weight function, defined as $W_{mod}(F_{t_i})/\sum_{i=1}^{n} W_{mod}(F_{t_i})$, while $\bar{n}_{mod}(F_{t_i})$ is a *normalized* t_i's non-modified function coverage, defined as $\bar{N}_{mod}(F_{t_i})$ / $\sum_{i=1}^{n} \bar{N}_{mod}(F_{t_i})$. The $\bar{N}_{mod}(F_{t_i})$ is the cardinality of the relative complement of t_i's modified function coverage, defined as $|(F_{t_i} - F_{mod})|$, or $|F_{t_i}|$-$N_{mod}(F_{t_i})$. The *cost factor* λ_{cost} (default value λ_{cost} = 0.5) and *coverage factor* $\lambda_{cov}(\lambda_{cov} = 1 - \lambda_{cost})$ are used to trade off the modified function execution rate against the non-modified function coverage. In this algorithm one factor can take precedence over the other by setting different values to the cost factor or coverage factor. If λ_{cost} = 1, the test suite obtained is the same as that in Algorithm 3 with full-modified function coverage, i.e. a test suite of safe-mode. If testers want to expand function coverage of a test suite, a larger value of λ_{cov} will be set to cover more non-modified functions. Alternatively, function $f_{cv}(F_{t_i}, F_{mod}, \lambda_{cost}, \lambda_{cov})$, defined as $w_{mod}(F_{t_i}) \times \lambda_{cost} + n_{mod}(F_{t_i}) \times \lambda_{cov}$, applies to balance the modified function execution rate and the modified function coverage. However, this alternative method cannot gain too much benefit by increasing unmodified function coverage. Algorithm CW-CostCov-B in Algorithm 4 is not only concerned with the modified function execution rate, but also provides testers a mechanism to enlarge their non-modified function coverage. In other words, if the modified function space covered by a safe-mode algorithm is inadequate, the CW-CostCov-B algorithm can offer a better way by covering a larger unmodified function space for a reduced test suite. An greedy heuristic approach is employed here.

Algorithm 4. CW-CostCov-B algorithm (only statements different from those in Algorithm 2 are shown).

```
1     Input $F_{mod}$, $t_i$, $T_{all}$, $\lambda_{cost}$, $\lambda_{cov}$
10        while ($T_{all} \neq \phi \wedge W_{mod}(F_{t_i}) \neq 0$)
11            $t_{sel} = \text{argmax}_{t_i \in T_{all}} f_{cv}(F_{t_i}, F_{mod}, \lambda_{cost}, \lambda_{cov})$; //balance a modified
      function execution rate and a non-modified function coverage.
```

3.2.4. An example for the CW-CostCov-B algorithm

We here dissect the CW-CostCov-B algorithm because the complexity of this algorithm is greater than two other algorithms. In Table 5, only five test cases are selected arbitrarily for demonstrating the steps of CW-CostCov-B algorithm. Test cases with ID (identification) from 1 to 5 are executed in 2, 3, 4, 5, and 4 min. These test cases can cover 500, 300, 700, 1,000, and 500 modified functions, as well as 500, 700, 500, 2,000 and 1,000 non-modified functions. Factors of cost and non-modified function coverage (or extra coverage) are assigned with $\lambda_{cov} = 0.4$ and λ_{cost} = 0.6. Steps of two phases are shown in Table 5.

Steps of Phase 1:

1. Calculate the values of $W_{mod}(F_{t_i})$ and $\bar{N}_{mod}(F_{t_i})$ for each test case. For example, the value of $W_{mod}(F_{t_i})$ in test case t_1 is 500/2, or 250.
2. Add the value of $W_{mod}(F_{t_i})$ and $\bar{N}_{mod}(F_{t_i})$. In Table 5(a), the total of $W_{mod}(F_{t_i})$ is 850, and $\bar{N}_{mod}(F_{t_i})$, 4700; then $\bar{n}_{mod}(F_{t_i})$, and$w_{mod}(F_{t_i})$ can be calculated. For instance, the value of $\bar{n}_{mod}(F_{t_i})$ in test case t_1 is 500/4700, or 10.64%, and $w_{mod}(F_{t_i})$, 250/850, or 29.41%.
3. Calculate the value of $f_{cv}(F_{t_i}, F_{mod}, \lambda_{cost}, \lambda_{cov})$. For instance, the comprehensive value of t_1 is $10.64\% \times 0.4 + 29.41\% \times 0.6$, or 21.90%, with the coverage factor λ_{cov}, 0.4, and the cost factor λ_{cost}, 0.6.
4. Select the test case having a highest comprehensive value, and remove it from this table. For example, in Table 5(a) test case t_4 is selected into selected test suite, and it is also removed from the original test suite.

Steps of Phase 2:

1. After test t_4 is removed, if any function in the rest test cases is the same as those in t_4, the function must be removed from the rest test cases.
2. Update the number of modified functions for function $N_{mod}(F_{t_i})$, and the number of non-modified functions for function $\bar{N}_{mod}(F_{t_i})$, as shown in Table 5(b).
3. Calculate the values of $W_{mod}(F_{t_i})$,$\bar{n}_{mod}(F_{t_i})$, $w_{mod}(F_{t_i})$, and $f_{cv}(F_{t_i}, F_{mod}, \lambda_{cost}, \lambda_{cov})$ for each of the rest test cases one by one.
4. In Table 5(b) test case t_3 is chosen into the selected test suite, and then removed from the original test suite.

3.2.5. Steps of the rest

Repeat the same steps as those in Phase 2 until the values of all entries in the column of $N_{mod}(F_{t_i})$ or $W_{mod}(F_{t_i})$ become zeros. This led to full-modified function coverage for a selected test suite, and the execution of this algorithm terminated.

Table 5
An example for the CW-CostCov-B algorithm.

t_i	$C(t_i)$	$N_{mod}(F_{t_i})$	$\bar{N}_{mod}(F_{t_i})$	$W_{mod}(F_{t_i})$	$\bar{n}_{mod}(F_{t_i})(\%)$	$w_{mod}(F_{t_i})(\%)$	$f_{cv}(F_{t_i}, F_{mod}, \lambda_{cost}, \lambda_{cov})(\%)$
(a) *Phase 1: CW-CostCov-B algorithm* ($\lambda_{cov} = 0.4,\ \lambda_{cost} = 0.6$)							
1	2	500	500	250	10.64	29.41	21.90
2	3	300	700	100	14.89	11.76	13.02
3	4	700	500	175	10.64	20.59	16.61
4 (will be selected)	5	1000	2000	200	42.55	23.53	31.14
5	4	500	1000	125	21.28	14.71	17.33
Total			4700	850			
(b) *Phase 2: CW-CostCov-B algorithm* ($\lambda_{cov} = 0.4,\ \lambda_{cost} = 0.6$)							
1	2	100	200	50	16.67	23.08	20.51
2	3	200	300	66.67	25.00	30.77	28.46
3 (will be selected)	4	300	400	75	33.33	34.62	34.10
4 (has been removed)	–	–	–	–	–	–	–
5	4	100	300	25	25.00	11.54	16.92
Total			1200	216.67			

3.3. CW-CovMax algorithm

If testers want to proceed a regression testing under a tight schedule but also want to have an extensive function coverage, this is where the CW-CovMax algorithm will play. Unlike the PDF-SA algorithm that aims to remove infrastructure functions, this algorithm first removes test cases with execution time larger than a specified cost. Here test cases with large execution time are called elephant test cases. A test suite with a maximal amount of function coverage is constructed by repeatedly selecting the test case having a maximal function execution rate. However, this algorithm is for attacking the test case selection problem, rather than the test case prioritization problem.

The weakness of this algorithm is that it is not a modification-aware approach, so the reduced test suite does not guarantee to cover all modified functions where faults are likely to occur. This could compromise the fault detection capability of the reduced test suite. However, this algorithm can benefit by removing elephant test cases with time-aware approaches.

Algorithm 5. CW-CovMax algorithm.

```
1   Input $\tau$, $F_{all}$, $T_{all}$
2   Output $T_{sel}$
3   Declare $t_{sel}$: //the selected test case.
4   $UpdateT()$: //the same as the updateT() in Algorithm 2.
5   $InitT()$:
6     for $\forall t_i, 0 \le i \le |T_{all}|$, where $t_i \in T_{all}$
7       if $C(t_i) > \tau$ then $T_{all} = T_{all} - t_i$;
8     end-for
9   Begin
10    $InitT()$;
11    if $T_{all} = \varnothing$ then return;
12    while ($T_{all} \neq \varnothing$)
13      $t_{sel} = \mathrm{argmax}_{t_i \in T_{all}} \left(\frac{|F_{t_i}|}{C(t_i)}\right)$;
14      $T_{sel} = T_{sel} + t_{sel}$;
15      $T_{all} = T_{all} - t_{sel}$;
16      $UpdateT()$;
17    end-while
18    return $T_{sel}$;
19  End
```

Steps of the CW-CovMax algorithm 5 are as follows: Line 1: Given a restriction time τ, a function space F_{all}, and a test suite T_{all}. Line 4: Update the test suite T_{all}. Lines 5–8: An initialization process removes the elephant test cases having an execution time larger than the restriction time τ. Line 12: When T_{all} is *null*, the while loop terminates. Line 13-15: a test case with a maximal function execution rate is selected into the selected test suite T_{sel}, and it is also removed from T_{all}. Line 16: Update the test suite T_{all}, i.e. remove every function appearing in test case t_{sel} from the rest test cases left in the test suite T_{all}. Empty test cases are removed from T_{all}. Line 18: Return the test suite T_{sel} which has a maximal amount of function coverage.

3.4. CW-CostMin-C algorithm

The CW-CostMin-C algorithm is an extension of the CW-CostMin algorithm, but the CW function is defined as $|F_{t_i}|/C(t_i)$, rather than $N_{mod}(F_{t_i})/C(t_i)$, or $W_{mod}(F_{t_i})$. The goal of this algorithm is to construct a test suite that may cover an *effective-confidence level* μ of function coverage, i.e. the percent of function coverage, at a minimal cost. The *effective-confidence level* μ ranges from 0% to 100% (default value μ = 100%), and is applied to notify the algorithm to stop selecting test cases once the effective-confidence level of function coverage is achieved. A test case with highest function execution rate is keeping selected from the updated test suite.

The weakness of this algorithm is the same as that in the CW-CovMax algorithm, which is not a modification-aware algorithm. There is no guarantee that the reduced test suite would cover all modified functions; this may compromise the fault detection capability of this test suite. However, this algorithm applies a time-aware approach for selecting test cases of minimal cost for a selected test suite, until the function coverage of the reduced test suite is not less than a specified effective-confidence level.

Algorithm 6. CW-CostMin-C algorithm.

```
1   Input $\mu$, $F_{all}$, $T_{all}$
2   Output $T_{sel}$
3   Declare $t_{sel}$: //the test case selected.
4   $UpdateT()$: //the same as the updateT() in Algorithm 2.
5   Begin
6     while $\left(T_{all} \neq \phi \wedge \frac{|F_{sel}|}{|F_{all}|} < \mu\right)$
7       $t_{sel} = \mathrm{argmax}_{t_i \in T_{all}} \left(\frac{|F_{t_i}|}{C(t_i)}\right)$;
8       $T_{sel} = T_{sel} + t_{sel}$;
9       $T_{all} = T_{all} - t_{sel}$;
10      $UpdateT()$;
11    end-while
12    return $T_{sel}$;
13  End
```

The following is a brief of steps of the CW-CostMin-C algorithm: Line 1: Given an effective-confidence level μ, a function space F_{all}, and a test suite T_{all}. Line 6: The while loop will terminate when T_{all} is *null* or the percent of function coverage is greater than or equal to the effective-confidence level. Lines 7–9: A test case of a maximal function execution rate is selected into T_{sel}, and removed from T_{all}. Line 10: Update the test suite T_{all} as before. Line 12: Return the test suite T_{sel} when the function coverage of the selected test suite is larger than or equal to the specified effective-confidence level μ, in percent.

Alternatively, we may replace the effective-confidence level μ with a confident testing time to construct a reduced test suite.

Table 6
Test information.

Number of test cases	391
Number of functions	23308
Total execution time (min)	7746
Number of releases	5
Number of DDTS reports	127
Number of modified functions	302
Number of reachable DDTS reports	67
Number of reachable modified functions	129

4. Experimental results

4.1. System design and implementation

The right side of Fig. 1 shows the Regression Function Coverage Architecture (RFCA), a system used for regression testing on function coverage of the Cisco IOS program. The system has four components – RFC Converter, RFC Importer, RFC Database, and RFC Viewer. First, testing tools perform a regression test in the testing server in Fig. 1, and generate test reports imported into the RFC database. Next, the RFC Viewer is configured to run test case selection algorithms and send a list of selected test cases to the test server. Prior to performing the regression testing, we must implement code on a target platform, the MPLS test area of Cisco IOS, by testing tools such as Testwell CTC++ (Testwell, 2011), so the re-tested code can be properly located and measured.

4.2. Characteristics of the test-function mappings

The entire experiment platform in this study is a personal computer with an AMD Athlon 64 3800+ 2.41 GHz processor, 3GB RAM, and Microsoft Windows XP Professional SP2. It is impractical for taking about 36 test-bed-days to thoroughly exercise a test suite of 2320 test cases for Cisco IOS containing 57,758 functions. The MPLS test area, a subset of Cisco IOS, is thus selected as the target platform because there are more test cases in the MPLS area than in other areas. Table 6 shows that there are 391 test cases in the MPLS area that contain 23,308 functions. It takes about 7746 min if all test cases are processed. Five releases are examined underlying 127 Distributed Defect Tracking Systems (DDTS) reports and 302 modified functions. All the five releases are grouped into a single set of modifications and each release may contain several DDTS reports including bug-tracking records. Now, distributed bug trackers require that adding or updating bug reports to the database be convenient. According to the Cisco test reports, only 67 DDTS reports and 129 modified functions are reachable in the MPLS test area.

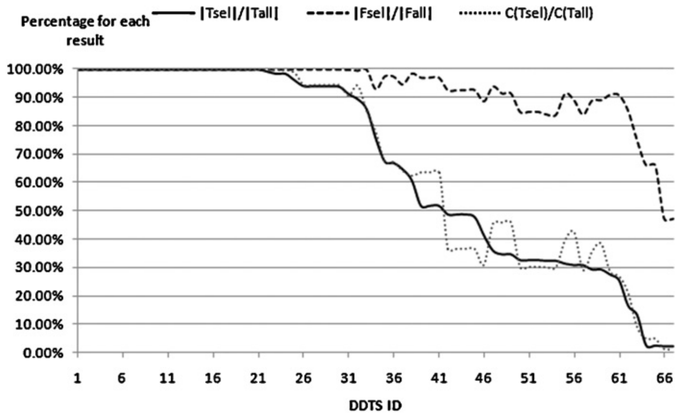


Fig. 3. Cost percent of the safe-mode selection.

Fig. 2(a) depicts the function reachability of the 391 test cases. Because of a series of testing procedures, the set of test cases now has a very high value of function reachability. Most test cases can cover about 40% up to 60% of the function coverage. Fig. 2(b) shows the test intensity of 23,308 functions. Over 25% of functions are covered by every test case, and they are considered infrastructure functions. The distribution of the function test intensity is quite non-uniform.

The *safe-mode* approach measures the cost of test case selection according to the DDTS reports. The values of $|F_{sel}|/|F_{all}|$ and $C(T_{sel})/C(T_{all})$ for each DDTS are sorted out by $|T_{sel}|/|T_{all}|$. In Fig. 3, most of the percentages of cost, $C(T_{sel})/C(T_{all})$, in DDTSs reports are still high, and higher than 30%. Only four of them are smaller. It shows 94% (63/67) of DDTS reports do not provide a substantial cost reduction even applying the safe-mode approach. Because the approach to safe-mode test selection is not as good as expected, a further reduction in cost is required.

Test case selection algorithms with varying function test intensities are compared. The PDF-SA algorithm examines the distribution of infrastructure functions based on test intensity thresholds. In Table 7, we specify several test intensity thresholds and various parameters.

To explore how a function test intensity affects a selection algorithm, we employed four thresholds, i.e. NA% (no threshold specified), 80%, 90%, and 100%. With different thresholds, we can verify which selection algorithm is more effective than others, after

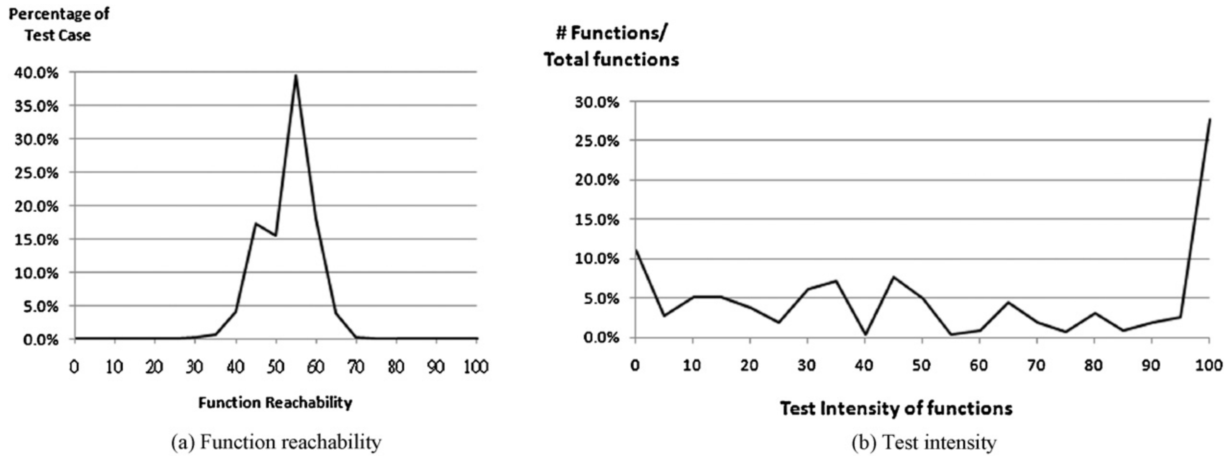


Fig. 2. Test's function reachability and function's test intensity.

Table 7
Summary of algorithms with thresholds and parameters.

	Algorithms	Test intensity threshold	Other parameters
1	PDF-SA	0, 5, . . ., 100	
2	CW-NumMin	NA, 80, 90, 100	
3	CW-CostMin	NA, 80, 90, 100	
4	CW-CostCov-B	NA, 80, 90, 100	$\lambda_{cov} = \{0, 0.1, \ldots, 1\}$
5	CW-CovMax	NA, 80, 90, 100	$\tau = \{500, 1000\}$
6	CW-CostMin-C	NA, 80, 90, 100	$\mu = \{10, 20, \ldots, 100\}$

removing infrastructure functions. Speedup of the PDF-SA algorithm under different test intensity thresholds is examined first. Both CW-NumMin and CW-CostMin algorithm are used for examine how many test cases and how much cost can be reduced. While the CW-CostCov-B algorithm investigates the impact of λ_{cov} and λ_{cost} on testing performance, the CW-CovMax algorithm selects 500 and 1000 min as the restriction time with the execution time of each test case from 10 to 100 min.

4.3. Result analysis

We first discuss test coverage under different test intensity thresholds, and then illustrate the performance of these six algorithms.

4.3.1. Test coverage under different test intensity thresholds

CW_η indicates a subset of registered functions having test intensities less than η, i.e. $\eta_{f_j} < \eta$. Here η is a test intensity threshold. CW represents an entire set of registered functions without any threshold, or $\eta_{f_j} \leq 100\%$. In contrast, $CW_{\eta100}$ is a subset of functions whose function's test intensities are less than 100%; in other words, $\eta_{f_j} <$ 100%. Functions with 100% test intensity are excluded. We examine the impact of various thresholds η's on test cost and coverage. Table 8(a) shows the results of exercising CW-NumMin algorithm with different thresholds corresponding to $CW_{\eta80}$, $CW_{\eta90}$, $CW_{\eta100}$, and CW, respectively. For instance, $CW_{\eta80}$ is a function set in which all function's test intensities are less than 80% ; in other words, functions with test intensities more than or equal to 80% are considered infrastructure functions and removed from the original function set.

Infrastructure functions are considered with little impacts on deteriorating the accuracy of regressing testing, so they can be removed to speed up test case selection algorithms. Only $CW_{\eta100}$ is employed to illustrate experimental results.

4.3.2. CW-NumMin and CW-CostMin: 2.32% and 1.1%

The result of performing the CW-NumMin algorithm is shown in Table 8(a). With $CW_{\eta100}$, only 2.56% test cases are selected, but 92.96% function coverage and 64.82% function coverage for non-modified functions are achieved. This drastically reduces the cost of the selected test cases to 2.3%. Similarly, the result of exercising the CW-CostMin algorithm is shown in Table 8(b). With $CW_{\eta100}$, only 2.56% test cases are selected, but 90.44% function coverage and 62.3% function coverage for non-modified functions are obtained. The cost of selected test cases was further reduced to 1.10%. The cost reduction by the CW-CostMin algorithm is better than that by the CW-NumMin algorithm, because each time the CW-CostMin algorithm always selects a test cases with the lowest cost, while the CW-NumMin algorithm chooses a test case with the largest coverage.

Table 8
Conventional selection vs. CW-NumMin and CW-CostMin.

Percent	$CW_{\eta80}$ (%)	$CW_{\eta90}$ (%)	$CW_{\eta100}$ (%)	CW (%)
(a) *CW-NumMin*				
$\lvert T_{sel}\rvert / \lvert T_{all}\rvert$	2.56	2.56	2.56	2.56
$\lvert F_{sel}\rvert / \lvert F_{all}\rvert$	92.96	92.96	92.96	92.96
$\bar{N}_{mod}(F_{sel})/\lvert F_{all}\rvert$	56.49	60.41	64.82	92.41
$C(T_{sel})/C(T_{all})$	2.32	2.32	2.32	2.32
(b) *CW-CostMin*				
$\lvert T_{sel}\rvert / \lvert T_{all}\rvert$	2.56	2.56	2.56	2.56
$\lvert F_{sel}\rvert / \lvert F_{all}\rvert$	90.44	90.44	90.44	90.44
$\bar{N}_{mod}(F_{sel})/\lvert F_{all}\rvert$	53.98	57.89	62.30	89.89
$C(T_{sel})/C(T_{all})$	1.10	1.10	1.10	1.10

Table 9
CW-CostCov-B under $CW_{\eta100}$.

	$CW_{\eta100}, \lambda_{cov} = 0.0$	$CW_{\eta100}, \lambda_{cov} = 1.0$
$\lvert T_{sel}\rvert / \lvert T_{all}\rvert$	1	1.20
$\lvert F_{sel}\rvert / \lvert F_{all}\rvert$	1	1.06
$\bar{N}_{mod}(F_{sel})/\lvert F_{all}\rvert$	1	1.09
$C(T_{sel})/C(T_{all})$	1	2.60

4.3.3. CW-CostCov-B: paying small cost for higher non-modified function coverage

The CW-CostCov-B algorithm employs both cost-driven and coverage-driven strategies. First, we evaluate the impacts of cost factor or coverage factor. The factor λ_{cov} is for non-modified function coverage and λ_{cost}, for cost. If λ_{cov} is larger than λ_{cost}, this means the non-modified function coverage takes precedence. A comparison of performing this algorithm with λ_{cov} from 0.0, 0.1, . . ., to 1.0, and λ_{cost} from 1.0, 0.9, . . ., to 0.0 is presented.

The curve of $\bar{N}_{mod}(F_{sel})/\lvert F_{all}\rvert$ where λ_{cov} ranges from 0.0 to 1.0 is shown in Fig. 4. Note, if $\lambda_{cost} = 1.0$, the CW-CostCov-B algorithm is the same as the CW-CostMin algorithm. Test cases selected at $\lambda_{cov} = 0.0$, or $\lambda_{cost} = 1.0$, generates 62.3% function coverage for *non-modified* functions. If λ_{cov} varies from 0.0 to 1.0, the function coverage of non-modified functions ranges from 62% to 69%. The function coverage of non-modified functions keeps less than 69%, because infrastructure functions possess about 27% function coverage. According to Fig. 4, the cost of the selected test suite, however, increased by 1% up to 3%, no matter what λ_{cov} is. Though the increase is small, it is still significant with the increase of non-modified function coverage. This may be derived from the test cases selected having large function reachability but with small cost.

The results of emphasizing either on the function coverage of non-modified functions or on cost are compared between $\lambda_{cov} = 0.0$ ($\lambda_{cost} = 1.0$) and $\lambda_{cov} = 1.0$ ($\lambda_{cost} = 0.0$), shown in Table 9, which have now normalized at $\lambda_{cov} = 0.0$. It appears that to obtain an extra 6% (68.5% − 62.5%) coverage for non-modified functions at $\lambda_{cov} = 1.0$ causes a cost 2.6 times the cost at $\lambda_{cov} = 0.0$ (the increase in cost is from 1.097% to 2.853%), and a total of test cases 1.2 times the total at $\lambda_{cov} = 0.0$. Thus, we conclude that specifying $\lambda_{cov} = 0.0$ can be a better choice for regression testing on the MPLS test area of Cisco IOS.

4.3.4. CW-CovMax: high cost for more coverage

The CW-CovMax algorithm applies cost-driven policy with restriction times (τ), such as 500 or 1000 min. In Table 10(a), when $\tau = 500$, the function coverage is 99.63%, and $\tau = 1000$, 100%. In Table 10(b), the results have now normalized at $\tau = 500$. Furthermore, $\tau = 1,000$, the function coverage increases by 0.37% (100.000%–99.630%), compared to that at $\tau = 500$, while the number of test cases at $\tau = 1000$ increased to 1.442 times the number at $\tau = 500$, and the cost at $\tau = 1000$ increased to 1.98 times the cost at $\tau = 500$. This means many test cases cover only a small number of extra functions. The increase of number of test cases soon causes a higher cost but with little improvement on expanding function coverage.

There are two possible explanations for this. First, there exist test cases of old-version, and new test cases are added without

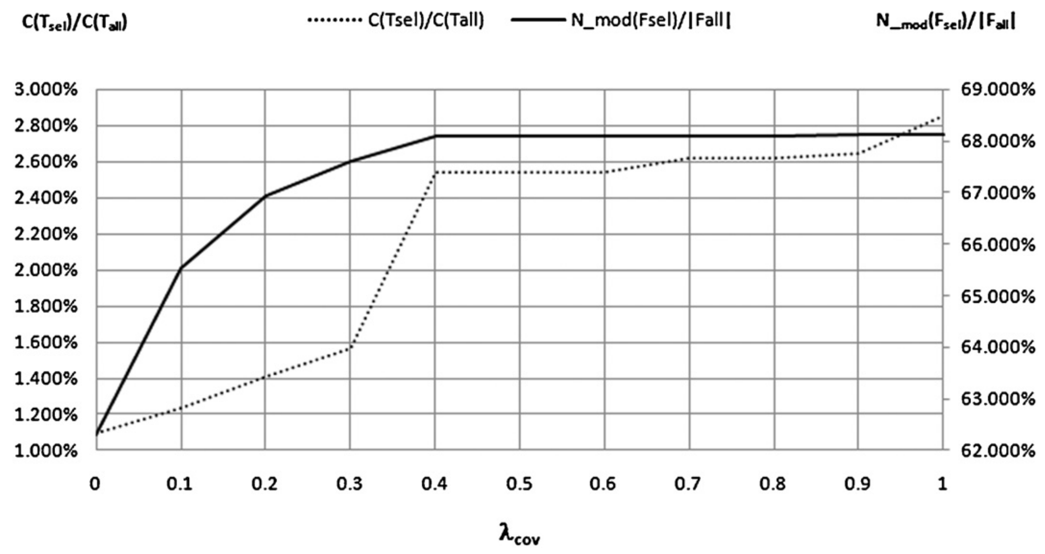


Fig. 4. Cost and non-modified function coverage of CW-CostCov-B.

Table 10
CW-CovMax under $CW_{\eta 100}$.

	$CW_{\eta 100}$, $\tau = 500$ (%)	$CW_{\eta 100}$, $\tau = 1000$ (%)
(a)		
$\lvert T_{sel}\rvert / \lvert T_{all}\rvert$	10.990	15.850
$\lvert F_{sel}\rvert / \lvert F_{all}\rvert$	99.630	100.000
$C(T_{sel})/C(T_{all})$	6.442	12.740
	$CW_{\eta 100}$, $\tau = 500$	$CW_{\eta 100}$, $\tau = 1000$
(b)		
$\lvert T_{sel}\rvert / \lvert T_{all}\rvert$	1.00	1.442
$\lvert F_{sel}\rvert / \lvert F_{all}\rvert$	1.00	1.004
$C(T_{sel})/C(T_{all})$	1.00	1.978

deleting old test cases. Functions, covered by old-version test cases, can also appear in the test cases for new-version. Second, the function coverage granularity is too coarse. When a function is touched by some test case, the function is marked as “covered”. In reality, some test cases may test different parts of the function. Because testing resources are limited, for ease of management, this study adopts function coverage as the coverage criterion. Therefore, the fault detection capability is weakened while a coarse granularity is adopted.

4.3.5. *CW-CostMin-C: coverage over 90% is inefficient*

The CW-CostMin-C algorithm is an extension of the CW-CostMin algorithm. It applies a coverage-driven method to achieve a minimal cost with sufficient function coverage. Each time the CW-CostMin algorithm always selects the test case of the smallest cost until the function coverage is adequate, as the effective-confidence level μ varies from 10% to 100%. Fig. 5 indicates that the CW-CostMin-C algorithm has 68.82% function coverage at $\mu = 10\%$. However, the coverage remains 68.82% when μ varies from 10% to 60%. This indicates that even new test cases are added in the selected test suite, the coverage does not expand. On the other hand, the cost of the selected test suite has increased drastically

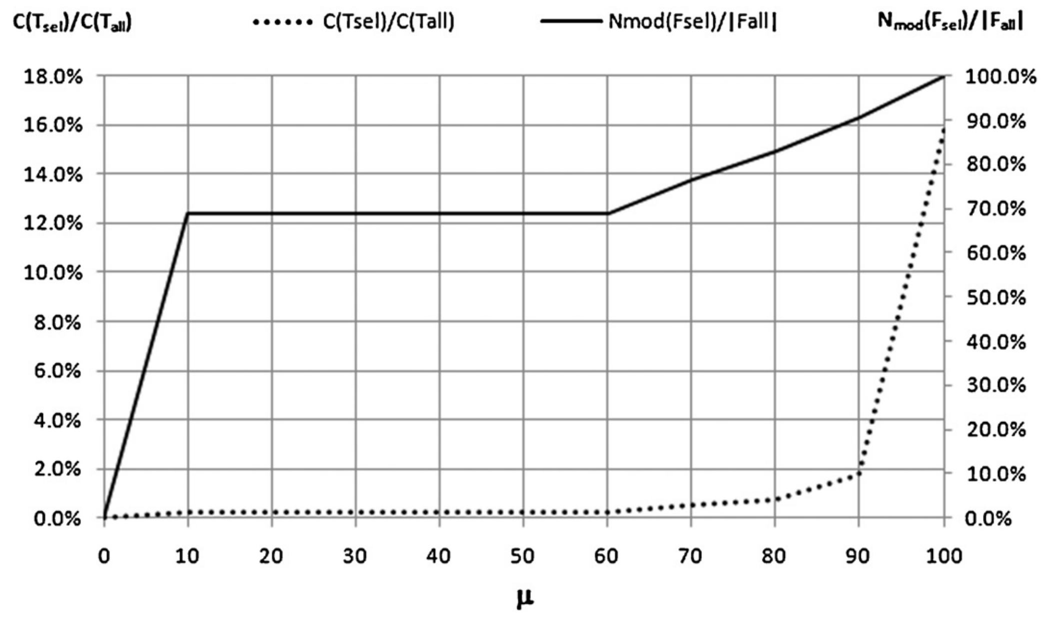


Fig. 5. Cost and coverage of CW-CovMin-C under $CW_{\eta 100}$.

Table 11
CW-CovMin-C under $CW_{\eta 100}$.

	$CW_{\eta 100}$, $\mu = 90$	$CW_{\eta 100}$, $\mu = 100$
$\|T_{sel}\|/\|T_{all}\|$	1.00	8.85
$\|F_{sel}\|/\|F_{all}\|$	1.00	1.11
$C(T_{sel})/C(T_{all})$	1.00	24.08

when μ varies from 90% to 100%. Because the increase is significant in cost, a further discussion is required.

At $\mu = 90\%$, as 1.79% test cases are selected, the cost increases by only 0.529%. At $\mu = 100\%$, as 15.85% test cases are chosen, the cost increases by 12.74%. Table 11 shows the normalized results. As the effective-confidence level varies from 90% to 100%, though the increase in the number of test cases at $\mu = 100\%$ is only 8.85 times the number of test cases at $\mu = 90\%$, the increase in the cost of the selected test suite at $\mu = 100\%$ becomes 24.08 times the cost of a test suite at $\mu = 90\%$.

4.3.6. *PDF-SA: the selection time is reduced to 10%∼70%*

Since the results of the above five algorithms have been examined, now we go forward to investigate how the PDF-SA algorithm can benefit other algorithms. Fig. 6 shows the probability density function (pdf) and cumulative density function (cdf). Both are functions of test intensity. To simplify the figure, each test intensity in Fig. 6 represents the aggregation percent of functions corresponding to every separate division as above. For example, the value at test intensity 20% means an aggregation value of the test intensity varies from 20% to 25%, where 20% is included, but 25% is excluded. The distribution of pdf is non-uniform as shown in Fig. 6. The coverage at both 100% and 0% test intensity has higher values than those at other test intensities. This indicates that a large portion of functions is covered by test cases due to the initial procedures and special features, but the rest coverage is left unreachable. Test cases can have no functions in this coverage.

If the test intensity threshold $\eta = 100$, functions with test intensity 100% are considered infrastructure functions. Thus, with $CW_{\eta 100}$, 6463 functions are considered infrastructure functions here. If all infrastructure functions are removed from the original function space, the function space may reduce by 27.73%. Two more test intensity thresholds, 80% and 90%, are used in Table 12.

Table 12
Function space reductions by PDF-SA.

	$CW_{\eta 80}$	$CW_{\eta 90}$	$CW_{\eta 100}$
The number of functions that can be ignored	8427	7510	6463
The reduced function space, in percent	36.20	32.20	27.73

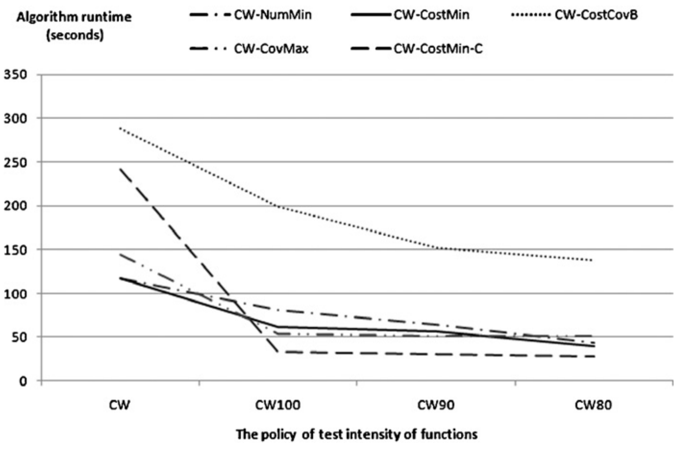


Fig. 7. Reducing the selection time by PDF-SA.

Under $CW_{\eta 80}$, there are 8,427 infrastructure functions, while under $CW_{\eta 90}$ 7,510 infrastructure functions. This led to 36.20% and 32.20% reductions in function space, respectively.

In Fig. 7 five piecewise lines represent costs for five corresponding algorithms – CW-NumMin, CW-CostMin, CW-CostCov-B, CW-CovMax, and CW-CostMin-C. *Y*-axis represents the execution time for each algorithm in seconds, while *X*-axis represents the function space with specified test intensity thresholds. In this figure different test intensity thresholds result in different amounts of reductions in execution time. There is a substantial reduction of cost for each algorithm because the function space is reduced to 27.73% under $CW_{\eta 100}$. The speedup for each algorithm from $CW_{\eta 100}$ to $CW_{\eta 90}$ or from $CW_{\eta 90}$ to $CW_{\eta 80}$ is not remarkable because the PDF-SA algorithm reduces the function space by only 4.47% and 4%, respectively. The CW-CostCov-B algorithm has a larger

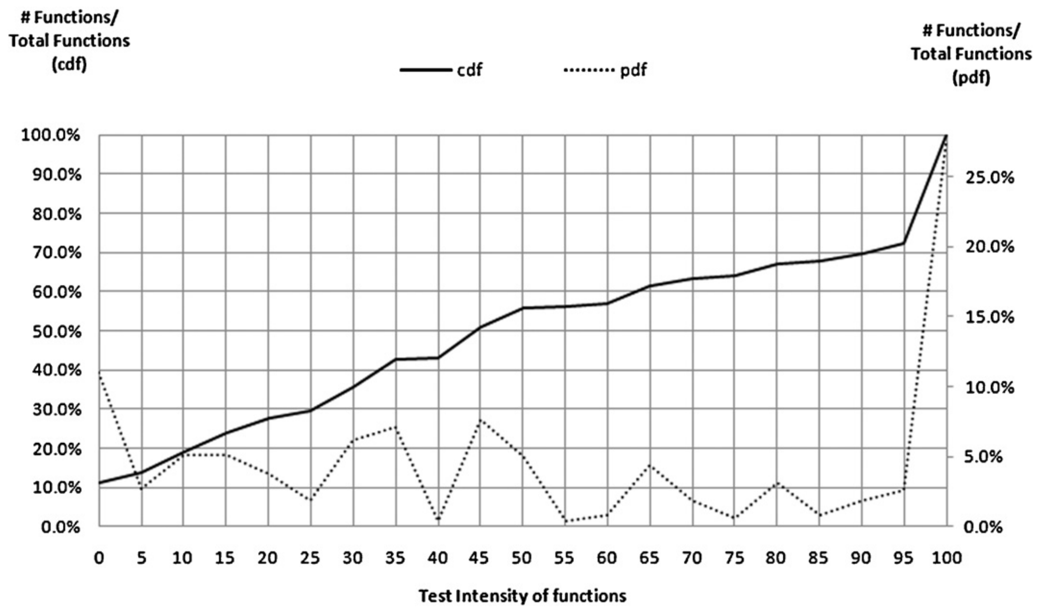


Fig. 6. Test intensity of functions.

Table 13
Selection times with and without PDF-SA.

	$CW_{\eta 80}$ (%)	$CW_{\eta 90}$ (%)	$CW_{\eta 100}$ (%)	CW (%)
CW-NumMin	37.61	54.70	62.39	100.00
CW-CostMin	34.19	48.72	52.99	100.00
CW-CostCov-B	48.76	52.96	69.09	100.00
CW-CostMax	35.42	35.76	35.15	100.00
CW-CostMin-C	11.76	12.42	13.83	100.00
Average	33.37	40.91	48.46	100.00

execution time than other algorithms because two parameters, λ_{cov} and λ_{cost}, are used for accumulating cost, and the value of $\bar{N}_{mod}(F_{t_i})$ is calculated for all test cases to solve the comprehensive function $f_{cv}(F_{t_i}, F_{mod}, \lambda_{cost}, \lambda_{cov})$.

In Table 13, the execution time of algorithms with $CW_{\eta 100}$, $CW_{\eta 90}$, or $CW_{\eta 80}$ can be reduced to 10% up to 70%. Selection algorithms with $CW_{\eta 100}$ take times to perform a variety of operations, such as union, intersection, and minus of set. Though removing infrastructure functions reduces the function space by only 27.73%, the runtime of algorithms can be reduced to 48.46%, on the average. Choosing a smaller η allows further reduction in execution time, but it is impractical if too many functions are considered infrastructure functions using a low test intensity threshold.

5. Conclusions

Regression testing of a large industrial software system with large code has become intractable. Hence, we implement a database-driven test case selection service for the MPLS testing area of Cisco IOS, and define two metrics, *function's test intensity* and *test's function reachability*, to characterize the coverage information. The former identifies whether a function is an infrastructure function by a specified threshold, and the latter is implicitly applied in Algorithms 2 and 4–6 while selecting test cases of maximal modified or non-modified function coverage. Infrastructure functions are trivial and can be removed in advance to speed up the time of test case selection and test suite execution. Approaches to test case selection problems in the study are derived from prior work, but modified to adapt to the environment of the automated regression test system on Cisco IOS. Algorithms are implemented with greedy heuristic methods.

In literature many test suite reduction techniques rely on the index of fault detection effectiveness to evaluate the performance of these techniques. However, to calculate the fault detection effectiveness in literature depends on two conditions. First, faults were seeded in subject programs of small code, and the total number of seeded-faults is known. Second, only one version of subject program was under test for test suite minimization problems, and faults seeded were not fixed, even they are detected. However, this is not the case here. We apply regression testing to an industrial software system, the Cisco IOS,which is not a small program, and the total number of real faults is *unknown*. In addition, faults detected during testing are supposed to be fixed in every subsequent release. Therefore, we argue that only real faults can reflect the phenomena and behaviors of faults occurred in a series of releases of an industrial software system. However, to obtain the total number of real faults that can be detected by performing an entire original test suite is rather costly; even there still exist many faults that cannot be detected after performing this test suite. Hence to acquire the total number of real faults is not only prohibitive, but also impossible by merely retesting all test cases of the original test suite.

In this study, the problems we attacked are test case selection problems, and the test cases we selected for reduced test suites are modification-traversing, which are used as substitutes for fault-revealing test cases. The selection strategy we adopted is the *safe-mode* regression test case selection strategy that can guarantee a certain degree of fault detection effectiveness. Because it is impossible to calculate the fault detection effectiveness for a reduced test suite in our study, further approaches to the topics of fault prediction on large software systems (Ostrand et al., 2005) and Pareto-like fault distribution (Catal, 2010; Ostrand and Weyuker, 2002) can support further studies in regression testing.

The algorithms developed in this study have reached the following achievements. First, the CW-NumMin algorithm, mainly aiming at minimizing the number of test cases, has reduced the size of test suite to 2.56%, and the testing cost to 2.32%. The CW-CostMin algorithm, emphasizing on minimizing the total cost of test cases, decreased the size of test suite to 2.56%, and the testing cost to 1.10%. The CW-CostCov-B algorithm, balancing the cost of test cases against the function coverage of non-modified functions, reached a better trade-off between cost-driven and coverage-driven strategies. Compared to the condition at $\lambda_{cov} = 1.0$, the cost of the test suite at $\lambda_{cov} = 0.0$ has increased 2.6 times ($1.097\% \rightarrow 2.853\%$) and the size of the test suite at $\lambda_{cov} = 0.0$ has expanded 1.2 times, but the size of code at $\lambda_{cov} = 0.0$ increased by only 6% ($96.25\% - 90.44\%$). The CW-CovMax algorithm, for maximizing function coverage, is a cost-driven algorithm. When the restriction time τ was relaxed from 1000 to 500, the function coverage increased by only 0.37%; in contrast, the size of test suite has enlarged 1.44 times and the testing cost has risen 1.98 times. Thus, specifying an appropriate restriction time is critical to speed up the CW-CovMax algorithm. Finally, the CW-CostMin-C algorithm is another cost-driven algorithm that minimizes test cost by an effective-confidence level. If the requested function coverage increased from 90% to 100%, the size of test suite will increase 8.85 times, and the testing cost, 24.08 times. Hence, selecting an appropriate *effective-confidence level* is crucial for regression testing. In addition, with $CW_{\eta 100}$, $CW_{\eta 90}$, and $CW_{\eta 80}$, if the PDF-SA algorithm is still applied for removing infrastructure functions, the execution time of the CW-CostMin-C algorithm can be further reduced to 48.46%, 40.91%, and 33.37%, respectively. This indicates that PDF-SA algorithm can provide other test case selection algorithms with a significant cost reduction in regression testing.

In future researchers can further explore the following issues for regression testing over a series of releases of large-scale software systems. First, if multiple test beds can test different features and multiple activities of regression testing can exercise simultaneously, allocate test cases among test beds and exercise them in parallel. Second, the function coverage generated by CTC++ tool may have flaws or be inadequate, further studies to evaluate the effectiveness of function coverage using different tools, such as attack tool, protocol fuzzier tool, and real traffic test data are required. Finally, it is worthwhile for studies in regression testing to further compare the efficiency and/or effectiveness between different coverage criteria such as function-level, statement-level, or path-level criteria.

Acknowledgements

This work was done when the first author had his sabbatical stay at Cisco Systems in San Jose in 2007–2008. It was supported in part by Cisco Systems and in part by National Science Council of Taiwan.

References

Black, J., Melachrinoudis, E., Kaeli, D., 2004. Bi-criteria models for all-uses test suite reduction. In: Proceedings of the 26th International Conference on Software Engineering, Edinburgh, UK, pp. 106–115.

Catal, C., 2010. Software fault prediction: A literature review and current trends. Expert Systems with Applications.

Chen, Y., Rosenblum, D., Vo, K., TestTube:, 1994. A system for selective regression testing. In: Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, pp. 211–220.

Chen, T.Y., Lau, M.F., 1998a. A new heuristic for test suite reduction. Information and Software Technology 40 (5–6), 347–354.

Chen, T., Lau, M., 1998b. A simulation study on some heuristics for test suite reduction. Information and Software Technology 40 (13), 777–787.

Garey, M., Johnson, D., 1979. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, San Francisco.

Harrold, M., Soffa, M., 1989. Interprocedual data flow testing. ACM SIGSOFT Software Engineering Notes 14 (8), 158–167.

Harrold, M., Gupta, R., Soffa, M., 1993. A methodology for controlling the size of a test suite. ACM Transactions on Software Engineering and Methodology (TOSEM) 2 (3), 270–285.

Harrold, M., 1999. Testing evolving software. Journal of Systems and Software 47 (2–3), 173–181.

Jeffrey, D., Gupta, N., 2005. Test suite reduction with selective redundancy. In: Proceedings of the 21st IEEE International Conference on Software Maintenance, Budapest, Hungary, pp. 549–558.

Jeffrey, D., Gupta, N., 2007. Improving fault detection capability by selectively retaining test cases during test suite reduction. IEEE Transactions on Software Engineering 33 (2), 108–123.

Leung, H., White, L., 1989. Insights into regression testing. In: Proceedings of the International Conference on Software Maintenance, Miami, FL, USA, pp. 60–69.

Leung, H., White, L., 1990. Insights into testing and regression testing global variables. Journal of Software Maintenance 2 (4), 209–222.

Ma, X., He, Z., Sheng, B., Ye, C., 2005. A genetic algorithm for test-suite reduction. In: Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, vol. 1, pp. 133–139.

Malishevsky, A., Rothermel, G., Elbaum, S., 2002. Modeling the cost-benefits tradeoffs for regression testing techniques. In: Proceedings of the 18th International Conference on Software Maintenance, Montreal, Canada, pp. 204–213.

Mansour, N., El-Fakih, K., 1999. Simulated annealing and genetic algorithms for optimal regression testing. Journal of Software Maintenance: Research and Practice 11 (1), 19–34.

Ostrand, T., Weyuker, E., 2002. The distribution of faults in a large industrial software system. In: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, Roma, Italy, pp. 55–64.

Ostrand, T., Weyuker, E., Bell, R., 2005. Predicting the location and number of faults in large software systems. IEEE Transactions on Software Engineering 31 (4), 340–355.

Rothermel, G., Harrold, M., 1993. A safe efficient algorithm for regression test selection. In: Proceedings of the Conference on Software Maintenance, Montreal, CA, pp. 358–367.

Rothermel, G., Harrold, M., 1994a. A framework for evaluating regression test selection techniques. In: Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, pp. 201–210.

Rothermel, G., Harrold, M., 1994b. Selecting tests and identifying test coverage requirements for modified software. In: Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis, ACM, pp. 169–184.

Rothermel, G., Harrold, M., 1996. Analyzing regression test selection techniques, Software Engineering. IEEE Transactions 22 (8), 529–551.

Rothermel, G., Harrold, M., 1997. A safe, efficient regression test selection technique. ACM Transactions on Software Engineering and Methodology (TOSEM) 6 (2), 173–210.

Rothermel, G., Harrold, M., Ostrin, J., Hong, C., 1998. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In: Proceedings of the International Conference on Software Maintenance, Bethesda, MD, USA, pp. 34–43.

Rothermel, G., Untch, R., Chu, C., Harrold, M., 1999. Test case prioritization: an empirical study. In: Proceedings of the International Conference on Software Maintenance, Oxford, UK, pp. 179–188.

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.

Testwell CTC++, 2011. URL http://www.testwell.fi/ctcdesc.html.

White, L., Leung, H., 1992. A firewall concept for both control-flow and data-flow in regression integration testing. In: Proceedings of the International Conference on Software Maintenance, Orlando, FL, USA, pp. 262–271.

Whitten, T., 1998. Method and computer program product for generating a computer program product test that includes an optimized set of computer program product test cases, and method for selecting same. US Patent 5,805,795 (September 8, 1998).

Wong, W., Horgan, J., London, S., Mathur, A., 1998. Effect of test set minimization on fault detection effectiveness. Software – Practice and Experience 28 (4), 347–369.

Wong, W., Horgan, J., Mathur, A., Pasquini, A., 1999. Test set size minimization and fault detection effectiveness: A case study in a space application. Journal of Systems and Software 48 (2), 79–89.

Yoo, S., Harman, M., 2010. Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability. Published online in Wiley InterScience (www.interscience.wiley.com). DOI:10.1002/stvr.430.

Zheng, J., Williams, L., Robinson, B., Smiley, K., 2007. Regression test selection for black-box dynamic link library components. In: Proceedings of the Second International Workshop on Incorporating COTS Software into Software Systems: Tools and Techniques, Minneapolis, MN, USA, pp. 9–14.

Zhong, H., Zhang, L., Mei, H., 2006. An experimental comparison of four test suite reduction techniques. In: Proceedings of the 28th International Conference on Software Engineering, Shanghai, China, pp. 636–640.