2012 26th International Conference on Advanced Information Networking and Applications Workshops

Test Coverage Optimization for Large Code Problems

Ying-Dar Lin, *Senior Member, IEEE,* Chi-Heng Chou, Yuan-Cheng Lai, Tse-Yau Huang, Simon Chung, Jui-Tsun Hung, and Frank C. Lin

***Abstract*—Because running all previous tests for the regression testing of a system is time-consuming, the size of a test suite of the system must be reduced intelligently with adequate test coverage and without compromising its fault detection capability. Five algorithms were designed for reducing the size of test suites where two metrics, *test's function reachability* and *function's test intensity,* were defined. Approaches to the algorithm CW-NumMin, CW-CostMin, or CW-CostCov-B are the safe-mode of test case selection with full-modified function coverage, while the CW-CovMax algorithm is of non-safe mode, which was performed under time restriction. In this study, the most efficient algorithm could reduce the cost (time) of a test suite down to 1.10%, on the average, over the MPLS area of Cisco IOS.**

***Index Terms*—test case selection, test coverage, regression testing, test intensity, function reachability**

I. Introduction

OVER the lifetime of a large software product, the size of a test suite may drastically increase as new versions are released, so software testers usually attempt to remove redundant or trivial tests, and select tests by certain criteria such as test coverage, resources constraints, or fault detection probability. In literature researchers have developed an array of test selection algorithms for regression testing by code coverage or fault detection capability. However, many existing algorithms still demand a long execution time or a number of tests to test a system with large code. Bearing in mind the factors of scalability and practicability, we analyzed code at the granularity of *function-level,* not *statement-level*. *Functions* and *tests* form an *interlaced net*, which can lead to metrics–function's *test intensity* and test's *function reachability.* The former indicates the percentage of tests that cover a function, and the latter the percentage of functions a test can reach.

Leung and White [1] proposed two subproblems of the reduction of a test suite, i.e. *test case selection problem* and *test plan update problem*. Solutions to the former problem emphasized on how to select test cases, and solutions to the latter on how to manage test plan update. In this work because Cisco provided no information on any test plan update, only the test selection problem could be dealt with. Yoo et al. [2] indicated that there were three problems of regression testing, *that is, test suite minimization, regression test-case selection (RTS), and test case prioritization*. They all share a common thread of optimization to reduce a test suite based on an existing test pool.

The test suite minimization problem is a minimal hitting-set problem, or a minimal set-cover problem [3]. This is an NP-complete problem, thus, heuristics methods were encouraged. Both greedy [4] and genetic [5] methods were commonly adopted. Other approaches, such as modeling cost-benefits [6], measuring the impact of test case reduction on fault detection capability [7], and analyzing fault detection capability, also have applied.

The test case selection and the test suite minimization differ only in how to deal with *changes* or modified code. The minimization problem applied to a single release of a system, but the selection problem demanded the changes between previous and the current version of the system. Hence approaches to the selection problem should be modification-aware, emphasizing the coverage of *code changes*. Rothermel and Harrold [8] introduced the concept of *modification-revealing test case*. They assumed that identifying *fault-revealing test cases* for a new software release could be possible through modification-revealing test cases. Rothermel also adopted a weaker criterion to select all *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 executes former code yet deleted in the new release.

The premise of selecting a subset of modification-traversing test cases and removing test cases without revealing faults in a new release was possible, and a solution [9] to *safe* regression test selection problem was introduced. Though it is not safe for this algorithm to detect all potential faults, but this algorithm provided a safe sense–always selecting a modification-traversing test case in a reduced test suite. Algorithms in section III that selects tests for a test suite with full-modified function coverage are of *safe-mode*, including CW-NumMin, CW-CostMin, and CW-CostCov-B algorithm.

The approach to the prioritization problem was first proposed by Wong et al. [7] and extended by Harrold [10]. The CW-CovMax algorithm in section III was a variant solution to the test case prioritization problem.

Regression testing run on the MPLS area of Cisco IOS was conducted by an automated production system, providing

Ying-Dar Lin, Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan, e-mail: ydlin@cs.nctu.edu.tw.
Chi-Heng Chou, Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan, e-mail: payton.chou@gmail.com.
Yuan-Cheng Lai, Department of Information Management, National Taiwan University of Science and Technology, Taipei, Taiwan, e-mail: laiyc@cs.ntust.edu.tw.
Tse-Yau Huang, Department of Communications Engineering, National Chiao Tung University, Hsinchu, Taiwan.
Simon Chung, Cisco Systems, Inc., USA.
Jui-Tsun Hung, Department of Computer Science, National Chiao Tung University, Hsinchu, Taiwan, e-mail: jason.hung.sb@gmail.com, jhung@cs.nctu.edu.tw.
Frank C. Lin, San Jose State University, USA.

978-0-7695-4652-0/12 $26.00 © 2012 IEEE
DOI 10.1109/WAINA.2012.137

Table I
LIST OF TEST SUITE REDUCTION PROBLEMS.

Name	Problem	Description
SA	Selection Acceleration Problem	Remove infrastructure functions to reduce the function space.
NumMin	Number-Minimization Problem	Given a set of modified functions, find a test suite with minimal number of test cases.
CostMin	Cost-Minimization Problem	Given a set of modified functions, find a test suite with minimal cost.
CostCov-B	Cost-and-Coverage Balance Problem	Given a set of modified functions, balance cost and non-modified function coverage.
CovMax	Coverage-Maximization Problem	Given a period of restricted time, find a test suite with maximal modified function's coverage.

Table II
TEST COVERAGE OPTIMIZATION PROBLEMS AND STRATEGIES.

Algorithm	Objective	Constraint	Strategy
PDF-SA	Enhance CW-algorithms performance.	Test intensity threshold	Remove infrastructure functions.
CW-NumMin	Minimize the number of test cases.	A set of modified functions	Decrease testing time.
CW-CostMin	Minimize the cost of test cases.	A set of modified functions	Decrease testing time.
CW-CostCov-B	Balance total cost of tests against non-modified function coverage.	A set of modified functions, cost factor, coverage factor	Applied to cost-driven and coverage-driven test cases.
CW-CovMax	Maximize function coverage.	Restriction time (min.)	For cost-driven tests to increase function coverage.

information on both code coverage traces and execution time.

This paper is a shortened version of the article [11] that investigated fault detection and fault prediction in more details and provided the CW-CostMin-C algorithm to solve the cost-minimization problem under an effective-confidence level. In section II five problems were proposed. In section III five algorithms for the above problems were designed. In section IV the implementation of database-driven test selection services was presented and the experimental results were addressed. Conclusions and future work were shown in section V.

II. TEST SUITE REDUCTION PROBLEMS

An empirical analysis of the MPLS area of Cisco Internetwork Operating System (IOS), containing 57,758 functions and 2,320 test cases, was performed. In the MPLS area a single test case may cost running time from10 minutes to 100 minutes with a sequence of configuration and testing steps. If all 2,320 test cases were exercised, it would have taken about five weeks, This was infeasible and a smaller test suite was required. With the safe regression test selection problem, we were interested in full-modified function coverage under certain constraints. Note, only *reachable* functions were registered in the RFC database, the Regression Function Coverage Architecture (RFCA), as shown in Figure 1 .

In this study we raised five problems, as shown in Table I. The NumMin and CostMin problem were solved by finding the minimal number of tests or the minimal cost for a reduced test suite still holding full modified function coverage. The CostCov-B problem was handled by balancing a cost-driven strategy against a coverage-driven strategy. A restriction time was used for the CovMax problem to remove test cases whose executing time was higher than restriction time. Then a test case with a maximal function execution rate would be selected into the reduced test suite. In SA problem the infrastructure function is defined as a function f_j whose test intensity η_{f_j} is greater than a threshold η. The solution was to remove infrastructure functions before selecting test cases to significantly reduce the cost (time) of test case selection and execution.

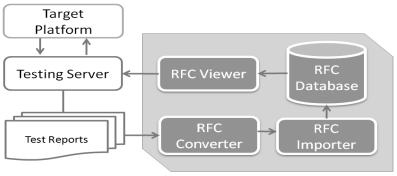


Figure 1. Regression Function Coverage Architecture.

III. FIVE CORRESPONDING ALGORITHMS

Table II indicates five algorithms related to five problems as mentioned in Section II. Algorithms were classified into two categories–PDF-SA (Probability Density Function–Selection Acceleration) and CW (characteristics weight). Test case selection has two phases. First, PDF-SA algorithm removes infrastructure functions under a test intensity threshold. Second, one CW-algorithm was performed to acquire a reduced test suite. A greedy approach based on the updated function coverage was used to select test cases to achieve a maximal modified function coverage, or reachability. Variables and function notations are respectively listed in Table III and Table IV.

A. PDF-SA Algorithm

The PDF-SA algorithm (see Algorithm 1) employed threshold η, from 0% to 100% (default 100%), as a minimum boundary value for removing infrastructure functions. The function's *test intensity* η_{f_j} represents the percentage of tests that can touch function f_j, or $\eta_{f_j} = \left|T_{f_j}\right| / |T_{all}| \times 100\%$, where $\left|T_{f_j}\right|$, the number of tests touching f_j. In this study we assumed that infrastructure functions provide only a little or no fault detection capability, but they can considerably degrade the effectiveness of regression testing.

B. CW-NumMin, CW-CostMin, CW-CostCov-B Algorithm

We assumed that changes in the subsequent releases of a system should all be re-tested; therefore, three algorithms (Al-

Table III
VARIABLES.

Variable	Description
f_j	Function f_j in RFC database, where $j = 1, \cdots, m$.
t_i	Test case t_i in a test suite, where $i = 1, \cdots, n$.
T_{all}	Test suite including all test cases in RFC database.
T_{sel}	A set of selected test cases, i.e. a subset of T_{all}.
Tf_j	Test coverage of function f_j.
$F_{all}, F_{T_{all}}$	Function coverage of all functions in 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. (min.)
η	Test intensity threshold.
γ_{t_i}	Function reachability of t_i.
η_{fj}	Test intensity of f_j.

Table IV
FUNCTIONS.

Function	Description
$C(t_i)$	Cost, or running time, function of test t_i.(min.)
$N_{mod}(F_{t_i})$	Number of modified functions in test t_i.
$\overline{N}_{mod}(F_{t_i})$	Number of non-modified functions in test t_i.
$N_{mod}(F_{all})$	Number of modified functions in all test cases.
$N_{mod}(F_{sel})$	Number of modified functions in all selected test cases.
$W_{mod}(F_{t_i})$	Modified function weight of test case t_i.
$w_{mod}(F_{t_i})$	Normalized modified function weight of test t_i.
$\overline{n}_{mod}(F_{t_i})$	Normalized non-modified function coverage of test t_i.
$f_{cv}(F_{t_i}, F_{mod}, \lambda_{cost}, \lambda_{cov})$	The comprehensive function of test t_i.

gorithm 2, Algorithm 3, and Algorithm 4) were designed for a test suites being able to cover all modified functions. Each has a *characteristic weight* (*CW*)— CW-NumMin, CW-CostMin, and CW-CostCov-B, and share the same pseudo code except *CW* functions. In Algorithm 3 and 4, only different statements from those in Algorithm 2 were shown.

1) CW-NumMin Algorithm: Function $N_{mod}(F_{t_i})$, a characteristic weight function, was used to select tests having maximal modified function coverage. Function $N_{mod}(F_{t_i})$ is the cardinality of the intersection of F_{mod} and F_{t_i}, or $|(F_{t_i} \cap F_{mod})|$. A modification-aware greedy approach applied here to acquire a minimal reduced test suite that covers all modified functions.

2) CW-CostMin Algorithm: This algorithm is for building a reduced test suite to retest all modified functions as soon as possible. A heuristic greedy approach, modification-aware and time-aware, was applied in test case selection. Test t_i applied the function $W_{mod}(F_{t_i})$ to calculate the average modified function execution rate, or the number of modified functions tested per minute by t_i. Here $W_{mod}(F_{t_i})$ is defined as $N_{mod}(F_{t_i})/C(t_i)$. Function $C(t_i)$ shows the execution time of test case t_i that includes modified and non-modified functions. This would guarantee that all modified functions were exercised at a minimum cost or a highest rate; however, the cost or the size of the selected test suite may not be minimal.

3) CW-CostCov-B Algorithm: Function $f_{cv}(F_{t_i}, F_{mod}, \lambda_{cost}, \lambda_{cov})$ is a linear combination of normalized modified function execution rate and normalized non-modified function coverage, or $w_{mod}(F_{t_i})\times\lambda_{cost}$ + $\overline{n}_{mod}(F_{t_i})\times\lambda_{cov}$. Function $w_{mod}(F_{t_i})$ is a *normalized* weight function, or $W_{mod}(F_{t_i})/\sum_{i=1}^{n} W_{mod}(F_{t_i})$, and $\overline{n}_{mod}(F_{t_i})$ is a *normalized* t_i's non-modified function coverage, or $\overline{N}_{mod}(F_{t_i})$ / $\sum_{i=1}^{n} \overline{N}_{mod}(F_{t_i})$. Function $\overline{N}_{mod}(F_{t_i})$ is the cardinality of the relative complement of t_i's modified function coverage, or $|(F_{t_i} - F_{mod})|$. *Cost factor* λ_{cost} (default 0.5) and *coverage factor* λ_{cov}, ($\lambda_{cov} = 1-\lambda_{cost}$), trade off the modified function execution rate against the non-modified function coverage. If $\lambda_{cost} = 1$, the reduced test suite is the same as that obtained by Algorithm 3.

To enlarge the function coverage of a test suite, testers should set λ_{cov} a larger value to cover more non-modified functions. Function $f_{cv}(F_{t_i}, F_{mod}, \lambda_{cost}, \lambda_{cov})$, defined as $w_{mod}(F_{t_i})\times\lambda_{cost}$ + $n_{mod}(F_{t_i})\times\lambda_{cov}$, can balance the modified function execution rate and the modified function coverage. The Algorithm CW-CostCov-B employs $f_{cv}(F_{t_i}, F_{mod}, \lambda_{cost}, \lambda_{cov})$ to deal with the modified function execution rate and the non-modified function coverage. If the modified function space covered by a safe-mode algorithm is inadequate,

Algorithm 1 PDF-SA algorithm.

```
1  Input η, F_all, T_all
2  Output F_sel
3  Begin
4      for ∀f_j, 0 ≤ j ≤ |F_all|, where f_j ∈ F_all
5          η_fj = |T_fj| / |T_all| × 100%;
6          if η_fj ≥ η then F_sel = F_sel + f_j;
7      end-for
8      return F_sel;
9  End
```

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 ∀t_i, 0 ≤ i ≤| T_all |, where t_i ∈ T_all
6          F_ti = F_ti − F_tsel;
7          if F_ti = Ø then T_all = T_all − t_i;
8      end-for
9  Begin
10     while (T_all ≠ φ ∧ N_mod(F_ti) ≠ 0)
11         t_sel = arg max_{t_i ∈ T_all} N_mod(F_ti);
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
```

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

```
10     while (T_all ≠ φ ∧ W_mod(F_ti) ≠ 0)
11         t_sel = arg max_{t_i ∈ T_all} W_mod(F_ti);
```

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

```
1  Input F_mod, t_i, T_all, λ_cost, λ_cov
10     while (T_all ≠ φ ∧ W_mod(F_ti) ≠ 0)
11 t_sel = arg max_{t_i ∈ T_all} f_cv(F_ti, F_mod, λ_cost, λ_cov);
```

Algorithm 5 CW-CovMax algorithm.

```
1   Input τ, F_all, T_all
2   Output T_sel
3   Declare t_sel: //the selected test case.
4   UpdateT(): //the same updateT() as in Algorithm 2.
5   InitT():
6       for ∀t_i, 0 ≤ i ≤ |T_all|, where t_i ∈ T_all
7           if C(t_i) > τ then T_all = T_all − t_i;
8       end-for
9   Begin
10      InitT();
11      if T_all = Ø then return;
12      while (T_all ≠ Ø)
13          t_sel = arg max_{t_i ∈ T_all} ( |F_{t_i}| / C(t_i) );
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
```

this algorithm would offer a better way by covering a larger unmodified function space.

C. CW-CovMax Algorithm

If a regression testing is under a tight schedule but expects extensive function coverage, CW-CovMax algorithm would be the one that helps. Here tests with execution time larger than a specified cost are removed first. Each time a test case with a maximal function execution rate will be selected into the test suite.

Because this algorithm is not of a modification-aware approach, the reduced test suite obtained does not guarantee to cover all modified functions in which faults are likely to occur. This could compromise the fault detection capability.

IV. Experimental Results

A. Characteristics of the test-function mappings

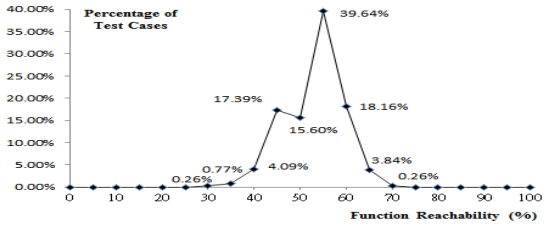
The experimental platform includes a personal computer of AMD Athlon 64 3800+ 2.41GHz processor, 3GB RAM, and Microsoft Windows XP Professional SP2. It is infeasible to spend about 36 test-bed-days to thoroughly execute a test suite with 2,320 tests. Therefore, the MPLS area in Cisco IOS is selected as the target platform, because the area has more test cases than others.

Table V shows 391 tests containing 23,308 functions. If all tests were executed, it would take about 7,746 minutes. Five releases were tested with 127 Distributed Defect Tracking Systems (DDTS) reports based on 302 modified functions. The reports addressed only 67 DDTS reports and 129 modified functions were reachable.

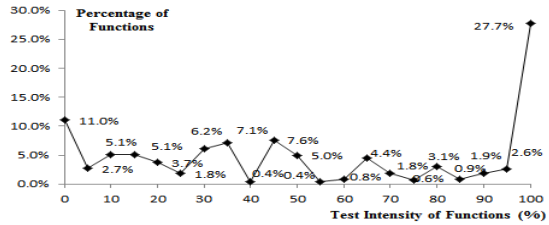
Figure 2(a) depicts function reachability of the 391 test cases. The function coverage of most test cases ranges from about 40% to 60%. Figure 2(b) shows test intensity of 23,308

Table V
Test information.

tests	functions	run time (min.)	releases	DDTS reports	modified functions	reachable DDTS reports	reachable modified functions
391	23308	7746	5	127	302	67	129



(a) The percentage of tests vs. function reachability(%).



(b) The percentage of functions vs. test intensity(%).

Figure 2. The test's function reachability and function's test intensity.

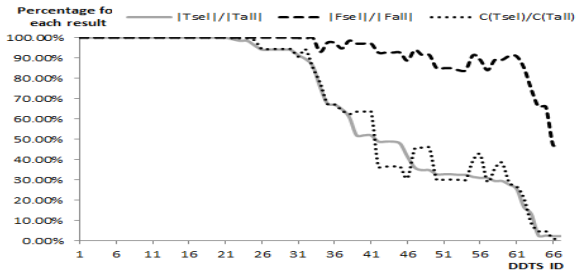


Figure 3. Cost percent of the safe-mode selection.

functions. Over 25% of functions were covered by each test case, and considered infrastructure functions.

The *safe-mode* approach calculates the cost of test case selection by 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 Figure 3, most cost percentages were higher than 30%. Only four of them were below. It showed that 94% (63/67) of DDTS reports did not provide a substantial cost reduction.

Now PDF-SA algorithm can examine the distribution of infrastructure functions by test intensity thresholds. Table VI shows different test intensity thresholds and parameters.

To realize how test intensity can affect the selection algorithms, we examined four test intensity thresholds — NA%(no threshold), 80%, 90%, and 100%. Speedup for the PDF-SA

Table VI
Summary of algorithms with thresholds and parameters.

Algorithm	Intensity Threshold	Other Parameters
PDF-SA	0, 5, …, 100	
CW-NumMin	NA, 80, 90, 100	
CW-CostMin	NA, 80, 90, 100	
CW-CostCov-B	NA, 80, 90, 100	λ_{cov} = {0, 0.1,…, 1}
CW-CovMax	NA, 80, 90, 100	τ = {500, 1000}

Table VII
CONVENTIONAL SELECTION VS. CW-NUMMIN AND CW-COSTMIN.

	$CW_{\eta 80}$ (%)	$CW_{\eta 90}$ (%)	$CW_{\eta 100}$ (%)	CW (%)
$\lvert T_{sel}\rvert / \lvert T_{all}\rvert$	2.56	2.56	2.56	2.56
$\lvert F_{sel}\rvert / \lvert F_{all}\rvert$	92.96	92.96	92.96	92.96
$\overline{N}_{mod}(F_{sel}) / \lvert F_{all}\rvert$	56.49	60.41	64.82	92.41
$C(T_{sel})/C(T_{all})$	2.32	2.32	2.32	2.32

(a) CW-NumMin.

	$CW_{\eta 80}$ (%)	$CW_{\eta 90}$ (%)	$CW_{\eta 100}$ (%)	CW (%)
$\lvert T_{sel}\rvert / \lvert T_{all}\rvert$	2.56	2.56	2.56	2.56
$\lvert F_{sel}\rvert / \lvert F_{all}\rvert$	90.44	90.44	90.44	90.44
$\overline{N}_{mod}(F_{sel}) / \lvert F_{all}\rvert$	53.98	57.89	62.30	89.89
$C(T_{sel})/C(T_{all})$	1.10	1.10	1.10	1.10

(b) CW-CostMin.

algorithm by various test intensity thresholds was also investigated. Run the CW-NumMin and CW-CostMin algorithm to calculate the number of tests and the amount of cost being reduced. The CW-CostCov-B algorithm verified the impact of λ_{cov} and λ_{cost}. The CW-CovMax algorithm used 500 and 1,000 minutes as restriction time when each execution time of tests ranges from 10 to 100 minutes.

B. Result Analysis

1) Test coverage of different test intensity thresholds: CW_{η} indicates a subset of registered functions with test intensities less than η, or $\eta_{f_j} < \eta$. Here η is a test intensity threshold. The CW stands for an entire set of registered functions without any threshold, or $\eta_{f_j} \leq 100\%$. On the other hand, $CW_{\eta 100}$ is a subset of functions with test intensities less than 100%, or $\eta_{f_j} < 100\%$, where functions with 100% test intensity were removed. Table VII(a) shows the results of performing the CW-NumMin algorithm at thresholds, $CW_{\eta 80}$, $CW_{\eta 90}$, $CW_{\eta 100}$, and CW.

2) CW-NumMin and CW-CostMin Algorithm—cost reduced to 2.32% and 1.1%, respectively: The outcome of performing the CW-NumMin algorithm were shown in Table VII(a). Under $CW_{\eta 100}$, selected only 2.56% tests but reached 92.96% function coverage and 64.82% function coverage for non-modified functions. The cost of the selected tests was significantly reduced to 2.3%. Similarly, Table VII(b) shows the results of exercising the CW-CostMin algorithm. Under $CW_{\eta 100}$, selected only 2.56% tests and reached 90.44% function coverage and 62.3% function coverage of non-modified function coverage. The cost of selected tests was further reduced to 1.10%. It showed that cost reduction by CW-CostMin algorithm is much better than that by CW-NumMin algorithm.

3) CW-CostCov-B Algorithm—small cost but higher non-modified function coverage: This algorithm employs both cost- and coverage-driven strategies. Factor λ_{cov} is for non-modified function coverage, and λ_{cost} for cost. If λ_{cov} is larger than λ_{cost}, the non-modified function coverage would take precedence. Performing this algorithm with $\lambda_{cov} = 0.0, 0.1, \ldots, 1.0$ and $\lambda_{cost} = 1.0, 0.9, \ldots, 0.0$. Figure 4 shows the curve of $\overline{N}_{mod}(F_{sel}) / \lvert F_{all}\rvert$, λ_{cov} 0.0 to 1.0. Test cases selected with $\lambda_{cov} = 0.0$, or $\lambda_{cost} = 1.0$, could reach 62.3% function coverage of *non-modified* functions. If λ_{cov} varies

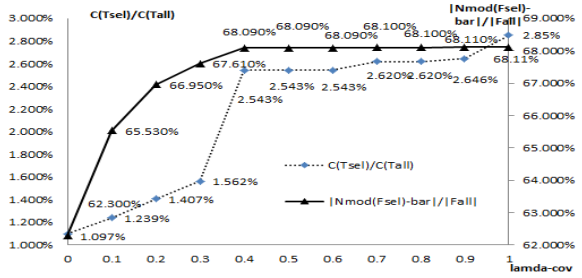


Figure 4. Cost and non-modified function coverage of CW-CostCov-B.

from 0.0 to 1.0, the function coverage of non-modified functions would range from 62% to 69%. The function coverage of non-modified functions was less than 69%, because the infrastructure functions held 27% function coverage. The cost of a selected test suite, however, has increased by 1% to 3%. Though the increase is small, it is still a significant increase for the non-modified function coverage.

Results in Table VIII emphasized on either the function coverage of non-modified functions or the cost with $\lambda_{cov} = 0.0$ ($\lambda_{cost} = 1.0$) and $\lambda_{cov} = 1.0$ ($\lambda_{cost} = 0.0$). Figure 4 appears that obtaining an extra 6% (68.5% - 62.5%) coverage for non-modified functions at $\lambda_{cov} = 1.0$ would cause a cost 2.6 times the cost at $\lambda_{cov} = 0.0$ (the increase in cost from 1.097% to 2.853%), and a total of tests 1.2 times the total at $\lambda_{cov} = 0.0$. Thus, $\lambda_{cov} = 0.0$ is a better choice.

4) CW-CovMax Algorithm—at high cost but with more coverage: The CW-CovMax algorithm employed both cost-driven policy and restriction times (τ) such as 500 or 1000 minutes. In Table IX(a), the function coverage was of 99.63% when $\tau = 500$, and 100% when $\tau = 1,000$. In Table IX(b), results was normalized at $\tau = 500$. The increase of function coverage was by 0.37% (100.000% - 99.630%) when $\tau = 1000$, compared to that when $\tau = 500$. The number of tests at $\tau = 1000$ increased to 1.442 times than that at $\tau = 500$, and the cost at $\tau = 1000$ 1.98 times than that at $\tau = 500$. The increase in the number of tests soon caused a higher cost with little improvement on function coverage.

5) PDF-SA—selection time is reduced to 10%∼70%: Figure 5 depicted curves of probability density function(pdf) and cumulative density function(cdf). The intensity in Figure 5 represented an aggregation percent of functions for every separate division. For example, the value at test intensity 20% means an aggregation value for the intensity varies from 20% to 25%, including 20%, but excluding 25%. Coverage at 100% and 0% of test intensity are higher than others. This implied a large portion of functions were covered by test cases with

Table VIII
CW-COSTCOV-B UNDER $CW_{\eta 100}$.

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

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

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

(a) Results. (b) Normalized values.

Table X
Function space reductions by PDF-SA.

	$CW_{\eta 80}$	$CW_{\eta 90}$	$CW_{\eta 100}$
Number of functions to be ignored	8427	7510	6463
Percent of function space reduced (%)	36.20	32.20	27.73

initial procedures and special features.

If $\eta = 100$, functions with test intensity 100% were considered infrastructure functions. Thus, under $CW_{\eta 100}$, 6,463 functions were infrastructure functions. If all infrastructure functions were removed, the function space would have reduced by 27.73%. Two intensity thresholds, 80% and 90%, were also shown in Table X. If under $CW_{\eta 80}$, 8,427 infrastructure functions were identified, and under $CW_{\eta 90}$ 7,510. This led to a decrease 36.20% and 32.20% in the function space.

In Table XI, the execution time of algorithms under $CW_{\eta 100}$, $CW_{\eta 90}$, or $CW_{\eta 80}$ were reduced to 10%~70%. Algorithms under $CW_{\eta 100}$ took times to perform various operations, such as union, intersection, and minus of set. Though removing infrastructure functions reduce the function space by only 27.73%, the runtime of algorithms were reduced to 48.46%, on the average. Choosing a smaller η allows a further reduction in execution time, but it is impractical if too many functions were considered infrastructure functions with a low intensity threshold.

V. Conclusions

Most problems we attacked here are test case selection problems. The modification-traversing approach, a substitute for the fault-revealing approach, applied for selecting test cases.

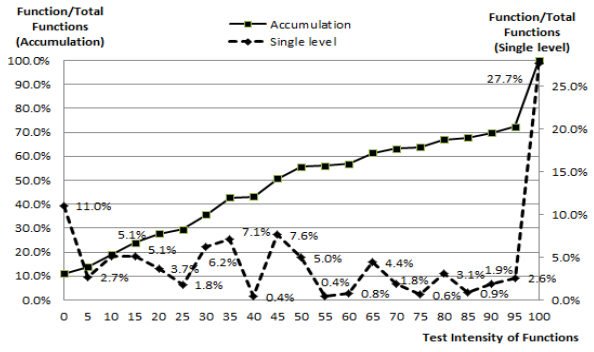


Figure 5. Test intensity of functions.

Table XI
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
Average	39.00	48.04	54.91	100.00

Algorithms proposed has reached the following achievements. First, the CW-NumMin algorithm could reduce the size of the test suite to 2.56%, and the cost to 2.32%. The CW-CostMin algorithm could decrease the size of test suite to 2.56%, and the cost to 1.10%. The CW-CostCov-B algorithm led to a better trade-off between cost-driven and coverage-driven strategies. Compared to the cost at $\lambda_{cov} = 1.0$, the cost at $\lambda_{cov} = 0.0$ has increased by 2.6 times (1.097% $\rightarrow$ 2.853%), and the size of the test suite has expanded by 1.2 times. However, the size of code at $\lambda_{cov} = 0.0$ increased only by 6% (96.25% - 90.44%). When restriction time τ was relaxed from 500 to 1000, the function coverage of CW-CovMax algorithm could increase by 0.37%, and the size of test suite by 1.44 times, the cost by 1.98 times.

Since precisely measuring the fault detection capability of a test suite for applications with large code is still infeasible, in future researchers could endeavor to "fault prediction" issues, instead of "fault detection", for the regression testing of large software system.

References

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

[2] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: a survey, Software Testing, Verification and Reliability-Published online in Wiley InterScience. doi:10.1002/stvr.430.
URL www.interscience.wiley.com

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

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

[5] X. Ma, Z. He, B. Sheng, C. Ye, A genetic algorithm for test-suite reduction, in: 2005 IEEE International Conference on Systems, Man and Cybernetics, Vol. 1, 2005, pp. 133–139.

[6] A. Malishevsky, G. Rothermel, S. Elbaum, Modeling the cost-benefits tradeoffs for regression testing techniques, in: Proceedings of the International Conference on Software Maintenance, 2002, pp. 204–213.

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

[8] G. Rothermel, M. Harrold, A framework for evaluating regression test selection techniques, in: Proceedings of the 16th international conference on Software engineering, IEEE Computer Society Press, 1994, pp. 201–210.

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

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

[11] Y.-D. Lin, C.-H. Chou, Y.-C. Lai, T.-Y. Huang, S. Chung, J.-T. Hung, F. C. Lin, Test coverage optimization for large code problems, Journal of Systems and Software 85 (2012) 16–27.