Science of Computer Programming 150 (2017) 1–25

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Empirically evaluating Greedy-based test suite reduction methods at different levels of test suite complexity

Chu-Ti Lin [a,*], Kai-Wei Tang [a], Jiun-Shiang Wang [a], Gregory M. Kapfhammer [b]

[a] Department of Computer Science and Information Engineering, National Chiayi University, Chiayi, Taiwan
[b] Department of Computer Science, Allegheny College, Meadville, PA, USA

ARTICLE INFO

Article history:
Received 29 August 2016
Received in revised form 10 May 2017
Accepted 14 May 2017
Available online 24 May 2017

Keywords:
Software testing
Regression testing
Test suite reduction
Test suite complexity

ABSTRACT

Test suite reduction is an important approach that decreases the cost of regression testing. A test suite reduction technique operates based on the relationship between the test cases in the regression test suite and the test requirements in the program under test. Thus, its effectiveness should be closely related to the complexity of a regression test suite – the product of the number of test cases and the number of test requirements. Our previous work has shown that cost-aware techniques (i.e., the test suite reduction techniques that aim to decrease the regression test suite's execution cost) generally outperform the others in terms of decreasing the cost of running the regression test suite. However, the previous empirical studies that evaluated cost-aware techniques did not take into account test suite complexity. That is, prior experiments do not reveal if the cost-aware techniques scale and work effectively on test suites with more test cases and more test requirements. This means that prior experiments do not appropriately shed light on how well test suite reduction methods work with large programs or test suites. Therefore, this paper focuses on the Greedy-based techniques and empirically evaluates the additional Greedy and two cost-aware Greedy techniques – at different levels of test suite complexity – from various standpoints including the cost taken to run the regression test suite, the time taken to reduce the test suites, the total regression testing costs, the fault detection capability, the fault detection efficiency, and the common rates of the representative sets. To the best of our knowledge, none of the previous empirical studies classify a considerable number of test suites according to their complexity. Nor do any prior experiments evaluate the test suite reduction techniques, in terms of the aforementioned criteria, at different levels of test suite complexity. This paper represents the first such attempt to carry out this important task. Based on the empirical results, we confirm the strengths and weaknesses of the cost-aware techniques and develop insights into how the cost-aware techniques' effectiveness varies as the test suite complexity increases.

1. Introduction

In many software development processes, the software developers may incrementally deliver or refine the functionality of a software system to satisfy the customers' needs. Regression testing [1,2] is an important activity that helps to maintain the quality of the modified software system. Each time the software developers change the system, they may also add some

* Corresponding author.
E-mail address: chutilin@mail.ncyu.edu.tw (C.-T. Lin).

http://dx.doi.org/10.1016/j.scico.2017.05.004
0167-6423/

new test cases to the regression test suite. As a result, the regression test suite's size may increase gradually, thus often causing some test requirements to be exercised by more than one test case. Thus, decreasing the execution cost of regression testing is important for practical software development processes, particularly when considerable amounts of testing have to be finished in a tight build schedule. We consider the IBM middleware product that was reported in [3] as an illustrative example of adopting a time constraint during testing. In this system, it takes more than four hours to execute all of the test cases in the test suite, but the time allocated to perform the typical smoke test scenario after each code commitment is only about an hour or less. This example is a clear case in point of forcing the test team to reduce the size and execution time of the test suite.

To decrease the execution cost of regression testing, software developers can apply test suite reduction techniques, like the automated methods evaluated in this paper, to produce a reduced test suite that still shows good test effectiveness. The test suite reduction problem can be defined as follows [4–7].

Given:

- $T = \{t_1, t_2, \ldots, t_n\}$ representing a set of the test cases in the original test suite, where $n = |T|$;
- $R = \{r_1, r_2, \ldots, r_m\}$ representing a set of test requirements in which each test requirement must be satisfied by at least one of the test cases in T, where $m = |R|$;
- $S = \{(t, r) \mid$ the test case t satisfies the test requirement $r,\ t \in T$ and $r \in R\}$ representing the binary relation between T and R.

Objective:

- Find a subset of the test suite T, denoted by a representative set RS, to satisfy all of the test requirements satisfied by T.

There may be many factors that could influence the complexity of a test suite, such as the number of test cases in the test suite, the number of test requirements in the program under test, the execution path of each test case, the type of code coverage that is considered by each test case, the third-party software integration, the operations involved in running each test case, the similarity between test cases in a test suite, and the effect of input/output operations (i.e., the effect caused by transferring data between a computer and a peripheral device). Yet, according to the aforementioned definition, the test suite reduction techniques operate based on the relationship between the test cases in the regression test suite and the test requirements in the program under test (i.e., the binary relation between T and R). Thus, this paper focuses the complexity of a test suite on two factors: the number of test cases in the test suite and the number of test requirements in the program under test. To the best of our knowledge, only Zhong et al. [8] defined the complexity of a test suite and their definition also focused on these two factors. Similar to Zhong et al.'s definition [8], we define the complexity of a test suite T as

$$Complexity(T) = m \times n, \tag{1}$$

where n is the number of test cases in T, and m is the number of test requirements satisfied by T. In addition to test suite reduction techniques, other regression testing techniques (e.g., test case prioritization techniques [9] and test case selection techniques [10]) also operate based on the relationship between T and R. The effectiveness of these techniques should also be closely related to the complexity of a regression test suite.

Because real programs are large and complex by nature [11,12], the complexity of real test suites is unavoidably high. Thus, it is important to evaluate these techniques' capability to scale with the complexity of the test suites. In the literature, several empirical studies (e.g., [6,13–19]) classify the test suites according to their size (i.e., the number of test cases) and, at different levels of test suite size, evaluate the regression testing techniques' effectiveness. Most of the aforementioned empirical studies [6,13,15–19] focus on the test suite reduction problem and indicate that the results of comparisons may not be consistent at different levels of test suite size. Yet, in these prior empirical studies, test suites are classified according to size, not complexity. Although [8] evaluates the test suite reduction techniques' effectiveness using several test suites of different complexities, only a few test suites are used in the empirical study. More specifically, the number of test suites used for each subject program in [8] ranges from only 5 to 12.

Moreover, although test cases' execution time is generally one of the critical factors in software testing [20], the empirical studies in [6,8,13,15–19] ignore the effect on the regression testing process caused by the individual execution costs of the test cases. To the best of our knowledge, most of the existing reduction techniques, including the Additional Greedy algorithm, ignore the significant differences in the execution costs of the tests. Actually, the difference in the execution cost of the tests is usually significant and, in some situations, it may take a lot of execution time to run a test suite consisting of a few long-running test cases [5]. In other words, most of the existing reduction techniques aim to decrease the regression test suite's size instead of the regression test suite's execution cost. Thus, Smith and Kapfhammer [21] and Lin et al. [5, 22] presented cost-aware test suite reduction techniques that aim to decrease the regression test suite's execution cost. In addition to the work described in [5,21,22], several test suite reduction techniques [23–25] also take into account the execution cost of test cases. However, these techniques try to strike a balance between the execution cost and the other testing objectives; their focused studies are not similar to those described in [5,21,22]. Thus, in order to control the scope

and focus of both this exposition and the empirical study, this paper only focuses on the techniques that aim to decrease the regression test suite's size or execution cost and excludes the techniques presented in [23–25].

Actually, in addition to the time taken to run the test cases, the total regression testing costs also include the time taken to produce the representative set of test cases. According to our initial observations in [22], in comparison with the others, the cost-aware techniques generally achieve lower execution costs for the regression test suite but take more time to perform the reduction tasks. However, the comparisons were conducted using the generated test suites that were not classified according to test suites' complexity. Thus, the current observations from [5,21] and [22] neither evaluate the cost-aware techniques' scalability (i.e., their capability to scale with the complexity of the original test suites) nor confirm if their effectiveness is generally satisfactory at different levels of test suites' complexity. In other words, prior experiments do not reveal if the cost-aware techniques scale and work effectively on test suites with more test cases and more test requirements. This means that prior experiments do not appropriately shed light on how well test suite reduction methods work with large programs and test suites.

In addition to scalability and cost reduction capability, the total regression testing cost, the fault detection capability, the fault detection efficiency (i.e., the number of fault detections per unit of cost), and the feasibility of conducting test suite reduction are also attractive considerations for realistically evaluating a test suite reduction technique. The effects on these measures as the complexity of the test suites grows are also worth discussing. Thus, this paper's experiment generates many test suites based on ten subject programs and the test cases collected from the Software-artifact Infrastructure Repository (SIR) [26]; next it divides the generated original test suites into five classes according to their complexity.[1] This paper also presents seven research questions. It then replies to these questions by empirically analyzing the aforementioned measures associated with the two cost-aware algorithms [5,21,22] and the Additional Greedy algorithm [27], always considering each level of test suite complexity.

To the best of our knowledge, none of the previous studies have looked at test suite complexity in the way that we did and followed our strategy for dividing the generated test suites. According to the empirical results, we find that (1) the total regression testing costs achieved by the two cost-aware algorithms are generally lower than those produced by the Additional Greedy algorithm; (2) the representative sets produced by the cost-aware technique presented in [22] generally have better fault detection efficiency (i.e., the greatest number of faults revealed per unit of regression test cost) than the others; (3) scalability should be a relatively minor consideration for choosing test suite reduction techniques; (4) considering the techniques compared in our study, the rankings in terms of various criteria do not significantly change as the level of test suite complexity varies; (5) the benefits of adopting the two cost-aware techniques increase as the test suite complexity grows. Please note that, without investigating test suite complexity, the prior empirical results might be misleading, and these aforementioned findings were not reported in the literature. Before this paper, it was an open question as to whether or not existing methods could reduce complex test suites. This paper provides the first evidence to suggest that they can indeed be efficiently and effectively applied to complex test suites. As such, our empirical study is different from and more comprehensive than those conducted in the past. In addition, our empirical results provide insights into how software developers should choose a suitable test suite reduction technique according to the situations that they are facing.

The reminder of this paper is organized as follows. Section 2 reviews three test suite reduction techniques that are empirically compared in this paper's study. Section 3 describes the experimental setup and the details of the data that are used in the empirical study. Also, this section presents and replies to seven research questions. Section 4 presents the threats to the validity of our empirical studies. Finally, Section 5 furnishes some concluding remarks and our future work.

2. Regression testing background

This section will briefly review the three test suite reduction techniques that are empirically compared in this paper.

2.1. Additional Greedy algorithm

The Additional Greedy algorithm [27] is a well-known method for addressing the set-covering problem. In the remainder of this paper, it is called Greedy for simplification. This algorithm has also been used to find the near-optimal solution to the test suite reduction problem [28,29]. It repetitively moves the test case that covers the most unsatisfied test requirements from the original test suite set T to the representative set RS until all of the test requirements are satisfied. In other words, Greedy evaluates the test cases using

$$Coverage(t) = |R_t|, \tag{2}$$

where R_t represents the set of unsatisfied test requirements that are covered by t, and attempts to repetitively choose the test case with the maximum $Coverage(t)$ in order to obtain the near-optimal solution from the locally optimal solutions. The worst-case time complexity of Greedy is $O(m \cdot n \cdot \min(m, n))$. Fig. 1 shows the pseudo code of Greedy.

[1] The reasons for choosing five levels will be described in Section 3.1.

```
1  algorithm Additional Greedy
2  input  T: the set of test cases for the program
3         R: the set of test requirements in the program
4         S: the relation S={(t, r)| t satisfies r, t∈T, and r∈R}
5  output RS: a representative set of T
6  Begin
7         RS = ϕ;
8         while (R ≠ ϕ)
9         {
10            t = the test case with the maximum Coverage(t);
11            RS = RS ∪ {t};
12            T = T − {t};
13            R = R − {r| (t, r)∈S};
14        }
15        return RS;
16 End
```

Fig. 1. The Additional Greedy algorithm.

2.2. Cost-aware Greedy algorithm associated with the Ratio metric

In comparison with the size of a test suite, the cost taken to run a test suite may be a more important metric to judge the efficiency of regression testing. Thus, instead of focusing on the increase in code coverage, Smith and Kapfhammer [21] evaluated the test cases using

$$Ratio(t) = \frac{Coverage(t)}{Cost(t)}, \tag{3}$$

where $Cost(t)$ represents the cost of running the test case t and $Coverage(t)$ represents the number of test requirements that are satisfied by t. A high value of $Ratio(t)$ indicates that t is expected to be a cost-effective test case; in contrast, a low value of $Ratio(t)$ may imply that t is expected to be less desirable.

Based on the framework of Greedy, Smith and Kapfhammer presented a cost-aware technique, called $\text{Greedy}_{\text{Ratio}}$ hereafter, that repeatedly chooses the test case with the maximum $Ratio(t)$ until all of the test requirements are satisfied. That is, $\text{Greedy}_{\text{Ratio}}$ chooses the test case that achieves the most coverage increase per unit of cost consumption. The worst-case time complexity of $\text{Greedy}_{\text{Ratio}}$ is $O(m \cdot n \cdot \min(m, n))$. Replacing $Coverage(t)$ in Fig. 1 by $Ratio(t)$, we can obtain the pseudo code of $\text{Greedy}_{\text{Ratio}}$.

2.3. Cost-aware Greedy algorithm associated with the EIrreplaceability metric

If the test requirement r that is satisfied by the test case t can also be satisfied by many other test cases, it is highly possible that r can still be satisfied although t is excluded from the representative set. In this case, Lin et al. [5,22] posited that t has a higher "replaceability" with respect to r. According to the empirical results in our previous work [5,22], Lin et al. had the following two observations:

1. A representative set may not have the lowest execution cost if it contains test cases with high replaceability.
2. Because the metric Ratio only takes into account the code coverage and execution cost of each test case, $\text{Greedy}_{\text{Ratio}}$ may include many test cases with high replaceability in the representative set.

In contrast to replaceability, a high value of irreplaceability means that it is not easy to find other test cases to replace t. Additionally, a test can be called the essential test case for a test requirement if this test requirement only can be satisfied by it [29]. Lin et al. [22] also considered the effect of essential test cases during test suite reduction. Given that r_s represents the s-th test requirement in R, they evaluated the irreplaceability of a test case using

$$EIrreplaceability(t) = \begin{cases} \infty, & \exists s,\ 1 \le s \le m,\ r_s \text{ can be satisfied by } t \text{ only;} \\ \frac{\sum_{s=1}^{m} Contribution(t, r_s)}{Cost(t)}, & \text{otherwise,} \end{cases} \tag{4}$$

where

$$Contribution(t, r_s) = \begin{cases} 0, & \text{if } t \text{ cannot satisfy } r_s; \\ \frac{1}{\text{the number of test cases that satisfy } r_s}, & \text{if } t \text{ satisfies } r_s. \end{cases} \tag{5}$$

Lin et al. [22] presented a cost-aware test suite reduction technique, called $\text{Greedy}_{\text{EIrreplaceability}}$, that repeatedly chooses the test case with the maximum $EIrreplaceability(t)$, as defined in Eq. (4), until all of the test requirements are satisfied. The

Table 1
The worst-case time complexities for the algorithms that are compared in this study.

Algorithms	Worst-case time complexity	Description of the factors in the expressions of time complexity
Greedy	$O(m \times n \times \min(m, n))$	• n: the number of test cases in the original test suite;
$\text{Greedy}_{\text{Ratio}}$	$O(m \times n \times \min(m, n))$	• m: the number of test requirements in the program under test;
$\text{Greedy}_{\text{EIrreplaceability}}$	$O(m \times n \times \min(m, n) \times k)$	• k: the maximum number of test requirements that can be satisfied by a single test case.

worst-case time complexity of $\text{Greedy}_{\text{EIrreplaceability}}$ is $O(m \cdot n \cdot \min(m, n) \cdot k)$, where k is the maximum number of test requirements that can be satisfied by a single test case. It should be noted that the value of $EIrreplaceability(t)$ for an essential test case is infinite. Similar to the pseudo code of $\text{Greedy}_{\text{Ratio}}$, if we replace $Coverage(t)$ in Fig. 1 by $EIrreplaceability(t)$, we can obtain the pseudo code of $\text{Greedy}_{\text{EIrreplaceability}}$.

2.4. Preliminary comparisons on the time complexity of the algorithms

Table 1 summarizes the worst-case time complexities of the algorithms that were reviewed in Sections 2.1 through 2.3. According to the table, we have two observations:

(1) Both the number of test cases in the original test suite and the number of test requirements in the program under test are common factors in the expressions of the three algorithms' time complexities.
(2) The worst-case time complexities of Greedy and $\text{Greedy}_{\text{Ratio}}$ are identical and they are lower than that of $\text{Greedy}_{\text{EIrreplaceability}}$.

Observation (1) confirms that it is suitable for this paper to focus its definition of test suite complexity on these two factors. Additionally, Observation (2) is consistent with our initial observations in [22]. That is, in comparison with Greedy, the cost-aware Greedy algorithms, especially $\text{Greedy}_{\text{EIrreplaceability}}$, generally achieve lower regression test suite execution cost but take more time to perform the tasks. Because the empirical studies in [22] do not take in account the complexities of the adopted test suites, these two observations confirm that it is worthwhile for this paper to evaluate the effectiveness of Greedy, $\text{Greedy}_{\text{Ratio}}$, and $\text{Greedy}_{\text{EIrreplaceability}}$ at different levels of test suite complexity.

3. Empirical studies

As mentioned in Section 1, the comparisons in the previous empirical studies are not sufficiently diverse and do not evaluate the effectiveness of the cost-aware techniques in terms of various criteria at different levels of test suite complexity. We judge that it is important and interesting to analyze whether the cost-aware techniques generally perform well at different levels of test suite complexity. These results may help practicing software developers understand if the cost-aware techniques scale and work effectively on test suites with more test cases or more test requirements. Thus, this section answers seven research questions by comparing Greedy, $\text{Greedy}_{\text{Ratio}}$, and $\text{Greedy}_{\text{EIrreplaceability}}$, at different levels of test suite complexity, from various standpoints. Section 3.1 briefly describes the SIR subject programs, the benchmarks that we adopted to compare the three test suite reduction techniques, and explains the experimental setup. Section 3.2 describes seven research questions and the metrics used for the comparisons. Section 3.3 answers the seven research questions based on the experimental data. From the standpoint of the criteria adopted to evaluate the techniques and the way used to classify test suites, Section 3.4 compares our empirical studies with those described in the literature.

3.1. Benchmark description and experimental setup

Our empirical studies focused on the SIR subject programs, including the suite of the seven Siemens programs [6,30], the space program [31], the gzip program [32], and the ant program [33–35]. The seven Siemens programs were assembled at Siemens Corporate Research for experiments with control-flow and data-flow test adequacy criteria [30] and each program is associated with several single fault versions. That is, each faulty version was seeded with only one fault. Considering the six different categories of seeded faults, please refer to [36] for more details; the space program was developed for the European Space Agency [31] and each of the associated faulty versions contains a single fault that was revealed during the program's development; the gzip program is the real-world Unix utility that was obtained from the GNU site [32, 37]. The gzip program has five sequential, previously-released versions that can be used for our empirical studies. Among those versions, we choose the version that includes the most test requirements (i.e., the 5th version); the ant program is a Java-based build tool that was supplied by the Apache project [33]. The ant program has eight sequential versions. In a similar manner, we choose the 7th version of ant because this version includes more test requirements than the others. Each of the selected sequential versions of the gzip and ant programs contains more than one seeded fault. In the remainder of this paper, we do not highlight the sequential versions of gzip and ant that are adopted in our empirical studies for simplification. Table 2 summarizes the detailed descriptions of the adopted SIR programs and test suites.

In order to determine the execution time taken by running each test case, we individually executed the test cases 1000 times on both Windows (Windows 7 SP1) and Linux (Ubuntu 10.10) workstations with Intel Core Duo processors

Table 2
The SIR subject programs and test suites.

Programs	*loc*[1]	$\|R\|$[2]	$\|A\|$[3]	Num. of faults[4]	Description
printtokens	402	140	4,130	7	A lexical analyzer
printtokens2	483	138	4,115	10	A lexical analyzer
replace	516	126	5,542	32	A program used for pattern replacement
schedule	299	46	2,650	9	A program used for priority scheduler
schedule2	297	72	2,710	10	A program used for priority scheduler
tcas	148	16	1,608	41	A program used for altitude separation
totinfo	346	44	1,052	23	A program used for information measure
space	6,218	1,067	13,585	38	An array definition language interpreter developed for the European Space Agency
gzip	7,997	1,923	214	9	A UNIX utility program that was designed for compression
ant	38,491	7,282	674	6	A Java-based build tool that was supplied by the Apache project

[1] *loc*: The lines of code in the subject program.
[2] $|R|$: The number of test requirements (i.e., the number of branches) in the subject program.
[3] $|A|$: The size of pool of tests designed for the subject program. It should be noted that the test cases of ant are method-level.
[4] Considering the Siemens programs and the space program, each program has several faulty versions and each of the associated faulty versions contains a single fault that was seeded for experiments or was revealed during the program's development. On the other hand, each of the selected versions for the gzip and ant programs contains more than one seeded fault, but some of the faults in gzip cannot be revealed by the test cases that were provided by SIR.

and 2 GB memory, and took the average across the runs for both workstations. Additionally, we also recorded the test requirements that are satisfied by each test case. Notice that the test requirements adopted in this empirical study are branches because branch coverage [38] is a common criterion used in many code coverage tools (e.g., EvoSuite that was cited in [39], Cobertura and CodeCover that were cited in [40], JaCoCo that was cited in [41], and EclEmma that was cited in [42]) and relevant empirical studies (e.g., [4,6,14,22]). Thus, we judge that it is a good option for this paper's experiments.

The experiment followed an empirical setup that is similar to the one adopted in [4,6] and [43]. Fig. 2 furnishes the pseudo code used to generate the experimental test suite for each SIR subject program. The detailed steps are described as follows:

Step 1. Randomly generate an integer v, $1 \leq v \leq 0.01 \times loc$, where *loc* represents the number of lines of code in the subject program;
Step 2. Generate the original test suite T by randomly choosing v test cases from the test pool for each subject program;
Step 3. Randomly pick one more test case that can satisfy at least one unsatisfied test requirement and include that test case in T if the tests in T cannot satisfy all of the test requirements;
Step 4. Repeat Step 3 until all test requirements are satisfied.

In Fig. 2, the operation shown on Line 9 is implemented to realize Step 1, those shown on Lines 10–16 are implemented to accomplish Step 2, and those shown on Lines 17–26 are implemented to achieve Steps 3–4. Please notice that selecting an arbitrary v from the range in Steps 1–2 is done to generate test suites of different sizes so that the diversities of the studied situations will be more satisfactory. On the other hand, Steps 3–4 are given in order to satisfy the branch adequacy criterion (i.e., each branch in the subject program will be executed by at least one test case in the generated test suite).

The experiment generates 1000 test suites for each subject program by following the aforementioned steps and divides the 1000 generated test suites into five classes (i.e., L_1, L_2, L_3, L_4, and L_5) according to the test suites' complexity. The reasons for choosing five levels include: (1) If we choose too few levels (i.e., 2 to 4), it is difficult to observe the trend from various analyses; (2) If we choose more levels, it will be difficult to clearly present the data in the figures (i.e., there will be too many bars in the figures). Additionally, because the interval of each level should be equal, choosing too many levels will make it difficult to ensure that the numbers of test cases in all of the levels are close. If there are significant differences in the number of test cases among different levels, the comparisons on the averages of different levels may be less convincing. Choosing five levels allowed us to effectively address these considerations.[2]

For each subject program, we implemented the pseudo code described in Fig. 3 to carry out the classification of the 1000 generated test cases. The operation shown on Line 18 of Fig. 3 is given to ensure that *MaxComp* is a multiple of a round integer so that the representation of complexity level looks concise. The value for the parameter *Round* is roughly defined according to the scale of *MaxComp*. In our experiments, we set *Round* to 100 (i.e., 0.1 K) for the Siemens programs, 1,000,000 (i.e., 1 M) for space, 10,000 (i.e., 10 K) for gzip, and 200,000 (i.e., 0.2 M) for ant. Notice that the complexity level L_1 will be empty if the complexities for all of the generated test suites are higher than *MaxComp*/5. Thus, the operations shown on

[2] We did not make the complexity adhere to a log scale because we cannot easily divide the complexity in terms of log scale into several levels of equal intervals. For example, by not using log scale, we can easily divide the complexity ranging from 1 to 100000 into five levels of equal intervals, i.e., $[1, 20000]$, $[20001, 40000]$, $[40001, 60000]$, $[60001, 80000]$, and $[80001, 100000]$. Although we cannot ensure that the numbers of test cases in different levels are equal, there are still a lot of test cases in each level so that the measures for each level are convincing. However, if the log-scale complexity ranges from 0 to 5 (i.e., ranges from $\log_{10} 1$ to $\log_{10} 100000$), while we can easily divide it into five levels of equal intervals, the number of test cases in different levels are significantly different and thus we cannot ensure that there are sufficient test cases in each level. As such, we did not formulate the complexity level in terms of a log scale.

```
1  algorithm TestSuiteGenerator
2  input   A: the pool of all available test cases for the program
3          R: the set of test requirements in the program
4          S_A: the relation S_A={(t, r)| t satisfies r, t∈A, and r∈R}
5          loc: the line of code
6  output T: a test suite for the experiment
7  begin
8          T = ϕ;
9          v = an arbitrary integer, 1 ≤ v ≤ 0.01 × loc;
10         for ( s = 1 to v)
11         {
12             randomly choose a test case t ∈ A;
13             A = A − {t};
14             T = T ∪ {t};
15             R = R − {r | (t, r)∈S_A};
16         }
17         while (R ≠ ϕ)
18         {
19             t = an arbitrary test case in A;
20             if ( ∃r∈R such that (t, r)∈S_A )
21             {
22                 A = A − {t};
23                 T = T ∪ {t};
24                 R = R − {r | (t, r)∈S_A};
25             }
26         }
27         return T;
28 end
```

Fig. 2. Pseudo code for the test suite generator.

Lines 18–21, 24, 26, 28, and 30 are given to avoid this situation. Table 3 shows the statistics extracted from the generated original test suites at different levels of complexity. For the original test suites at each complexity level, we reduced the test suites by using the three techniques that are compared in this study and took the averages of the collected information.

Please note that this paper exclusively focused on the programs and test pools collected from SIR due to the following three reasons. First, they are frequently chosen benchmarks for evaluating test suite reduction methods. For example, the empirical studies that are described in [6,14,33–36,44–48] also adopted the SIR programs. This will make our experimental results broadly comparable to those in prior work. Second, they clearly exhibit variability in the cost of the test cases according to the analyses in [22]. This is an important feature for evaluating the effectiveness of the test suite reduction techniques that take into account the differences in execution time among test cases. Thus, the selected SIR programs and the associated test pools should be a good starting point for our empirical studies. Section 5 notes that, in future work, we will conduct experiments with additional programs and test suites.

3.2. Research questions

Our empirical studies are designed to answer the following seven research questions.

- *Research Question 1* (*RQ1*) – *What is the effect on the cost reduction capability of the three techniques as the level of test suite complexity varies?*

The cost taken to run the representative set is a valid measure for evaluating the cost reduction capability of a test suite reduction technique. For *RS*, we calculate the execution cost of the representative set (ECRS) by using [49]

$$ECRS(RS) = \sum_{t \in RS} Cost(t), \tag{6}$$

where $Cost(t)$ represents the time taken to run the test case t. The lower the ECRS value is, the more satisfactory the cost reduction capability is. ECRS should be one of the key criteria for evaluating test suite reduction techniques because regression testing generally has to be finished in a tight build schedule. We will answer this research question by comparing the three techniques in terms of ECRS at different levels of test suite complexity.

- *Research Question 2* (*RQ2*) — *Are the three techniques' scalability* (i.e., *the capability to scale with the complexity of the original test suites*) *close to each other?*

```
1  algorithm TestSuiteClassification
2  input   T_i (for 1 ≤ i ≤ 1000): the 1000 test suites that are generated using TestSuiteGenerator shown in Fig. 2
3          R: the set of test requirements in the program
4          Round: the number is given in order to ensure that the representation of complexity level looks concise
5  output L_l (for 1 ≤ l ≤ 5): the five sets of test suites for different test suite complexity levels
6  begin
7          for ( l = 1 to 5 )
8              L_l = ϕ;
9          MaxComp = 0; // MaxComp represents the maximum of the complexities for all test suites to be classified.
10         MinComp = ∞; // MinComp represents the minimum of the complexities for all test suites to be classified.
11         for ( i = 1 to 1000 )
12         {
13             if ( |T_i| × |R| > MaxComp )
14                 MaxComp = |T_i| × |R|;
15             if ( |T_i| × |R| < MinComp )
16                 MinComp = |T_i| × |R|;
17         }
18         MaxComp = ⌈(MaxComp) / Round⌉ × Round ;
19         counter = 5;
20         while not ( (MaxComp − ( MaxComp/counter)×5) ≤ MinComp < (MaxComp − ( MaxComp/counter)×4) )
21             counter ++;
22         for ( i = 1 to 1000 )
23         {
24             if ( |T_i| × |R| ≤ MaxComp − (MaxComp / counter) × 4 )
25                 L_1 = L_1 ∪ {T_i};
26             else if ( |T_i| × |R| ≤ MaxComp − (MaxComp / counter) × 3 )
27                 L_2 = L_2 ∪ {T_i};
28             else if ( |T_i| × |R| ≤ MaxComp − (MaxComp / counter) × 2 )
29                 L_3 = L_3 ∪ {T_i};
30             else if ( |T_i| × |R| ≤ MaxComp − (MaxComp / counter) × 1 )
31                 L_4 = L_4 ∪ {T_i};
32             else
33                 L_5 = L_5 ∪ {T_i};
34         }
35         return L_1, L_2, L_3, L_4, L_5;
36 End
```

Fig. 3. Pseudo code for dividing the generated test suites into five classes according to the test suites' complexity.

Scalability is considered as one of the important problems for software engineering research and development [3,50,51]. This applies to test suite reduction work just as much as it does to other aspects of software engineering, such as software requirement analysis [52], software visualization [53], software maintenance [54], reverse engineering [55], selective mutative testing [56], and static analysis [57]. Scalability analysis with respect to test suite reduction may be intuitively connected with the time taken to perform a task. Thus, in addition to the cost taken to run the representative set, the time taken to produce the representative set (TPRS) should also be an important measure for evaluating reduction techniques. In order to answer this research question, we let *TPRS*(*RS*) represent the time taken to produce the representative set *RS* (i.e., the time taken to perform a test suite reduction technique). We will evaluate the scalability of reduction techniques by analyzing whether their values of *TPRS*(*RS*) are still acceptable as the levels of test suite complexity grow.

- *Research Question 3* (*RQ3*) – *What is the effect on the total regression testing cost achieved by the three techniques as the level of test suite complexity varies?*

As mentioned in Section 1, the total regression testing costs (TRTC) include ECRS and TPRS. That is, the TRTC of the representative set *RS* is defined as

$$TRTC(RS) = ECRS(RS) + TPRS(RS). \tag{7}$$

We will reply to this research question by analyzing and comparing the TRTC values of the three techniques at different levels of test suite complexity.

- *Research Question 4* (*RQ4*) – *Can the improvements in the time taken to run the representative sets of test cases achieved by the three techniques still compensate for the penalties of producing the representative sets as the levels of test suite complexity grow?*

Table 3
The statistics for the generated original test suites at different levels of test suite complexity.[1]

printtokens					printtokens2				
Complexity level	Suite size[2]	Suite cost[3]	Faults[4]	Faults/Cost[5]	Complexity level	Suite size[2]	Suite cost[3]	Faults[4]	Faults/Cost[5]
0–5.8 K	26.21	1395909	3.44	2.63×10^{-6}	0–5.2 K	22.38	899554	7.91	1.02×10^{-5}
5.8 K–11.6 K	62.84	3248158	4.30	1.36×10^{-6}	5.2 K–10.4 K	56.50	2275130	8.75	4.04×10^{-6}
11.6 K–17.4 K	103.28	5336796	4.99	9.44×10^{-7}	10.4 K–15.6 K	95.65	3851035	9.33	2.45×10^{-6}
17.4 K–23.2 K	144.49	7443717	5.12	6.91×10^{-7}	15.6 K–20.8 K	130.71	5259621	9.50	1.82×10^{-6}
23.2 K–29.0 K	178.57	9165315	5.71	6.23×10^{-7}	20.8 K–26.0 K	167.53	6776432	9.65	1.43×10^{-6}

replace					schedule				
Complexity level	Suite size[2]	Suite cost[3]	Faults[4]	Faults/Cost[5]	Complexity level	Suite size[2]	Suite cost[3]	Faults[4]	Faults/Cost[5]
0–7.2 K	34.29	2123686	12.64	6.50×10^{-6}	0–1.4 K	17.12	889994	3.93	5.10×10^{-6}
7.2 K–14.4 K	85.48	5447269	18.53	3.48×10^{-6}	1.4 K–2.8 K	46.15	2447603	5.40	2.25×10^{-6}
14.4 K–21.6 K	140.29	8931111	21.64	2.46×10^{-6}	2.8 K–4.2 K	75.75	4022230	6.05	1.52×10^{-6}
21.6 K–28.8 K	197.94	12673558	22.73	1.80×10^{-6}	4.2 K–5.6 K	106.93	5663437	6.43	1.15×10^{-6}
28.8 K–36.0 K	240.84	15143388	23.61	1.56×10^{-6}	5.6 K–7.0 K	135.63	7238972	7.05	9.77×10^{-7}

schedule2					tcas				
Complexity level	Suite size[2]	Suite cost[3]	Faults[4]	Faults/Cost[5]	Complexity level	Suite size[2]	Suite cost[3]	Faults[4]	Faults/Cost[5]
0–2.1 K	18.42	1156227	2.62	2.61×10^{-6}	0–300	10.79	423732	9.74	2.43×10^{-5}
2.1 K–4.2 K	44.61	2766730	4.23	1.55×10^{-6}	300–600	27.30	1075408	15.60	1.48×10^{-5}
4.2 K–6.3 K	73.93	4571786	5.19	1.15×10^{-6}	600–900	47.67	1865553	21.15	1.15×10^{-5}
6.3 K–8.4 K	102.04	6327875	6.07	9.68×10^{-7}	900–1.2 K	65.06	2559134	23.66	9.29×10^{-6}
8.4 K–10.5 K	130.51	8060580	6.35	7.89×10^{-7}	1.2 K–1.5 K	77.85	3073643	25.92	8.45×10^{-6}

totinfo					space				
Complexity level	Suite size[2]	Suite cost[3]	Faults[4]	Faults/Cost[5]	Complexity level	Suite size[2]	Suite cost[3]	Faults[4]	Faults/Cost[5]
0–1.6 K	10.79	970630	9.74	1.78×10^{-5}	0–1.0 M	527.09	3.28×10^{8}	31.73	1.32×10^{-7}
1.6 K–3.2 K	27.30	2634726	15.60	7.40×10^{-6}	1.0 M–2.0 M	1380.88	8.88×10^{8}	33.98	3.98×10^{-8}
3.2 K–4.8 K	47.67	4413865	21.15	4.63×10^{-6}	2.0 M–3.0 M	2374.20	1.53×10^{9}	34.65	2.30×10^{-8}
4.8 K–6.4 K	65.06	6240201	23.66	3.41×10^{-6}	3.0 M–4.0 M	3284.87	2.10×10^{9}	34.77	1.66×10^{-8}
6.4 K–8.0 K	77.85	7721254	25.92	2.74×10^{-6}	4.0 M–5.0 M	4143.87	2.66×10^{9}	34.94	1.32×10^{-8}

gzip					ant				
Complexity level	Suite size[2]	Suite cost[3]	Faults[4]	Faults/Cost[5]	Complexity level	Suite size[2]	Suite cost[3]	Faults[4]	Faults/Cost[5]
30 K–60 K	29.01	2.71×10^{8}	4.57	1.71×10^{-8}	0.91 M–1.37 M	168.54	127935	6.00	4.72×10^{-5}
60 K–90 K	38.82	3.34×10^{8}	4.65	1.43×10^{-8}	1.37 M–1.83 M	219.33	154847	6.00	3.91×10^{-5}
90 K–120 K	54.26	4.31×10^{8}	4.70	1.12×10^{-8}	1.83 M–2.29 M	283.66	191174	6.00	3.16×10^{-5}
120 K–150 K	70.76	5.40×10^{8}	4.79	9.10×10^{-9}	2.29 M–2.74 M	344.80	227819	6.00	2.65×10^{-5}
150 K–180 K	84.25	6.25×10^{8}	4.81	7.84×10^{-9}	2.74 M–3.20 M	397.12	259383	6.00	2.32×10^{-5}

[1] Each value in this table indicates the average across all of the generated original test suites at each complexity level.
[2] Indicates the average number of test cases in the original test suite.
[3] Indicates the average cost (microsecond, μs) taken to run the test cases in the original test suite.
[4] Indicates the average number of faults that can be revealed by the test cases in the original test suite.
[5] Indicates the average number of faults that can be revealed per unit of regression test cost in the original test suite.

While the cost-aware techniques are effective at minimizing ECRS, the algorithms adopted by them take more computational time than Greedy and may result in higher TPRS values. Thus, it is necessary to analyze whether the improved TRTC (i.e., the sum of ECRS and TPRS) is lower than the time taken to perform regression test using the original test suite. In order to answer this research question, we will compare the three techniques' TRTC values with the time taken to run the original test suite at different levels of test suite complexity.

- *Research Question 5 (RQ5) – What is the effect on the number of revealed faults achieved by the three techniques as the level of test suite complexity varies?*

While test suite reduction decreases ECRS, it may also decrease the number of revealed faults (NRF) in this round of regression testing. We let $NRF(RS)$ represent the number of faults revealed by the test cases in RS. It is noted that some test cases in a test suite may reveal the same faults, but a fault revealed by more than one test case is counted only once.

A higher value of NRF represents that the fault detection loss caused by the reduction technique is less significant. We will reply to this research question in terms of the NRF metric.

- *Research Question 6 (RQ6) – What is the effect on the fault detection efficiency for the three techniques as the level of test suite complexity varies?*

Since both cost and fault detection are important considerations for regression testing, our empirical studies will also evaluate the fault detection efficiency using the number of faults revealed per unit of regression test cost (FPTC). The FPTC of the representative set *RS* is defined as

$$FPTC(RS) = \frac{NRF(RS)}{TRTC(RS)}. \tag{8}$$

A higher value of FPTC indicates that the reduction technique that produces *RS* realizes higher fault detection efficiency. In other words, the FPTC value should be a suitable criterion for evaluating test suite reduction techniques in order to maximize the number of revealed faults when adopting a time constraint during testing. We will answer this research question by comparing the three test reduction techniques in terms of FPTC at different levels of test suite complexity.

- *Research Question 7 (RQ7) – Do the representative sets produced by the three techniques have many test cases in common at each level of test suite complexity?*

Recall that Greedy intends to decrease the number of test cases in the representative set, while $\text{Greedy}_{\text{ElIrreplaceability}}$ and $\text{Greedy}_{\text{Ratio}}$ are cost-aware techniques that aim to decrease the cost taken to run all of the test cases in the representative set. Because they evaluate the test cases using different metrics during the reduction process, this motivates us to analyze whether the representative sets produced by different techniques have many test cases in common. According to the concepts described in [21] and [22], the common rate of the representative sets (CRRS) produced by different techniques is defined as

$$CRRS(RS_1, RS_2) = \frac{|RS_1 \cap RS_2|}{|RS_1 \cup RS_2|} \times 100\%, \tag{9}$$

where RS_1 and RS_2 represent the representative sets produced by two different techniques. A high value of CRRS indicates that the representative sets have a lot of test cases in common. Two techniques' cost reduction capability may be close to each other if they often produce representative sets that have a high percentage of test cases in common. This may indicate that the two techniques can be used interchangeably. That is to say, if the two cost-aware techniques and Greedy often produce representative sets that have a high percentage of test cases in common, the cost-aware techniques would not be judged as effective. We will answer this research question by analyzing the CRRS between the three techniques at different levels of test suite complexity.

3.3. Empirical results

Reply to RQ1 – Fig. 4 compares the ECRS values produced by the three techniques at different complexity levels. Because the values represented by the bar charts in the figure differ in magnitude, we insert the horizontal regions in order to both clarify the small values and avoid cutting the large values. As seen from this figure, $\text{Greedy}_{\text{ElIrreplaceability}}$ achieves the lowest ECRS values in all studied situations, and $\text{Greedy}_{\text{Ratio}}$ usually performs better than Greedy. But, when the complexity is relatively low, $\text{Greedy}_{\text{Ratio}}$ performs worse than Greedy for printtokens, printtokens2, schedule, and schedule2. Fig. 4 also indicates that in all studied situations the three techniques' ECRS values are much lower than the costs taken to run the test cases in the original test suites. That is, the ECRS values are significantly decreased no matter which of the three techniques was adopted.

Additionally, it should be noted that, except for the schedule program, the ECRS values achieved by both $\text{Greedy}_{\text{ElIrreplaceability}}$ and $\text{Greedy}_{\text{Ratio}}$ monotonically decrease as the complexity increases. This is because the number of test cases to be selected generally increases with the growth of the complexity. As a result, $\text{Greedy}_{\text{ElIrreplaceability}}$ and $\text{Greedy}_{\text{Ratio}}$ can choose the most preferable ones to minimize the execution cost. However, this phenomenon does not hold for Greedy because it does not aim to minimize the execution cost of a test suite even though more test cases are available for selection.

Reply to RQ2 – Fig. 5 compares the TPRS values for the three techniques at different complexity levels. As seen from Fig. 5, the cost-aware techniques generally take more time to produce the representative sets at different levels of test suite complexity. Especially for $\text{Greedy}_{\text{ElIrreplaceability}}$, its TPRS values are 2 to 42 times higher than those of Greedy and 2 to 41 times higher than those of $\text{Greedy}_{\text{Ratio}}$. Although Greedy sometimes performs slower than $\text{Greedy}_{\text{Ratio}}$ when the test suite complexity is high, its TPRS values are usually the lowest on the whole. Additionally, Fig. 5 also indicates that the TPRS values for the three techniques monotonically increase, in most of the studied situations, as the test suite complexity increases.

Reply to RQ3 – Instead of comparing the ECRS and TPRS values separately, Fig. 6 compares the TRTC values (i.e., the summations of ECRS and TPRS values) of the three techniques together with the costs taken to run all of the test cases in the original test suites at different complexity levels. Fig. 6 indicates that in all of the studied situations the three techniques' TRTC values are much lower than the costs taken to run all tests in the original test suites. That is, the regression testing costs are significantly decreased no matter which of the three techniques was adopted to improve the regression testing process.

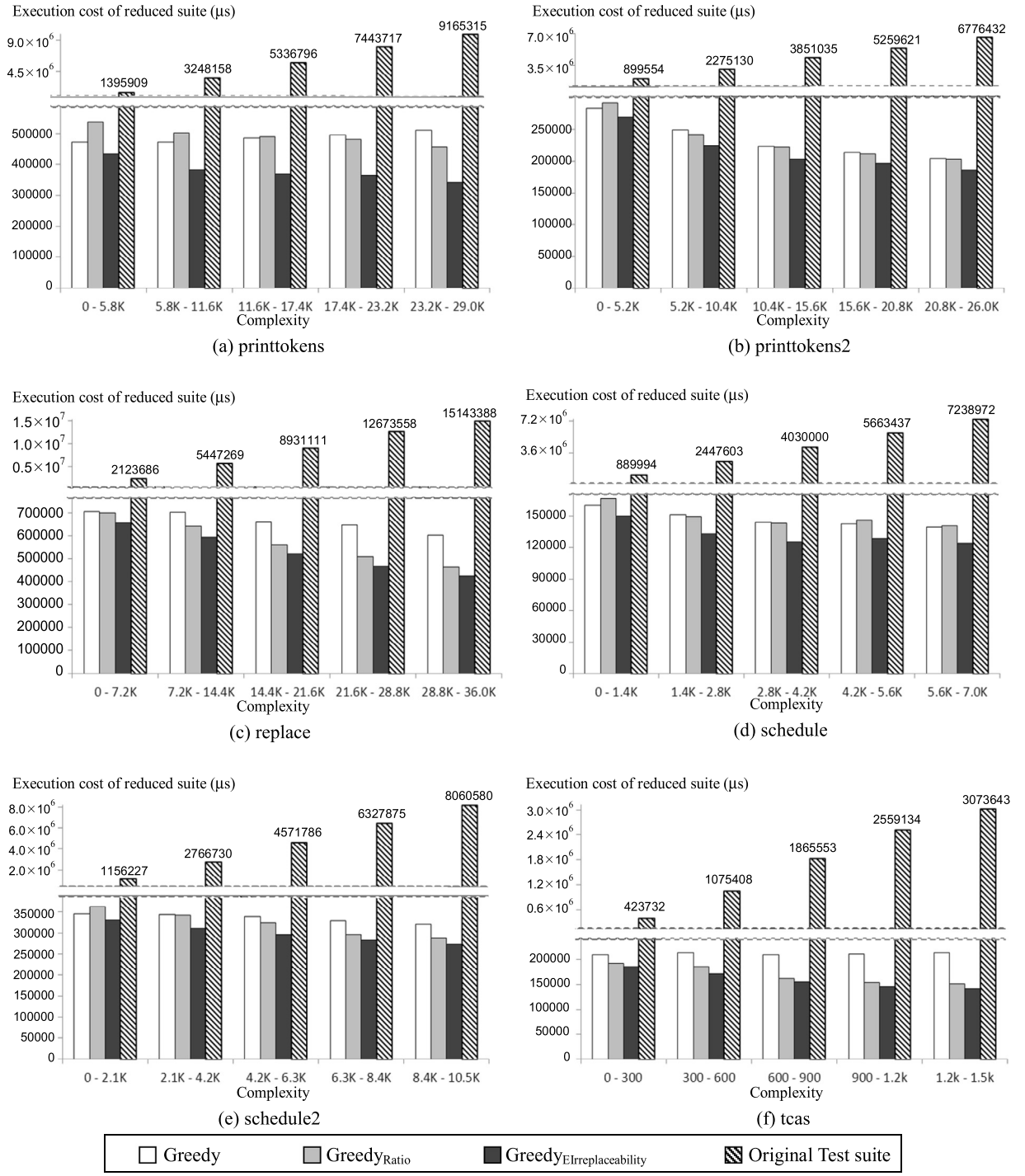

Fig. 4. The comparisons on the execution costs of the representative set (ECRS).

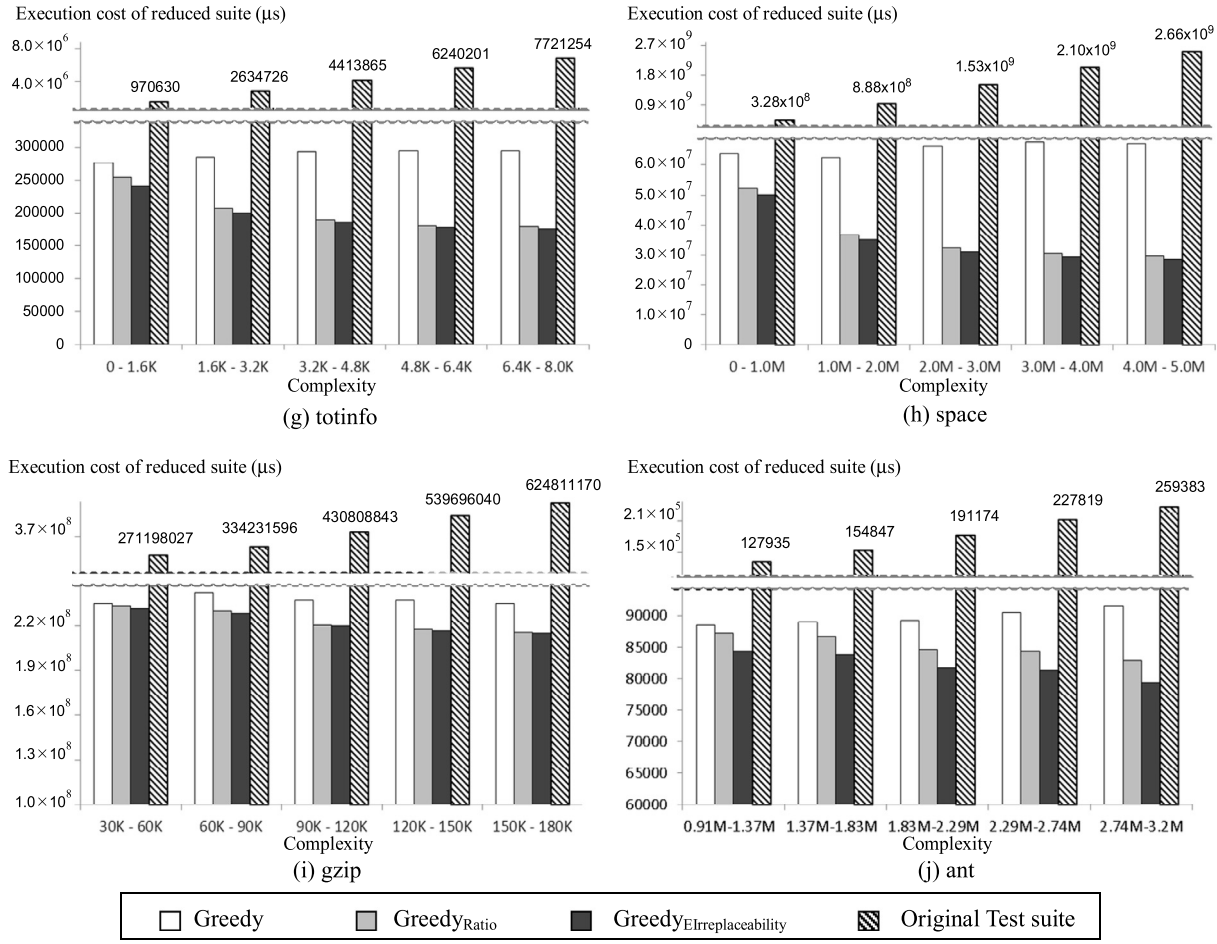

Fig. 4. (*continued*)

It is noted that the TPRS value accounts for a negligible percentage of the TRTC value, ranging from 0.000003% to 0.26511%, in all studied situations. As a result, Fig. 6 almost looks identical to Fig. 4. In other words, similar to the ECRS comparisons in the reply to RQ1, the TRTC values for the cost-aware techniques generally tend to monotonically decrease as the test suite complexity increases, but this phenomenon does not hold for Greedy. Moreover, as seen from Fig. 6, $\text{Greedy}_{\text{EIrreplaceability}}$ achieves the lowest TRTC values in all studied situations; $\text{Greedy}_{\text{Ratio}}$ usually performs better than Greedy. In general, the TRTC values achieved by both $\text{Greedy}_{\text{EIrreplaceability}}$ and $\text{Greedy}_{\text{Ratio}}$ monotonically decrease as the complexity increases, except for schedule. However, this trend does not hold for Greedy. This phenomenon is similar to Fig. 4, and the reply to RQ1 has already discussed the reason.

Reply to RQ4 – As seen from Figs. 4 through 6, for all of the subject programs, the TPRS values are much lower in comparison with ECRS values at all levels of test suite complexity. Thus, the improvements in ECRS achieved by any of the three techniques can compensate TPRS (i.e., the penalties of performing the test suite reduction). As mentioned in the reply to RQ3, any of the three techniques can decrease the costs of regression testing. We can also notice that, compared to execution cost of the representative set, the cost of performing a test suite reduction technique should be a relatively minor consideration for improving regression testing.

Reply to RQ5 – Fig. 7 compares the NRF values of the three techniques at different levels of test suite complexity. This figure indicates that the differences in the NRF values between the cost-aware techniques and Greedy are insignificant in most of the studied situations, except for schedule2 at the high complexity levels. In other words, the fault revealing capabilities of the cost-aware techniques are not significantly better than that of Greedy. This is because the cost-aware techniques intend to minimize the execution costs rather than maximize the fault revealing capability. Additionally, for most of the subject programs, the three techniques' NRF values do not clearly change as the test suite complexity grows. Only for the subject program schedule2, the cost-aware techniques' NRF values tend to decrease with the growth of the test suite complexity while those of Greedy contrariously tend to increase.

Fig. 7 also indicates that the three techniques result in a significant decrease in fault detection effectiveness in most of the studied situations, except for gzip and ant. Notice that the three techniques do not cause any fault detection loss for

the ant program. Overall, the fault detection loss caused by a test suite reduction technique is not directly related to its cost reduction capability.

Reply to RQ6 – According to the replies to RQ1 and RQ5, the two cost-aware techniques generally achieve better ECRS and TRTC values than Greedy but result in an ordinary NRF. Considering both cost reduction capability and fault detection loss, Fig. 8 compares the fault detect efficiency achieved by three techniques in terms of FPTC. As seen from this figure,

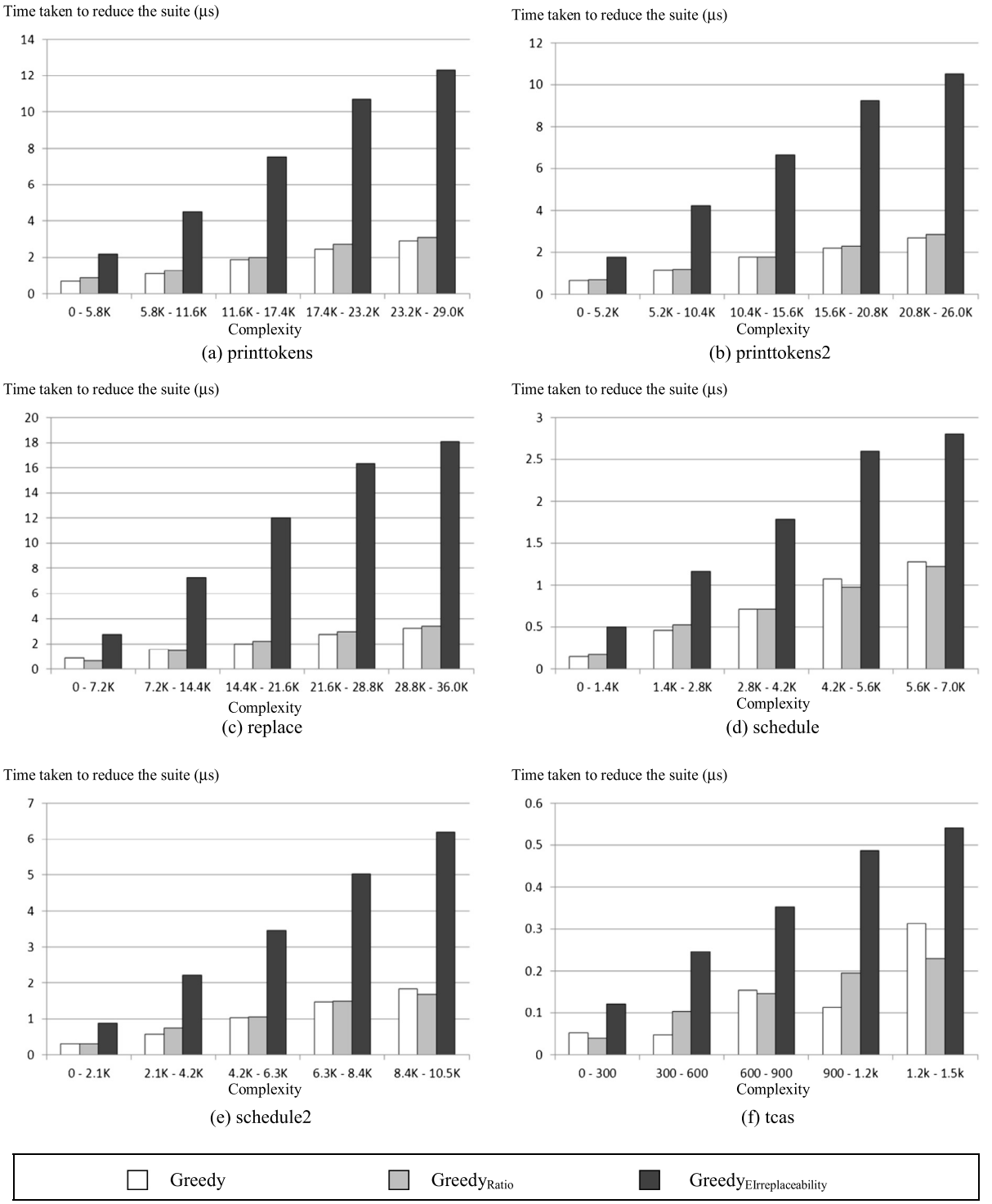

Fig. 5. The comparisons on the time taken to produce the representative sets (TPRS).

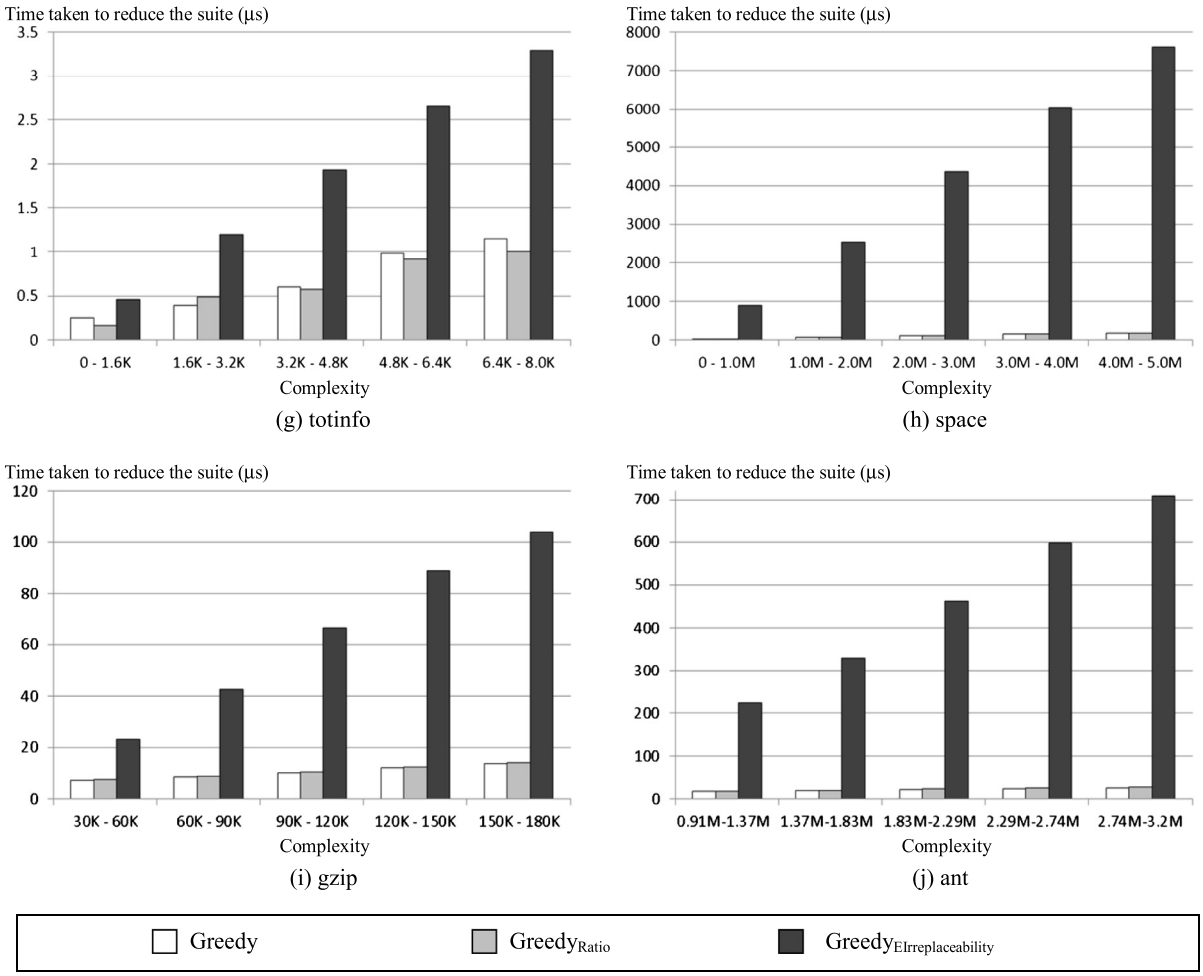

Fig. 5. (*continued*)

$Greedy_{EIrreplaceability}$ achieves higher FPTC values than the others in most of the studied situations; considering the other two techniques, $Greedy_{Ratio}$ works better than Greedy in general. Overall, the cost-aware techniques outperform Greedy in terms of FPTC.

Additionally, Fig. 8 also shows that, as the test suite complexity grows, the FPTC values achieved by both cost-aware techniques tend to increase in most of the studied situations, except for schedule2. However, the FPTC values achieved by Greedy do not generally tend to increase due to the following two reasons (1) as in the reply to RQ5, the NRF values of the three techniques for most of the subject programs do not clearly change as the test suite complexity grows. However, considering schedule2, the NRF values of the cost-aware techniques tend to decrease with the growth of the test suite complexity while those of Greedy contrariously tend to increase; (2) as mentioned in the reply to RQ3, the TRTC values for the cost-aware techniques tend to monotonically decrease as the test suite complexity increases, but this phenomenon does not hold for Greedy.

Moreover, the test suites produced by the three techniques significantly outperform the original test suites in terms of FPTC for all of the subject programs. Thus, although the three techniques' NRF values are less than those of the original test suite in most of the situations, as described in the reply to RQ5, they can result in more efficient regression testing. Fig. 8 also indicates that the FPTC values of original test suites generally decrease as the test suite complexity grows. This is because, although both the TRTC and NRF values of the original test suites monotonically grow as the test suite complexity increases, the growths of TRTC are clearly more substantial than those of NRF.

Reply to RQ7 – Fig. 9 shows the radar charts of the common rates at different levels of test suite complexity for all of the subject programs. Each of the five axes in the radar chart indicates the common rates for a specific pair of representative sets at each level of complexity. According to this figure, the representative sets produced by $Greedy_{Ratio}$ and $Greedy_{EIrreplaceability}$ contain the most test cases in common at all complexity levels for most of the subject programs, except for printtokens and ant. This may confirm that the two cost-aware techniques reduce test suites with the same intention (i.e., decreasing the test execution costs), thus meaning that they generally choose the most test cases in common. On the other hand, Greedy

reduces test suites with another intention (i.e., decreasing the number of test cases), thus resulting that the common rates of the other two pairs (i.e., the pair of $\text{Greedy}_{\text{EIrreplaceability}}$ and Greedy and the pair of $\text{Greedy}_{\text{Ratio}}$ and Greedy) are generally relatively lower. Additionally, although the common rates of the pair of $\text{Greedy}_{\text{EIrreplaceability}}$ and Greedy are usually higher than those of the pair of $\text{Greedy}_{\text{Ratio}}$ and Greedy, the differences between these two pairs are usually minor, except for printtokens. Especially for tcas, totinfo, and space, the common rates for these two pairs are almost totally identical.

Finally, it is observed that the common rates of all possible pairs for most of the subject programs tend to decrease as the complexity of the test suites increases. This is because more test cases are gradually available as the test suite

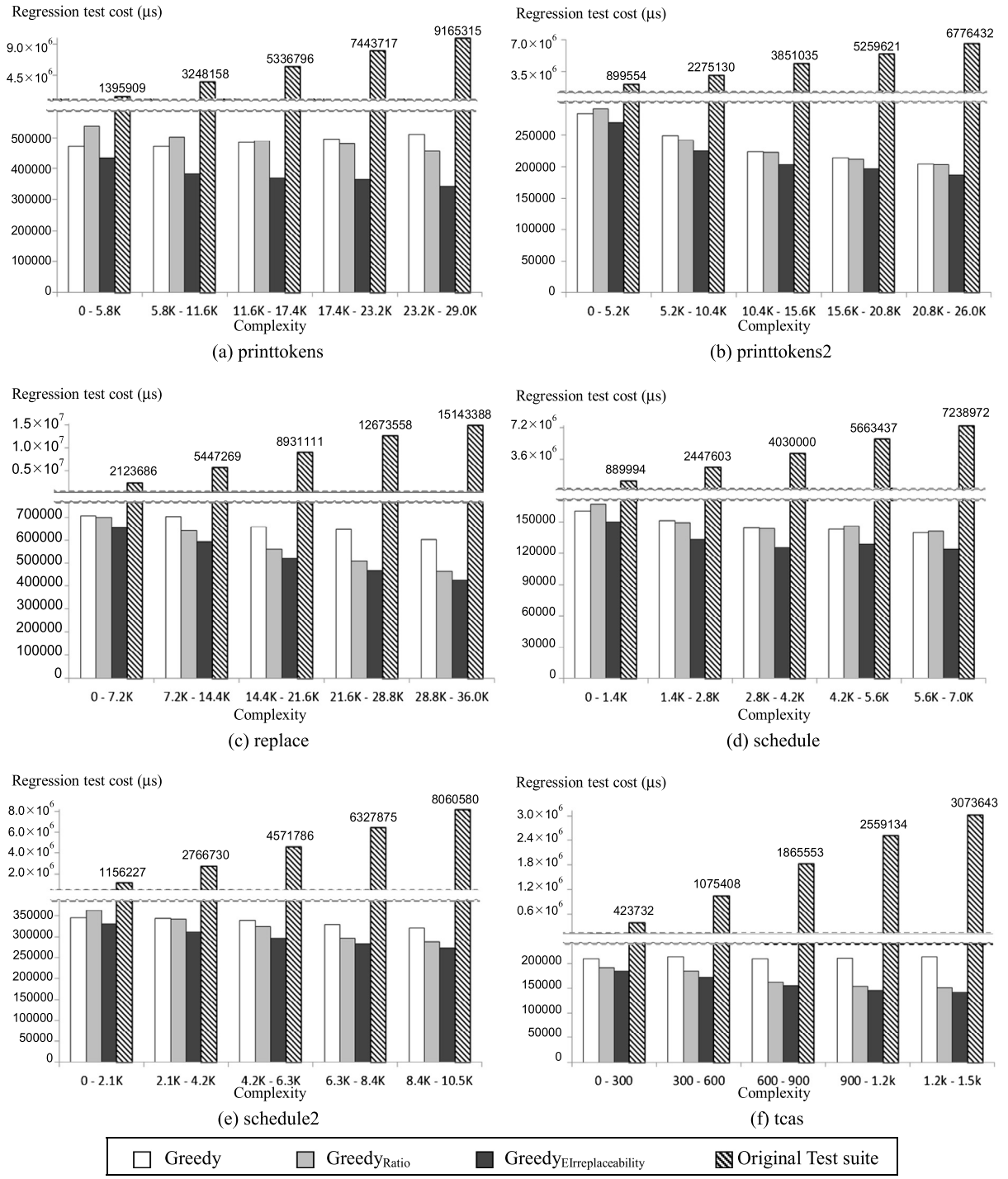

Fig. 6. The comparisons on the total regression test cost (TRTC).

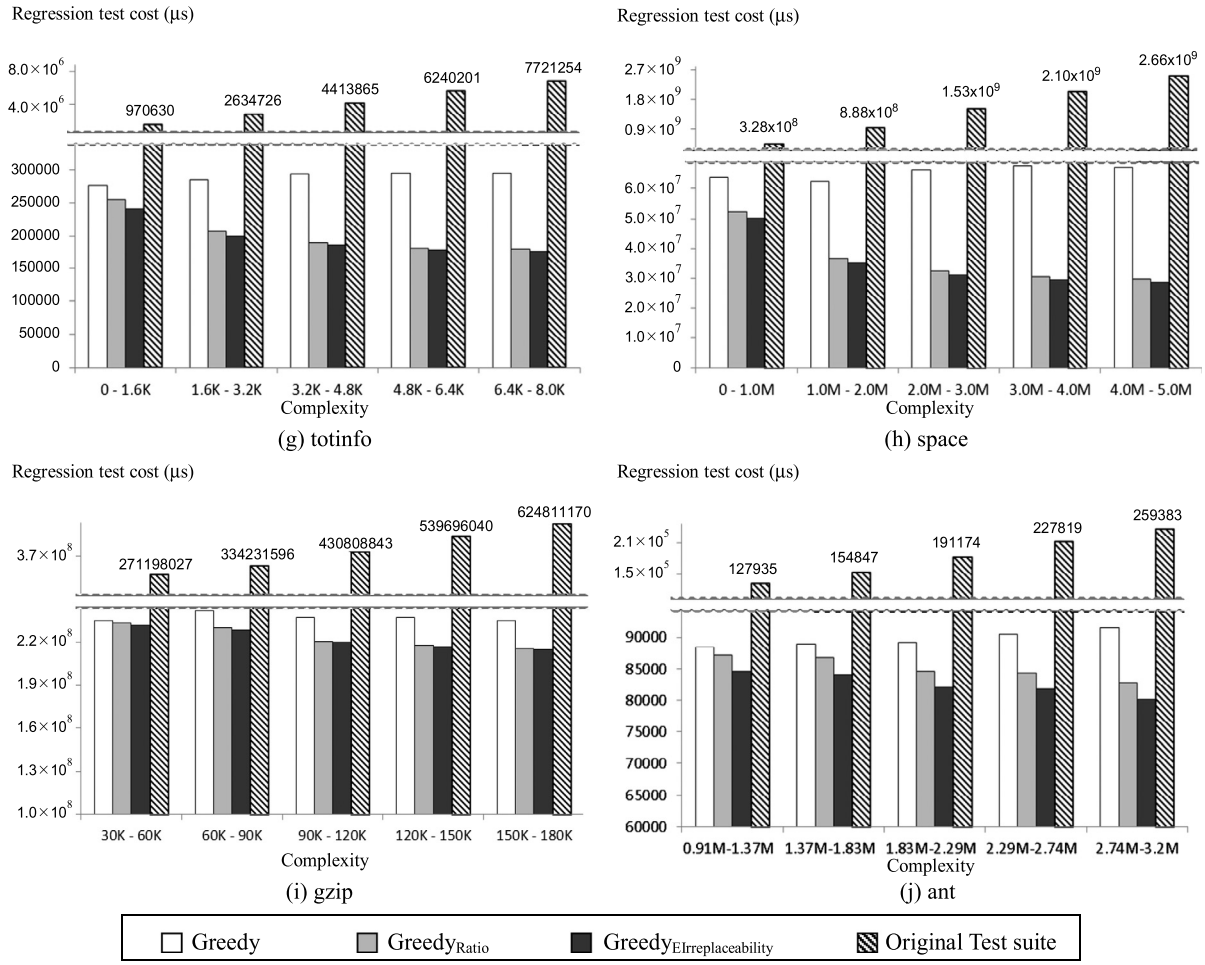

Fig. 6. (*continued*)

complexity increases, thus decreasing the probability that a test case is included in the representative sets produced by different techniques. Furthermore, if we more specifically analyze the trends of different pairs, we can see that the common rates between $\text{Greedy}_{\text{EIrreplaceability}}$ and $\text{Greedy}_{\text{Ratio}}$ slightly decrease as the test suite complexity increases, whereas the common rates of the other two pairs significantly decrease. The cause of this phenomenon is similar to the one described in the immediately previous paragraph. That is, the two cost-aware techniques, $\text{Greedy}_{\text{Ratio}}$ and $\text{Greedy}_{\text{EIrreplaceability}}$, reduce test suites with the same intention while Greedy reduces test suites with another goal. Thus, even though more test cases are available, the decreases in common rates are minor. However, if more test cases are available, Greedy has more chances to choose test cases with a different intention, thus significantly decreasing the common rates. In other words, as the test suite complexity increases, the difference between the representative sets produced by Greedy and either cost-aware technique will be amplified.

Practical Implications – The following implications for practicing software developers arise from this paper's empirical results. It is important to note that all of these implications are generally applicable at different levels of test suite complexity.

- All of the three test suite reduction techniques work well for all of the studied situations.
- The representative sets that were produced by the three techniques have similar fault revealing capabilities.
- If testers want to minimize the execution cost of the representative set, the total regression testing cost, and the number of faults revealed per unit of regression test cost, then they should pick $\text{Greedy}_{\text{EIrreplaceability}}$.
- If minimizing the time taken to perform the test suite reduction technique is the testers' main consideration, Greedy should be the best technique for testers.
- In comparison with the others (e.g., cost reduction capability and fault detection capability), scalability should be a less troublesome issue for test suite reduction. This provides an initial confirmation that it is feasible to adopt test suite reduction techniques during real-world software development.

- Most of the adopted measured values, except for scalability, that were observed for the cost-aware techniques tend to improve as the test suite complexity grows. However, these phenomena do not generally hold for Greedy. Thus, the higher the complexity of the original test suite is, the more beneficial it is to adopt the cost-aware techniques.

Recall that the aforementioned implications are generally applicable at all levels of test suite complexity. To the best of our knowledge, these results were not previously confirmed in the literature. On the whole, the cost-aware techniques should be better choices for reducing complex test suites. Yet, we suggest that the number of test cases or test requirements should not be the key factor for determining the most suitable technique among the three that were compared in our study.

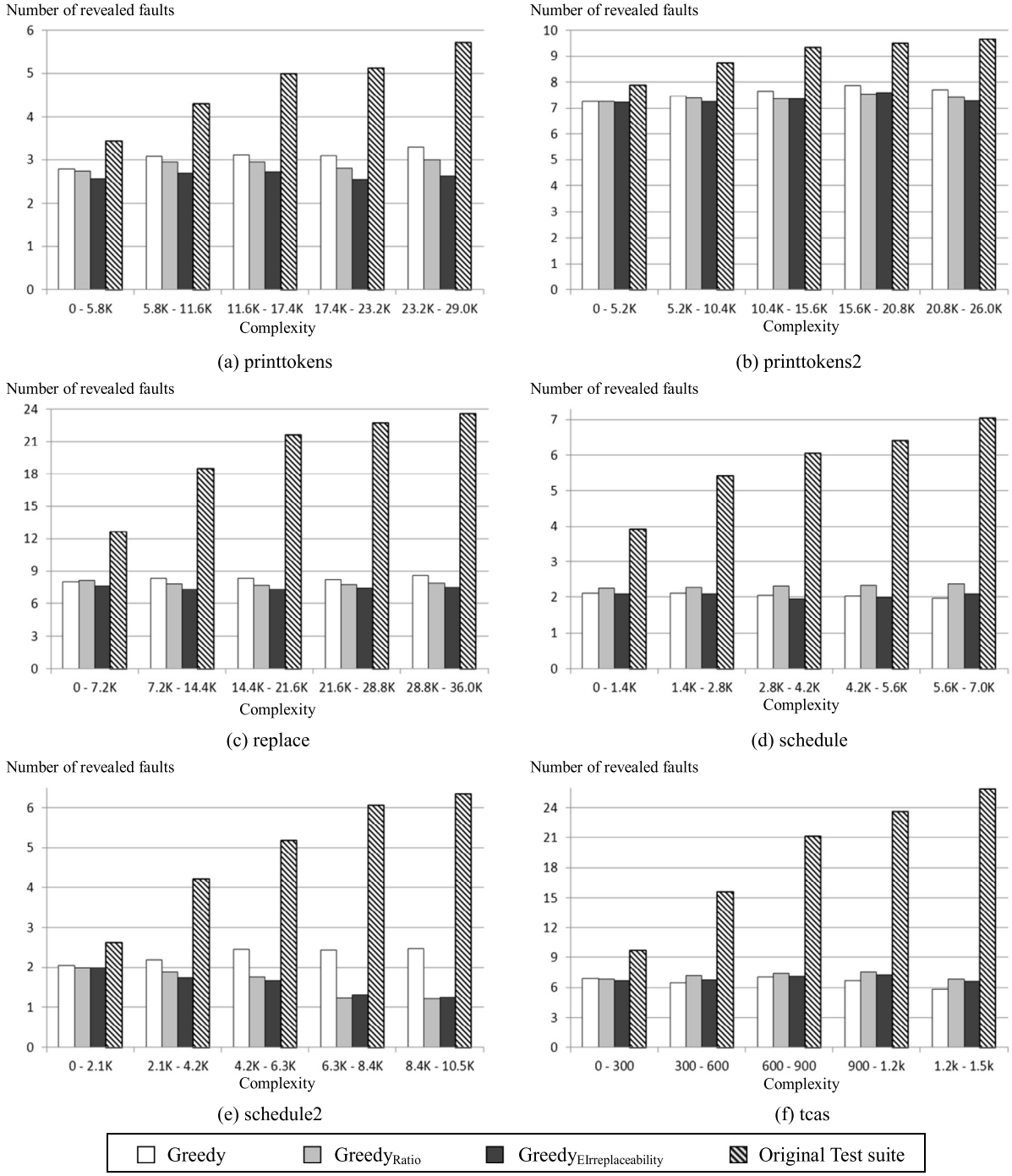

Fig. 7. The comparisons on the number of revealed faults (NRF).

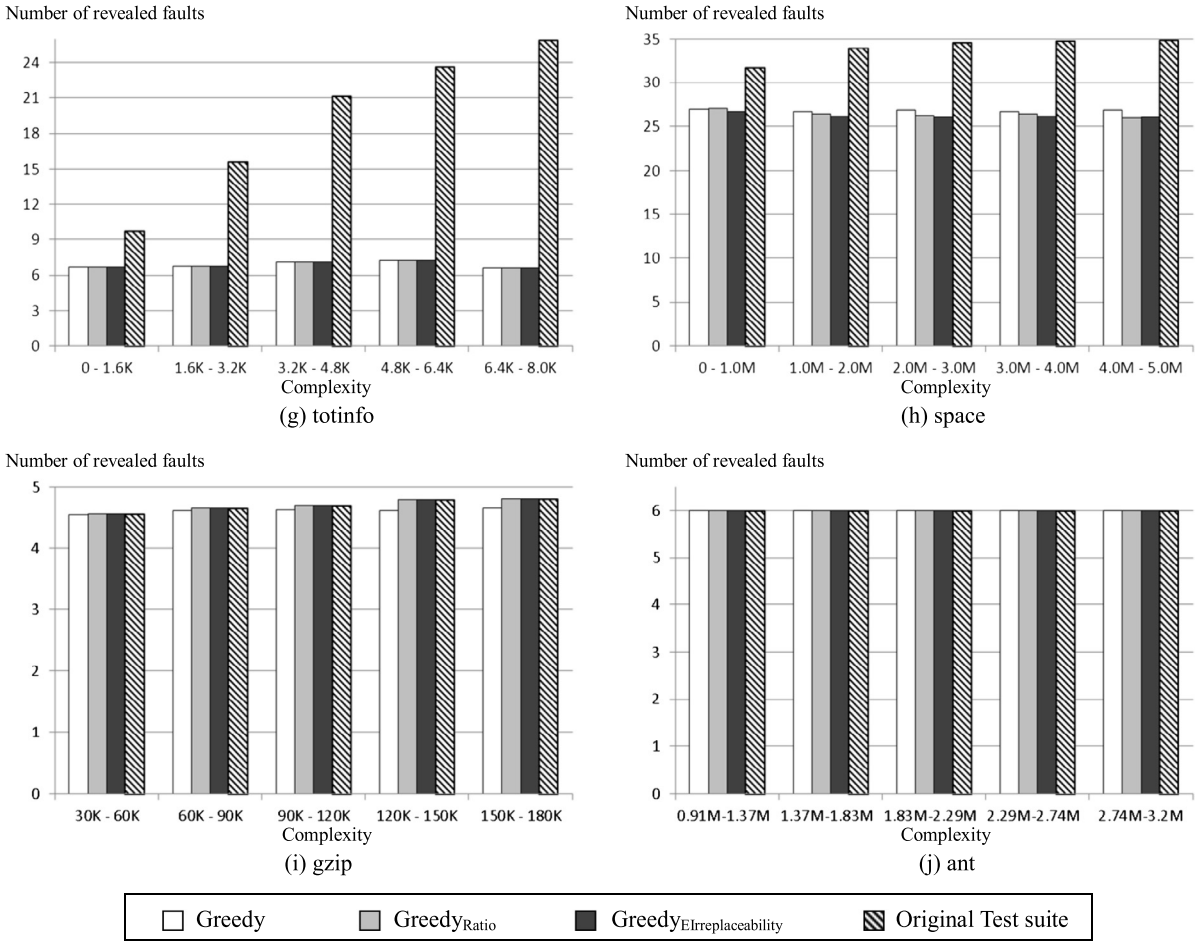

Fig. 7. (*continued*)

Instead, the selection of techniques should depend on the metrics that are adopted by practicing software developers (e.g., adopting Greedy$_{EIrreplaceability}$ if the testers aim to minimize the total regression testing cost).

Research Implications – The results in this paper also suggest that researchers in the field of regression testing should consider the following issues:

- Greedy$_{EIrreplaceability}$ generally takes more time than the others to produce the representative set of test cases. However, the cost of performing Greedy$_{EIrreplaceability}$ is trivial in comparison with the decrease in the total regression testing costs achieved by Greedy$_{EIrreplaceability}$. Therefore, researchers should focus on improving regression testing by incorporating other considerations into the Greedy method. While these improved versions of Greedy may have an inferior worst-case time complexity, in comparison to the standard Greedy algorithm, this is not likely to be noticed in practice.
- Most of the metrics adopted in our empirical studies tend to monotonically increase or decrease as the test suite complexity grows. This confirms that the effectiveness of a test suite reduction technique should be positively or negatively related to the test suite complexity. Thus, it is worthwhile for researchers to focus a study of regression testing techniques on the effect caused by changing the complexity of the original test suite.

3.4. Comparison to previous empirical studies

Considering test suite techniques' effectiveness, most of the existing empirical studies in the literature evaluate techniques' capability to reduce the size of a test suite. Recall that the time taken to run a test suite consisting of a few test cases may not always be short. In the literature, only several empirical studies (e.g., [5,21,22,58,59]) focused on comparing the execution cost of the test cases in the representative suite. Additionally, there exist some empirical studies that evaluated the techniques by comparing the time taken to perform a reduction task (e.g., [4,8,15,16,24,31,44,59–63]). Yet, unlike our paper, few empirical studies took into account the total regression testing costs (i.e., the sum of the time taken to run the test cases and the time taken to produce the representative set of test cases).

On the other hand, although a majority of prior empirical studies evaluate test suite reduction techniques in terms of their fault detection effectiveness, only a few of them (e.g., [4,6,45,62]) adopt fault detection efficiency as the criterion. Moreover, unlike the empirical study in this paper, the empirical studies in [4,6,45,62] assess fault detection efficiency using the number of faults revealed per test case. Actually, the criterion adopted in this paper to evaluate fault detection efficiency (i.e., the number of faults revealed per unit of regression test cost) should be more representative than the one adopted in [4,6,45,62] due to the following two reasons: (1) it considers the time taken to perform a reduction task; and (2) in comparison with the number of test cases, the number of time units is a more accurate metric for describing the cost of regression testing.

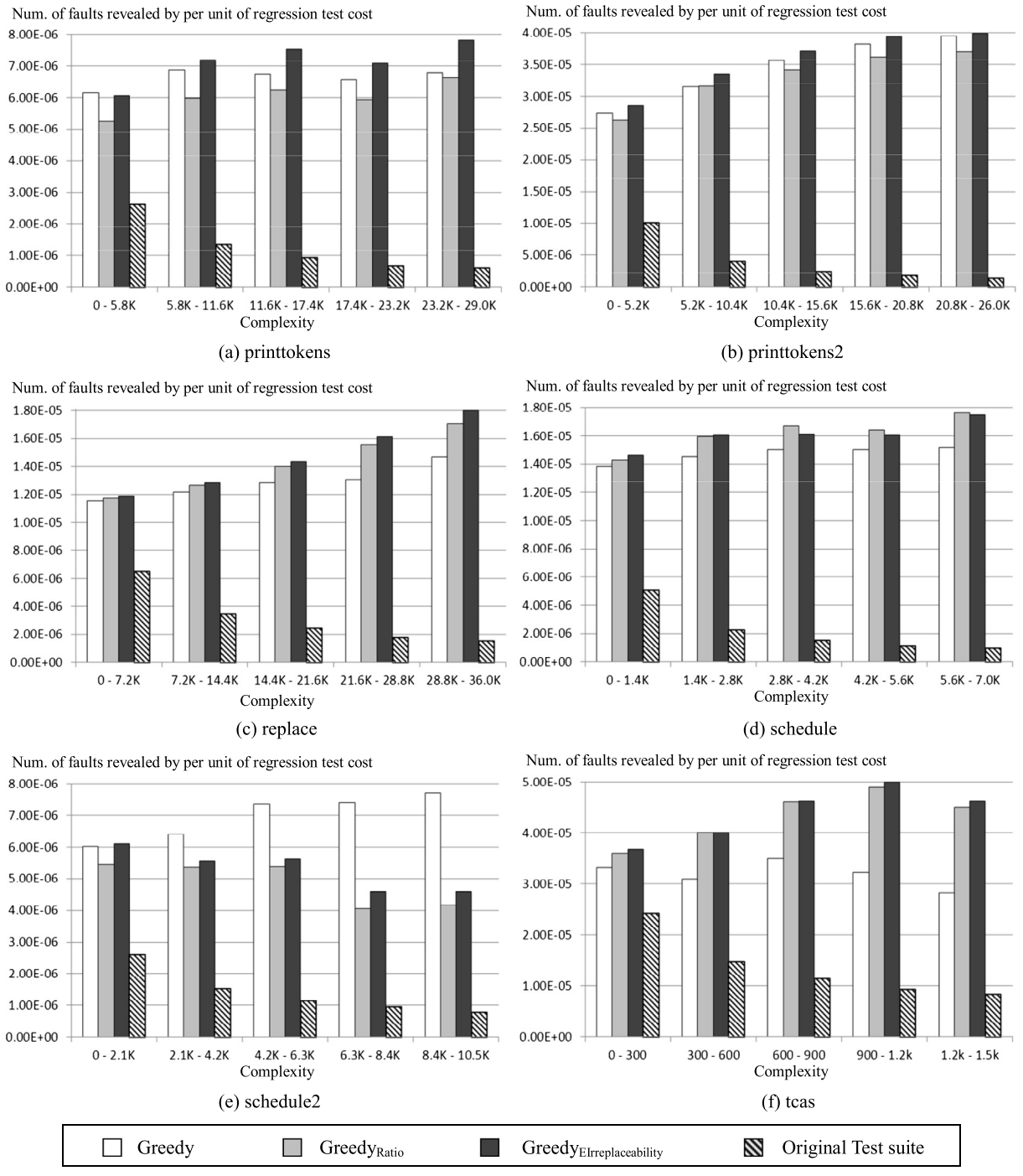

Fig. 8. The comparisons on the number of faults revealed per unit of regression test cost (FPTC).

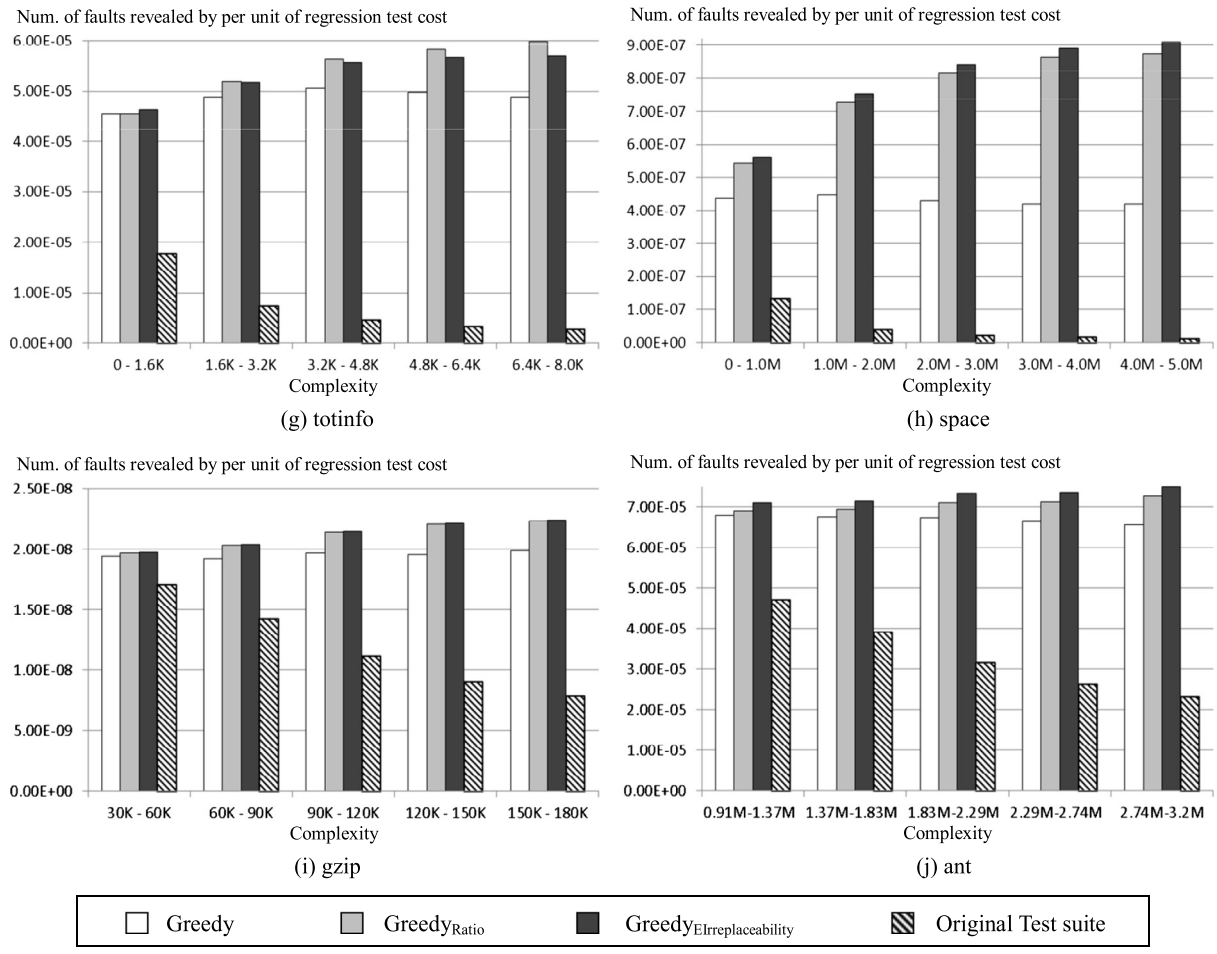

Fig. 8. (*continued*)

Furthermore, only two empirical studies in the literature (i.e., [8,22]) analyze the common rate of the representative sets produced by different techniques. Most importantly, to the best of our knowledge, none of the prior studies used many test suites, that were classified according to their complexity, to evaluate the reduction techniques according to the aforementioned criteria. This paper represents the first such attempt to carry out this important and interesting task.

4. Threats to validity

The precision of measuring the execution time of each test case is the primary threat to the internal validity of our empirical studies. Our strategy for controlling this threat is to execute each test case 1000 times on both of two frequently used workstations, a Windows machine and a Linux machine, and take the average of the 2000 execution results.

Second, we only used source code branches as the coverage criterion and this choice could affect the outcome of our empirical results. We consider it as another threat to the internal validity of our empirical studies because regression testing techniques may be sensitive to the granularity of the selected coverage criterion [46]. However, our empirical studies include quite a few subject programs and evaluation metrics. In order to avoid blurring the contribution of this study, we focused it on a commonly used coverage criterion. In order to further confirm our observations, we also plan, in our future work, to do new studies with coverage criteria of different granularities. Yet, as mentioned in Section 3.1, branch coverage is a common criterion used in many code coverage tools (e.g., the tools that were cited in [39–42]) and relevant empirical studies (e.g., [4,6,14,22]). Additionally, Gligoric et al. [47] suggested that researchers should use branch coverage to compare non-adequate test suites due to its high effectiveness and low overhead. The empirical results described in [64] also indicate that branch coverage appears to be the most viable option for code-based control flow testing. Although the focused studies in [47] and [64] are not totally identical to our study, they confirmed that branch coverage is suitable for testing researchers aiming to evaluate test suites. Thus, we judge that it is a good option for this paper's experiments.

Moreover, Inozemtseva and Holmes [65] focused on five large software systems and measured the branch coverage, statement coverage, and modified condition coverage that were achieved by a considerable number of test suites. According

to their empirical results, the adopted coverage criterion had little influence on the correlation between code coverage and test suite effectiveness if the number of test cases in the suite is controlled for. Therefore, although our studies did not include the other coverage criteria, our empirical results still have referential value for researchers and practitioners in the field of regression testing.

Third, the eight Siemens programs, the space program, the gzip program, and the ant program are frequently used benchmarks for evaluating the effectiveness of the test suite reduction techniques. Additionally, their sizes range from small to large. Even so, we still cannot definitively confirm that this paper's observations will stand for other programs. Thus, we consider it as a threat to the external validity of our empirical studies, which, as mentioned in Section 5, we plan to address in our future work.

Fourth, our empirical results indicate that the NRF values for gzip and ant are almost identical in all of the studied situations. Additionally, although the cost-aware techniques' ECRS, TRTC, FPTC, and CRRS values for gzip and ant tend to improve as the test suite complexity grows, their improvements are less significant than those of the other subject programs.

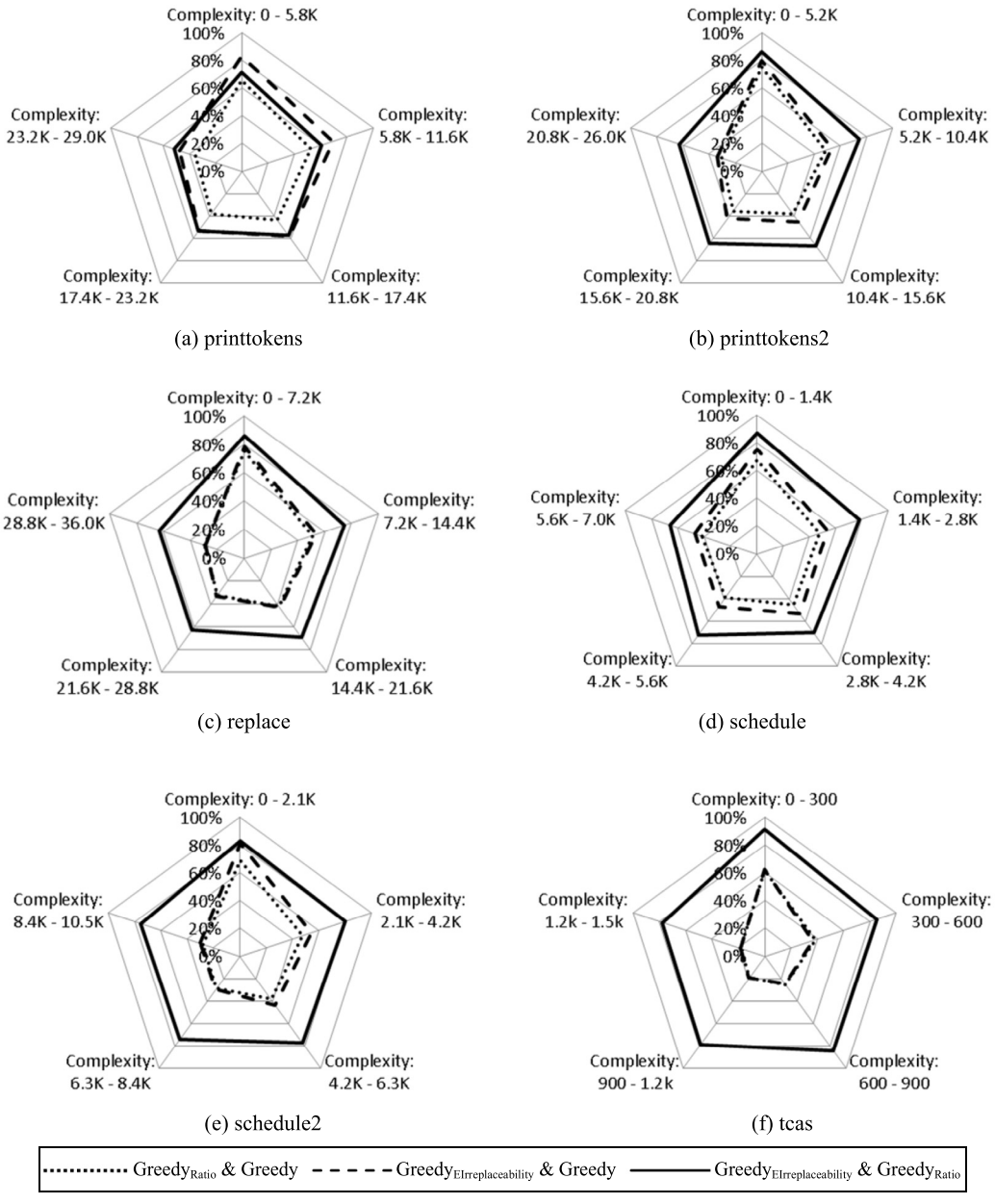

Fig. 9. The comparisons on the common rate of the representative sets (CRRS).

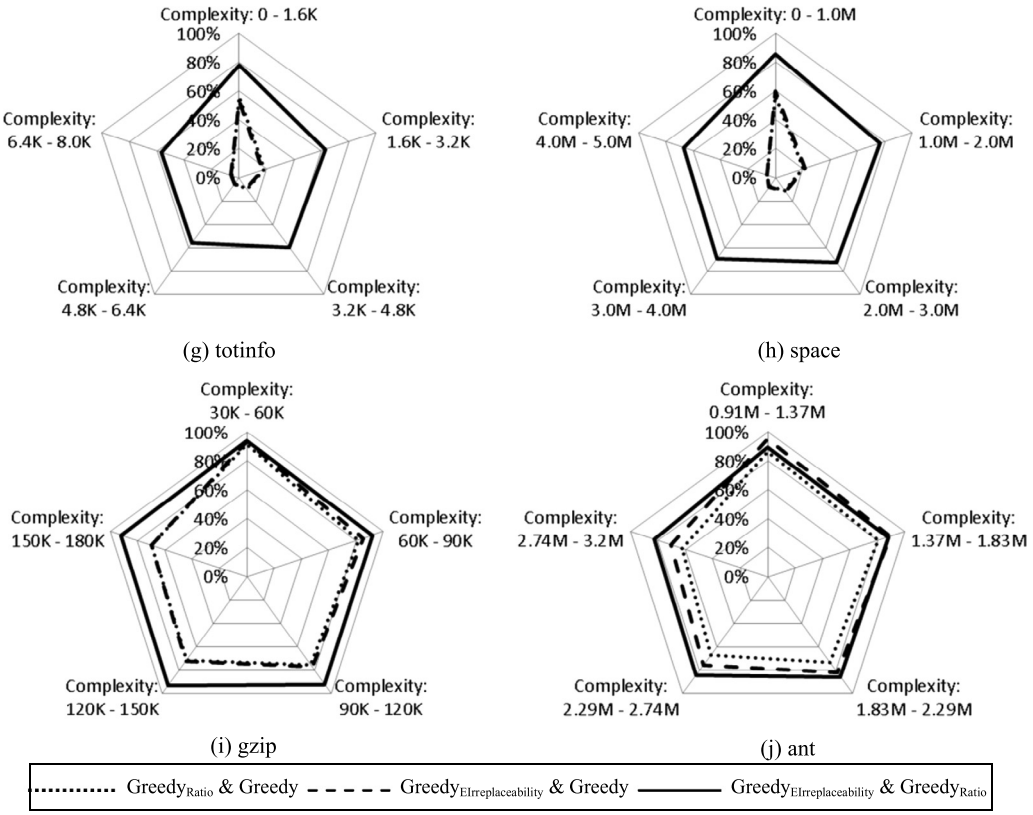

Fig. 9. (*continued*)

This is likely due to the fact that the sizes of pools of test cases provided in SIR for these two subject programs are less than those of the others – even though they still include several hundreds of test cases. This decreases the diversity in the 1,000 original test suites that are generated using the aforementioned Steps 1–4 described in Section 3.1. As a result, no matter how big the generated original test suites are, the set of test cases with high metric values (i.e., *Coverage*(t), *Ratio*(t), or *Irreplaceability*(t)) in different test suites may be overlapping. Therefore, although the sizes of the gzip and ant programs are larger than the other subject programs in our empirical studies, we consider the sizes of the test pools as another threat to the external validity of our studies, which, as mentioned in Section 5, we plan to address in our future work. Even so, it should be noted that our findings still hold for gzip and ant.

Fifth, the threat to the construct validity of the empirical studies is that we may have introduced some defects when implementing the reduction algorithms that are compared in this study. In order to control this threat, we manually studied the behavior of the test suite reduction technique when it was applied to small programs and test suites and found that the results are totally consistent with the expected behavior of our implementation. This operation confirmed the confidence in our implementation.

5. Conclusions and future work

Prior empirical studies showed that the cost-aware test suite reduction techniques, Greedy$_{EIrreplaceability}$ and Greedy$_{Ratio}$, decrease regression testing costs more than Greedy. Yet, their comparisons are too limited and were not conducted using test suites of different complexity levels. Thus, this paper empirically compared these three techniques in terms of cost reduction capability, scalability (i.e., the capability of a technique to scale with the complexity of the original test suites), total regression testing costs, fault detection capability, and fault detection efficiency (i.e., the number of fault detections per unit of cost) at different levels of original test suite complexity. Additionally, at different levels of test suite complexity, this paper also discussed the effectiveness of applying test suite reduction to regression testing and analyzed the common rates between the representative sets produced by different techniques. Because the structure of the program under test is complicated and both the execution paths and execution costs of the test cases are diverse, the operations of test suite reduction methods are generally hard to predict. Empirical studies that do not consider these factors could yield misleading results.

Considering the effectiveness of applying test suite reduction techniques to regression testing, the empirical results show that (1) no matter which of the three techniques is adopted, the cost reduction capability and the total regression testing

costs improve, but the fault detection capability gets worse in general. Nevertheless, since the improvement in the total regression testing cost is much more significant than the deterioration in the fault detection capability, the fault detection efficiency also significantly improves in general; (2) in comparison with Greedy, the cost-aware techniques generally show better results for the cost reduction capability, the total regression testing costs, and the fault detection efficiency. However, they generally show worse results for the scalability; (3) the representative sets produced by the two cost-aware techniques have more test cases in common than those produced by the other pairs of techniques.

On the other hand, considering the reduction techniques' effectiveness at different levels of test suite complexity, the empirical results show that (1) as the test suite complexity grows, the cost reduction capability, the total regression testing costs, and the fault detection efficiency achieved by the cost-aware techniques generally improve. However, the trends are not as clear for Greedy; (2) as the test suite complexity grows, the common rates between the two cost-aware techniques are stable, but the common rates of the other pairs generally decrease; (3) as the test suite complexity grows, the time taken to perform a reduction task monotonically increases for all of the three techniques. Yet, in comparison with the execution costs of the reduced test suite, it is almost negligible. Overall, the benefits of adopting the two cost-aware techniques generally increase as the test suite complexity grows.

In future work, we will address some open issues in order to further understand and improve regression testing. First of all, we aim to further decrease the threats to the internal/external validity that are described in Section 4 by conducting additional experiments with more subject programs and test suites and with coverage criteria of different granularities. Second, as described in Section 3.3, some exceptions exist for the aforementioned observations. Thus, we plan to analyze the characteristics of these exceptions to our observations and improve the effectiveness of the presented cost-aware test suite reduction techniques based on the observed characteristics. Since the improved techniques may take more computational time to perform the reduction task, we also intend to refactor them so that they can be accelerated using general purpose computing on graphical processing units (GPGPU) [3]. In addition, we plan to study how the presented cost-aware test suite reduction techniques interact with the existing statistical fault localization methods [66,67]. Overall, the combination of this paper's observations and the achievements completed during future work will improve both test suite reduction methods and other affiliated techniques related to repeated testing (e.g., automatic fault localization). Ultimately, this will improve the regressing testing and debugging activities that are a crucial part of the software development lifecycle.

Acknowledgements

This work was supported by the Ministry of Science and Technology, Taiwan, under Grants MOST 103-2221-E-415-010 and MOST 104-2628-E-415-001-MY3. The authors would like to thank the anonymous referees for their constructive and insightful suggestions, which led to a significant improvement of this paper.

References

[1] G.M. Kapfhammer, Regression Testing, The Encyclopedia of Software Engineering, Taylor and Francis – Auerbach Publications, 2010.
[2] D. Binkley, Semantics guided regression test cost reduction, IEEE Trans. Softw. Eng. 23 (8) (1997) 498–516.
[3] S. Yoo, M. Harman, S. Ur, Highly scalable multi objective test suite minimisation using graphics cards, in: Proceedings of the 3rd International Symposium on Search-Based Software Engineering, 2011, pp. 219–236.
[4] J.W. Lin, C.Y. Huang, Analysis of test suite reduction with enhanced tie-breaking techniques, Inf. Softw. Technol. 51 (4) (2009) 679–690.
[5] C.T. Lin, K.W. Tang, C.D. Chen, G.M. Kapfhammer, Reducing the cost of regression testing by identifying irreplaceable test cases, in: Proceedings of the 6th International Conference on Genetic and Evolutionary Computing, 2012, pp. 257–260.
[6] D. Jeffrey, N. Gupta, Improving fault detection capability by selectively retaining test cases during test suite reduction, IEEE Trans. Softw. Eng. 33 (2) (2007) 108–123.
[7] M.J. Harrold, R. Gupta, M.L. Soffa, A methodology for controlling the size of a test suite, ACM Trans. Softw. Eng. Methodol. 2 (3) (1993) 270–285.
[8] H. Zhong, L. Zhang, H. Mei, An experimental study of four typical test suite reduction techniques, Inf. Softw. Technol. 50 (6) (2008) 534–546.
[9] Y.C. Huang, K.L. Peng, C.Y. Huang, A history-based cost-cognizant test case prioritization technique in regression testing, J. Syst. Softw. 85 (3) (2012) 626–637.
[10] E. Engstrom, P. Runeson, M. Skoglund, A systematic review on regression test selection techniques, Inf. Softw. Technol. 52 (1) (2010) 14–30.
[11] J.A. Whittaker, S. Atkin, Software engineering is not enough, IEEE Softw. 19 (4) (2002) 108–115.
[12] M. Fahndrich, BANE: A Framework for Scalable Constraint Based Program Analysis, Ph.D. Dissertation, EECS Department, Univeristy of California, Berkeley, March 1999.
[13] D. Jeffrey, N. Gupta, Test suite reduction with selective redundancy, in: Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005, pp. 549–558.
[14] Z. Li, M. Harman, R.M. Hierons, Search algorithm for regression test case prioritization, IEEE Trans. Softw. Eng. 33 (4) (2007) 225–237.
[15] S. McMaster, A. Memon, Call-stack coverage for test suite reduction, in: Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005, pp. 539–548.
[16] S. McMaster, A. Memon, Call-stack coverage for GUI test suite reduction, IEEE Trans. Softw. Eng. 34 (1) (2008) 99–115.
[17] A. Khalilian, S. Parsa, Bi-criteria test suite reduction by cluster analysis of execution profiles, in: Proceedings of the 4th IFIP TC 2 Central and East European Conference on Advances in Software Engineering Techniques, 2012, pp. 243–256.
[18] W.E. Wong, J.R. Horgan, A.P. Mathur, A. Pasquini, Test set size minimization and fault detection effectiveness: a case study in a space application, J. Syst. Softw. 48 (2) (1999) 79–89.
[19] W.E. Wong, J.R. Horgan, A.P. Mathur, A. Pasquini, Test set size minimization and fault detection effectiveness: a case study in a space application, in: Proceedings of the 21st Annual IEEE International on Computer Software and Applications Conference, 1997, pp. 522–528.
[20] S. Arlt, A. Podelski, M. Wehrle, Reducing GUI test suites via program slicing, in: Proceedings of the 2014 ACM International Symposium on Software Testing and Analysis, 2014, pp. 270–281.

[21] A.M. Smith, G.M. Kapfhammer, An empirical study of incorporating cost into test suite reduction and prioritization, in: Proceedings of the 24th ACM Symposium on Applied Computing, Software Engineering Track, 2009, pp. 461–467.
[22] C.T. Lin, K.W. Tang, G.M. Kapfhammer, Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests, Inf. Softw. Technol. 56 (10) (2014) 1322–1344.
[23] S. Yoo, M. Harman, Pareto efficient multi-objective test case selection, in: Proceedings of the 2007 ACM International Symposium on Software Testing and Analysis, 2007, pp. 140–150.
[24] S. Yoo, M. Harman, Using hybrid algorithm for Pareto efficient multi-objective test suite minimisation, J. Syst. Softw. 83 (4) (2010) 689–701.
[25] M. Bozkurt, Cost-aware Pareto optimal test suite minimization for service-centric systems, in: Proceedings of the 15th Annual ACM Conference on Genetic and Evolutionary Computation, 2013, pp. 1429–1436.
[26] H. Do, S. Elbaum, G. Rothermel, Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact, Empir. Softw. Eng. 10 (4) (2005) 405–435.
[27] V. Chvatal, A Greedy heuristic for the set-covering problem, Math. Oper. Res. 4 (3) (1979) 233–235.
[28] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: a survey, software testing, Softw. Test. Verif. Reliab. 22 (2) (2012) 67–120.
[29] T.Y. Chen, M.F. Lau, A new heuristic for test suite reduction, Inf. Softw. Technol. 40 (5–6) (1998) 347–354.
[30] M. Hutchins, H. Foster, T. Goradia, T. Ostrand, Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria, in: Proceedings of the 16th ACM/IEEE International Conference on Software Engineering, 1994, pp. 191–200.
[31] F.I. Vokolos, P.G. Frankl, Empirical evaluation of the textual differencing regression testing technique, in: Proceedings of the 14th IEEE International Conference on Software Maintenance, 1998, pp. 44–53.
[32] Z. Zhang, B. Jiang, W.K. Chan, T.H. Tse, X. Wang, Fault localization through evaluation sequences, J. Syst. Softw. 83 (2) (2010) 174–187.
[33] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, G. Rothermel, A static approach to prioritizing JUnit test cases, IEEE Trans. Softw. Eng. 38 (6) (2012) 1258–1275.
[34] H. Do, G. Rothermel, A. Kinneer, Prioritizing JUnit test cases: an empirical assessment and cost-benefits analysis, Empir. Softw. Eng. 11 (1) (2006) 33–70.
[35] H. Do, G. Rothermel, On the use of mutation faults in empirical assessments of test case prioritization techniques, IEEE Trans. Softw. Eng. 32 (9) (2006) 733–752.
[36] A. Khaliloan, M.A. Azgomi, Y. Fazlalizadeh, An improved method for test case prioritization by incorporating historical test case data, Sci. Comput. Program. 78 (1) (2012) 93–116.
[37] Y.C. Huang, K.L. Peng, C.Y. Huang, A history-based cost-cognizant test case prioritization technique in regression testing, J. Syst. Softw. 85 (3) (2012) 626–637.
[38] A. Bergel, V. Peña, Increasing test coverage with Hapao, Sci. Comput. Program. 79 (1) (2014) 86–100.
[39] G. Fraser, A. Arcuri, EvoSuite: automatic test suite generation for object-oriented software, in: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 2011, pp. 416–419.
[40] R. Gopinath, C. Jensen, A. Groce, Code coverage for suite evaluation by developers, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 72–82.
[41] Y. Zheng, S. Kell, L. Bulej, H. Sun, W. Binder, Comprehensive multiplatform dynamic program analysis for Java and Android, IEEE Softw. 33 (4) (2016) 55–63.
[42] L. Mariani, F. Pastore, MASH: tool integration made easy, Softw. Pract. Exp. 43 (4) (2013) 419–433.
[43] G. Rothermel, M.J. Harrold, J. Ostrin, C. Hong, An empirical study of the effects of minimization on the fault detection capabilities of test suites, in: Proceedings of the 14th IEEE International Conference on Software Maintenance, 1998, pp. 34–43.
[44] J.A. Jones, M.J. Harrold, Test suite reduction and prioritization for modified condition/decision coverage, IEEE Trans. Softw. Eng. 29 (3) (2003) 195–209.
[45] M. Marre, A. Bertolino, Using spanning sets for coverage testing, IEEE Trans. Softw. Eng. 29 (11) (2003) 974–984.
[46] S. Elbaum, A.G. Malishevsky, G. Rothermel, Test case prioritization: a family of empirical studies, IEEE Trans. Softw. Eng. 28 (2) (2002) 159–182.
[47] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M.A. Alipour, D. Marinov, Guidelines for coverage-based comparisons of non-adequate test suites, ACM Trans. Softw. Eng. Methodol. 24 (4) (2015), Article No. 22.
[48] S. Mirarab, S. Akhlaghi, L. Tahvildari, Size-constrained regression test case selection using multi-criteria optimization, IEEE Trans. Softw. Eng. 38 (4) (2012) 936–956.
[49] S. Wang, S. Ali, A. Gotlieb, Cost-effective test suite minimization in product lines using search techniques, J. Syst. Softw. 103 (2015) 370–391.
[50] I. Sommerville, Software Engineering, 9th ed., Addison–Wesley, 2010.
[51] R. Pressman, Software Engineering: A Practitioner's Approach, 3rd ed., McGraw-Hill, 1992.
[52] U.K. Kudikyala, R.B. Vaughn, Software requirement understanding using pathfinder networks: discovering and evaluating mental models, J. Syst. Softw. 74 (1) (2005) 101–108.
[53] R. Koschke, Software visualization in software maintenance, reverse engineering, and re-engineering: a research survey, J. Softw. Maint. Evol. 15 (2) (2003) 87–109.
[54] H. Ibrahim, B.H. Far, A. Eberlein, Scalability improvement in software evaluation methodologies, in: Proceedings of the 2009 IEEE International Conference on Information Reuse & Integration, 2009, pp. 236–241.
[55] X. Dong, M.W. Godfrey, A hybrid program model for object-oriented reverse engineering, in: Proceedings of the 15th IEEE International Conference on Program Comprehension, 2007, pp. 81–90.
[56] J. Zhang, M. Zhu, D. Hao, L. Zhang, An empirical study on the scalability of selective mutation testing, in: Proceedings of the 25th IEEE International Symposium on Software Reliability Engineering, 2014, pp. 277–287.
[57] P. Parízek, O. Lhoták, Model checking of concurrent programs with static analysis of field accesses, Sci. Comput. Program. 98 (4) (2015) 735–763.
[58] A.M. Smith, J. Geiger, G.M. Kapfhammer, M. Soffa, Test suite reduction and prioritization with call trees, in: Proceedings of the 21th IEEE/ACM International Conference on Automated Software Engineering, 2007, pp. 539–540.
[59] X.Y. Ma, Z.F. He, B.K. Sheng, C.Q. Ye, A genetic algorithm for test-suite reduction, in: Proceedings of the 2005 IEEE International Conference on Systems, Man and Cybernetics, vol. 1, 2005, pp. 133–139.
[60] S. Tallam, N. Gupta, A concept analysis inspired Greedy algorithm for test suite minimization, in: Proceedings of the 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, 2005, pp. 35–42.
[61] H.Y. Hsu, A. Orso, MINTS: a general framework and tool for supporting test-suite minimization, in: Proceedings of the 31st ACM/IEEE International Conference on Software Engineering, 2009, pp. 419–429.
[62] S. Sprenkle, S. Sampath, E. Gibson, L. Pollock, A. Souter, An empirical comparison of test suite reduction techniques for user-session-based testing of web applications, in: Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005, pp. 587–596.
[63] H. Zhong, L. Zhang, H. Mei, An experimental comparison of four test suite reduction techniques, in: Proceedings of the 28th ACM/IEEE International Conference on Software Engineering, 2006, pp. 636–640.

[64] A. Gupta, P. Jalote, An experimental comparison of the effectiveness of control flow based testing approaches on seeded faults, in: Proceedings of the 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, 2006, pp. 365–378.
[65] L. Inozemtseva, R. Holmes, Coverage is not strongly correlated with test suite effectiveness, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 435–445.
[66] D. Gong, T. Wang, X. Su, P. Ma, A test-suite reduction approach to improving fault-localization effectiveness, Comput. Lang. Syst. Struct. 39 (3) (2013) 95–108.
[67] Y. Yu, J.A. Jones, M.J. Harrold, An empirical study of the effects of test-suite reduction on fault localization, in: Proceedings of the 30th International Conference on Software Engineering, 2008, pp. 201–210.