Information and Software Technology xxx (2014) xxx–xxx

Contents lists available at ScienceDirect

Information and Software Technology

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

Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests

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

[a] Department of Computer Science and Information Engineering, National Chiayi University, Chiayi, Taiwan
[b] Cloud System Software Institute, Institute for Information Industry, Taipei, Taiwan
[c] Department of Computer Science, Allegheny College, Meadville, PA, USA

ARTICLE INFO

Article history:
Received 16 August 2013
Received in revised form 10 April 2014
Accepted 14 April 2014
Available online xxxx

Keywords:
Software testing
Regression testing
Test suite reduction
Code coverage
Test irreplaceability

ABSTRACT

Context: In software development and maintenance, a software system may frequently be updated to meet rapidly changing user requirements. New test cases will be designed to ensure the correctness of new or modified functions, thus gradually increasing the test suite's size. Test suite reduction techniques aim to decrease the cost of regression testing by removing the redundant test cases from the test suite and then obtaining a representative set of test cases that still yield a high level of code coverage.
Objective: Most of the existing reduction algorithms focus on decreasing the test suite's size. Yet, the differences in execution costs among test cases are usually significant and it may take a lot of execution time to run a test suite consisting of a few long-running test cases. This paper presents and empirically evaluates cost-aware algorithms that can produce the representative sets with lower execution costs.
Method: We first use a cost-aware test case metric, called Irreplaceability, and its enhanced version, called EIrreplaceability, to evaluate the possibility that each test case can be replaced by others during test suite reduction. Furthermore, we construct a cost-aware framework that incorporates the concept of test irreplaceability into some well-known test suite reduction algorithms.
Results: The effectiveness of the cost-aware framework is evaluated via the subject programs and test suites collected from the Software-artifact Infrastructure Repository — frequently chosen benchmarks for experimentally evaluating test suite reduction methods. The empirical results reveal that the presented algorithms produce representative sets that normally incur a low cost to yield a high level of test coverage.
Conclusion: The presented techniques indeed enhance the capability of the traditional reduction algorithms to reduce the execution cost of a test suite. Especially for the additional Greedy algorithm, the presented techniques decrease the costs of the representative sets by 8.10–46.57%.

1. Introduction

The goal of software testing is to execute the software system, locate the faults that cause failures, and improve the quality of the software by removing the detected faults. Testing is the primary method that is widely adopted to ensure the quality of the software under development [1]. According to the IEEE definition [2], a test case is a set of input data and expected output results which are designed to exercise a specific software function or test requirement. During testing, the testers, or the test harnesses, will execute the underlying software system to either examine the associated program path or to determine the correctness of a software function. It is difficult for a single test case to satisfy all of the specified test requirements. Hence, a considerable number of test cases are usually generated and collected in a test suite [3].

If the requirement set covered by a test case overlaps the set covered by another test case, then the two test cases may be redundant to each other. At the beginning of software testing, a large number of test cases may be automatically generated by a test case generator without considering the redundancies of test cases [4,5]. Yet, because the resources allocated to a testing team are usually limited, it may be impractical to execute all of the generated test cases. If software developers can reduce the test suite by removing the redundant test cases, while still ensuring that all test requirements are satisfied by the reduced test suite, then testing may be more efficient and the effectiveness of testing in unlikely to be compromised.

* Corresponding author. Tel.: +886 5 2717227.
E-mail address: chutilin@mail.ncyu.edu.tw (C.-T. Lin).

http://dx.doi.org/10.1016/j.infsof.2014.04.013
0950-5849/

The process of removing the redundant test cases is called test suite reduction or test suite minimization [6,7].

Additionally, evolutionary development, incremental delivery, and software maintenance are common in software development [8,9]. In such development processes, the functionality of a software system may be refined to meet the customer's needs or may be delivered incrementally. Each time the software developers modify the system, they may also introduce some faults. New test cases should be added to ensure the quality of new functions. The existing test cases should also be re-executed in order to detect the faults caused by imperfect debugging. Such an activity is called regression testing [2,10]. In the process of software development, more and more test cases will be included, thus often causing some test requirements to be associated with more than one test case. To reduce the cost of regression testing, test suite reduction can also be applied to remove the redundant test cases [11].

The test suite reduction problem can be defined as follows [6,7,9]:

Given:

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

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.

Let us consider the binary relation between $T_0 = \{t_1, t_2, t_3, t_4, t_5\}$ and $R_0 = \{r_1, r_2, r_3, r_4, r_5, r_6\}$ that is shown in Table 1 as an illustration. If the set of test cases $\{t_1, t_2\}$ are executed instead of T_0, all of the requirements in R_0 are still satisfied. Thus, the test cases $\{t_3, t_4, t_5\}$ do not need to be executed. In this example, the subset $\{t_1, t_2\}$ is the minimal representative set.

In fact, the test suite minimization problem can be reduced to the minimum set cover problem [6]. Karp proved that the set cover problem is NP-complete [12]; yet, many techniques have been proposed to obtain the near-optimal solution for the test suite reduction problem. Even though the representative sets produced by these techniques are not guaranteed to be optimal, they can significantly decrease both the size of the test suite and the cost associated with its execution. However, to the best of our knowledge, most of the existing reduction algorithms ignore the significant differences in execution costs among test cases. In response to this limitation, this paper presents a cost-aware test suite reduction technique based on the concept of test irreplaceability.

The main contributions of this paper include: (C1) presenting a cost-aware test case metrics, called Irreplaceability and EIrreplaceability, based on the concept of test irreplaceability; (C2) constructing a cost-aware framework that incorporates the concept of test irreplaceability into some well-known test suite reduction algorithms; (C3) empirically evaluating the effectiveness of the presented test suite reduction techniques and evaluating whether these methods select many test cases in common during the test suite reduction process. It is important to note that, in comparison with the preliminary version of this work (i.e., [9]), this paper provides several additional contributions. Considering (C1) as an example, although the test case metric, Irreplaceability, was presented in [9], this paper further presents the enhanced version, EIrreplaceability, which strictly dominates Irreplaceability. Considering (C2), the work in [9] only explained how to use Irreplaceability to evaluate the test cases and selected the test cases based on the concept of the additional Greedy algorithm. This paper further incorporates the presented cost-aware test case metrics into two additional well-known reduction algorithms. Considering (C3), in addition to the small subject programs used in [9], this paper further includes empirical studies with a larger subject program. Moreover, this paper evaluates the common rates of the representative sets according to the concept developed by Zhong et al. in [3].

The remainder of this paper is organized as follows. Section 2 reviews some well-known test suite reduction algorithms and some cost-aware regression testing techniques. In Section 3, we discuss the differences in execution costs among test cases and explain how the differences influence test suite reduction. Based on these discussions, we show how to use test irreplaceability to evaluate each test case. In Section 4, we propose the cost-aware algorithms by incorporating test irreplaceability into the existing reduction algorithms. Section 5 reports on the results from the empirical studies. Finally, Section 6 furnishes some concluding remarks.

Table 1
An example of a test suite.

Test suite T_0	Requirement set R_0					
	r_1	r_2	r_3	r_4	r_5	r_6
t_1		•	•		•	
t_2	•			•		•
t_3					•	•
t_4		•	•	•		
t_5				•		•

2. Related work

Test suite reduction, test case prioritization, and test case selection are three primary techniques to make regression testing more efficient and effective [13]. As described in Section 1, test suite reduction techniques remove the redundant test cases and then produce a representative set of test cases that still yield a high level of code coverage. Test case prioritization techniques reorder the test cases according to some specific criteria such that the tests with better fault detection capability are executed early during the regression testing process. Test case selection techniques run a subset of the test cases that execute the modified source code in order to ensure the correctness of the functionalities of the updated program [14]. Recall that this paper aims to present and empirically evaluate the cost-aware test suite reduction algorithms. In order to introduce the state-of-the-art relevant techniques, Sections 2.1.1–2.1.3 first review three well-known test suite reduction algorithms that will be used to explain the presented technique in Section 4, and then Section 2.1.4 reviews some existing cost-aware test suite reduction techniques. To the best of our knowledge, few cost-aware test case selection techniques have been proposed in the literature, while a lot of cost-aware test case prioritization techniques have been proposed and have received considerable attention. Thus, Section 2.2 focuses on the reviews of the cost-aware regression test case prioritization techniques.

2.1. Reviews of well-known test suite reduction techniques

2.1.1. Additional Greedy algorithm

The additional Greedy algorithm [15], hereafter called Greedy for simplicity, is a well-known method for finding the near-optimal solution to the test suite reduction problem. This algorithm repeatedly moves the test which covers the most unsatisfied test

Please cite this article in press as: C.-T. Lin et al., Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests, Inform. Softw. Technol. (2014), http://dx.doi.org/10.1016/j.infsof.2014.04.013

requirements from the test suite set T to RS until all of the requirements are covered. The detailed steps are given as follows:

1. Create an empty set RS and mark all test requirements as "unsatisfied".
2. Identify a test case t in T, where t is the test case that satisfies the most unsatisfied test requirements.
3. Move the test case t from T into RS.
4. Mark the test requirements which are satisfied by t as "satisfied".
5. Repeat Steps 2–4 until all test requirements are satisfied, and then return the representative set RS.

The time complexity of the Greedy algorithm is $O(m \cdot n \cdot \min(m, n))$ [16]. Many existing test suite reduction methods are based on the concept of the Greedy algorithm [13]. In other words, many algorithms repetitively choose the "best" test case to obtain the near-optimal solution from the locally optimal solutions.

2.1.2. GE and GRE algorithms

If a test requirement only can be satisfied by a specific test case, then that test can be called the essential test case for that requirement [16]. Considering the example in Table 1, t_2 is the essential test case for the requirement r_1 because r_1 only can be satisfied by t_2. If the essential test cases are not inserted into the representative set early in the reduction process, some of the selected test cases may become redundant with a high probability. However, the Greedy algorithm does not specifically deal with the essential test cases as early as is possible. Because the Greedy algorithm does not ensure that the essential test cases are chosen first, Chen and Lau [16] proposed two algorithms, called GE and GRE, to address this problem. Here "G", "R", and "E" represent the abbreviations of "Greedy", "Redundant", and "Essential", respectively.

The GE algorithm will choose all of the essential test cases first and then will apply the Greedy algorithm to the remaining test suite. It has the same time complexity as the Greedy algorithm. The GRE algorithm is the enhanced version of GE, which includes the following strategies:

(1) Essential strategy: it will first choose all essential test cases.
(2) 1-to-1 redundant strategy: if the test case t_x can cover all the test requirements satisfied by another test case t_y, t_y is said to be 1-to-1 redundant to t_x. Considering the example in Table 1, t_5 is 1-to-1 redundant to t_2 because both the requirements satisfied by t_5 (i.e., r_4 and r_6) can also be satisfied by t_2. This strategy is used to remove the 1-to-1 redundant test cases.
(3) Greedy strategy: the Greedy algorithm will be applied to the remaining test cases.

In the test suite reduction process, the GRE algorithm will first adopt the essential strategy and then the 1-to-1 redundant strategy. Only when no essential test case can be found will the Greedy strategy then be adopted. GRE finds the optimal solution if only the essential and 1-to-1 redundant strategies are adopted during the reduction process [16]. The time complexity of the GRE algorithm is $O(\min(m, n) \cdot (m + n^2 \cdot k))$, where k is the maximum number of requirements that can be satisfied by a single test case.

2.1.3. HGS algorithm

Harrold et al. [6] also proposed a test suite reduction approach, called the HGS (i.e., the abbreviation of the authors' last names, Harrold, Gupta, and Soffa) algorithm, that has received considerable attention (e.g., [3,7,11,13,16–18]). Let S_i (for $i = 1, 2, 3, \ldots, m$) represent the subsets of T, with each subset S_i containing all of the test cases that satisfy the i-th test requirement. The HGS algorithm will determine the representative test cases for each subset and include them in the representative set. The time complexity of the HGS algorithm is $O(m \cdot (m + n) \cdot maxCard)$, where $maxCard$ is the maximum cardinality of all the S_is. The experimental results reported by Chen and Lau [17] and Zhong et al. [18] indicated that both the HGS algorithm and the GRE algorithm can significantly reduce the size of a test suite.

2.1.4. Other test suite reduction techniques

Similar to the current study, Ma et al. [19] and Smith and Kapfhammer [20] also considered the differences in execution costs among test cases when reducing a test suite. Instead of using code coverage criteria (e.g., statements, branches, define-use pairs, basic blocks, or functions), Ma et al. [19] proposed a cost-aware criterion that combines the block-based coverage (i.e., the node coverage or the segment coverage) and the execution cost of each test case. Later, Smith and Kapfhammer [20] generalized the cost-aware criterion proposed by Ma et al. [19] by considering any type of test requirement and used the generalized criterion to greedily reduce the test suite. An empirical study demonstrated that this cost-aware criterion can aid a test suite reduction method in generating a representative set with a low execution cost. This approach will be clearly described in Section 3.3. Additionally, this approach, together with Greedy, GRE, and HGS, will be empirically compared to the presented cost-aware algorithms in terms of the metrics adopted by Zhong et al. (see [3] for more details).

Yoo and Harman [21] formulated a multi-objective test suite reduction problem and illustrated this with two versions, including a two-objective formulation that combines both execution costs of test cases and statement coverage and a three-objective formulation that further considers past fault-detection history. Based on the concept of Pareto efficiency, they explained how to use the non-dominating sorting genetic algorithm, called NSGA-II, [22] and an island genetic algorithm variant of NSGA-II, called vNSGA-II, to address the two- and three-objective instances of the test suite reduction problem. Their empirical results show that the search-based algorithms outperform Greedy for smaller programs while Greedy demonstrates good effectiveness for a large program. Because of this observation, Yoo and Harman [23] further presented a hybrid variant of NSGA-II, called HNSGA-II, that combines the efficient approximation of Greedy with NSGA-II to address the multi-objective test suite reduction problem. The empirical results indicate that the approximation produced by Greedy is complemented by the intermediate solutions that are detected by NSGA-II. As a result, the proposed hybrid approach indeed works better than Greedy and NSGA-II. In addition to execution cost and code coverage, Bozkurt [24] recently further incorporated the concern about the reliability of a test suite (i.e., the validity of test inputs in the test suite) into the multi-objective test suite reduction problem and addressed the problem based on HNSGA-II. The experimental results also showed evidence for the effectiveness of HNSGA-II for analyzing the trade-off between execution cost, code coverage, and test suite reliability.

It is true that the studies in [21,23,24] are relevant to the execution cost of test cases. However, these papers intend to produce a representative set of test cases that balances the trade-offs between the execution cost and the other objectives while our paper aims to produce a representative set that decreases the execution cost but yields the same level of code coverage. Thus, it is not absolutely necessary to empirically compare the techniques presented in [21,23,24] with our presented techniques.

In addition to these algorithms reviewed in Sections 2.1.1–2.1.4, many studies (e.g., [7,11,25–30]) that have received considerable attention also aim to reduce the test suites under various considerations; the focuses of their empirical studies are diverse. Yet, it is not advisable to include all of the considerations and foci of prior

Please cite this article in press as: C.-T. Lin et al., Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests, Inform. Softw. Technol. (2014), http://dx.doi.org/10.1016/j.infsof.2014.04.013

work in this paper. Additionally, these studies did not consider the differences in the execution costs of the tests. Therefore, in order to control the scope and focus of both this exposition and the empirical study, in this section we only review and discuss the papers that are frequently cited and most relevant to the current study (i.e., [6,15,16], the studies that proposed the Greedy, GRE, and HGS algorithms, and [19–21,23,24], the studies that proposed the cost-aware test suite reduction techniques).

2.2. Cost-aware test case prioritization techniques

The average percentage of fault detected (APFD) [31], a commonly used metric for evaluating the fault detection rate of prioritized test cases, assumes that all test cases and fault severities are uniform. Elbaum et al. [32] considered the differences in execution costs among test cases and the differences in the severity of faults in practice, and thus presented a new metric, called the cost-cognizant average percentage of faults detected ($APFD_C$). They also described how to adjust several prioritization techniques to be cost-cognizant [33]. Park et al. [34] used historical information to estimate costs and fault severities, and further proposed a historical value-based technique for cost-cognizant test case prioritization. The experimental results show that this historical value-based technique works better, in terms of $APFD_C$, than functional coverage-based prioritization techniques. Recently, Huang et al. [35] proposed a cost-cognizant test case prioritization technique that uses a genetic algorithm to determine the most effective prioritization according to the historical records gathered from the latest regression testing. As a result of the experiment, the proposed cost-cognizant approach significantly outperforms the coverage-based techniques and the other history-based techniques.

In addition to the differences in execution costs among test cases, some test case prioritization studies have considered the testing time constraint or budget. Walcott et al. [36] reported that previous test case prioritization techniques did not explicitly consider a testing time constraint. Thus, they presented a time-aware test case prioritization technique that used a genetic algorithm to reorder the test cases so that regression testing will terminate in the allotted time. They also defined a new metric for evaluating the effectiveness of test case prioritization in an explicit testing time constraint. To address the similar time-constrained test case prioritization problem, Alspaugh et al. [37] applied knapsack solvers to perform time-aware prioritization. Given a testing time constraint, the reordered test suite created by the proposed knapsack-based approaches can cover the test requirements in a short time and always stop execution in a given time constraint. The empirical results also indicate that the knapsack-based approaches, without considering the overlap in test case coverage, result in a low time overhead and a moderate to high space overhead while the created prioritizations lead to a minor to moderate decrease in the effectiveness. Zhang et al. [38] presented the first time-aware test case prioritization approach that uses integer linear programming. In this study, they empirically evaluated and compared the time-aware prioritization and traditional techniques in the context of time-constrained prioritization. The results indicate that their proposed techniques outperform others, especially when the given time budget is tight.

While several time-aware test case prioritization techniques have been proposed to satisfy the time budget of regression testing, You et al. [39] analyzed whether it is effective to consider the time cost of each test case when prioritizing the regression test cases. Their empirical studies evaluate the effectiveness of the test case prioritization techniques based on two common criteria, statement coverage and fault detection, and the results indicate that the time-aware techniques are only slightly better than others.

Although the aforementioned studies do not focus on the test suite reduction problem, they clearly show that discussions on the time costs of regression testing have received considerable attention. This further confirms that it is worthwhile for this paper to focus on the cost of regression testing.

3. Test suite cost reduction

3.1. Execution costs of test cases

Most traditional test suite reduction approaches aim to decrease the number of test cases in the representative set and ignore the individual execution cost of each test case. That is, the execution costs of different test cases in a test suite are treated as if they are the same. In fact, this assumption may be unreasonable for most test suites. For example, suppose that two test cases t_1 and t_2 are applied to a program that only includes a loop. During testing, the execution associated with t_1 will repeat the loop twice while t_2 will cause the loop to be executed one thousand times. If the test requirement in this example is statement coverage or branch coverage, both t_1 and t_2 can yield 100% coverage for this code segment. However, the time required to execute t_2 is significantly greater than that required to execute t_1. In a real-world example, Mücke and Huhn reported that the execution time for different test cases should not be omitted in railway interlocking system [40]. In such a system, the executions of test cases may be related to the traffic or process control systems and the relevant actions may cause a long waiting time.

Fig. 1 shows the results from an investigation into the test case cost of the subject programs from the Software-artifact Infrastructure Repository (SIR) [41], which includes the seven Siemens suite programs [7,42] and the Space program [43]. Table 2 shows the statistics related to the execution time of the test cases. As seen from the figure and the table, the differences in execution time among test cases are significant for the SIR subject programs. Thus, it may be unsuitable to ignore the issues related to execution time when looking for the representative set of a test suite.

In addition to execution time, running test cases may lead to other resource consumptions such as memory usage, network-bandwidth, disk input/output, and energy [44]. The consumptions of these resources are also important factors to consider during cost-effective software testing and it may be helpful to take into account their differences when reducing the test suite. However, because regression tests are usually run outside of working hours, finishing the tests before the next work period should be the most important consideration of testers. The other resources may be less critical in comparison to time consumption. Therefore, even though the presented methods are general enough to handle any measurable type of resource consumption, this paper only focuses on time consumption and leaves the others for study in future work.

3.2. Execution cost of a test suite

Due to the differences in execution costs among the test cases, the representative set with the smallest number of tests may not be the one with the minimum execution cost [9]. Let us consider the code segment of function *func* in Fig. 2 as an illustration. The code segment includes 6 *if*-statements. If test requirements are defined in terms of branch coverage and b_y^T and b_y^F denote the true case and the false case for the y-th *if*-statement b_y, respectively, then all test requirements (12 possible branches) are shown in Table 3. Table 4 lists the individual execution cost (i.e., execution time) for each test case in the test suite and shows the binary relations between the test cases and the test requirements. As shown in the table, $RS_a = \{t_1, t_2, t_5, t_6\}$, with cardinality 4 (i.e., $|RS_a| = 4$)

Please cite this article in press as: C.-T. Lin et al., Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests, Inform. Softw. Technol. (2014), http://dx.doi.org/10.1016/j.infsof.2014.04.013

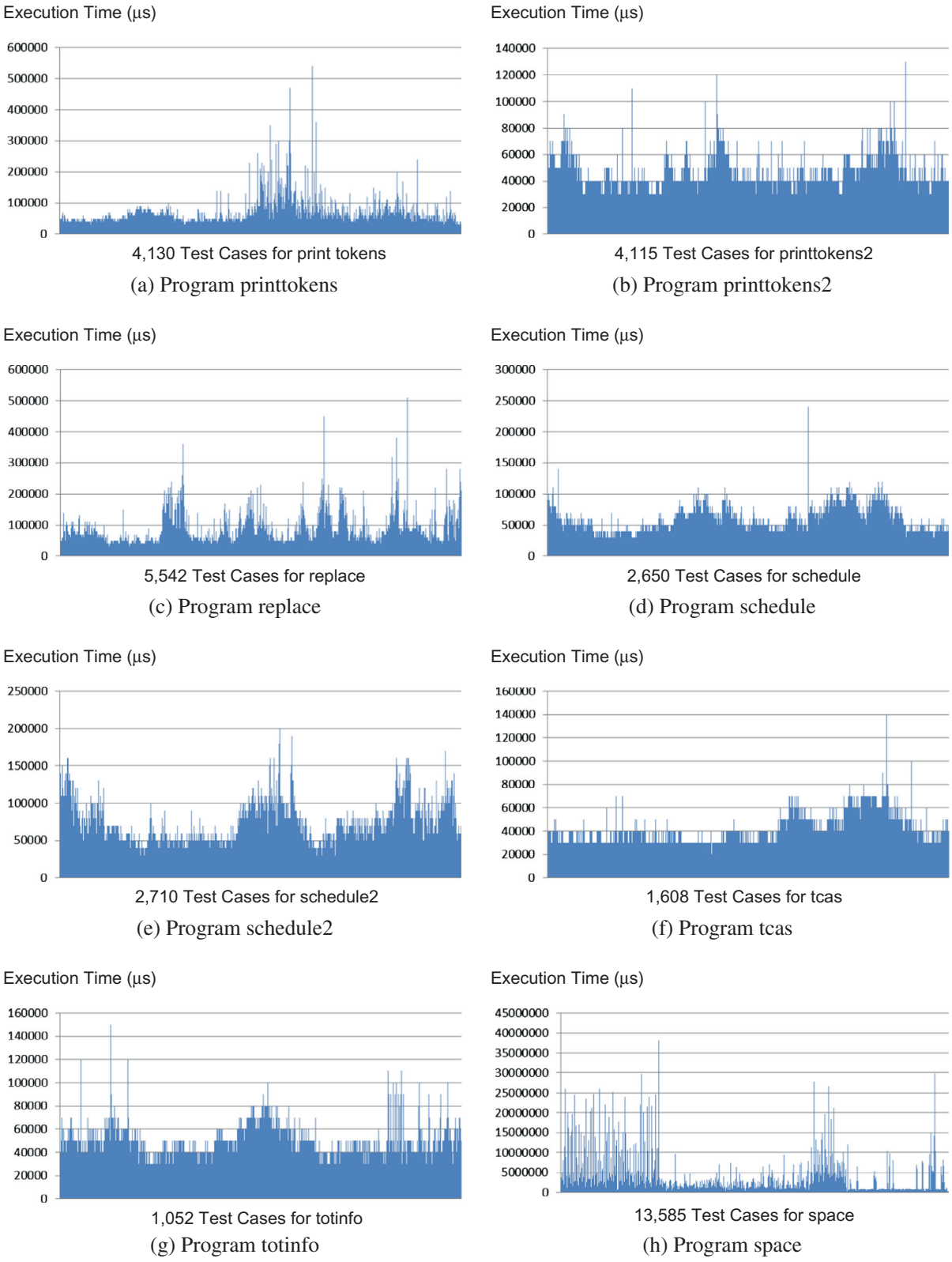

Fig. 1. Execution time distribution of the test cases created for the SIR subject programs.

and an execution cost of 700, is the minimal representative set that can achieve 100% branch coverage. In fact, if t_2 is replaced by t_3 and t_4, it can yield another representative set $RS_b = \{t_1, t_3, t_4, t_5, t_6\}$ with cardinality 5 (i.e., $|RS_b| = 5$) and cost 600. Although the cardinality of RS_b is larger than that of RS_a, the associated execution cost decreases to 600 time units. From this example, it can be seen that the representative set containing the fewest test cases may not exhibit the lowest execution time.

As such, the cost of a test should be a more important consideration for achieving cost-effective testing than the size of the test

Please cite this article in press as: C.-T. Lin et al., Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests, Inform. Softw. Technol. (2014), http://dx.doi.org/10.1016/j.infsof.2014.04.013

Table 2
The statistics related to the execution time of the test cases designed for the SIR subject programs.

Subject program	Num. of test cases	Execution time (μs)				
		Maximum	Minimum	Average	Std. deviation	Coefficient of variance (Stdev./Avg.)
printtokens	4130	540,031	20,001	51177.26	27527.31	0.54
printtokens2	4115	130,008	20,001	40291.49	11648.51	0.29
replace	5542	510,029	20,001	63838.01	35030.60	0.55
schedule	2650	240,014	20,001	53267.20	19535.76	0.37
schedule2	2710	200,011	20,001	61966.64	23904.06	0.39
tcas	1608	140,008	20,001	39255.98	12998.87	0.33
totinfo	1052	150,008	30,001	48919.14	14148.49	0.29
space	13,585	38,112,180	60,003	643380.93	1546418.47	2.40

```
1  bool func(int x, int y , int z)
2  {
3      if(z<0)                    // if-statement 1
4          return false;
5      if(x<y)                    // if-statement 2
6      {
7          if(x*y!=0)             // if-statement 3
8              x*=y;
9          else
10             x++;
11     }
12     if(x>0)                    // if-statement 4
13     {
14         x*=x;
15         if(y>5)                // if-statement 5
16             y+=y;
17     }
18     if(z>10)                   // if-statement 6
19         sleep(200);            // this statement takes 200 ms
20     sync(x+y);                 // this statement takes 100 ms
21     return true;
22 }
```

Fig. 2. Function *func*.

suite. Thus, it is necessary to consider individual execution costs when choosing the test cases. This paper proposes a cost-aware framework to enhance the test suite reduction algorithms reviewed in Sections 2.1–2.3.

3.3. Evaluating the test cases

This subsection will first review two existing metrics, called Coverage and Ratio, that are used to evaluate and choose test cases during the test suite reduction process. We will point out the possible weaknesses of these existing metrics and then propose two new metrics, called Irreplaceability and EIrreplaceability, based on the concept of test irreplaceability. EIrreplaceability is the enhanced version of Irreplaceability.

3.3.1. Reduction using the existing metrics called Coverage and Ratio

The well-known traditional algorithms reviewed in Sections 2.1.1–2.1.3 repetitively choose the "best" test cases to obtain the near-optimal solution. For example, the Greedy algorithm repetitively includes the test case that covers the maximum number of uncovered test requirements in the representative set *RS*. The Greedy strategy of the GRE algorithm and the HGS algorithm are also variations of the Greedy algorithm [13]. In fact, their common metric used to evaluate the test case can be defined as

$$Coverage(t) = |R_t|, \tag{1}$$

where R_t represents the set of uncovered test requirements satisfied by the test case t.

Similarly, the HGS algorithm also identifies the test case that exists in the unmarked subsets S_is and satisfies the most uncovered test cases. As mentioned in Sections 2.1.1–2.1.3, the goals of Greedy, GRE, and HGS are to determine a near-optimal representative set that can achieve high coverage. However, compared to the size of a test suite, the execution time of a test suite may be a more important consideration for the software developer to judge the efficiency of software testing. Thus, the coverage increase per unit of cost consumption may be an intuitive metric to evaluate the test case. Ma et al. [19] and Smith and Kapfhammer [20] evaluated the test cases using

$$Ratio(t) = \frac{Coverage(t)}{Cost(t)}, \tag{2}$$

where $Cost(t)$ represents the execution cost of the test case t; this paper calls this metric Ratio. A higher value of $Ratio(t)$ implies that the test case is expected to be more cost-effective; in contrast, a lower value of $Ratio(t)$ may indicate that the test is less desirable.

Table 3
All possible branches of the function *func* in Fig. 2.

If-statements	1		2		3		4		5		6	
Outcomes	True	False	True	False	True	False	True	False	True	False	True	False
Branches	b_1^T	b_1^F	b_2^T	b_2^F	b_3^T	b_3^F	b_4^T	b_4^F	b_5^T	b_5^F	b_6^T	b_6^F

Table 4
Test cases for function *func*.

Test cases (*x*, *y*, *z*)	Cost	b_1^T	b_1^F	b_2^T	b_2^F	b_3^T	b_3^F	b_4^T	b_4^F	b_5^T	b_5^F	b_6^T	b_6^F
t_1 (2, 2, −1)	<1	•											
t_2 (10, 20, 30)	300		•	•		•		•		•		•	
t_3 (10, 15, 8)	100		•		•			•		•			•
t_4 (−5, 5, 5)	100		•	•		•			•				•
t_5 (−5, 0, 2)	100		•	•			•		•				•
t_6 (10, 8, 20)	300		•		•			•			•	•	

Please cite this article in press as: C.-T. Lin et al., Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests, Inform. Softw. Technol. (2014), http://dx.doi.org/10.1016/j.infsof.2014.04.013

The Greedy algorithm repeatedly includes the test case with the maximum *Coverage*(t) until all test requirements are satisfied. Instead of *Coverage*(t), Smith and Kapfhammer [20] incorporated the Ratio metric into the Greedy algorithm, hereafter called $\text{Greedy}_{\text{Ratio}}$, which repeatedly includes the test case with the maximum *Ratio*(t) until all of the test requirements are satisfied. The time complexity of $\text{Greedy}_{\text{Ratio}}$ is $O(m \cdot n \cdot \min(m, n))$.

Table 5
Example 1 – an example including the test requirements, the test cases, and the associated execution costs.

Test suite T_1		Requirement set R_1		
No.	Cost	r_1	r_2	r_3
t_1	6	•	•	
t_2	2			•
t_3	1		•	
t_4	3	•		

The $\text{Greedy}_{\text{Ratio}}$ and Greedy algorithms are compared via Example 1 (i.e., the test suite in Table 5 that satisfies three requirements with four tests that have varying cost). Table 6 demonstrates the detailed steps of applying Greedy to reduce the test suite in the example. This table shows the set of test requirements R, the set of test cases T, and the associated execution costs. In addition, the changes to R and the representative set RS_1 are shown step by step. Similarly, Table 7 demonstrates the steps of applying $\text{Greedy}_{\text{Ratio}}$ to the same example. We find that Greedy reduces the test suite to the representative set $RS_1 = \{t_1, t_2\}$ with cost 8. If the adopted test suite reduction algorithm is replaced by $\text{Greedy}_{\text{Ratio}}$, we can obtain the representative set $RS_2 = \{t_2, t_3, t_4\}$ with cost 6. Although $|RS_2| > |RS_1|$, the total execution cost associated with running RS_2 is lower than that associated with running RS_1. Thus, $\text{Greedy}_{\text{Ratio}}$ outperforms Greedy from the standpoint of execution cost in this example, thus indicating that it may be

Table 6
Reduction steps when applying Greedy to Example 1.

Initial	$T_1 = \{t_1, t_2, t_3, t_4\}$, $R_1 = \{r_1, r_2, r_3\}$, $RS_1 = \{\}$					
	Test cases	Cost	r_1	r_2	r_3	*Coverage*(t)
	t_1	6	•	•		2
	t_2	2			•	1
	t_3	1		•		1
	t_4	3	•			1
Step 1	$T_1 = \{t_2, t_3, t_4\}$, $R_1 = \{r_3\}$, $RS_1 = \{t_1\}$					
	Test cases	Cost	–	–	r_3	*Coverage*(t)
	$\boxed{t_1}$	6	–	–		–
	t_2	2	–	–	•	1
	t_3	1	–	–		0
	t_4	3	–	–		0
Step 2	$T_1 = \{t_3, t_4\}$, $R_1 = \{\}$, $RS_1 = \{t_1, t_2\}$, Cost of $RS_1 = 8$					
	Test cases	Cost	–	–	–	*Coverage*(t)
	$\boxed{t_1}$	6	–	–	–	–
	$\boxed{t_2}$	2	–	–	–	–
	t_3	1	–	–	–	0
	t_4	3	–	–	–	0

Table 7
Reduction steps when applying $\text{Greedy}_{\text{Ratio}}$ to Example 1.

Initial	$T_1 = \{t_1, t_2, t_3, t_4\}$, $R_1 = \{r_1, r_2, r_3\}$, $RS_2 = \{\}$					
	Test cases	Cost	r_1	r_2	r_3	*Ratio*(t)
	t_1	6	•	•		0.33
	t_2	2			•	0.50
	t_3	1		•		1.00
	t_4	3	•			0.33
Step 1	$T_1 = \{t_1, t_2, t_4\}$, $R_1 = \{r_1, r_3\}$, $RS_2 = \{t_3\}$					
	Test cases	Cost	r_1	–	r_3	*Ratio*(t)
	t_1	6	•	–		0.17
	t_2	2		–	•	0.50
	$\boxed{t_3}$	1		–		–
	t_4	3	•	–		0.33
Step 2	$T_1 = \{t_1, t_4\}$, $R_1 = \{r_1\}$, $RS_2 = \{t_2, t_3\}$					
	Test cases	Cost	r_1	–	–	*Ratio*(t)
	t_1	6	•	–	–	0.17
	$\boxed{t_2}$	2		–	–	–
	$\boxed{t_3}$	1		–	–	–
	t_4	3	•	–	–	0.33
Step 3	$T_1 = \{t_1, t_4\}$, $R_1 = \{\}$, $RS_2 = \{t_2, t_3, t_4\}$, Cost of $RS_2 = 6$					
	Test cases	Cost	–	–	–	*Ratio*(t)
	t_1	6	–	–	–	0.17
	$\boxed{t_2}$	2	–	–	–	–
	$\boxed{t_3}$	1	–	–	–	–
	$\boxed{t_4}$	3	–	–	–	–

Please cite this article in press as: C.-T. Lin et al., Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests, Inform. Softw. Technol. (2014), http://dx.doi.org/10.1016/j.infsof.2014.04.013

suitable for the Greedy algorithm to replace the Coverage metric by Ratio to further decrease the execution cost.

Yet, in preliminary studies, we found some circumstances in which $Greedy_{Ratio}$ did not produce the best reduced test suite. As an illustrative example, let us consider Example 2 shown in Table 8. Tables 9 and 10 demonstrate the steps that Greedy and $Greedy_{Ratio}$ would take to reduce the test suite of Example 2, respectively. Contrary to Example 1, $RS_4 = \{t_1, t_2, t_3\}$ with cost 14 (reduced by $Greedy_{Ratio}$) takes more cost than $RS_3 = \{t_2, t_3\}$ with cost 10 (reduced by Greedy). Therefore, the Ratio metric does not always lead to the test suite with the lowest execution time.

Considering Example 2, the representative set $\{t_2, t_3\}$, with cost 10, is enough to satisfy all of the test requirements. In other words, the cost of a representative set will not be the lowest if test cases other than t_2 and t_3 are chosen during the test suite reduction process. Table 10 indicates that $Greedy_{Ratio}$ identified t_1 as the best candidate at Step 1 because it provides the maximum $Ratio(t)$. Thus, t_1 was included in the representative set, which means that $Greedy_{Ratio}$ will not create the representative set with the lowest execution cost in this case. Actually, the test requirements satisfied by t_1 (i.e., $\{r_1, r_2, r_3\}$) are also satisfied by other test cases. For example, t_2 satisfies $\{r_2, r_3\}$ whereas t_3 satisfies $\{r_1\}$. If $\{t_1\}$ is replaced by $\{t_2, t_3\}$, although the execution cost at the early step of reduction increases, two extra test requirements, r_4 and r_5, can also be satisfied. For examples like this one, $Greedy_{Ratio}$ cannot generate the representative set with the lowest cost because the ratio metric did not take into account the aforementioned trade-off.

Table 8
Example 2– An example including the test requirements, the test cases, and the associated execution costs.

Test suite T_2		Requirement set R_2					
No.	Cost	r_1	r_2	r_3	r_4	r_5	r_6
t_1	4	•	•	•			
t_2	7		•	•	•	•	
t_3	3	•					•
t_4	4			•			•

Table 9
Reduction steps when applying Greedy to Example 2.

Initial	$T_2 = \{t_1, t_2, t_3, t_4\}$, $R_2 = \{r_1, r_2, r_3, r_4, r_5, r_6\}$, $RS_3 = \{\}$								
	Test cases	Cost	r_1	r_2	r_3	r_4	r_5	r_6	$Coverage(t)$
	t_1	4	•	•	•				3
	t_2	7		•	•	•	•		4
	t_3	3	•					•	2
	t_4	4			•			•	2
Step 1	$T_2 = \{t_1, t_3, t_4\}$, $R_2 = \{r_1, r_6\}$, $RS_3 = \{t_2\}$								
	Test cases	Cost	r_1	–	–	–	–	r_6	$Coverage(t)$
	t_1	4	•	–	–	–	–		1
	$\boxed{t_2}$	7		–	–	–	–		–
	t_3	3	•	–	–	–	–	•	2
	t_4	4		–	–	–	–	•	1
Step 2	$T_2 = \{t_1, t_4\}$, $R_2 = \{\}$, $RS_3 = \{t_2, t_3\}$, Cost of $RS_3 = 10$								
	Test cases	Cost	–	–	–	–	–	–	$Coverage(t)$
	t_1	4	–	–	–	–	–	–	0
	$\boxed{t_2}$	7	–	–	–	–	–	–	–
	$\boxed{t_3}$	3	–	–	–	–	–	–	–
	t_4	4	–	–	–	–	–	–	0

Table 10
Reduction steps when applying $Greedy_{Ratio}$ to Example 2.

Initial	$T_2 = \{t_1, t_2, t_3, t_4\}$, $R_2 = \{r_1, r_2, r_3, r_4, r_5, r_6\}$, $RS_4 = \{\}$								
	Test cases	Cost	r_1	r_2	r_3	r_4	r_5	r_6	$Ratio(t)$
	t_1	4	•	•	•				0.75
	t_2	7		•	•	•	•		0.57
	t_3	3	•					•	0.67
	t_4	4			•			•	0.50
Step 1	$T_2 = \{t_2, t_3, t_4\}$, $R_2 = \{r_4, r_5, r_6\}$, $RS_4 = \{t_1\}$								
	Test cases	Cost	–	–	–	r_4	r_5	r_6	$Ratio(t)$
	$\boxed{t_1}$	4	–	–	–				–
	t_2	7	–	–	–	•	•		0.29
	t_3	3	–	–	–			•	0.33
	t_4	4	–	–	–			•	0.25
Step 2	$T_2 = \{t_2, t_4\}$, $R_2 = \{r_4, r_5\}$, $RS_4 = \{t_1, t_3\}$								
	Test cases	Cost	–	–	–	r_4	r_5	–	$Ratio(t)$
	$\boxed{t_1}$	4	–	–	–			–	–
	t_2	7	–	–	–	•	•	–	0.29
	$\boxed{t_3}$	3	–	–	–			–	–
	t_4	4	–	–	–			–	0
Step 3	$T_2 = \{t_4\}$, $R_2 = \{\}$, $RS_4 = \{t_1, t_2, t_3\}$, Cost of $RS_4 = 14$								
	Test cases	Cost	–	–	–	–	–	–	$Ratio(t)$
	$\boxed{t_1}$	4	–	–	–	–	–	–	–
	$\boxed{t_2}$	7	–	–	–	–	–	–	–
	$\boxed{t_3}$	3	–	–	–	–	–	–	–
	t_4	4	–	–	–	–	–	–	0

Please cite this article in press as: C.-T. Lin et al., Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests, Inform. Softw. Technol. (2014), http://dx.doi.org/10.1016/j.infsof.2014.04.013

3.3.2. Reduction using the cost-aware metric called Irreplaceability

If the test requirement r that is satisfied by the test case t can also be satisfied by many other test cases, there is a high probability that r can still be satisfied even though t is not included in the representative set. Thus, we posit that t has a higher replaceability with respect to r in this case. According to our initial observations from the reduction steps in Table 10, we surmise that:

1. A representative set may not have the lowest execution cost if it includes test cases with high replaceability.
2. Because the Ratio metric only considers the code coverage and execution cost of a test case, the test cases with high replaceability frequently may be selected for inclusion in the representative set.

In prior work, Jones and Harrold [27] pointed out that test suite reduction algorithms may choose a test case according to its contribution, or goodness, based on some characteristics of a program. One measure of the contribution of a test case t to the test suite T can be defined as

$$ContributionToSuite(t) = \sum_{s=1}^{m} Contribution(t, r_s), \quad (3)$$

where $R = \{r_1, r_2, \ldots, r_m\}$ represents all of the test requirements, r_s is the s-th test requirement in R, and

$$Contribution(t, r_s) = \begin{cases} 0, & \text{if } t \text{ cannot satisfy } r_s, \\ \frac{1}{\text{the number of test cases that satisfy } r_s}, & \text{if } t \text{ satisfies } r_s. \end{cases} \quad (4)$$

In fact, the number of test cases that satisfy the test requirement r_s in Eq. (4) is positively related to the replaceability of t with respect to r_s. A higher value of replaceability indicates that more test cases can be used to replace t while maintaining the original test coverage. In contrast, a higher value of irreplaceability means that it is not easy to find other test cases to replace t. Consequently, we use Eq. (3) to evaluate the irreplaceability of test cases in the test suite. Based on this concept, we combine the execution cost of a test with Eq. (3) and evaluate the irreplaceability of test cases by [9]

$$Irreplaceability(t) = \frac{\sum_{s=1}^{m} Contribution(t, r_s)}{Cost(t)}. \quad (5)$$

Fig. 3 shows the procedure used to calculate the value of Eq. (5). The procedure accepts the following three input parameters: (1) t – the test case to be evaluated; (2) CT – the set of available test cases which include t; (3) UR – the set of test requirements which are not yet satisfied. At the beginning, the procedure first checks whether t belongs to the set CT or not (Lines 7–8). If not, it will return an error message and interrupt the procedure's execution. If the condition holds, the for-each loop (Lines 10–17) will identify all of the test requirements that can be satisfied by t but have not yet been covered by the representative set. For each test requirement r, the procedure determines how many test cases which are not in the representative set yet can satisfy r, and then evaluates the irreplaceability of each test case on r (Lines 14–15). Based on the irreplaceability of the test case with respect to each test requirement, the irreplaceability of each test case to the test suite can then be obtained.

In Section 3.3.1, we have shown how to incorporate *Coverage*(t) and *Ratio*(t) into the Greedy algorithm to evaluate and choose the test cases. Similarly, we can also incorporate *Irreplaceability*(t) into the Greedy algorithm to evaluate test cases and repeatedly choose the test with the maximum value until all of the test requirements are satisfied. The Greedy algorithm integrated with

```
1  algorithm EvaluateIrreplaceability
2  input     t: a test case
3            CT: the set of considered test cases
4            UR: the set of uncovered requirements
5  output    the value of Irreplaceability(t), i.e., Eq. (5)
6  begin
7      if ( t ∉ CT)
8          return ERROR;
9      irreplaceability = 0;
10     for each (requirement r covered by t)
11     {
12         if (r ∈ UR)
13         {
14             covNum = the number of test cases in CT covering r;
15             irreplaceability += 1 / covNum;
16         }
17     }
18     return  irreplaceability / Cost(t);
19 end
```

Fig. 3. The pseudo code for evaluating the Irreplaceability of a test case.

Irreplaceability(t) is called $Greedy_{Irreplaceability}$ in this paper. The time complexity of $Greedy_{Irreplaceability}$ is $O(m \cdot n \cdot \min(m, n) \cdot k)$. Recall that k indicates the maximum number of requirements that can be satisfied by a single test case. In comparison with the time complexities of Greedy and $Greedy_{Ratio}$, the worst-case time complexity of $Greedy_{Irreplaceability}$ is not much worse. Table 11 shows the steps associated with applying $Greedy_{Irreplaceability}$ to Example 2. According to this table, it is evident that RS_5={t_2, t_3} with cost 10 (reduced by $Greedy_{Irreplaceability}$) takes less cost than RS_4 with cost 14 (reduced by $Greedy_{Ratio}$).

3.3.3. Reduction using the enhanced cost-aware metric called EIrreplaceability

Although the $Greedy_{Irreplaceability}$ algorithm can yield the representative set with the lowest cost in Example 2, there is still room for improvement. As an illustrative example, let us consider Example 3 shown in Table 12, in which eight requirements are satisfied by five test cases. In this example, the representative set with the lowest cost is {t_2, t_3, t_4} with cost 28. According to the steps of applying $Greedy_{Irreplaceability}$ to Example 3 shown in Table 13, $Greedy_{Irreplaceability}$ creates the representative set RS_6={t_1, t_2, t_3, t_4} with cost 32. Compared to the representative set with the lowest cost, {t_2, t_3, t_4}, t_1 is also included. In fact, the test requirement satisfied by t_1 can also be satisfied by other test cases. On the other hand, it should be noted that r_7 and r_8 only can be satisfied by t_3 and t_4, respectively. Thus, here we call t_3 and t_4 the essential test cases according to the concept of the GRE algorithm. That is, it is impossible to exclude t_3 and t_4 if the representative set can satisfy all test requirements. In addition to r_7 and r_8, many requirements, such as r_2, r_3, r_4, r_5, and r_6, will also be satisfied early in the reduction process if t_3 and t_4 are chosen at the beginning. Consequently, many of the non-essential test cases may become redundant so that they will not be placed into the representative set, thus potentially supporting a further decrease in total execution cost. As mentioned in Section 2.2, the essential test cases should be selected as early as is possible during the test suite reduction process. Otherwise, the representative set may, with high probability, include redundant test cases. Yet, to the best of our knowledge, all test suite reduction algorithms that consider the execution cost of test cases do not take into account the influence of the essential test cases.

Now we will enhance the procedure in Fig. 3 by considering the essential test cases. In Fig. 3, if the value of *covNum* (Line 14) equals 1, it indicates that a specific test requirement r is satisfied by one test case t only (i.e., t is an essential test case). Because the essential test cases should be added to the representative set as early as is possible, we must assign t a large irreplaceability value to ensure that the algorithm will select t early. Thus, the enhanced cost-aware metric used to evaluate the test case is defined as

Please cite this article in press as: C.-T. Lin et al., Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests, Inform. Softw. Technol. (2014), http://dx.doi.org/10.1016/j.infsof.2014.04.013

Table 11
Reduction steps when applying $\text{Greedy}_{\text{Irreplaceability}}$ to Example 2.

Initial	$T_2 = \{t_1, t_2, t_3, t_4\}$, $R_2 = \{r_1, r_2, r_3, r_4, r_5, r_6\}$, $RS_5 = \{\}$							
	Test cases	Cost	r_1	r_2	r_3	r_4	r_5	r_6	*Irreplaceability*(t)
	t_1	4	•	•	•				0.33
	t_2	7		•	•	•	•		0.40
	t_3	3	•					•	0.33
	t_4	4			•			•	0.21
Step 1	$T_2 = \{t_1, t_3, t_4\}$, $R_2 = \{r_1, r_6\}$, $RS_5 = \{t_2\}$								
	Test cases	Cost	r_1	–	–	–	–	r_6	*Irreplaceability*(t)
	t_1	4	•	–	–	–	–		0.13
	$\boxed{t_2}$	7		–	–	–	–		–
	t_3	3	•	–	–	–	–	•	0.33
	t_4	4		–	–	–	–	•	0.13
Step 2	$T_2 = \{t_1, t_4\}$, $R_2 = \{\}$, $RS_5 = \{t_2, t_3\}$, Cost of $RS_5 = 10$								
	Test cases	Cost	–	–	–	–	–	–	*Irreplaceability*(t)
	t_1	4	–	–	–	–	–	–	0.00
	$\boxed{t_2}$	7	–	–	–	–	–	–	–
	$\boxed{t_3}$	3	–	–	–	–	–	–	–
	t_4	4	–	–	–	–	–	–	0.00

$$EIrreplaceability(t) = \begin{cases} \infty, & \exists s,\ 1 \leqslant s \leqslant m,\ r_s \text{ can be satisfied by } t \text{ only;} \\ \frac{\sum_{s=1}^{m} Contribution(t, r_s)}{Cost(t)}, & \text{otherwise.} \end{cases} \quad (6)$$

To evaluate Eq. (6), we propose the new procedure *EvaluateEnhancedIrreplaceability* in Fig. 4 which is built upon the procedure *EvaluateIrreplaceability* in Fig. 3. Compared to *EvaluateIrreplaceability*,

Table 12
Example 3– An example including the test requirements, the test cases, and the associated execution costs.

Test suite T_3		Requirement set R_3							
No.	Cost	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8
t_1	4	•	•			•			
t_2	3	•					•		
t_3	10		•	•	•			•	
t_4	15					•	•		•
t_5	10			•					

Table 13
Reduction steps when applying $\text{Greedy}_{\text{Irreplaceability}}$ to Example 3.

Initial	$T_3 = \{t_1, t_2, t_3, t_4, t_5\}$, $R_3 = \{r_1, r_2, r_3, r_4, r_5, r_6\ r_7, r_8\}$, $RS_6 = \{\}$									
	Test cases	Cost	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	*Irreplaceability*(t)
	t_1	4	•	•			•				0.38
	t_2	3	•					•			0.33
	t_3	10		•	•	•			•		0.30
	t_4	15					•	•		•	0.13
	t_5	10			•						0.05
Step 1	$T_3 = \{t_2, t_3, t_4, t_5\}$, $R_3 = \{r_3, r_4, r_6\ r_7, r_8\}$, $RS_6 = \{t_1\}$									
	Test cases	Cost	–	–	r_3	r_4	–	r_6	r_7	r_8	*Irreplaceability* (t)
	$\boxed{t_1}$	4	–	–			–				–
	t_2	3	–	–			–	•			0.17
	t_3	10	–	–	•	•	–		•		0.25
	t_4	15	–	–			–	•		•	0.10
	t_5	10	–	–	•		–				0.05
Step 2	$T_3 = \{t_2, t_4, t_5\}$, $R_3 = \{r_6, r_8\}$, $RS_6 = \{t_1, t_3\}$									
	Test cases	Cost	–	–	–	–	–	r_6	–	r_8	*Irreplaceability* (t)
	$\boxed{t_1}$	4	–	–	–	–	–		–		–
	t_2	3	–	–	–	–	–	•	–		0.17
	$\boxed{t_3}$	10	–	–	–	–	–		–		–
	t_4	15	–	–	–	–	–	•	–	•	0.10
	t_5	10	–	–	–	–	–		–		0
Step 3	$T_3 = \{t_4, t_5\}$, $R_3 = \{r_8\}$, $RS_6 = \{t_1, t_2, t_3\}$									
	Test cases	Cost	–	–	–	–	–	–	–	r_8	*Irreplaceability* (t)
	$\boxed{t_1}$	4	–	–	–	–	–	–	–		–
	$\boxed{t_2}$	3	–	–	–	–	–	–	–		–
	$\boxed{t_3}$	10	–	–	–	–	–	–	–		–
	t_4	15	–	–	–	–	–	–	–	•	0.07
	t_5	10	–	–	–	–	–	–	–		0
Step 4	$T_3 = \{t_5\}$, $R_3 = \{\}$, $RS_6 = \{t_1, t_2, t_3, t_4\}$, Cost of $RS_6 = 32$									
	Test cases	Cost	–	–	–	–	–	–	–	–	*Irreplaceability* (t)
	$\boxed{t_1}$	4	–	–	–	–	–	–	–	–	–
	$\boxed{t_2}$	3	–	–	–	–	–	–	–	–	–
	$\boxed{t_3}$	10	–	–	–	–	–	–	–	–	–
	$\boxed{t_4}$	15	–	–	–	–	–	–	–	–	–
	t_5	10	–	–	–	–	–	–	–	–	0

```
1  algorithm EvaluateEnhancedIrreplaceability
2  input     t: a test case
3            CT: the set of considered test cases
4            UR: the set of uncovered requirements
5  output    the value of EIrreplaceability(t), i.e., Eq. (6)
6  begin
7      if ( t ∉ CT)
8          return ERROR;
9      eirreplaceability = 0;
10     for each (requirement r covered by t)
11     {
12         if ( r ∈ UR)
13         {
14             covNum = the number of test cases in CT covering r;
15             if (covNum == 1)    // t is an essential test case
16             {
17                 eirreplaceability = ∞;
18                 break;
19             }
20             eirreplaceability += 1 / covNum;
21         }
22     }
23     return    eirreplaceability / Cost(t);
24 end
```

Fig. 4. The pseudo code for evaluating the EIrreplaceability of a test case.

the new procedure *EvaluateEnhancedIrreplaceability* will check whether the number of test cases satisfying r equals 1. If the value equals 1, we let $EIrreplaceability(t)=\infty$ and exit the for-each loop, thus indicating that t is an essential test case that cannot be replaced by any other test cases. Incorporating $EIrreplaceability(t)$ into the traditional Greedy algorithm, we obtain $Greedy_{EIrreplaceability}$. The time complexity of $Greedy_{EIrreplaceability}$ is $O(m \cdot n \cdot \min(m, n) \cdot k)$. Please notice that it has the same time complexity as $Greedy_{Irreplaceability}$ and is not much more costly than Greedy and $Greedy_{Ratio}$ in the worst case. Table 14 demonstrates the reduction steps of applying $Greedy_{EIrreplaceability}$ to Example 3. As seen in the table, it creates the representative set $RS_7 = \{t_2, t_3, t_4\}$ which, for this example, has the lowest execution cost of 28.

4. Cost-aware test suite reduction algorithms

In this section, we will develop the cost-aware test suite reduction algorithms by incorporating the cost-aware metrics (i.e., Ratio, Irreplaceability, and EIrreplaceability) into the traditional test suite reduction algorithms.

4.1. Incorporating the cost-aware metrics into the Greedy algorithm

Inserting the cost-aware metrics into the Greedy algorithm is intuitive; we have applied $Greedy_{Ratio}$, $Greedy_{Irreplaceability}$, and $Greedy_{EIrreplaceability}$ to Examples 2 and 3 in Section 3. Fig. 5 shows the pseudo code of $Greedy_{EIrreplaceability}$ as an illustrative example. Compared to the Greedy algorithm, we modify Line 10 of Fig. 5 to consider the influence of execution time on test suite reduction. The pseudo code of $Greedy_{EIrreplaceability}$ is described as follows:

Step 1. Determine the test case t with the maximum value of $EIrreplaceability(t)$.
Step 2. Move the test case t from T to RS.
Step 3. Remove from R the requirements that are satisfied by t.
Step 4. Repeat Steps 1–3 until all of the requirements are satisfied.

It is important to observe that, if the test cases are sorted according to the value of irreplaceability at the beginning of the while-loop (Lines 8–14), the time needed to execute Step 1 will significantly decrease, and the time required to finish the algorithm will decrease as well. Similarly, if we replace $EIrreplaceability(t)$ by $Ratio(t)$ and $Irreplaceability(t)$, we can obtain the $Greedy_{Ratio}$ and $Greedy_{Irreplaceability}$ algorithms, respectively. As shown in Section 3.3, the time complexities of $Greedy_{Ratio}$, $Greedy_{Irreplaceability}$ and $Greedy_{EIrreplaceability}$ are $O(m \cdot n \cdot \min(m, n))$, $O(m \cdot n \cdot \min(m, n) \cdot k)$ and $O(m \cdot n \cdot \min(m, n) \cdot k)$, respectively.

Table 14
Reduction steps when applying $Greedy_{EIrreplaceability}$ to Example 3.

Initial	$T_3 = \{t_1, t_2, t_3, t_4, t_5\}$, $R_3 = \{r_1, r_2, r_3, r_4, r_5, r_6\ r_7, r_8\}$, $RS_7 = \{\}$										
	Test cases	Cost	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8	$EIrreplaceability(t)$
	t_1	4	•	•			•				0.38
	t_2	3	•					•			0.33
	t_3	10		•	•	•			•		∞
	t_4	15					•	•		•	∞
	t_5	10			•						0.05
Step 1	$T_3 = \{t_1, t_2, t_4, t_5\}$, $R_3 = \{r_1, r_5, r_6, r_8\}$, $RS_7 = \{t_3\}$										
	Test cases	Cost	r_1	–	–	–	r_5	r_6	–	r_8	$EIrreplaceability(t)$
	t_1	4	•	–	–	–	•		–		0.25
	t_2	3	•	–	–	–		•	–		0.33
	$\boxed{t_3}$	10		–	–	–			–		–
	t_4	15		–	–	–	•	•	–	•	∞
	t_5	10		–	–	–			–		0
Step 2	$T_3 = \{t_1, t_2, t_5\}$, $R_3 = \{r_1\}$, $RS_7 = \{t_3, t_4\}$										
	Test cases	Cost	r_1	–	–	–	–	–	–	–	$EIrreplaceability(t)$
	t_1	4	•	–	–	–	–	–	–	–	0.13
	t_2	3	•	–	–	–	–	–	–	–	0.17
	$\boxed{t_3}$	10		–	–	–	–	–	–	–	–
	$\boxed{t_4}$	15		–	–	–	–	–	–	–	–
	t_5	10		–	–	–	–	–	–	–	0
Step 3	$T_3 = \{t_1, t_5\}$, $R_3 = \{\}$, $RS_7 = \{t_2, t_3, t_4\}$, Cost of $RS_7 = 28$										
	Test cases	Cost	–	–	–	–	–	–	–	–	$EIrreplaceability(t)$
	t_1	4	–	–	–	–	–	–	–	–	0
	$\boxed{t_2}$	3	–	–	–	–	–	–	–	–	–
	$\boxed{t_3}$	10	–	–	–	–	–	–	–	–	–
	$\boxed{t_4}$	15	–	–	–	–	–	–	–	–	–
	t_5	10	–	–	–	–	–	–	–	–	0

Please cite this article in press as: C.-T. Lin et al., Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests, Inform. Softw. Technol. (2014), http://dx.doi.org/10.1016/j.infsof.2014.04.013

```
1  algorithm Greedy_EIrreplaceability
2  input     T: the set of test cases
3            R: the set of requirements
4            S: the relation between T and R, S={(t, r)| t satisfies r, t∈ T, and r∈ R}
5  output    RS: a representative set of T
6  begin
7      RS = { };
8      while (R is not empty)
9      {
10         t = the test case in T that has the max. return value of EvaluateEnhancedIrreplaceability(t,T,R);
11         RS = RS ∪ {t};
12         T = T − {t};
13         R = R − the set including all of the requirements covered by t;
14     }
15     return RS;
16 end
```

Fig. 5. The $\text{Greedy}_{\text{EIrreplaceability}}$ algorithm.

4.2. Incorporating the cost-aware metrics into the GRE algorithm

In this section, we propose the cost-aware GRE algorithms (i.e., $\text{GRE}_{\text{Ratio}}$, $\text{GRE}_{\text{Irreplaceability}}$, and $\text{GRE}_{\text{EIrreplaceability}}$) by incorporating the cost-aware metrics into the original GRE algorithm. Here we consider $\text{GRE}_{\text{EIrreplaceability}}$ as an illustration. The pseudo code of $\text{GRE}_{\text{EIrreplaceability}}$ is provided in Fig. 6, and the detailed algorithm is described as follows:

Step 1. Determine all essential test cases, and move them from T to RS.
Step 2. Remove the requirements that are satisfied by the essential test cases from R.
Step 3. Remove all of the 1-to-1 redundant test cases from R.
Step 4. Calculate the value of $EIrreplaceability(t)$ for all test cases t remaining in T and sort the test cases in descending order.
Step 5. Determine all essential test cases and move them from T to the set *list*. Please notice that, although the essential test cases are removed in Step 1, removing the one-to-one redundant test cases in Step 5 may yield some new essential test cases.
Step 6. Move the test case with the maximum value of irreplaceability from T to *list* (i.e., the Greedy strategy).

```
1  algorithm GRE_EIrreplaceability
2  input     T: the set of test cases
3            R: the set of requirements
4            S: the relation between T and R, S={(t, r)| t satisfies r, t∈ T, and r∈ R}
5  output    RS: a representative set of T
6  begin
7      // essential strategy
8      RS = the set including all of the essential test cases in T;
9      R = R − the set including all of the requirements covered by RS;
10     T = T − RS;
11
12     while (R is not empty)
13     {
14       list = { };
15
16       // 1-to-1 redundant strategy
17       T = T − the set including all of the 1-to-1 redundant test cases in T;
18
19       sort the test cases in T in descending order according to
20                          the return value of EvaluateEnhancedIrreplaceability(t,T,R);
21
22       // essential strategy
23       if (T contains essential test cases)
24           list = the set including all of the essential test cases in T;
25
26       // Greedy strategy
27       else
28           list = the set including the first test case in T;
29
30       RS = RS ∪ list;
31       T = T − list;
32       R = R − the set including all of the requirements covered by list;
33     }
34     return RS;
35 end
```

Fig. 6. The $\text{GRE}_{\text{EIrreplaceability}}$ algorithm.

Step 7. Check the set R, deleting the requirements that are satisfied by the test cases in *list* and moving all test cases from *list* to RS.
Step 8. Repeat Steps 3–7 until all requirements are satisfied (i.e., until R becomes empty).

In Fig. 6, the highlighted part of $\text{GRE}_{\text{EIrreplaceability}}$ (i.e., Line 19) is different from the GRE algorithm. In the GRE algorithm, the test cases will be sorted before conducting the 1-to-1 redundant strategy. However, because the 1-to-1 redundant strategy will remove some test cases, the irreplaceability of the remaining test cases may change. If the sorting operation takes place before the 1-to-1 redundant strategy, the sorting operation will be repeated again. As such, the first sorting operation is unnecessary. Additionally, the sorting operation does not impact the 1-to-1 redundant strategy. Thus, we can exchange the order of the sorting and the 1-to-1 redundant strategy in $\text{GRE}_{\text{EIrreplaceability}}$. The time complexity of $\text{GRE}_{\text{EIrreplaceability}}$ is $O(\min(m, n) \cdot (m + n^2 \cdot k) \cdot k)$. Similar to the $\text{Greedy}_{\text{EIrreplaceability}}$ algorithm, $\text{GRE}_{\text{Ratio}}$ and $\text{GRE}_{\text{Irreplaceability}}$ can be easily obtained by substituting $Ratio(t)$ and $Irreplaceability(t)$ for $EIrreplaceability(t)$, respectively. The time complexities of $\text{GRE}_{\text{Ratio}}$ and $\text{GRE}_{\text{Irreplaceability}}$ are $O(\min(m, n) \cdot (m + n^2 \cdot k))$ and $O(\min(m, n) \cdot (m + n^2 \cdot k) \cdot k)$, respectively.

4.3. Incorporating the cost-aware metrics into the HGS algorithm

As mentioned in Section 2, the HGS algorithm can be divided into the main program and the subprogram. The main program iteratively chooses a set of candidate test cases and calls the subprogram to find and return the best one. Here, we integrate the cost-aware test-case metrics with the HGS algorithm and propose the cost-aware HGS algorithms (i.e., the $\text{HGS}_{\text{Ratio}}$, $\text{HGS}_{\text{Irreplaceability}}$, and $\text{HGS}_{\text{EIrreplaceability}}$ algorithms). Let us consider $\text{HGS}_{\text{EIrreplaceability}}$ as an illustration. Compared with the pseudo code of HGS that can be found in [6], the main program of the $\text{HGS}_{\text{EIrreplaceability}}$ algorithm shown in Fig. 7a is identical, but the highlighted part of the subprogram shown in Fig. 7b is modified. The $\text{HGS}_{\text{EIrreplaceability}}$ algorithm is described as follows:

Main program:

Step 1. Let the representative set RS be the union of all S_is with a single element (i.e., cardinality = 1), and label all S_is containing the elements in RS as "marked".
Step 2. Include the elements of all unmarked S_is that have two elements (i.e., cardinality = 2) into the temporary set *list*, call the subprogram *selectTest* to choose a test case, add the test case into RS, and label all subsets containing the test case as "marked".
Step 3. Repeat Step 2 for the unmarked S_is with cardinality = 3, 4, 5, ..., until the unmarked S_is of maximum cardinality are examined.

```
1  algorithm HGS_EIrreplaceability
2  input      S_i: the set of test cases that covers the i-th requirement, 1 ≤ i ≤ m
3  output     RS: a representative set
4  declare    card(S_i): returns the number of test cases in S_i;
5  begin
6      maxCard = the maximum cardinality of S_is;
7      RS = all t ∈ S_i where card(S_i) == 1;
8      mark all S_i containing elements in RS;
9      curCard = 2;
10     while (curCard <= maxCard)
11     {
12         while ( there are unmarked S_i such that card(S_i) == curCard )
13         {
14             list = the set including all test cases in unmarked S_is where card(S_i) == curCard;
15             nextTest = selectTest(curCard, list);
16             RS = RS ∪ {nextTest};
17             mayReduce = false;
18             // mark all S_i containing elements in nextTest;
19             for each (S_i containing nextTest)
20             {
21                 mark S_i;
22                 if (card(S_i) == maxCard)
23                     mayReduce = true;
24             }
25             // update the max cardinality
26             if (mayReduce == true)
27                 maxCard = the maximum cardinality of unmarked S_is;
28         }
29         curCard = curCard + 1;
30     }
31     return RS;
32 end
```

Fig. 7a. The main program of the $HGS_{EIrreplaceability}$ algorithm.

```
1  function selectTest(size, list)
2  begin
3      count[ ]: an array
4      for each (t_i in list)
5      {
6          CT = the union of unmarked S_is;
7          UR = the corresponding requirements of unmarked S_is;
8          count[i] = the return value of EvaluateEnhancedIrreplaceability(t_i,CT,UR);
9      }
10     testList = the set including the test cases in list for which count[i] is the maximum;
11     if (card(testList)==1)
12         return the test case in testList;
13     else if (size == maxCard)
14         return any test case in testList;
15     else
16         return selectTest(size+1, testList);
17 end
```

Fig. 7b. The subprogram of the $HGS_{EIrreplaceability}$ algorithm.

Subprogram *selectTest*:

Step 1. Calculate the value of *EIrreplaceability*(t) for all test cases t in *list*.

Step 2. Identify the test case possessing the maximum *EIrreplaceability*(t).

Step 3.

Condition 1: If there is only one test case t possessing the maximum value of *EIrreplaceability*(t), then return t.

Condition 2: If *size* is equal to *maxCard*, randomly return one of the test cases possessing the maximum *EIrreplaceability*(t) value.

Condition 3: Recursively call *selectTest*(*size* + 1, *testList*) until there exists only one test case possessing the maximum *EIrreplaceability*(t) value.

Compared to the subprogram of the HGS algorithm, the subprogram of $HGS_{EIrreplaceability}$ defines the value of *count*[i] according to the return value of the function *EvaluateEnhancedIrreplaceability*. The time complexity of $HGS_{EIrreplaceability}$ is $O(m \cdot (m+n) \cdot maxCard \cdot k)$. Again, we can obtain the HGS_{Ratio} and $HGS_{EIrreplaceability}$ algorithms by substituting *Ratio*(t) and *Irreplaceability*(t) for *EIrreplaceability*(t), respectively. The time complexities of HGS_{Ratio} and $HGS_{Irreplaceability}$ are $O(m \cdot (m+n) \cdot maxCard)$ and $O(m \cdot (m+n) \cdot maxCard \cdot k)$, respectively.

5. Experimental analyses

In this section, we will describe the details of the subject programs and the algorithms compared in the empirical studies, explain the steps of the experiments, and then present and discuss the empirical results.

5.1. Experimental setup

5.1.1. Algorithms and subject programs for comparison

In the experiments, we compare the capabilities of three metrics (i.e., *Coverage*(t), *Ratio*(t), and *EIrreplaceability*(t)) when they are incorporated into the Greedy algorithm, the GRE algorithm, and the HGS algorithm, respectively. Because *EIrreplaceability*(t) is the enhanced version of *Irreplaceability*(t), it will strictly dominate *Irreplaceability*(t). Thus, *Irreplaceability*(t) is not included in the comparisons. Table 15 shows the nine algorithms to be compared in the experiment. Please notice that we exclusively focused on the SIR programs because: (1) the SIR subject programs (i.e., the suite of the seven Siemens programs [7,42] and the Space program [43]) are frequently chosen benchmarks for evaluating test suite reduction methods; and (2) the SIR subject programs clearly exhibit variability in the cost of the test cases, as demonstrated in Section 3.1. The detailed descriptions of the SIR subject programs and test suites are given in Table 16.

Before the experiments, we individually recorded the average of the execution times of each test case on both Linux (Ubuntu 10.10) and Windows (Windows 7 SP1) machines with Intel Core Duo processors and 2 GB of memory. The consumed execution times are measured in terms of microseconds. Besides, we also analyzed the test requirements that are satisfied by each test case. In addition to R, T, and RS, we define the following terms in order to enable discussions of the experiments:

- *A:* the pool of all test cases designed for the subject program, with $|A| \geqslant 1$.
- *loc:* the number of lines of code in the subject program.

Please cite this article in press as: C.-T. Lin et al., Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests, Inform. Softw. Technol. (2014), http://dx.doi.org/10.1016/j.infsof.2014.04.013

Table 15
Algorithms that are compared in the empirical studies.

Algorithm	Algorithm description	Time Complexity	Classification
Greedy	The original Greedy algorithm	$O(m \cdot n \cdot \min(m, n))$	Greedy-based algorithms
$Greedy_{Ratio}$	Incorporating Ratio into Greedy	$O(m \cdot n \cdot \min(m, n))$	
$Greedy_{EIrreplaceability}$	Incorporating EIrreplaceability into Greedy	$O(m \cdot n \cdot \min(m, n) \cdot k)$	
GRE	The original GRE algorithm	$O(\min(m, n) \cdot (m + n^2 \cdot k))$	GRE-based algorithms
GRE_{Ratio}	Incorporating Ratio into GRE	$O(\min(m, n) \cdot (m + n^2 \cdot k))$	
$GRE_{EIrreplaceability}$	Incorporating EIrreplaceability into GRE	$O(\min(m, n) \cdot (m + n^2 \cdot k) \cdot k)$	
HGS	The original HGS algorithm	$O(m \cdot (m + n) \cdot maxCard)$	HGS-based algorithms
HGS_{Ratio}	Incorporating Ratio into HGS	$O(m \cdot (m + n) \cdot maxCard)$	
$HGS_{EIrreplaceability}$	Incorporating EIrreplaceability into HGS	$O(m \cdot (m + n) \cdot maxCard \cdot k)$	

Table 16
Descriptions of the SIR subject programs and test suites.

Programs	$\|A\|$: size of test pool	$\|R\|$: num. of test requirements	Description
printtokens	4130	140	A lexical analyzer
printtokens2	4115	138	A lexical analyzer
replace	5542	126	A program used for pattern replacement
schedule	2650	46	A program used for priority scheduler
schedule2	2710	72	A program used for priority scheduler
tcas	1608	16	A program used for altitude separation
totinfo	1052	44	A program used for information measure
space	13,585	1067	An array definition language interpreter

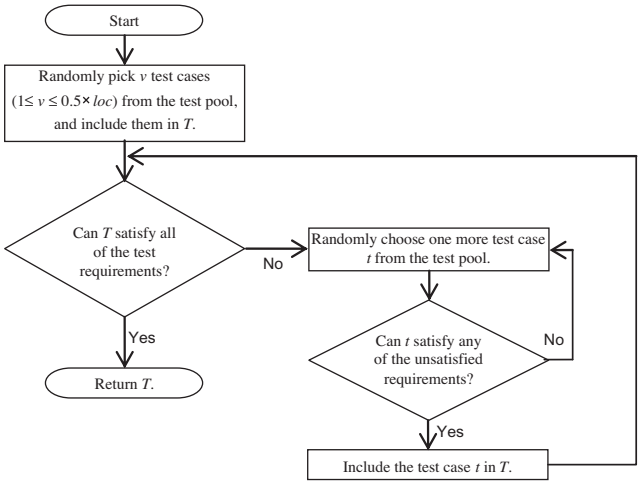

Fig. 8. The process of generating the test suite for each subject program.

5.1.2. Experimental steps

After collecting the execution times of the tests, we followed an empirical setup similar to the one described in [7,45]. The detailed steps are shown in Fig. 8 and described as follows:

1. Randomly generate an integer v, $1 \leqslant v \leqslant 0.5 \times loc$.
2. Randomly pick v test cases from the test pool for each subject program, and include those v test cases in T.
3. Check whether the test cases in T can satisfy all of the test requirements. If not, randomly choose one more test case that can satisfy one or more unsatisfied test requirements, and include the test case into T.
4. Repeat Step 3 until all test requirements are satisfied.

Following the aforementioned steps, we generate 1000 test suites for each program. We perform test suite reduction for each of the 1000 generated test suites, collect the related information, and take the average. Table 17 shows the averages of size and execution cost of the 1000 generated test suites for each subject program.

Table 17
The averages of size and execution cost of the generated test suites for all subject programs.

Subject programs	Suite size	Execution cost of test suite (μs)
printtokens	97.89	505,426
printtokens2	94.47	380,847
replace	133.46	848,038
schedule	78.11	414,835
schedule2	70.82	438,657
tcas	41.37	162,646
totinfo	8.82	419,690
space	2318.30	1.49×10^8

5.2. Research questions

As mentioned at the end of Section 2.4, because the focal points of the previous empirical studies related to test suite reduction problem are diverse, it is not advisable to address all of these past considerations in this paper. Thus, with a focus on the metrics adopted in [3], our empirical studies are designed to answer the following three research questions.

Research Question 1 (RQ1) – *Can the algorithms integrated with the Ratio metric and the EIrreplaceability metric provide better cost reduction capability than the traditional algorithms (i.e., Greedy, GRE, and HGS)?*

The size of the representative set is a direct and valid measure for comparing the effectiveness of the test suite reduction algorithms. However, instead of the size of the representative set, here our focus is the cost of executing the test cases in the representative set. The execution cost of test suite T is defined as

$$SuiteCost(T) = \sum_{t \in T} ExecutionTime(t), \tag{7}$$

where $ExecutionTime(t)$ represents the time required to execute the test case t.

Additionally, we also use the percentage of suite cost reduction (SCR) [9], as defined in Eq. (8), to evaluate the reduction capability.

$$SCR(T, RS) = \frac{SuiteCost(T) - SuiteCost(RS)}{SuiteCost(T)} \times 100\%, \tag{8}$$

where $SuiteCost(T)$ represents the cost required to execute the original test suite T, and $SuiteCost(RS)$ represents the cost associated with

running the representative set *RS*. A high SCR value indicates that the cost reduction capability is satisfactory.

Moreover, we perform statistical tests on the SCR values to examine whether the difference in the capabilities between two specific reduction algorithms is statistically significant and whether the experimental results are repeatable. Because the sample data in the experiments are not approximately normally distributed and are highly skewed left, nonparametric tests should be more appropriate for our empirical studies in comparison to parametric ones. Also, according to the discussions in [46,47], the Mann–Whitney *U*-test, a well-known nonparametric test, is suggested for comparing two data samples when analyzing data in the domain of software engineering. The "null hypothesis" is that there is no difference in the capabilities between two specific reduction algorithms. In the statistical tests, we consider a confidence level of 95% (i.e., the *p*-value below 0.05). If the *p*-value is less than 0.05, the null hypothesis is rejected (i.e., the difference in the reduction capability between two algorithms is statistically significant).

Research Question 2 (RQ2) – *Do the representative sets produced by the algorithms integrated with different metrics have many test cases in common?*

As mentioned in Section 3, the traditional algorithms aim to decrease the number of test cases in the representative set, while the algorithms integrated with cost-aware test case metrics are designed to decrease the time required to execute all of the test cases in the representative set. Thus, it is important to analyze whether the representative sets produced by them have many test cases in common. According to Zhong et al.'s concept in [3], we define the common rate of the representative sets produced by the algorithms in the comparison as

$$CommonRate(RS_1, RS_2, \ldots, RS_z) = \frac{|\bigcap_{h=1}^{z} RS_h|}{|\bigcup_{h=1}^{z} RS_h|} \times 100\%, \quad (9)$$

where z is the number of algorithms in the comparison, and RS_h denotes the representative set reduced by the h-th algorithm. A higher value of common rate reveals that the representative sets have more test cases in common. If two test suite reduction algorithms often create representative sets that have a high percentage of test cases in common, then their cost reduction capabilities may be close to each other and they can be used interchangeably. In other words, if a traditional algorithm and its cost-aware version often create representative sets that have a high percentage of test cases in common, the presented technique is unlikely to represent a significant improvement over the traditional method.

Research Question 3 (RQ3) – *Considering all of the nine algorithms that are compared in the study, what is the ranking in terms of the cost reduction capability?*

We plan to answer this question via multiple comparisons in the nine algorithms in terms of the average SCR across all subject programs. Based on the multiple comparisons, we will rank all of the nine algorithms and determine whether the difference between each pair of algorithms is statistically significant. Yet, because the number of algorithms that are compared in this study is considerable, we will not involve the common rates of representative sets in the multiple comparisons.

5.3. Experimental results and analyses

The analyses are divided into two parts. First, in Sections 5.3.1–5.3.3, we respond to RQ1 and RQ2 by analyzing how the three traditional algorithms improve when the Coverage metric is replaced by EIrreplaceability and Ratio, respectively. Second, we reply to RQ3 by comparing the overall capabilities of all of the nine algorithms in Section 5.3.4.

5.3.1. Discussion 1: comparing the Greedy-based algorithms

For the ease of the comparison, we let $RS_{native1}$, RS_{Ratio1}, and $RS_{EIrreplaceability1}$ denote the representative sets reduced from T via the Greedy-based algorithms (i.e., Greedy, $Greedy_{Ratio}$, and $Greedy_{EIrreplaceability}$), respectively.

5.3.1.1. Replying to RQ1. Table 18 lists the execution costs and the SCR values of the representative sets produced by the three Greedy-based algorithms for each subject program. From the high SCR values in this table, we can see that all of the three algorithms significantly reduce the execution costs of the test suites, and $Greedy_{EIrreplaceability}$ provides the best reduction capability for all of the subject programs. Additionally, the reduction capabilities in the order of largest to smallest are $Greedy_{EIrreplaceability}$, $Greedy_{Ratio}$, and Greedy for most of the subject programs, except for printtokens and schedule. Considering printtokens and schedule, even though the Ratio metric is designed under the consideration of minimizing execution cost, the cost reduction capability of $Greedy_{Ratio}$ is second to that of Greedy. Overall, $Greedy_{Ratio}$ still outperforms Greedy according to the average execution cost and SCR across all of the subject programs.

It should be noticed that, the execution cost of the original test suite for each subject program is so high that the SCR values of the three representative sets are close to each other. In fact, the differences in the execution costs of the representative sets are considerable. Table 19 shows the improvement on the Greedy algorithm when *Coverage*(t) is replaced by *Ratio*(t) and *EIrreplaceability*(t), respectively. The table indicates that the improvements in the execution cost of the representative sets are significant, especially for the large-scale program, space. It is clear that the EIrreplaceability metric can improve the Greedy algorithm for all subject programs, and the differences in the cost reduction capability between $Greedy_{EIrreplaceability}$ and Greedy are all statistically significant. On

Table 18
Reduction capability for the three Greedy-based algorithms.

Program	Suite						
	Original	$RS_{native1}$		RS_{Ratio1}		$RS_{EIrreplaceability1}$	
	Cost[#]	Cost[#]	SCR (%)	Cost[#]	SCR (%)	Cost[#]	SCR (%)
printtokens	505425.70	48402.75	90.42	49434.84	90.22	37867.13	92.51
printtokens2	380846.80	23362.42	93.87	23294.41	93.88	21470.30	94.36
replace	848037.60	66697.85	92.14	58051.27	93.15	53657.01	93.67
schedule	414834.80	14701.83	96.46	14859.85	96.42	13140.77	96.83
schedule2	438657.40	33636.91	92.33	32523.82	92.59	30022.68	93.16
tcas	162646.30	21161.35	86.99	17156.95	89.45	16202.89	90.04
totinfo	419689.50	28868.62	93.12	20382.22	95.14	19752.15	95.29
space	1.49×10^8	6.55×10^6	95.59	3.64×10^6	97.55	3.50×10^6	97.64
Mean	1.90×10^7	848555.22	92.62	482514.17	93.55	461544.87	94.19

[#] Measured in microsecond (μs).

Please cite this article in press as: C.-T. Lin et al., Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests, Inform. Softw. Technol. (2014), http://dx.doi.org/10.1016/j.infsof.2014.04.013

Table 19
The Comparisons Regarding the Improvement on the Greedy Algorithm when Coverage(t) is replaced by Ratio(t) and EIrreplaceability(t), respectively.[a]

Program	$Greedy_{Ratio}$ versus Greedy			$Greedy_{EIrreplaceability}$ versus Greedy		
	p-Value[b]	Improvement		p-Value[b]	Improvement	
		Cost[c]	Percentage[d] (%)		Cost[e]	Percentage[f] (%)
printtokens	.746	−1032.09	−2.13	*.000*	*10535.62*	*21.77*
printtokens2	.892	68.01	0.29	*.009*	*1892.12*	*8.10*
replace	*.000*	*8646.58*	*12.96*	*.000*	*13040.84*	*19.55*
schedule	.514	−158.02	−1.07	*.001*	*1561.06*	*10.62*
schedule2	.094	1113.09	3.31	*.000*	*3614.23*	*10.74*
tcas	*.000*	*4004.40*	*18.92*	*.000*	*4958.46*	*23.43*
totinfo	*.000*	*8486.40*	*29.40*	*.000*	*9116.47*	*31.58*
space	*.000*	*2.91×10^6*	*44.37*	*.000*	*3.05×10^6*	*46.57*
Mean	–	366041.05	13.26	–	387010.35	21.55

[a] The italic part indicates that the improvement is significantly different according to the Mann–Whitney U-test.
[b] Indicates the p-value of statistical test on the SCR values.
[c] Indicates $SuiteCost(RS_{native1}) - SuiteCost(RS_{Ratio1})$.
[d] Indicates $(SuiteCost(RS_{native1}) - SuiteCost(RS_{Ratio1}))/SuiteCost(RS_{native1})$.
[e] Indicates $SuiteCost(RS_{native1}) - SuiteCost(RS_{EIrreplaceability1})$.
[f] Indicates $(SuiteCost(RS_{native1}) - SuiteCost(RS_{EIrreplaceability1}))/SuiteCost(RS_{native1})$.

the other hand, the Ratio metric degrades the effectiveness of the Greedy algorithm for two programs, printtokens and schedule. Even though it can improve the Greedy algorithm for six out of the eight subject programs, not all improvements are statistically significant. On the whole, both the EIrreplaceability and Ratio metrics can enhance the capability of the Greedy algorithm, but the EIrreplaceability metric outperforms the Ratio metric.

Although $Greedy_{EIrreplaceability}$ generally provides the best cost reduction capability for all of the subject programs, Table 15 indicates that $Greedy_{EIrreplaceability}$ shows less satisfactory time complexity than Greedy and $Greedy_{Ratio}$. Actually, the costs taken to perform regression testing include the time taken to produce the representative set of test cases and the time taken to execute the test cases. Thus, it is important to determine if the decrease in the cost of executing the test cases achieved by $Greedy_{EIrreplaceability}$ can compensate for the increase in the cost of producing the representative set. Table 20 lists the aforementioned regression testing costs associated with the three Greedy-based algorithms for each subject program. As seen in Table 20, it is evident that the EIrreplaceability metric indeed improves the total regression testing costs for all of the subject programs.

5.3.1.2. Replying to RQ2. Table 21 lists the common rates of different pairs of representative sets for each subject program, respectively. As seen from the table, the common rate between $RS_{EIrreplaceability1}$ and RS_{Ratio1} is usually much higher than others for most of the subject programs, except for printtokens. That is, $Greedy_{EIrreplaceability}$ and $Greedy_{Ratio}$ usually choose more test cases in common than other pairs during the reduction process. This phenomenon is explained by the fact that both the Ratio and the EIrreplaceability metrics take into account the minimization of the test execution cost. However, even though the common rates between $RS_{EIrreplaceability1}$ and RS_{Ratio1} are high, $Greedy_{EIrreplaceability}$ does not pick exactly the same test cases as $Greedy_{Ratio}$. Because $Greedy_{EIrreplaceability}$ shows the best SCR values for all of the subject programs, then this would suggest that it picks better test cases than $Greedy_{Ratio}$ during the cost reduction process. Additionally, although the common rate of the pair of $RS_{native1}$ and RS_{Ratio1} and

Table 20
Regression testing cost comparisons for the three Greedy-based algorithms.[a]

Program	Suite					
	$RS_{native1}$[d]		RS_{Ratio1}[d]		$RS_{EIrreplaceability1}$[d]	
	Cost[b]	Wasted time[c]	Cost[b]	Wasted time[c]	Cost[b]	Wasted time[c]
	Cost for regression test		Cost for regression test		Cost for regression test	
printtokens	48402.75	1.72	49434.84	1.91	37867.13	7.15
	48404.47		49436.75		37874.28	
printtokens2	23362.42	1.68	23294.41	1.76	21470.30	6.51
	23364.10		23296.17		21476.81	
replace	66697.85	2.00	58051.27	2.05	53657.01	10.93
	66699.85		58053.32		53667.94	
schedule	14701.83	0.75	14859.85	0.74	13140.77	1.81
	14702.58		14860.59		13142.58	
schedule2	33636.91	1.00	32523.82	1.02	30022.68	3.41
	33637.91		32524.84		30026.09	
tcas	21161.35	0.11	17156.95	0.13	16202.89	0.32
	21161.46		17157.08		16203.21	
totinfo	28868.62	0.64	20382.22	0.60	19752.15	1.83
	28869.26		20382.82		19753.98	
space	6.55×10^6	103.28	3.64×10^6	107.35	3.50×10^6	4245.64
	6.55×10^6		3.64×10^6		3.50×10^6	
Mean	848555.22	13.90	482514.17	14.45	461544.87	534.70
	848569.12		482528.62		462079.57	

[a] Measured in microsecond (μs).
[b] Indicates the time cost taken to execute the test cases in the test suite.
[c] Indicates the time taken to run the test suite reduction algorithm.
[d] $RS_{native1}$, RS_{Ratio1}, and $RS_{EIrreplaceability1}$ represent the representative sets produced by Greedy, $Greedy_{Ratio}$, and $Greedy_{EIrreplaceability}$, respectively.

Please cite this article in press as: C.-T. Lin et al., Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests, Inform. Softw. Technol. (2014), http://dx.doi.org/10.1016/j.infsof.2014.04.013

Table 21
Common rate for the three Greedy-based algorithms.

Program	Common rate			
	$RS_{native1}$ & RS_{Ratio1} (%)	$RS_{native1}$ & $RS_{EIrreplaceability1}$ (%)	RS_{Ratio1} & $RS_{EIrreplaceability1}$ (%)	All
printtokens	47.65	63.10	59.19	43.44
printtokens2	45.92	51.31	71.07	41.53
replace	46.72	48.32	71.38	41.04
schedule	47.67	54.72	73.91	44.75
schedule2	42.12	47.48	77.88	40.07
tcas	32.21	31.85	83.50	29.54
totinfo	19.26	18.40	62.44	14.97
space	19.86	21.08	74.07	18.33
Mean	37.68	42.03	71.68	34.21

that of the pair of $RS_{native1}$ and $RS_{EIrreplaceability1}$ are close to each other for most of the subject programs, the former usually provides the lowest common rate. It is also important to note that, according to Tables 17 and 21, the common rates of all possible combinations are not related to the size of the original test suite. This observation may indicate that the effectiveness of adopting cost-aware test case metrics is satisfactory for the test suites of various scales.

5.3.2. Discussion 2: comparing the GRE-based algorithms

Similar to Section 5.3.1, $RS_{native2}$, RS_{Ratio2} and $RS_{EIrreplaceability2}$ denote the representative sets reduced from T via the GRE-based algorithms (i.e., GRE, GRE_{Ratio}, and $GRE_{EIrreplaceability}$), respectively.

5.3.2.1. Replying to RQ1. The results of test suite reduction for three GRE-based algorithms are shown in Table 22. The table indicates that all GRE-based algorithms can significantly reduce the execution costs of the test suites. The cost reduction capabilities in order of rank are $GRE_{EIrreplaceability}$, GRE_{Ratio}, and GRE for five out of eight subject programs (i.e., printtokens, replace, tcas, totinfo, and space). Again, $GRE_{EIrreplaceability}$ and GRE provide the best and the worst reduction capabilities for most of the subject programs, respectively. Notice that GRE_{Ratio} performs worse than GRE for three programs (i.e., printtokens2, schedule, and schedule2). However, considering the averages across all of the subject programs, the reduction capabilities in order of rank are still $GRE_{EIrreplaceability}$, GRE_{Ratio}, and GRE.

Table 23 quantifies the influence on the GRE algorithms caused by the Ratio and EIrreplaceability metrics. On the whole, both Ratio and EIrreplaceability can upgrade the reduction capability of the GRE algorithm. It should be observed that they impair the cost capability of the GRE algorithm on the subject program schedule, but the impairments are very slight; the statistical test also

Table 22
Reduction capability for the three GRE-based algorithms.

Program	Suite						
	Original	$RS_{native2}$		RS_{Ratio2}		$RS_{EIrreplaceability2}$	
	Cost[#]	Cost[#]	SCR (%)	Cost[#]	SCR (%)	Cost[#]	SCR (%)
printtokens	505425.70	43293.50	91.43	41063.38	91.88	40846.38	91.92
printtokens2	380846.80	22809.39	94.01	23119.43	93.93	22661.40	94.05
replace	848037.60	63973.69	92.46	61372.51	92.76	60722.47	92.84
schedule	414834.80	14183.82	96.58	14464.82	96.51	14379.82	96.53
schedule2	438657.40	32480.80	92.60	32555.81	92.58	32344.81	92.63
tcas	162646.30	20418.31	87.45	19361.21	88.10	19306.20	88.13
totinfo	419689.50	28179.61	93.29	24532.51	94.15	24292.50	94.21
space	1.49×10^8	6.11×10^6	95.89	5.01×10^6	96.63	4.83×10^6	96.75
Mean	1.90×10^7	792098.77	92.96	652844.71	93.32	630455.45	93.38

[#] Measured in microsecond (μs).

Table 23
The Comparisons regarding the improvement on the GRE algorithm when Coverage(t) is replaced by Ratio(t) and EIrreplaceability(t), respectively.[a]

Program	GRE_{Ratio} versus GRE			$GRE_{EIrreplaceability}$ versus GRE		
	p-Value[b]	Improvement		p-Value[b]	Improvement	
		Cost[c]	Percentage[d] (%)		Cost[e]	Percentage[f] (%)
printtokens	.083	2230.12	5.15	*.047*	*2447.12*	*5.65*
printtokens2	.456	−310.04	−1.36	.977	147.99	0.65
replace	.101	2601.18	4.07	*.030*	*3251.22*	*5.08*
schedule	.414	−281.00	−1.98	.550	−196.00	−1.38
schedule2	.830	−75.01	−0.23	.951	135.99	0.42
tcas	*.034*	*1057.10*	*5.18*	*.028*	*1112.11*	*5.45*
totinfo	*.000*	*3647.10*	*12.94*	*.000*	*3887.11*	*13.79*
space	*.000*	*1.10×10^6*	*18.08*	*.000*	*1.28×10^6*	*20.98*
Mean	–	139254.06	5.23	–	161643.32	6.33

[a] The italic part indicates that the improvement is significantly different according to the Mann–Whitney U-test.
[b] Indicates the p-value of statistical test on the SCR values.
[c] Indicates SuiteCost($RS_{native2}$)−SuiteCost(RS_{Ratio2}).
[d] Indicates (SuiteCost($RS_{native2}$)−SuiteCost(RS_{Ratio2}))/SuiteCost($RS_{native2}$).
[e] Indicates SuiteCost($RS_{native2}$)−SuiteCost($RS_{EIrreplaceability2}$).
[f] Indicates (SuiteCost($RS_{native2}$)−SuiteCost($RS_{EIrreplaceability2}$))/SuiteCost($RS_{native2}$).

Please cite this article in press as: C.-T. Lin et al., Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests, Inform. Softw. Technol. (2014), http://dx.doi.org/10.1016/j.infsof.2014.04.013

Table 24
Regression testing cost comparisons for the three GRE-based algorithms.[a]

Program	Suite					
	$RS_{native2}$[d]		RS_{Ratio2}[d]		$RS_{EIrreplaceability2}$[d]	
	Cost[b] Cost for regression test	Wasted time[c]	Cost[b] Cost for regression test	Wasted time[c]	Cost[b] Cost for regression test	Wasted time[c]
printtokens	43293.50	4.29	41063.38	4.27	40846.38	4.41
	43297.79		41067.65		40850.79	
printtokens2	22809.39	5.82	23119.43	6.28	22661.40	6.52
	22815.21		23125.71		22667.92	
replace	63973.69	6.99	61372.51	7.16	60722.47	7.67
	63980.68		61379.67		60730.14	
schedule	14183.82	1.38	14464.82	1.36	14379.82	1.31
	14185.20		14466.18		14381.13	
schedule2	32480.80	2.03	32555.81	2.03	32344.81	2.10
	32482.83		32557.84		32346.91	
tcas	20418.31	0.19	19361.21	0.19	19306.20	0.21
	20418.50		19361.40		19306.41	
totinfo	28179.61	1.63	24532.51	1.69	24292.50	1.78
	28181.24		24534.20		24294.28	
space	6.11×10^6	12082.67	5.01×10^6	14188.01	4.83×10^6	22362.05
	6.12×10^6		5.02×10^6		4.85×10^6	
Mean	792098.77	1513.13	652844.71	1776.37	630455.45	2798.26
	793611.90		654621.08		633253.71	

[a] Measured in microsecond (μs).
[b] Indicates the time cost taken to execute the test cases in the test suite.
[c] Indicates the time taken to run the test suite reduction algorithm.
[d] $RS_{native2}$, RS_{Ratio2}, and $RS_{EIrreplaceability2}$ represent the representative sets produced by GRE, GRE_{Ratio}, and $GRE_{EIrreplaceability}$, respectively.

indicates that the impairments are not statistically significant. With the exception of schedule, EIrreplaceability improves the GRE algorithm for the other seven subject programs, and the improvements on five out of the seven programs are statistically significant. On the other hand, Ratio degrades the capability of the GRE algorithm for printtokens2, schedule, and schedule2. Although it still improves the GRE algorithm for the other five programs, the improvements are statistically significant for only three out of the five programs. Moreover, the improvement achieved by the EIrreplaceability metric tops that which was achieved by the Ratio metric for all of the subject programs.

From Tables 19 and 23, it is evident that Ratio and EIrreplaceability make more significant improvement in the Greedy algorithm than in the GRE algorithm. This is because the Greedy algorithm only takes into account the value, or importance, of the candidates when selecting test cases. Consequently, the adopted test case metric directly decides whether the test cases will be placed into the representative set; thus, changing the test case metric may lead to the most significant improvement. On the other hand, because the GRE algorithm involves the 1-to-1 redundant strategy, the test case metric is not the only factor that controls the reduction process.

Additionally, Table 24 lists the total regression testing costs associated with the three GRE-based algorithms for each subject program. Table 24 indicates that even though $GRE_{EIrreplaceability}$ generally takes more time to perform the test suite reduction in comparison with the other two GRE-based algorithms, it still shows lower total regression testing costs for seven out of eight subject programs.

5.3.2.2. Replying to RQ2. Table 25 shows the common rates for all possible combinations of the representative sets produced by the GRE-based algorithms. These results indicate that, for all of the subject programs, the common rate of the pair of $RS_{EIrreplaceability2}$ and RS_{Ratio2} is much higher than those of other combinations, and the common rates of the other two pairs are close to each other. Yet, $RS_{EIrreplaceability2}$ and RS_{Ratio2} still have different test cases. That is, $GRE_{EIrreplaceability}$ does not pick exactly the same test cases as GRE_{Ratio}, thus resulting in a better cost reduction. Besides, according to Tables 17 and 25, it seems that the common rates for all combinations are not related to the size of the original test suite. This may show that the effectiveness of adopting cost-aware test case metrics is satisfactory for the test suites of various scales.

5.3.3. Discussion 3: comparing the HGS-based algorithms

Again, $RS_{native3}$, RS_{Ratio3} and $RS_{EIrreplaceability3}$ denote the representative sets reduced from T via the HGS-based algorithms (i.e., HGS, HGS_{Ratio}, and $HGS_{EIrreplaceability}$), respectively.

Table 25
Common rate for the three GRE-based algorithms.

Program	Common rate			
	$RS_{native2}$ & RS_{Ratio2} (%)	$RS_{native2}$ & $RS_{EIrreplaceability2}$ (%)	RS_{Ratio2} & $RS_{EIrreplaceability2}$ (%)	All (%)
printtokens	84.98	85.75	96.80	84.21
printtokens2	62.55	64.99	85.31	59.68
replace	71.12	71.57	86.47	66.84
schedule	82.47	82.16	96.03	80.99
schedule2	71.39	71.82	94.43	69.88
tcas	72.26	71.71	94.50	70.22
totinfo	45.69	44.47	72.58	38.91
space	70.42	67.56	78.20	62.22
Mean	70.11	70.00	88.04	66.62

Please cite this article in press as: C.-T. Lin et al., Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests, Inform. Softw. Technol. (2014), http://dx.doi.org/10.1016/j.infsof.2014.04.013

Table 26
Reduction capability for the three HGS-based algorithms.

Program	Suite						
	Original	$RS_{native3}$		RS_{Ratio3}		$RS_{EIrreplaceability3}$	
	Cost[#]	Cost[#]	SCR (%)	Cost[#]	SCR (%)	Cost[#]	SCR (%)
printtokens	505425.70	43461.50	91.40	42062.41	91.68	40725.33	91.94
printtokens2	380846.80	26165.54	93.13	28089.59	92.62	24904.49	93.46
replace	848037.60	67134.83	92.08	62384.54	92.64	60838.41	92.83
schedule	414834.80	15185.88	96.34	15389.88	96.29	14730.84	96.45
schedule2	438657.40	32922.86	92.49	33318.87	92.40	32101.80	92.68
tcas	162646.30	20495.31	87.40	18495.12	88.63	18366.13	88.71
totinfo	419689.50	26870.58	93.60	25591.54	93.90	24503.51	94.16
space	1.49×10^8	5.93×10^6	96.01	4.31×10^6	97.10	4.27×10^6	97.13
Mean	1.90×10^7	770765.56	92.81	567277.49	93.16	560854.31	93.42

[#] Measured in microsecond (μs).

Table 27
The comparisons regarding the improvement on the HGS algorithm when Coverage(t) is replaced by Ratio(t) and EIrreplaceability(t), respectively.[a]

Program	HGS_{Ratio} versus HGS			$HGS_{EIrreplaceability}$ versus HGS		
	p-Value[b]	Improvement		p-Value[b]	Improvement	
		Cost[c]	Percentage[d] (%)		Cost[e]	Percentage[f]
printtokens	.392	1399.09	3.22	*.027*	*2736.17*	*6.30*
printtokens2	.004	−1924.05	−7.35	.082	1261.05	4.82
replace	*.007*	*4750.29*	*7.08*	*.000*	*6296.42*	*9.38*
schedule	.521	−204.00	−1.34	.416	455.04	3.00
schedule2	.639	−396.01	−1.20	.399	821.06	2.49
tcas	*.000*	*2000.19*	*9.76*	*.000*	*2129.18*	*10.39*
totinfo	.074	1279.04	4.76	*.001*	*2367.07*	*8.81*
space	*.000*	1.62×10^6	27.32	*.000*	1.66×10^6	28.03
Mean	–	203488.07	5.28	–	209911.25	9.15

[a] The italic part indicates that the improvement is significantly different according to the Mann–Whitney U-test.
[b] Indicates the p-value of statistical test on the SCR values.
[c] Indicates $SuiteCost(RS_{native3}) - SuiteCost(RS_{Ratio3})$.
[d] Indicates $(SuiteCost(RS_{native3}) - SuiteCost(RS_{Ratio3}))/SuiteCost(RS_{native3})$.
[e] Indicates $SuiteCost(RS_{native3}) - SuiteCost(RS_{EIrreplaceability3})$.
[f] Indicates $(SuiteCost(RS_{native3}) - SuiteCost(RS_{EIrreplaceability3}))/SuiteCost(RS_{native3})$.

5.3.3.1. Replying to RQ1. Table 26 compares the cost reduction capabilities of the three HGS-based algorithms. $HGS_{EIrreplaceability}$ achieves the best cost reduction capability in all of the subject programs. Although HGS_{Ratio} provides better capabilities than HGS for five out of eight programs (i.e., printtokens, replace, tcas, totinfo, and space), it performs worse for the remaining three subject programs. However, the averages across all of the subject programs indicate that the reduction capabilities in order of rank are $HGS_{EIrreplaceability}$, HGS_{Ratio}, and HGS.

Table 27 quantifies how Ratio and EIrreplaceability enhance the HGS algorithm. According to the table, EIrreplaceability upgrades the cost reduction capability of the HGS algorithm for all of the subject programs, and the improvements are statistically significant for five out of the eight programs. On the other hand, Ratio can improve the HGS algorithm for only some of the subject programs, and the improvements are statistically significant for only three subject programs. It is clear that selecting EIrreplaceability as the test case metric is a better choice to improve the cost reduction capability of the HGS algorithm. In general, the results reveal that EIrreplaceability and Ratio are better at improving the HGS algorithm than they are at bettering GRE. Yet, these two metrics cannot improve HGS as much as they can enhance the Greedy algorithm.

Moreover, Table 28 lists the total regression testing costs associated with the three HGS-based algorithms for each subject program. As seen from Table 28, although $HGS_{EIrreplaceability}$ may take a little bit more time than the other two HGS-based algorithms to produce the representative set of test cases, it still achieves the lowest total regression testing costs for all of the subject programs.

5.3.3.2. Replying to RQ2. Table 29 compares the common rates of all possible combination of the representative sets produced by the HGS-based algorithms. Similar to the results given in Sections 5.3.1.2 and 5.3.2.2, the pair of $RS_{EIrreplaceability3}$ and RS_{Ratio3} shows higher common rates compared to the other pairs. However, while $RS_{EIrreplaceability3}$ and RS_{Ratio3} show high common rates, they still have different test cases. Because $HGS_{EIrreplaceability}$ shows better SCR values than HGS_{Ratio}, this would suggest that it picks better test cases than HGS_{Ratio} during the cost reduction process. The common rates of the other two pairs are close to each other for most of the subject programs, except for printtokens2. Additionally, we cannot identify the correlation between the common rate and the original suite size according to Tables 17 and 29. Again, this may indicate that the effectiveness of adopting cost-aware test case metrics is satisfactory for the test suites of various scales. Overall, the results look similar to those in Sections 5.3.1.2 and 5.3.2.2.

5.3.4. Discussion 4: the overall comparison of the nine test suite reduction algorithms

5.3.4.1. Replying to RQ3. Table 30 presents the reduction capabilities and the rankings of the nine algorithms with respect to each subject program. From this table, it is clear that $Greedy_{EIrreplaceability}$ provides the best reduction capability for all of the subject programs. $Greedy_{Ratio}$ and the other two algorithms integrated with EIrreplaceability (i.e., $GRE_{EIrreplaceability}$ and $HGS_{EIrreplaceability}$) also demonstrate good effectiveness for most of the subject programs. Although the remaining two algorithms integrated with Ratio (i.e., GRE_{Ratio} and HGS_{Ratio}) demonstrate effectiveness that is

Table 28
Regresion test cost comparisons for the three HGS-based algorithms.[a]

Program	Suite					
	$RS_{native3}$[d]		RS_{Ratio3}[d]		$RS_{EIrreplaceability3}$[d]	
	Cost[b] Cost for regression test	Wasted time[c]	Cost[b] Cost for regression test	Wasted time[c]	Cost[b] Cost for regression test	Wasted time[c]
printtokens	43461.50 43461.75	0.25	42062.41 42062.63	0.22	40725.33 40725.63	0.30
printtokens2	26165.54 26165.76	0.22	28089.59 28089.86	0.27	24904.49 24904.78	0.29
replace	67134.83 67135.12	0.29	62384.54 62384.84	0.30	60838.41 60838.79	0.38
schedule	15185.88 15185.94	0.06	15389.88 15389.95	0.07	14730.84 14730.89	0.05
schedule2	32922.86 32922.96	0.10	33318.87 33318.97	0.10	32101.80 32101.92	0.12
tcas	20495.31 20495.34	0.03	18495.12 18495.15	0.03	18366.13 18366.16	0.03
totinfo	26870.58 26870.67	0.09	25591.54 25591.61	0.07	24503.51 24503.61	0.10
space	5.93×10^6 5.93×10^6	18.91	4.31×10^6 4.31×10^6	9.03	4.27×10^6 4.27×10^6	12.19
Mean	770765.56 770768.05	2.49	567277.49 567278.75	1.26	560854.31 560855.99	1.68

[a] Measured in microsecond (μs).
[b] Indicates the time cost taken to execute the test cases in the test suite.
[c] Indicates the time taken to run the test suite reduction algorithm.
[d] $RS_{native3}$, RS_{Ratio3}, and $RS_{EIrreplaceability3}$ represent the representative sets produced by HGS, HGS_{Ratio}, and $HGS_{EIrreplaceability}$, respectively.

Table 29
Common rate for the three HGS-based algorithms.

Program	Common rate			
	$RS_{native3}$ & RS_{Ratio3} (%)	$RS_{native3}$ & $RS_{EIrreplaceability3}$ (%)	RS_{Ratio3} & $RS_{EIrreplaceability3}$ (%)	All (%)
printtokens	67.98	73.87	88.96	67.03
printtokens2	50.99	67.72	69.55	48.49
replace	55.01	62.61	83.26	53.44
schedule	65.75	72.53	88.49	64.43
schedule2	59.50	66.93	85.58	58.16
tcas	46.21	52.04	86.59	44.77
totinfo	44.08	54.83	69.88	39.71
space	39.20	43.13	87.37	38.24
Mean	53.59	61.71	82.46	51.78

Table 30
Reduction capability in terms of SCR for all of the nine algorithms.

Program	Suite SCR (Ranking)								
	$RS_{native1}$	RS_{Ratio1}	$RS_{EIrreplaceability1}$	$RS_{native2}$	RS_{Ratio2}	$RS_{EIrreplaceability2}$	$RS_{native3}$	RS_{Ratio3}	$RS_{EIrreplaceability3}$
printtokens	90.42%(8)	90.22%(9)	92.51%(1)	91.43%(6)	91.88%(4)	91.92%(3)	91.40%(7)	91.68%(5)	91.94%(2)
printtokens2	93.87%(6)	93.88%(5)	94.36%(1)	94.01%(3)	93.93%(4)	94.05%(2)	93.13%(8)	92.62%(9)	93.46%(7)
replace	92.14%(8)	93.15%(2)	93.67%(1)	92.46%(7)	92.76%(5)	92.84%(3)	92.08%(9)	92.64%(6)	92.83%(4)
schedule	96.46%(5)	96.42%(7)	96.83%(1)	96.58%(2)	96.51%(4)	96.53%(3)	96.34%(8)	96.29%(9)	96.45%(6)
schedule2	92.33%(9)	92.59%(5)	93.16%(1)	92.60%(4)	92.58%(6)	92.63%(3)	92.49%(7)	92.40%(8)	92.68%(2)
tcas	86.99%(9)	89.45%(2)	90.04%(1)	87.45%(7)	88.10%(6)	88.13%(5)	87.40%(8)	88.63%(4)	88.71%(3)
totinfo	93.12%(9)	95.14%(2)	95.29%(1)	93.29%(8)	94.15%(5)	94.21%(3)	93.60%(7)	93.90%(6)	94.16%(4)
space	95.59%(9)	97.55%(2)	97.64%(1)	95.89%(8)	96.63%(6)	96.75%(5)	96.01%(7)	97.10%(4)	97.13%(3)
Mean	92.62%(9)	93.55%(2)	94.19%(1)	92.96%(7)	93.32%(5)	93.38%(4)	92.81%(8)	93.16%(6)	93.42%(3)

less satisfactory overall, they still perform better than the three traditional algorithms for most of the subject programs. In Table 31, we combine the test suites of all subject programs, report the pairwise comparisons of all algorithms and present the result of the statistical tests. For each pair of algorithms, the mean difference between the results of applying the algorithms is given along with the significance level. The SCR values in order of rank are $Greedy_{EIrreplaceability}$, $Greedy_{Ratio}$, $HGS_{EIrreplaceability}$, $GRE_{EIrreplaceability}$, GRE_{Ratio}, HGS_{Ratio}, GRE, HGS, and Greedy. However, it should be noted that there is no significant difference in the reduction capability among $HGS_{EIrreplaceability}$, $GRE_{EIrreplaceability}$, and GRE_{Ratio}. Additionally, both the difference between HGS_{Ratio} and GRE and the difference between HGS and Greedy are not statistically significant. Overall, the rankings shown in Tables 30 and 31 indicate that the EIrreplaceability metric demonstrates better improvement than the Ratio metric for all of the three traditional algorithms.

Please cite this article in press as: C.-T. Lin et al., Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests, Inform. Softw. Technol. (2014), http://dx.doi.org/10.1016/j.infsof.2014.04.013

Table 31
Pairwise comparisons (mean SCR across all subject programs).

Reduced Set x (rank)	Statistics	Reduced Set y (rank)								
		$RS_{native1}$ (9)	RS_{Ratio1} (2)	$RS_{EIrreplaceability1}$ (1)	$RS_{native2}$ (7)	RS_{Ratio2} (5)	$RS_{EIrreplaceability2}$ (4)	$RS_{native3}$ (8)	RS_{Ratio3} (6)	$RS_{EIrreplaceability3}$ (3)
$RS_{native1}$ (9)	Difference+	–	*−0.93*	*−1.57*	*−0.34*	*−0.70*	*−0.76*	−0.19	*−0.54*	*−0.80*
	p-Value		*.000*	*.000*	*.000*	*.000*	*.000*	.156	*.000*	*.000*
RS_{Ratio1} (2)	Difference+	*0.93*	–	−0.64	0.59	0.23	0.17	0.74	0.39	0.13
	p-Value	*.000*		*.000*	*.000*	*.000*	*.000*	*.000*	*.000*	*.000*
$RS_{EIrreplaceability1}$ (1)	Difference+	*1.57*	*0.64*	–	1.23	0.87	0.81	1.38	1.03	0.77
	p-Value	*.000*	*.000*		*.000*	*.000*	*.000*	*.000*	*.000*	*.000*
$RS_{native2}$ (7)	Difference+	*0.34*	*−0.59*	*−1.23*	–	*−0.36*	*−0.42*	*0.15*	−0.20	*−0.46*
	p-Value	*.000*	*.000*	*.000*		.000	.000	*.023*	.143	*.000*
RS_{Ratio2} (5)	Difference+	*0.70*	*−0.23*	*−0.87*	*0.36*	–	−0.06	*0.51*	*0.16*	−0.10
	p-Value	*.000*	*.000*	*.000*	*.000*		.324	*.000*	*.010*	.448
$RS_{EIrreplaceability2}$ (4)	Difference+	*0.76*	*−0.17*	*−0.81*	*0.42*	0.06	–	*0.57*	*0.22*	−0.04
	p-Value	*.000*	*.000*	*.000*	*.000*	.324		*.000*	*.000*	.816
$RS_{native3}$ (8)	Difference+	0.19	*−0.74*	*−1.38*	*−0.15*	*−0.51*	*−0.57*	–	−0.35	*−0.61*
	p-Value	.156	*.000*	*.000*	*.023*	*.000*	*.000*		*.000*	*.000*
RS_{Ratio3} (6)	Difference+	*0.54*	*−0.39*	*−1.03*	0.20	*−0.16*	*−0.22*	*0.35*	–	*−0.26*
	p-Value	*.000*	*.000*	*.000*	.143	*.010*	*.000*	*.000*		*.001*
$RS_{EIrreplaceability3}$ (3)	Difference+	*0.80*	*−0.13*	*−0.77*	*0.46*	0.10	0.04	*0.61*	*0.26*	–
	p-Value	*.000*	*.000*	*.000*	*.000*	.448	.816	*.000*	*.001*	

*The italic part indicates that the mean difference between the pair of reduced suites is statistically significant.
+ Indicates SCR_x–SCR_y, where SCR_x and SCR_y are the mean SCR values for the reduced sets x and y, respectively.

5.3.5. Threats to validity

5.3.5.1. Threats to internal validity. The primary threat to the internal validity of our empirical studies is the measurement of the execution time of each test case. That is, the measure for each test case may change if we performed the experiment in another environment. To reduce this threat, we executed each test 1000 times in both of two frequently used execution environments, the Windows and Linux operating systems, and took the average of the 2000 trials.

5.3.5.2. Threats to external validity. The Siemens programs are of a small size and the space program is of a medium size. Therefore, we cannot definitively claim that our findings are representative for all programs. As part of future work, we aim to reduce this threat by conducting additional experiments with larger subject programs and test suites.

5.3.5.3. Threats to construct validity. Defects in implementing the compared reduction algorithms could be a threat to construct validity of the empirical studies; we controlled this threat by examining our implementation with several small programs. We manually inspected the differences in step-by-step traces between our implementation and the manual operation, and found that the results are totally consistent with each other. Another threat to construct validity is related to the metric, SCR, which was used to evaluate the effectiveness of the test suite reduction algorithms. It is true that, given a specific original test suite, a representative set with lower costs than the others always yields a higher SCR value. However, the execution costs of the original test suites generated for the empirical studies are usually so high that the SCR values of the representative sets are close to each other; thus, it is difficult to clearly understand the amounts of the differences in the cost reduction capability between two algorithms. To address this concern, we use Tables 19, 23 and 27 to quantify the amount of the difference in execution cost between two representative sets and highlight whether or not the difference in the reduction capability between two algorithms is statistically significant.

6. Conclusions and future work

In the literature, most of the existing test suite reduction algorithms attempt to reduce the size of a test suite (e.g., [6,16,26]). However, due to the differences in execution costs among test cases, the representative set with the fewest test cases may not be the one with the lowest execution cost. This paper considered this phenomenon and presented a cost-aware metric, called EIrreplaceability, that evaluates the possibility that each test case can be replaced by others when reducing test suites. This paper further presented several cost-aware test suite reduction algorithms in which EIrreplaceability and an existing test case metric (i.e., Ratio) were incorporated into three well-known test suite reduction algorithms (i.e., Greedy, GRE, and HGS). The cost reduction capabilities of the presented cost-aware algorithms were evaluated through an empirical study that was based on eight subject programs collected from SIR. From the results, we have some findings that are revealed through the replies to the three research questions.

(1) *Replying to RQ1*: On the whole, both Ratio and EIrreplaceability can improve the reduction capabilities of the three traditional reduction algorithms. Please notice that:
   i. For each traditional algorithm, the improvements achieved by EIrreplaceability are usually more significant than those achieved by Ratio, as revealed in Tables 19, 23 and 27.
   ii. Both Ratio and EIrreplaceability make more significant improvements to the Greedy algorithm than to the GRE and HGS algorithms. That is, among the three traditional algorithms, the Greedy algorithm benefits more than the others from the use of test case metrics. Additionally, the overall improvements for GRE are relatively minor, as revealed in Tables 19, 23 and 27.

(2) *Replying to RQ2*: If we incorporate Ratio or EIrreplaceability into the Greedy algorithm, then more than half of the test cases in the representative set for most of the subject programs are replaced by different test cases. This phenomenon is rarely seen for GRE and HGS, especially for GRE. Please also notice that:
   i. The common rates between $Greedy_{Ratio}$ and $Greedy_{EIrreplaceability}$ are usually greater than those between Greedy and $Greedy_{EIrreplaceability}$ and those between Greedy and $Greedy_{Ratio}$. This may imply that, although Ratio and EIrreplaceability evaluate test cases based on different criteria, they may achieve similar results. This phenomenon also exists for the GRE-based and the HGS-based algorithms, as shown in Tables 21, 25 and 29.

Please cite this article in press as: C.-T. Lin et al., Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests, Inform. Softw. Technol. (2014), http://dx.doi.org/10.1016/j.infsof.2014.04.013

ii. The common rate between a traditional algorithm and its cost-aware version (e.g., between Greedy and $Greedy_{EIrreplaceability}$) is often low and this phenomenon usually leads to a significant improvement in the reduction capability of the traditional reduction algorithms, thus indicating that the improvements achieved by the presented technique are satisfactory; see Tables 19, 21, 23, 25, 27 and 29 for more details.

(3) *Replying to RQ3*: When we involve all the nine algorithms in the comparisons, the $Greedy_{EIrreplaceability}$ algorithm always provides the best cost reduction capability for all of the selected subject programs. The three traditional algorithms, especially the Greedy algorithm, generally demonstrate less satisfactory cost reduction capabilities. Additionally, it should be noticed that, with the exception of $Greedy_{Ratio}$, the algorithms integrated with Ratio generally perform worse than those integrated with EIrreplaceability. That is, although $Greedy_{Ratio}$ normally demonstrates worse reduction capability than $Greedy_{EIrreplaceability}$, it often outperforms $GRE_{EIrreplaceability}$ and $HGS_{EIrreplaceability}$, as revealed in Tables 30 and 31.

On the whole, the empirical studies indicate that incorporating the concept of test case irreplaceability into the traditional test suite reduction algorithms, especially for the Greedy algorithm, can lead to the representative sets with low execution costs.

In future work, there are some open issues that we want to address in order to further improve the efficiency of regression testing. First, we plan to examine the effectiveness of applying the proposed approach to programs from different domains, such as database applications [48–50], since we currently focused on standard programs from SIR. Second, the presented method is general and can handle the different types of execution costs, such as memory usage, network-bandwidth, disk input/output, and energy. Thus, we also intend to examine the effectiveness of applying it to reduce the other types of costs associated with test suites. Third, we will apply the concept of test irreplaceability to address the test case prioritization problem [13,20,27]. Fourth, we aim to upgrade the presented approach by enhancing the capability of revealing faults since the current version only focuses on reducing the execution cost of a test suite. For example, we intend to investigate the interaction between test irreplaceability and fault localization techniques [51,52] and then incorporate fault localization information into the cost-aware reduction algorithms to achieve better fault detection capabilities. We also want to use fault seeding tools such as MAJOR [53] to insert faults into the subject programs and conduct an empirical study to evaluate the upgraded algorithms (i.e., the algorithms associated with fault localization information) in terms of fault detection rate (i.e., the number of faults detected by per unit of execution cost). Ultimately, combining the concept of test case irreplaceability with the results from the suggested future work will yield a comprehensive framework for achieving efficient and effective regression testing.

Acknowledgement

This work was supported by the National Science Council Taiwan, under Grants NSC 100-2221-E-415-007-MY2 and NSC 102-2221-E-415-009.

References

[1] P. Ammann, J. Offutt, Introduction to Software Testing, Cambridge University Press, 2008.
[2] D. Binkley, Semantics guided regression test cost reduction, IEEE Trans. Softw. Eng. 23 (8) (1997) 498–516.
[3] H. Zhong, L. Zhang, H. Mei, An experimental study of four typical test suite reduction techniques, Inform. Softw. Technol. 50 (6) (2008) 534–546.
[4] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, B. Meyer, Efficient unit test case minimization, in: Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, ACM, November 2007, pp. 417–420.
[5] M. Prasanna, S.N. Sivanandam, R. Venkatesan, R. Sundarrajan, A survey on automatic test case generation, Acad. Open Internet J. 15 (2005).
[6] 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.
[7] 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.
[8] I. Sommerville, Software Engineering, ninth ed., Addison-Wesley, 2010.
[9] 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, IEEE Computer Society, August 2012, pp. 257–260.
[10] G.M. Kapfhammer, Regression Testing, The Encyclopedia of Software Engineering, Taylor and Francis – Auerbach Publications, 2010.
[11] J.W. Lin, C.Y. Huang, Analysis of test suite reduction with enhanced tie-breaking techniques, Inform. Softw. Technol. 51 (4) (2009) 679–690.
[12] R.M. Karp, Reducibility among Combinatorial Problems, Complexity of Computer Computations, Plenum Press, 1972. pp. 85–103.
[13] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: a survey, Softw. Test., Verif. Reliab. 22 (2) (2012) 67–120.
[14] E. Engstrom, P. Runeson, M. Skoglund, A systematic review on regression test selection techniques, Inform. Softw. Technol. 52 (1) (2010) 14–30.
[15] V. Chvatal, A Greedy heuristic for the set-covering problem, Math. Oper. Res. 4 (3) (1979) 233–235.
[16] T.Y. Chen, M.F. Lau, A new heuristic for test suite reduction, Inform. Softw. Technol. 40 (5–6) (1998) 347–354.
[17] T.Y. Chen, M.F. Lau, A simulation study on some heuristics for test suite reduction, Inform. Softw. Technol. 40 (13) (1998) 777–787.
[18] 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, ACM, May 2006, pp. 636–640.
[19] X.Y. Ma, Z.F. He, B.K. Sheng, C.Q. Ye, A genetic algorithm for test-suite reduction, in: Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, IEEE, October 2005, pp. 133–139.
[20] 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, ACM, March 2009, pp. 461–467.
[21] S. Yoo, M. Harman, Pareto efficient multi-objective test case selection, in: Proceedings of the 16th ACM International Symposium on Software Testing and Analysis, ACM, July 2007, pp. 140–150.
[22] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multiobjective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput. 6 (2) (2002) 182–197.
[23] S. Yoo, M. Harman, Using hybrid algorithm for Pareto efficient multi-objective test suite minimisation, J. Syst. Softw. 83 (4) (2010) 689–701.
[24] M. Bozkurt, Cost-aware pareto optimal test suite minimization for service-centric systems, in: Proceedings of the 15th ACM International Conference on Genetic and Evolutionary Computation, ACM, July 2013, pp. 1429–1436.
[25] C.G. Chung, J.G. Lee, An enhanced zero-one optimal path set selection method, J. Syst. Softw. 39 (2) (1997) 145–164.
[26] S. Tallam, N. Gupta, A concept analysis inspired greedy algorithm for test suite minimization, Softw. Eng. Notes 31 (1) (2006) 35–42.
[27] 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.
[28] G.V. Jourdan, P. Ritthiruangdech, H. Ural, Test suite reduction based on dependence analysis, Lect. Notes Comput. Sci. 4263 (2006) 1021–1030.
[29] S. McMaster, A. Memon, Call-stack coverage for GUI test suite reduction, IEEE Trans. Softw. Eng. 34 (1) (2008) 99–115.
[30] J.G. Lee, C.G. Chung, An optimal representative set selection method, Inform. Softw. Technol. 42 (1) (2000) 17–25.
[31] S. Elbaum, A.G. Malishevsky, G. Rothermel, Test case prioritization: a family of empirical studies, IEEE Trans. Softw. Eng. 28 (22) (2002) 159–182.
[32] S. Elbaum, A. Malishevsky, G. Rothermel, Incorporating varying test costs and fault severities into test case prioritization, in: Proceedings of the 23rd ACM/IEEE International Conference on Software Engineering, IEEE Computer Society, May 2001, pp. 329–338.
[33] A.G. Malishevsky, J.R. Ruthruff, G. Rothermel, S. Elbaum, Cost-Cognizant Test Case Prioritization, Technical Report TR-UNL-CSE-2006-0004, University of Nebraska-Lincoln, March 2006, Lincoln, USA.
[34] H. Park, H. Ryu, J. Baik, Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing, in: Proceedings of the 2nd IEEE International Conference on Secure System Integration and Reliability Improvement, IEEE Computer Society, July 2008, pp. 39–46.
[35] 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.

[36] K.R. Walcott, M.L. Soffa, G.M. Kapfhammer, R.S. Roos, Time-aware test suite prioritization, in: Proceedings of the 15th ACM International Symposium on Software Testing and Analysis, ACM, July 2006, pp. 1–12.
[37] S. Alspaugh, K.R. Walcott, M. Belanich, G.M. Kapfhammer, M.L. Soffa, Efficient time-aware prioritization with knapsack solvers, in: Proceedings of the 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies, ACM, November 2007, pp. 17–31.
[38] L. Zhang, S.S. Hou, C. Guo, T. Xie, H. Mei, Time-aware test-case prioritization using integer linear programming, in: Proceedings of the 18th ACM International Symposium on Software Testing and Analysis, ACM, July 2009, pp. 213–224.
[39] D. You, Z. Chen, B. Xu, B. Luo, C. Zhang, An empirical study on the effectiveness of time-aware test case prioritization techniques, in: Proceedings of the 26th ACM Symposium on Applied Computing, ACM, March 2011, pp. 1451–1456.
[40] T. Mücke, M. Huhn, Minimizing test execution time during test generation, Int. Fed. Inform. Process. 227 (2007) 223–235.
[41] H. Do, S. Elbaum, G. Rothermel, Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact, Empirical Softw. Eng. 10 (4) (2005) 405–435.
[42] 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, IEEE Computer Society, May 1994, pp. 191–200.
[43] 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, IEEE Computer Society, November 1998, pp. 44–53.
[44] L. Corral, A.B. Georgiev, A. Sillitti, G. Succi, A method for characterizing energy consumption in Android smartphones, in: Proceedings of the 2nd International Workshop on Green and Sustainable Software, IEEE, May 2013, pp. 38–45.
[45] 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, IEEE Computer Society, November 1998, pp. 34–43.
[46] A. Arcuri, L. Briand, A practical guide for using statistical tests to assess randomized algorithms in software engineering, in: Proceedings of the 33rd ACM/IEEE International Conference on Software Engineering, ACM, May 2011, pp. 1–10.
[47] S. Kpodjedo, F. Ricca, G. Antoniol, P. Galinier, Evolution and search based metrics to improve defects prediction, in: Proceedings of the 1st International Symposium on Search Based Software Engineering, IEEE Computer Society, May 2009, pp. 23–32.
[48] S.R. Clark, J. Cobb, G.M. Kapfhammer, J.A. Jones, M.J. Harrold, Localizing SQL faults in database applications, in: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, IEEE, November 2011, pp. 213–222.
[49] G.M. Kapfhammer, M.L. Soffa, Database-aware test coverage monitoring, in: Proceedings of the 1st India Software Engineering Conference, ACM, February 2008, pp. 77–86.
[50] F. Haftmann, D. Kossmann, E. Lo, A framework for efficient regression tests on database applications, Int. J. Very Large Data Bases 16 (1) (2007) 145–164.
[51] 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 ACM/IEEE International Conference on Software Engineering, ACM, May 2008, pp. 201–210.
[52] B. Jiang, Z. Zhang, W.K. Chan, T.H. Tse, T.Y. Chen, How well does test case prioritization integrate with statistical fault localization?, Inform. Softw. Technol. 54 (7) (2012) 739–758.
[53] R. Just, F. Schweiggert, G.M. Kapfhammer, MAJOR: an efficient and extensible tool for mutation analysis in a Java compiler, in: Proceedings of the 26th IEEE/ACM International Conference on Automated Software Engineering, IEEE, November 2011, pp. 612–615.

Please cite this article in press as: C.-T. Lin et al., Test suite reduction methods that decrease regression testing costs by identifying irreplaceable tests, Inform. Softw. Technol. (2014), http://dx.doi.org/10.1016/j.infsof.2014.04.013