Reducing the Cost of Regression Testing by Identifying Irreplaceable Test Cases

Chu-Ti Lin[1], Kai-Wei Tang[2], Cheng-Ding Chen[1], Gregory M. Kapfhammer[3]

[1]Dept. of Computer Sci. and Info. Eng.
National Chiayi University
Chiayi, Taiwan
{chutilin,s0990394}@mail.ncyu.edu.tw

[2]Cloud System Software Institute
Institute for Information Industry
Taipei, Taiwan
kwtang@iii.org.tw

[3]Dept. of Computer Science
Allegheny College
Meadville, PA
gkapfham@allegheny.edu

***Abstract*—Test suite reduction techniques decrease the cost of software testing by removing the redundant test cases from the test suite while still producing a reduced set of tests that yields the same level of code coverage as the original suite. Most of the existing approaches to reduction aim to decrease the size of the test suite. Yet, the difference in the execution cost of the tests is often significant and it may be costly to use a test suite consisting of a few long-running test cases. Thus, this paper proposes an algorithm, based on the concept of test irreplaceability, which creates a reduced test suite with a decreased execution cost. Leveraging widely used benchmark programs, the empirical study shows that, in comparison to existing techniques, the presented algorithm is the most effective at reducing the cost of running a test suite.**

Keywords-regression testing; test suite reduction; code coverage

I. INTRODUCTION

According to the IEEE definition [1], 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 underlying software system will be executed to 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 [2].

Evolutionary development, incremental delivery, and software maintenance are common in software development [3]. 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 tests 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 [1]. 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.

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. That is, given the original test suite $T=\{t_1, t_2, t_3, ..., t_n\}$ and a set of test requirements $R=\{r_1, r_2, r_3, ..., r_m\}$, the goal is to find a subset of the test suite *T*, denoted by a representative set *RS*, to satisfy all the test requirements satisfied by *T*. The process of finding the representative set is called test suite reduction [4], [5].

The minimum set cover problem can be reduced to the test suite minimization problem in polynomial time [6]. Karp proved that the set cover problem is NP-complete [7]. 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. Yet, to the best of our knowledge, most of the existing reduction algorithms ignore the significant differences in the execution costs of the tests. In response to this limitation, this paper describes and empirically evaluates a technique that uses the concept of test irreplaceability to create a representative set with low execution cost.

II. RELATED WORK

A. Greedy Algorithm

The Greedy algorithm is a commonly-used method for finding the near-optimal solution to the test suite reduction problem [8]. This algorithm repeatedly removes the test which covers the most unsatisfied test requirements from the test suite set *T* to *RS* until all of the requirements are covered. Many existing test suite reduction methods are based on the concept of the Greedy algorithm [9]. In other words, many algorithms repetitively choose the "best" test case to obtain the near-optimal solution from the locally optimal solutions.

B. 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 [10]. 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 [10] proposed two algorithms, called GE and GRE, to address this problem. The GE algorithm will choose all of the essential test cases first and then will apply the

Greedy algorithm to the remaining test suite. The GRE algorithm is the enhanced version of GE. 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 only when the essential and 1-to-1 redundant strategies are adopted during the reduction process [10].

C. HGS Algorithm

The HGS algorithm proposed by Harrold et al. [4] is another approach to test suite reduction that has received considerable attention. Let T_i (for $i = 1, 2, 3, \ldots, m$) represent the subsets of T, with each subset T_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.

III. Reducing the Execution Cost of a Test Suite

Due to the differences in the execution costs between the test cases, the representative set with the smallest number of tests may not be the one with the minimum execution cost. As such, the cost of a test should be a more important consideration for achieving cost-effective testing than the size of the test suite. Thus, it is necessary to consider individual execution costs when choosing the test cases.

A. Review of the ReduceWithRatio Algorithm

The coverage increase per unit of cost consumption may be an intuitive metric to evaluate a test case. Given that *Coverage*(t) represents the number of uncovered test requirements satisfied by test case t, Ma et al. [11] and Smith and Kapfhammer [12] evaluated the test cases using

$$Ratio(t) = \frac{Coverage(t)}{Cost(t)}\ , \tag{1}$$

where *Cost*(t) represents the execution cost of the test case t. 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.

The Greedy algorithm repeatedly includes the test case with the maximum *Coverage*(t) until all of the test requirements are satisfied. Instead of *Coverage*(t), Smith and Kapfhammer's algorithm, hereafter called ReduceWithRatio, repeatedly includes the test case with the maximum *Ratio*(t) until all of the test requirements are satisfied.

B. Reducing with Irreplaceability

In preliminary studies, we found circumstances in which ReduceWithRatio did not produce the best reduced test suite. As an illustrative example, let's consider Table I's test suite that satisfies six requirements with four tests that have varying cost. Table II demonstrates the steps that ReduceWithRatio would take to reduce the test suite from Table I. This table shows that the ratio-based method creates the representative set $RS_1=\{t_1, t_2, t_3\}$ with an execution cost of 14. However, the subset $\{t_2, t_3\}$, with cost 10, is enough to satisfy all test requirements. That is, the best representative set will exclude t_1 in order to minimize the execution cost.

In fact, 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. Therefore, we posit that t has a higher replaceability with respect to r in this case. According to our initial observations, we found that:

1. A representative set may not have the lowest execution cost if it includes a test case with high replaceability.
2. Because the ReduceWithRatio algorithm only considers the 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 [13] 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

$$ContributionToSuite(t) = \sum_{i=1}^{m} Contribution(t, r_i)\ , \tag{2}$$

where $R=\{r_1, r_2, r_3, ..., r_m\}$ is the test requirements and

$$Contribution(t, r_i) = \begin{cases} 0, \text{ if } t \text{ cannot satisfy } r_i, \\ \dfrac{1}{\text{the number of test cases that satisfy } r_i}, \text{ if } t \text{ satisfies } r_i. \end{cases} \tag{3}$$

The number of test cases that satisfy the test requirement r_i in (3) is positively related to the replaceability of t with respect to r_i. 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 (2) to evaluate the irreplaceability of test cases in the test suite. Based on this concept, we combine the execution cost of a test with (2) and evaluate the irreplaceability of test cases by

$$Irreplaceability(t) = \frac{\sum_{i=1}^{m} Contribution(t, r_i)}{Cost(t)}\ . \tag{4}$$

As mentioned previously, *Coverage*(t) and *Ratio*(t) have been adopted to evaluate and choose the test cases. Similarly, *Irreplaceability*(t) can be used to evaluate test cases and repeatedly choose the test with the maximum value until all of the test requirements are satisfied. Fig. 1 gives the pseudo code of ReduceWithIrreplaceability and Table III shows the steps associated with applying this algorithm to the example in Table I. In comparison to the sets produced by the other methods, it is evident that the ReduceWithIrreplaceability algorithm creates the representative set $RS_2=\{t_2, t_3\}$ with the lowest overall execution cost.

TABLE I. An Example of a Test Suite and Test Requirements.

Test case		Requirements to be satisfied					
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 II. APPLYING ReduceWithRatio TO THE EXAMPLE.

Initial	$T = \{t_1, t_2, t_3, t_4\}$, $R = \{r_1, r_2, r_3, r_4, r_5, r_6\}$, $RS_1 = \{\ \}$								
	Test case	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	$T = \{t_2, t_3, t_4\}$, $R = \{r_4, r_5, r_6\}$, $RS_1 = \{t_1\}$								
	Test case	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	$T = \{t_2, t_4\}$, $R = \{r_4, r_5\}$, $RS_1 = \{t_1, t_3\}$								
	Test case	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	$T = \{t_4\}$, $R = \{\ \}$, $RS_1 = \{t_1, t_2, t_3\}$, Cost of $RS_1 = 14$								
	Test case	Cost	–	–	–	–	–	–	*Ratio*(t)
	$\boxed{t_1}$	4	–	–	–	–	–	–	–
	$\boxed{t_2}$	7	–	–	–	–	–	–	–
	$\boxed{t_3}$	3	–	–	–	–	–	–	–
	t_4	4	–	–	–	–	–	–	0

TABLE III. APPLYING ReduceWithIrreplaceability TO THE EXAMPLE.

Initial	$T = \{t_1, t_2, t_3, t_4\}$, $R = \{r_1, r_2, r_3, r_4, r_5, r_6\}$, $RS_5 = \{\ \}$								
	Test case	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	$T = \{t_1, t_3, t_4\}$, $R = \{r_1, r_6\}$, $RS_2 = \{t_2\}$								
	Test case	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	$T = \{t_1, t_4\}$, $R = \{\ \}$, $RS_2 = \{t_2, t_3\}$, Cost of $RS_2 = 10$								
	Test case	Cost	–	–	–	–	–	–	*Irreplaceability*(t)
	t_1	4	–	–	–	–	–	–	0.00
	$\boxed{t_2}$	7	–	–	–	–	–	–	–
	$\boxed{t_3}$	3	–	–	–	–	–	–	–
	t_4	4	–	–	–	–	–	–	0.00

IV. EXPERIMENTAL ANALYSES

A. *Experiment Description*

In addition to the proposed algorithm that uses test irreplaceability, we also selected Greedy, GRE, HGS and ReduceWithRatio for comparison. Moreover, since the Siemens suite of programs [5], [14], as described in Table IV and obtained from the Software-artifact Infrastructure Repository (SIR) [15], are frequently chosen benchmarks for evaluating test suite reduction methods, we used them in the experiments. Following an empirical setup similar to the one in [5] and [16], we took these steps:

1. Randomly generate an integer z, $1 \leq z \leq 0.5 \times loc$;
2. Randomly pick z test cases from the test pool for each subject program, and include those z test cases in T;
3. Check whether the test cases in T can satisfy all of the test requirements or not. 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.

After collecting the execution times of the tests and taking their average, we performed test suite reduction for each of the 1000 generated test suites. In these experiments, we use the percentage of suite cost reduction (SCR), as defined in (5), to evaluate the reduction capability.

$$\mathrm{SCR}(T, RS) = \frac{Cost(T) - Cost(RS)}{Cost(T)} \times 100\%, \qquad (5)$$

where *Cost*(T) represents the cost required to execute the original test suite T, and *Cost*(RS) represents the cost associated with running the representative set RS.

TABLE IV. DESCRIPTION OF THE SIR SUBJECT PROGRAMS.

Subject Programs	**Size of Test Pool**	**Num. of Test Requirements**
printtokens	4,130	140
printtokens2	4,115	138
replace	5,542	126
schedule	2,650	46
schedule2	2,710	72
tcas	1,608	16
totinfo	1,052	44

B. *Experimental Results*

Table V furnishes the experimental results for the five algorithms. Here $\mathrm{RS}_{\mathrm{Greedy}}$, $\mathrm{RS}_{\mathrm{GRE}}$, $\mathrm{RS}_{\mathrm{HGS}}$, $\mathrm{RS}_{\mathrm{ReduceWithRatio}}$ and $\mathrm{RS}_{\mathrm{ReduceWithIrreplaceability}}$ denote the representative sets obtained by applying the selected algorithms, respectively. From this table, it is clear that all selected algorithms can significantly reduce the execution costs of test suites; the SCR values are greater than 87% in all cases. Among all of the algorithms, ReduceWithIrreplaceability achieves the best SCR scores for all of the subject programs. With the exception of schedule, the Greedy algorithm produces the worst representative set for the programs. Moreover, even though ReduceWithRatio considers test execution cost, it performs worse than the traditional algorithms (i.e., GRE and HGS) for printtokens and printtokens2. On the whole, both ReduceWithIrreplaceability and ReduceWithRatio exhibit excellent cost reduction capabilities, but the SCR scores of ReduceWithRatio are not as good as those of ReduceWithIrreplaceability. It should be noted that the SCR values shown in Table V are close to each other because the execution cost of the original generated test suite is quite high. However, the differences in the execution costs of the representative sets are considerable.

One threat to the validity of our empirical study includes the measurement of the execution time of each test case. Although we executed each test 1000 times and took the average of the execution costs, the measure for each test case may change if we performed the experiment in another environment. The second threat to validity is related to the fact that the experiments focused on the relatively small Siemens programs. While these programs are a good starting point for experimentation, our findings may not be relevant to larger programs and test suites.

```
algorithm ReduceWithIrreplaceability
input    T: the set of test cases
         R: the set of requirements
         S: the relation between T and R, S={(t, r)| t satisfies r, t ∈ T, and r ∈ R}
output   RS: a representative set of T
begin
    RS = { };
    while (R is not empty)
    {
         t = the test case in T that has the maximum irreplaceability;
         RS = RS ∪ {t};
         T = T − {t};
         R = R − {the requirements covered by t};
    }
    return RS;
end
```

Figure 1. Pseudo code of the ReduceWithIrreplaceability algorithm.

TABLE V. COMPARISON OF THE REPRESENTATIVE SETS PRODUCED BY THE SELECTED ALGORITHMS.

Test Suite / Program	Original	RS_{Greedy}		RS_{GRE}		RS_{HGS}		$RS_{ReduceWithRatio}$		$RS_{ReduceWithIrreplaceability}$	
	Cost[a]	Cost[a]	SCR	Cost[a]	SCR	Cost[a]	SCR	Cost[a]	SCR	Cost[a]	SCR
Printtokens	914.67	117.32	87.17%	100.63	89.00%	98.26	89.26%	115.04	87.42%	81.73	91.06%
printtokens2	717.84	58.29	91.88%	53.98	92.48%	56.65	92.11%	56.19	92.17%	48.53	93.24%
Replace	1068.90	88.28	91.74%	83.51	92.19%	87.46	91.82%	81.06	92.42%	76.06	92.88%
Schedule	493.77	18.71	96.21%	18.14	96.33%	19.45	96.06%	16.35	96.69%	15.32	96.90%
schedule2	651.82	40.14	93.84%	37.70	94.22%	37.04	94.32%	28.60	95.61%	26.80	95.89%
Tcas	219.39	23.74	89.18%	22.85	89.58%	22.85	89.58%	21.53	90.19%	20.74	90.55%
Totinfo	690.97	52.15	92.45%	48.62	92.96%	43.82	93.66%	26.43	96.17%	26.14	96.22%

a. Indicates the cost required to execute the original test suite or the representative set, which is measured in millisecond (ms).

V. CONCLUSION AND FUTURE WORK

Most existing test suite reduction algorithms attempt to minimize the size of a regression test suite. Since the ReduceWithRatio algorithm does not always perform in a satisfactory manner, this paper presents an algorithm that uses irreplaceability to evaluate the importance of tests and ultimately produce reduced test suites with a substantially decreased execution cost. The experimental results indicate that ReduceWithIrreplaceability is the best method for decreasing the cost of test suite execution, according to the SCR metric. In future work, we intend to enhance the cost reduction capabilities of GRE and HGS by incorporating test irreplaceability into their operation. Furthermore, we will conduct additional experiments with larger subject programs.

ACKNOWLEDGMENT

This research was sponsored by the National Science Council of Taiwan (Grant: NSC 100-2221-E-415-007-MY2).

REFERENCES

[1] D. Binkley, "Semantics Guided Regression Test Cost Reduction," *IEEE Trans. on Software Engineering*, Vol. 23, No. 8, pp. 498-516, August 1997.

[2] H. Zhong, L. Zhang, and H. Mei, "An Experimental Study of Four Typical Test Suite Reduction Techniques," *Information and Software Technology*, Vol. 50, No. 6, pp. 534-546, May 2008.

[3] I. Sommerville, *Software Engineering*, Addison-Wesley, ninth ed., 2010.

[4] M. J. Harrold, R. Gupta, and M. L. Soffa, "A Methodology for Controlling the Size of a Test Suite," *ACM Trans. on Software Engineering and Methodology*, Vol. 2, No. 3, pp. 270-285, July 1993.

[5] D. Jeffrey and N. Gupta, "Improving Fault Detection Capability by Selectively Retaining Test Cases During Test Suite Reduction," *IEEE Trans. on Software Engineering*, Vol. 33, No. 2, pp. 108-123, February 2007.

[6] J. W. Lin and C. Y. Huang, "Analysis of Test Suite Reduction with Enhanced Tie-Breaking Techniques," *Information and Software Technology*, Vol. 51, No. 4, pp. 679-690, April 2009.

[7] R. M. Karp, "Reducibility among Combinatorial Problems," *Complexity of Computer Computations*, Plenum Press, pp. 85-103, 1972.

[8] V. Chvatal, "A Greedy Heuristic for the Set-Covering Problem," *Mathematics Operations Research*, Vol. 4, No. 3, pp. 233-235, August 1979.

[9] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Prioritization: a Survey," *Software Testing, Verification and Reliability*, Vol. 22, No. 2, March 2012.

[10] T. Y. Chen and M. F. Lau, "A New Heuristic for Test Suite Reduction," *Information and Software Technology*, Vol. 40, No. 5-6, pp. 347-354, July 1998.

[11] X. Y. Ma, Z. F. He, B. K. Sheng, and C. Q. Ye, "A Genetic Algorithm for Test-Suite Reduction," *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, October 2005.

[12] A. M. Smith and G. M. Kapfhammer, "An Empirical Study of Incorporating Cost into Test Suite Reduction and Prioritization," *Proceedings of the 24th ACM SIGAPP Symposium on Applied Computing, Software Engineering Track*, March 2009.

[13] J. A. Jones and M. J. Harrold, "Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage," *IEEE Trans. on Software Engineering*, Vol. 29 No. 3, pp. 195-209, March 2003.

[14] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments on the Effectiveness of Dataflow- and Controlflow-based Test Adequacy Criteria," *Proceedings of the 16th International Conference on Software Engineering*, pp. 191-200, May 1994.

[15] H. Do, S. Elbaum, and G. Rothermel, "Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and its Potential Impact," *Empirical Software Engineering*, Vol. 10, No. 4, pp. 405-435, October 2005.

[16] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong, "An Empirical Study of the Effects of Minimization on the Fault Detection Capabilities of Test Suites," *Proceedings of the 14th International Conference on Software Maintenance*, pp. 34-43, November 1998.