2010 10th International Conference on Quality Software

A Simulation Study on Some Search Algorithms for Regression Test Case Prioritization

Sihan Li, Naiwen Bian, Zhenyu Chen*, Dongjiang You, Yuchen He
State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China
Software Institute, Nanjing University, Nanjing 210093, China
*zychen@software.nju.edu.cn

***Abstract*—Test case prioritization is an approach aiming at increasing the rate of faults detection during the testing phase, by reordering test case execution. Many techniques for regression test case prioritization have been proposed. In this paper, we perform a simulation experiment to study five search algorithms for test case prioritization and compare the performance of these algorithms. The target of the study is to have an in-depth investigation and improve the generality of the comparison results. The simulation study provides two useful guidelines: (1) Two search algorithms, Additional Greedy Algorithm and 2-Optimal Greedy Algorithm, outperform the other three search algorithms in most cases. (2) The performance of the five search algorithms will be affected by the overlap of test cases with regard to test requirements.**

Keywords- regression testing, test case prioritization, simulation study, search algorithms

I. INTRODUCTION

Regression testing, which intends to ensure that a software program works as specified after changes have been made to it, is an important phase in software development lifecycle. Unfortunately, regression testing could become more and more costly with changes increasing. To deal with this problem, many test approaches have been proposed. This paper focuses on one of these approaches-- test case prioritization [8-11] [13-16].

Test case prioritization is an approach aiming at increasing the rate of faults detection during the testing phase, by reordering test case execution. Test case prioritization makes regression testing effective because it provides a way to detect faults as early as possible and reserve more time to fix the bugs found.

In previous studies, researchers have proposed many techniques for regression test case prioritization. Their results show that most of the existing techniques could improve the rate of faults detection, but they have different performance and complexity in various situations. Some of the techniques are coverage based [5][10-12][14], and noncoverage based techniques [4][6][9][16] also take up a portion. Here, coverage serves as a surrogate measure for faults found [7]. In [7], Li et al. investigated five search algorithms: three greedy algorithms (Total Greedy, Additional Greedy, and 2-Optimal Greedy), together with two meta-heuristic search techniques (Hill Climbing and Genetic Algorithms), and conducted an empirical study to compare the performance of these five algorithms by applying them to six programs ranging from 374 to 11,148 lines of code. However, the experiment in [7] studied only a small set of relatively small programs, additional studies examining a wider range of factors are needed.

In order to have an in-depth investigation and improve the generality of the comparison results, we perform a simulation experiment. In our experiment, we use test requirements, which can be blocks, decisions, statements, and other coverage criteria in practice, as the test objective to be satisfied by test cases. The numbers of test requirements and test cases are randomly generated so as to simulate all circumstances in reality, and of course, these numbers are ought to comply with some constraints.

A simulation study can propose a guideline about how to choose these five algorithms in various cases for test case prioritization. In the simulation experiment, large quantities of data are analyzed to compare the performance of these five search algorithms, thus improving the generality of the comparison results. In particular, our simulation study makes two significant contributions:

- Two search algorithms, Additional Greedy Algorithm and 2-Optimal Greedy Algorithm, outperform the other three search algorithms in most cases. Hence, these two search algorithms are recommended to be used preferentially.
- A measure, namely the ratio of overlapping [14], is introduced to better investigate the performance of these five search algorithms. Our results indicate that the performance of two best search algorithms, Additional Greedy Algorithm and 2-Optimal Greedy Algorithm, reaches the maximum when the ratio of overlapping is around 3 and it will decline when the ratio of overlapping increases.

The remainder of this paper is organized as follows. Section II reviews the test case prioritization problem and describes the five search algorithms used to prioritize test cases. Section III describes our simulation experiment including its design, parameters, procedures, measurements. Section IV reports the results of our experiment and our analysis. Section V reviews

The work described in this article was partially supported by the National Natural Science Foundation of China (90818027, 60803007) and the Opening Project of Shanghai Key Laboratory of Computer Software Evaluating & Testing(09DZ2272600).

1550-6002/10 $26.00 © 2010 IEEE
DOI 10.1109/QSIC.2010.15

the previous related work. Section VI presents the conclusion and our future work.

II. PRELIMINARIES

A. Test Case Prioritization

Test case prioritization sorts the test cases so that test cases with higher priority, according to some criteria, are executed earlier in the regression testing process. The test case prioritization problem is formulized by Rothermel et al. in [11] as follows:

Given: T, a test suite; PT, the set of permutations of T; f, a function from PT to the real numbers.

Problem: Find $T' \in PT$ such that

$$(\forall T'')(T'' \in PT)(T'' \neq T')[f(T') \geq f(T'')]$$

Here, PT represents the set of all possible orderings of T and f is a function that is applied to any such ordering and yields an award value for that ordering.

Although test case prioritization is done for many possible goals, in this paper, we focus on the rate of faults found — a measure of how quickly faults are found within the testing process. Since the rate of faults found cannot be determined in advance, we use APRC as a surrogate measure. Average Percentage Requirement Coverage, abbreviated as APRC, measures the rate at which a test suite covers the requirements. We compute APRC with the equation below, in which N_i denotes the value of the order (from 1 to m) of the first test case that satisfies the i_{th} requirement.

$$\text{APRC} = 1 - \frac{N_1 + N_2 + ... + N_m}{mn} + \frac{1}{2n} \qquad (1)$$

For example in Table Ⅰ, if the order of the test cases is T_4, T_3, T_2, T_1, then N_1 should be 2, since R_1 is first satisfied by T_3, the second test case in this order. n and m should be 4 and 10 respectively. Thus, the APRC score under this order is 0.675.

B. Algorithms

1) Total Greedy Algorithm

The principle of all Greedy Algorithms is to select the most promising element at the current moment. This algorithm maintains two sets of test cases, one is the chosen set, the other is the candidate set. Its strategy is to repeatedly select one test case which satisfies the most test requirements from the candidate set, regardless of the requirement's current state (satisfied or not). If there are several test cases satisfying the same number of the requirements, one of them is randomly picked. Under the circumstances of test case prioritization, this process terminates when all the test requirements are satisfied. After full coverage of the test requirements has been accomplished, the rest of the test cases in candidate set (if any) can be selected using any strategy. In our experiments, we select these test cases in a random order for simplicity. For the instance in Table Ⅰ, T_3 will be selected first. It satisfies six requirements, the maximum of the four test cases. Next, we can either select T_1 or T_2 since they both cover five requirements. T_4, which satisfies only three requirements, is the last test case to be selected. Thus, the order of test case execution could either be T_3, T_1, T_2, T_4 *or* T_3, T_2, T_1, T_4.

TABLE I. TEST SUITE REQUIREMENT COVERAGE

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1		X	X	X			X		X	
T_2	X				X	X		X		X
T_3	X		X	X		X	X	X		
T_4		X							X	X

2) Additional Greedy Algorithm

The main difference between Total Greedy Algorithm and Additional Greedy Algorithm is that Additional Greedy Algorithm selects the test case that satisfies the most not-yet-satisfied requirements. This means that each step makes decision on the basis of information from previous steps. For the instance in Table Ⅰ, T_3 is chosen first since it covers the most uncovered requirements. After this, the uncovered requirements are R_2, R_5, R_9 and R_{10}. T_1, T_2 and T_4 cover 2, 2 and 3 uncovered requirements respectively. So T_4 is selected next leaving R_5 to be the only uncovered requirement. T_2 covers R_5 while T_1 does not. Thus, the order should be T_3, T_4, T_2, T_1.

3) 2-Optimal Greedy Algorithm

2-Optimal Greedy Algorithm is an instance of k-Optimal Greedy Algorithm. Considering the balance between time and effectiveness, we choose 2-Optimal for our experiment. This algorithm selects the next two test cases that together satisfy the most test requirements which have not been satisfied so far. Note that, the internal order of the chosen pair of test cases also should be considered. We arrange the test case that satisfies the more test requirements before the other, for this could improve the performance of the algorithm with little overheads. Still for the instance in Table Ⅰ, T_1 and T_2 are the first to be select. They cover ten requirements together, the maximum that can be covered by a pair of test cases. These two test cases can be arranged in any order since they both cover five requirements. Since T_1 and T_2 have covered all the requirements, any order of T_3 and T_4 has the same effect. So T_1, T_2, T_3, T_4 can be one of the solutions.

All greedy approaches can be called "short-sighted" algorithms, for they always take the local best element iteratively. This step-by-step recipe may produce a suboptimal solution instead of the global optimal solution. Table Ⅱ presents a case in which Total Greedy Algorithm does not produce an optimal solution. Obviously, the algorithm will return a solution T_1, T_2, T_3, T_4. And the APRC score of the solution is 0.675. However, a solution with this order T_1, T_3, T_2, T_4 has an APRC score of 0.775, which is higher than that produced by Total Greedy Algorithm.

TABLE II. A CASE IN WHICH TOTAL GREEDY ALGORITHM DOES NOT PROCUCE AN OPTIMAL SOLUTION

	R_1	R_2	R_3	R_4	R_5	R_6	R_7	R_8	R_9	R_{10}
T_1	X	X	X	X	X	X				
T_2	X	X	X	X	X					
T_3							X	X	X	X
T_4	X	X	X							

4) Hill Climbing

Hill Climbing belongs to the family of local search algorithms. It starts with a random solution and iteratively improves the solution a little by finding a fitter neighbor. It terminates when no further improvement can be achieved. The Hill Climbing approach in test case prioritization works as follows:

1. It starts with an initial state which is a randomly generated sequence of test cases.
2. It randomly selects two test cases and exchanges their positions.
3. The new state is accepted if and only if it has a better fitness than the previous solution.
4. Repeat the previous two steps until the current state remains unchanged for a given number of iterations.
5. Return the current state as the solution to the problem.

In our experiment, the given number of iterations is set to be 1000 for the balance between time and effectiveness.

5) Genetic Algorithms

Genetic Algorithms (GAs) are a class of adaptive heuristic search algorithms that use techniques inspired by the evolutionary idea of natural selection. Fig. 1 shows the framework of Genetic Algorithm. A GA starts with a randomly generated population. And the chromosome of each individual in the population represented a solution to the problem. There is a function to evaluate the fitness of each individual. An individual with higher fitness will have a higher probability to be selected for the reproduction. Two genetic operators, crossover and mutation, will be performed on the selected individuals to generate the next population. The algorithm ends when the specified termination condition is reached. And then the best individual which represents the solution found by the GA will be output. The detailed implementation of GA in our experiment follows the pattern adopted by Li et al. [7].

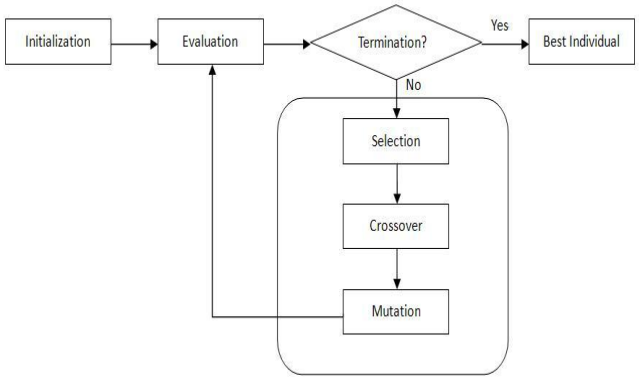

Figure 1. The framework of Genetic Algorithm.

III. EXPERIMENT

We have done an experiment to study the performance of those five search algorithms introduced in Section Ⅱ. This section describes the details of our experiment including the design, parameters, procedures and measurements of the experiment.

A. Experimental Design

This is a simulation experiment, in which we use a program to simulate the test case prioritization problem. Consider a test suite containing n test cases that covers m requirements. It can be denoted as a matrix with n rows and m columns. If the i_{th} test case satisfies the j_{th} requirement, the element in the i_{th} row j_{th} column of the matrix is set to be 1, otherwise 0. Since one test case can satisfy more than one requirement, we assume that the number of rows of the matrix is less than or equal to the number of columns, that is, $n \leq m$. Our experiment is carried out as follows:

1. Many matrices, whose elements are 1's and 0's with eligible distribution, are randomly generated, and the details of matrix generation are described in the next subsection.
2. An APRC score is computed based on each primitive matrix.
3. Five algorithms, introduced in Section Ⅱ, are applied to each matrix to implement test case ordering. Similarly, an APRC score based on each reordered matrix is computed. Thus each algorithm corresponds to an APRC score for one matrix.
4. Compare the performance of these five algorithms according to the APRC scores they improved.

Here, to avoid the redundancy in our statement, we denote the APRC score improved by each algorithm as APRCI (APRC improved) and define it as follows.

$$\text{APRCI} = \text{APRC} - \text{Initial APRC} \qquad (2)$$

Where the Initial APRC is the APRC score of the primitive matrix and the APRC is the APRC score of the reordered matrix. Hence, the value of APRCI is the only criterion to judge the performance of each algorithm.

To improve the generality of our results, various matrix with different rows and columns and different distributions of 1's and 0's are generated. Still for this reason, an abstract concept, test requirement, is employed in our experiment. And the requirements can be blocks, statements and decisions, depending on different testing criteria in practice.

As C++ is an effective programming language, we choose it as the implementation language and the developing environment is Microsoft Visual Studio 2008. We run this program on a PC with a 2.67GHz Intel Pentium 4 CPU and 4 GB memory.

B. The Parameters and Procedures in Matrix Generation

The strategy of matrix generation proposed by Chen et al. in [3] is adopted in our experiment. Consider the parameters and their values. The number of requirements is denoted as m,

and the number of test cases is denoted as n. x represents the number of requirements satisfied by a test case. μ and σ represent the mean and standard deviation of x, respectively. x_i denotes the number of requirements satisfied by the i_{th} test case. The values of the parameters are supposed to comply with some constraints. In our experiment, we assume x_i follows the normal distribution, that is $x \sim N(\mu, \sigma^2)$. The value of m is fixed at 1000, and we have suggested that $n \leq m$. Also, since the test cases in a single test suite must cover all the test requirements, the value of μn should be no less than that of m. Note that the number of test cases, the value of n, can't be too small, otherwise, the performance of different algorithms is likely to be more or less the same. Therefore, we decide to set the value of μ less than 10% of m so that more than 10 test cases are needed to satisfy all the requirements. After the value of μ is determined, the range of n is within m/μ to m. Finally, we focus on the value of σ. Since x denotes the number of requirements satisfied by a test case, its value must be greater than zero. We confirm that if the value of σ is selected within the range from 0.1μ to 0.5μ, more than 97% of the values of x in the normal distribution are greater than zero.

Table Ⅲ presents the values of the parameters used in our experiments. We set m to be 1000 and μ to be 1%, 2.5%, 5% and 10% of m. For each fixed set of m and μ, we set σ to be 0.1μ, 0.3μ, 0.5μ, and select some important points between m/μ and m for the value of n. For each fixed set of these parameters (m, n, μ, and σ), we repeat the matrix generation procedure for 100 times.

Fig. 2 shows the procedure of matrix generation. In Step1, we first generate n numbers(x_1,...,x_n) randomly and store them in an array named *normalDeviateX*. Note that if $sum(x_i) \leq m$, the *DO-WHILE* loop will not terminate, and this ensures that all m requirements are satisfied by this test suite containing n test cases, each of which satisfies x_i requirements. And then in Step2, the first *FOR* loop guarantees that each requirement must be satisfied by one test case. Each time it loops, the $k_{th}(k=1,...,m)$ test case is chosen to satisfy the $j_{th}(j=1,...,n)$ requirement. After the second *FOR* loop, the total number of requirements satisfied by the j_{th} test case is x_j for $j=1,...,n$. In this step, the array *countX* is used to record the number of requirements already satisfied by each test case during the procedure. The input parameters of the procedure in Fig. 2 are a pointer to the output matrix and four variables *mu*, *sigma*, *n* and *m* which are respectively equal to the values of μ, σ, n and m we discussed above. And the output is a pointer to a randomly generated matrix that meets all the requirements mentioned above.

C. Effectiveness Measures

A higher value of APRCI indicates that the corresponding algorithm applied performs better in the test case prioritization problem. And 100 trials with one fixed set of m, n, μ, and σ are fulfilled to obviate significant loss of generality. Afterwards, we obtain the mean of 100 APRCIs as the final measurement of each algorithm.

TABLE III. THE VALUES OF THE PARAMETERS

Group	m	μ	σ	n
i	1000	10	1-5	100-1000
ii	1000	25	2.5-12.5	40-1000
iii	1000	50	5-25	20-1000
iv	1000	100	10-50	10-1000

```
CREATEMATRIX parameters are int* matrix, int mu,int sigma,int n,
                int m

    //Step1
    DO
        FOR each i from 0 to n-1
            DO
                generate a normal deviate z following N(0,1)
                set x as roundNumber(mu + z * sigma)
            WHILE !1≤x ≤ m
            END
            store x in normalDeviateX[i]
        END
        compute sum( normalDeviateX[i]) where i is from 0 to n-1
    WHILE sum<m
    END

    //Step 2
    initialize array countX with n 0's
    FOR each j from 0 to m-1
        DO
            randomly generate a number k which satisfies 0≤k≤ n-1
        WHILE countX[k] ≥normalDeviateX[k]
        END
        increase countX[k] by 1
        set matrix[k*m+j] to 1
    END

    FOR each i from 0 to n-1
        WHILE countX[i] ≤normalDeviateX[i]
            DO
                randomly generate a number j where 0≤j≤ m-1
            WHILE matrix[i*m+j] equals 1
            END
            increase countX[i] by 1
            set matrix[i*m+j] to 1
        END
    END
```

Figure 2. Matrix generation procedure.

IV. RESULTS AND ANALYSIS

This section reports the results of our experiment and the analysis we have done on these results. Also, we list the threats to validity in this paper.

A. Study of results

First, we introduce a concept named ratio of overlapping. The ratio of overlapping, proposed by Chen et al. in [3], is denoted as $r_{overlap}$ and defined as $\mu n/m$, where μ, n, m are defined in Section Ⅲ. For example, if μ=10 and n=200, then the value of $r_{overlap}$ is 2. With the overlapping ratio introduced into our experiment, we can easily compare the performance of each algorithm across different values of μ.

Fig. 3-6 show the mean values of APRCI of each algorithm against different $r_{overlap}$ for each set of μ and σ. Generally speaking, the performance of each algorithm declines with the increase of μ and dramatically rises with the increase of σ. This is a reasonable result. A large value of μ indicates that each test case covers considerable requirements. Hence, all the requirements are possibly covered after the execution of the

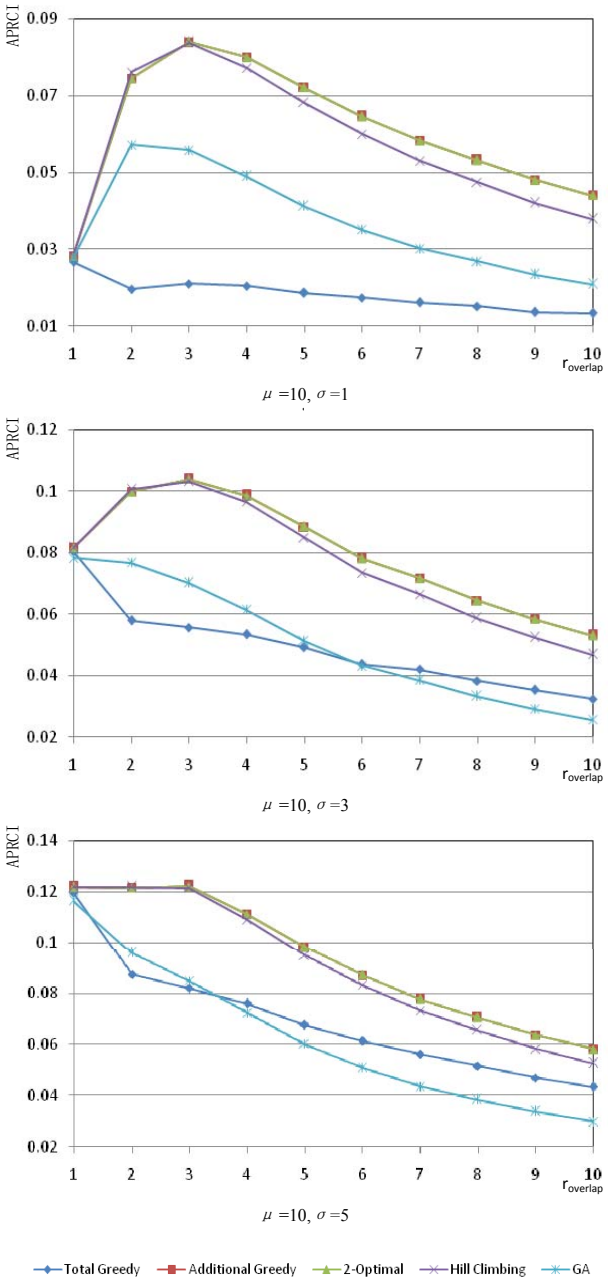

Figure 3. Plots of mean values of APRCIs corresponding to each algorithm against $r_{overlap}$ for various σ when μ=10.

first few test cases and then no further APRC scores can be improved. Also, a large value of σ suggests that the number of requirements satisfied by each test case differs a lot so that there are more potential chances for search algorithms to improve APRC score. In brief, the five search algorithms work better when the number of test requirements satisfied by each test case in a test suite is small on average and differs a lot.

Now we focus on the performance of each algorithm for different values of $r_{overlap}$. When $r_{overlap}$=1, we can easily find that all algorithms perform almost the same. Since $r_{overlap}$=1 means that each requirement is covered once on average by test cases and we have assumed that each requirement must be covered at least once. Hence, each requirement is covered exactly once and test cases do not overlap each other. Under the circumstances, a local optimal solution find by a search algorithm is actually close to the global optimal solution so that these five algorithms have similar performance. Theoretically, the three greedy algorithms are identical in this case. While in these figures when $r_{overlap}$=1, there is a slight difference between the performance of the three greedy algorithms. The reason lies in the matrix generation procedure. To guarantee the number of test requirements, x, covered by a test case follows the normal distribution, the sum of x_i is fluctuant and can not always be exactly equal to a fixed number. Thus, the value of $r_{overlap}$ fluctuates around a given number. Since every test requirement should be covered at least once, we abandon the matrix whose $r_{overlap}$ is less than 1 in our experiment. Hence, the $r_{overlap}$ assigned by the value of 1 in these figures is actually a little more than 1.

When $r_{overlap}$ is relatively large, the performance of each algorithm declines with the increase of $r_{overlap}$. As a result, when $r_{overlap}$ is extremely large ($r_{overlap}$>50), the effect of each algorithm is really limited and the APRC scores improved by them are less than 0.01. This is understandable since we have $r_{overlap}=\mu n/m$ and when m is fixed and $n \leq m$, the larger value of $r_{overlap}$ means the larger value of μ and we have explained why few APRC scores can be improved by each algorithm when μ is large at the beginning of this subsection.

When $r_{overlap}$ is moderate, we have the following observations. All five algorithms can significantly improve the APRC scores and no algorithm can always outperform the other four algorithms. But in most cases, Additional Greedy Algorithm and 2-Optimal Greedy Algorithm perform better than the other three algorithms. On the other hand, the Total Greedy Algorithm performs much weaker than the other four algorithms overall. And Hill Climbing Algorithm performs good but not as good as Additional Greedy and 2-Optimal Greedy Algorithms. As for GA, it is worse than Hill Climbing Algorithm overall and the performance of GA dramatically declines with the increase of $r_{overlap}$ and sometimes it is the worst of these five algorithms. This is very different from the result reported by Li et al.[7], in which, the difference between the performance of GA and that of the greedy approach is not significant. We think the reason lies in that our experiment is based on simulated data while the experiment in [7] is based on real data. We also find that in most cases, the APRCIs of Additional Greedy and 2-Optimal Greedy algorithms reach the maximum value when $r_{overlap}$ is around 3.

From Fig. 3-6, since the five curves of these algorithms can be easily distinguished from each other except the curves corresponding to Additional Greedy Algorithm and 2-Optimal Greedy Algorithm—they overlap all the time, we therefore have done a Student's t-test only between these two algorithms to figure out whether they have significant difference or not.

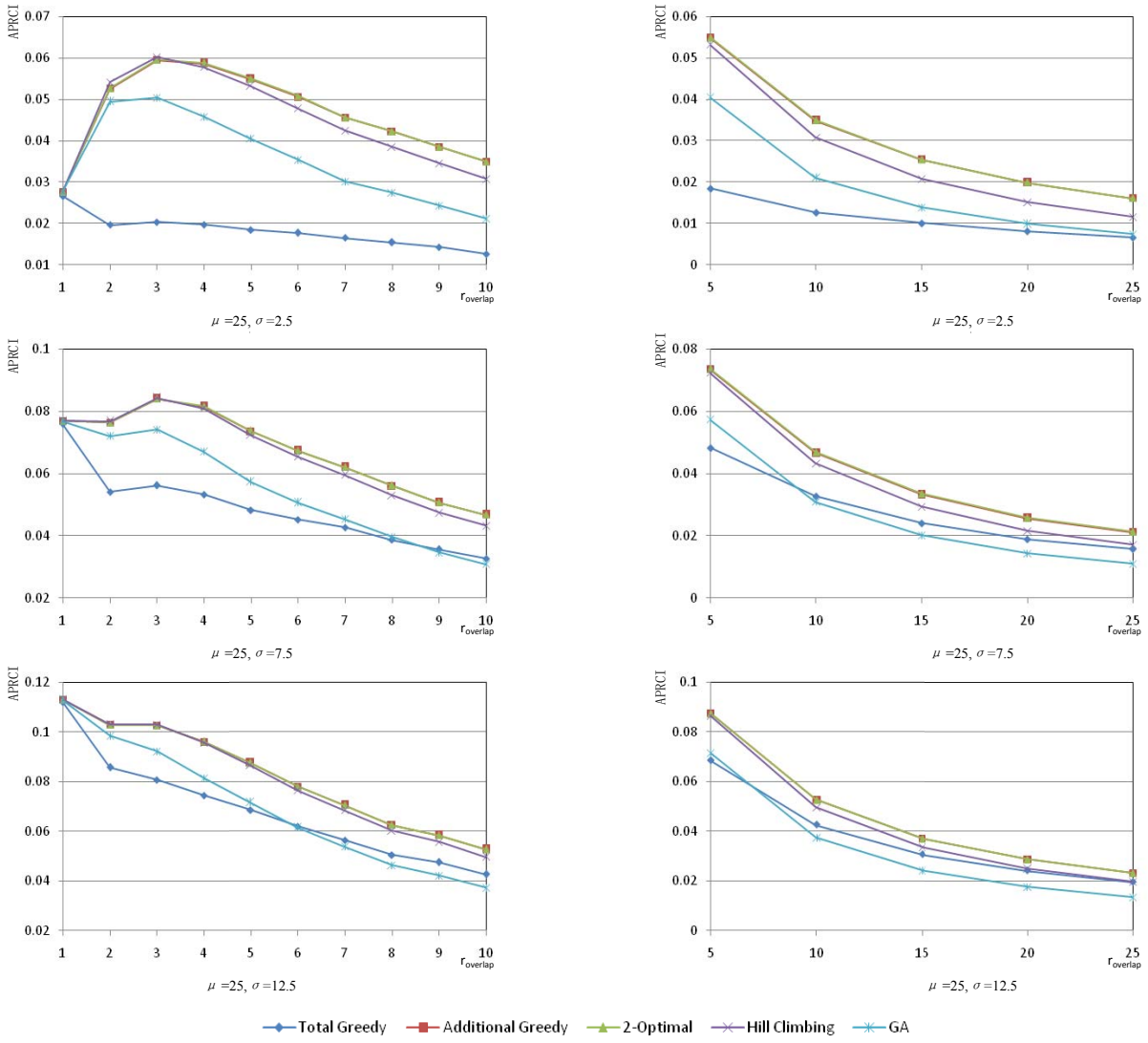

Figure 4. Plots of mean values of APRCIs corresponding to each algorithm against $r_{overlap}$ for various σ when μ=25.

As a reminder, the experiment described in Section Ⅲ is arranged in this manner, for each fixed set of m, n, μ and σ, the simulation program repeats 100 times. Thus, there are 100 APRCI values corresponding to each algorithm and we use these values as a sample set. Afterwards, we perform a Student's t-test on the two sample sets respectively corresponding to Additional Greedy Algorithm and 2-Optimal Greedy Algorithm for each set of m, n, μ and σ. All the comparison results are shown in Table Ⅳ. We group the Student's t-test results to 21 sets, and present the ranges of the $r_{overlap}$, the mean difference and the significance within each group. We choose the 0.05 level as the threshold for statistical significance. Thus, if the significance value is smaller than 0.05, the difference between Additional Greedy and 2-Optimal Greedy is statistically significant. The results in Table Ⅳ indicate that the significant value is much larger than the threshold in all cases. On the strength of the Student's t-test results, we come to the conclusion that there is no significant difference between Additional Greedy and 2-Optimal Greedy, that is, their performance is almost identical. This result is acceptable since the two algorithms share the same philosophy and they are two instances of k-Optimal Greedy Algorithm.

B. Comparison with prior studies

This subsection mainly compares our work to the studies from [7] since our experiment is very similar with theirs. The candidate algorithms in both experiments are the same. Although the detailed implementation in each experiment may

have little difference, it does not much affect the overall performance of each algorithm. The main difference is the objects that these algorithms are applied to. The objects in [7] are a set of relatively small programs while objects in our experiment are generated coverage matrices. Since they use the real programs as the objects, they could directly compare the performance of each algorithm on a specified program. However, our experiment is based on simulation and hundreds of coverage matrices are generated during the simulation process, we have to employ the ratio of overlapping to compare the performance of these algorithms. In addition, our experiment extended the effective measure (APBC, APDC, APSC) in [7] to APRC in which the requirement could be block, decision, statement or any other requirement, and we also studied other factors that may affect the performance of these algorithms.

Our results of the Greedy approaches are consistent with those in [7]. Both experiments show that Additional Greedy Algorithm and 2-Optimal Greedy Algorithm are surprisingly effective and there is no significance between these two algorithms. Besides, the performance of Greedy Algorithm is much worse than other two Greedy approaches.

Nevertheless, when it comes to metaheuristic algorithms, there are some obvious difference between our results and theirs. Their study about Hill Climbing Algorithm focused on the "fitness landscapes" and showed that the performance of Hill Climbing Algorithm is not stable, the lengths of boxplots

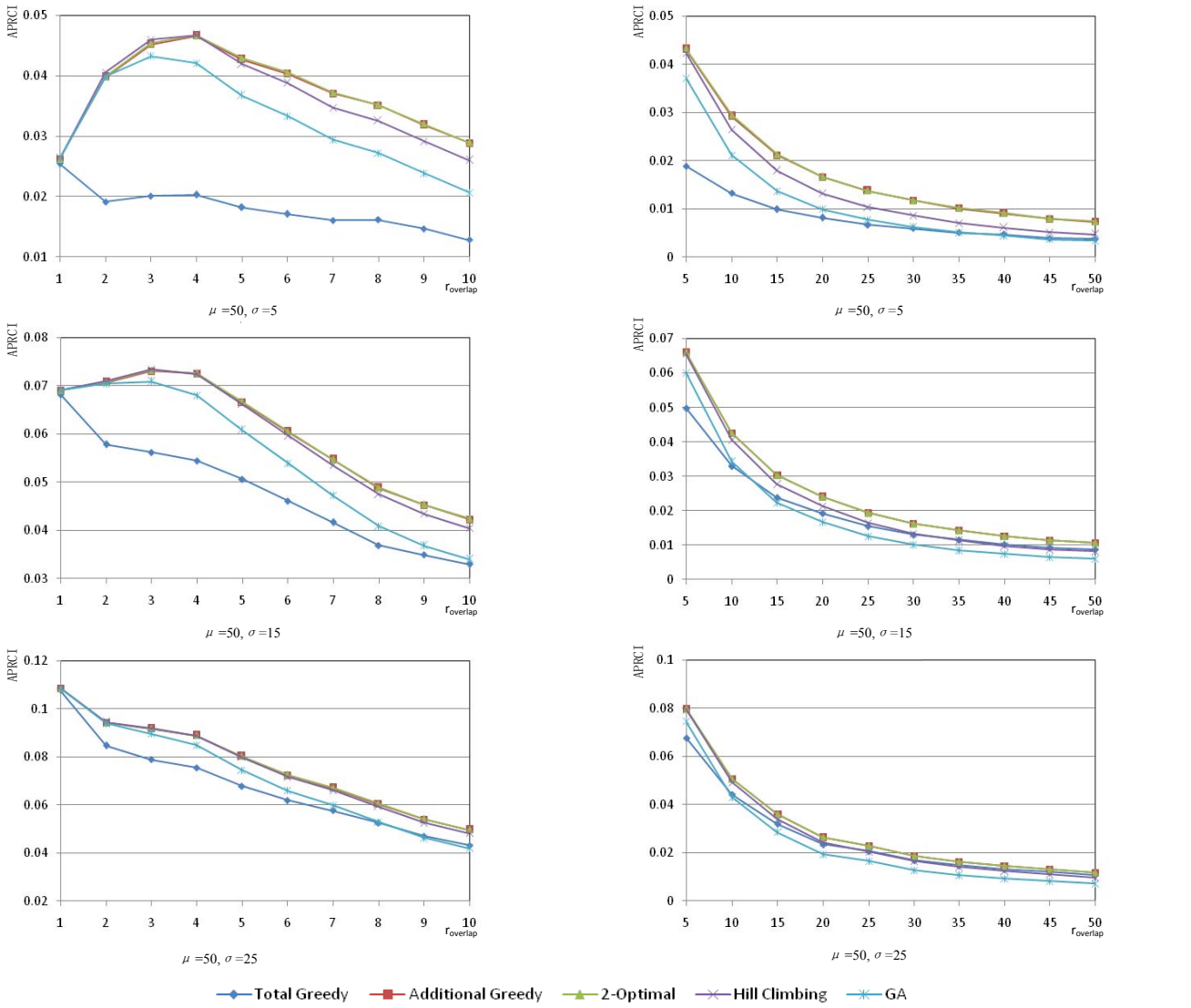

Figure 5. Plots of mean values of APRCIs corresponding to each algorithm against $r_{overlap}$ for various σ when μ=50.

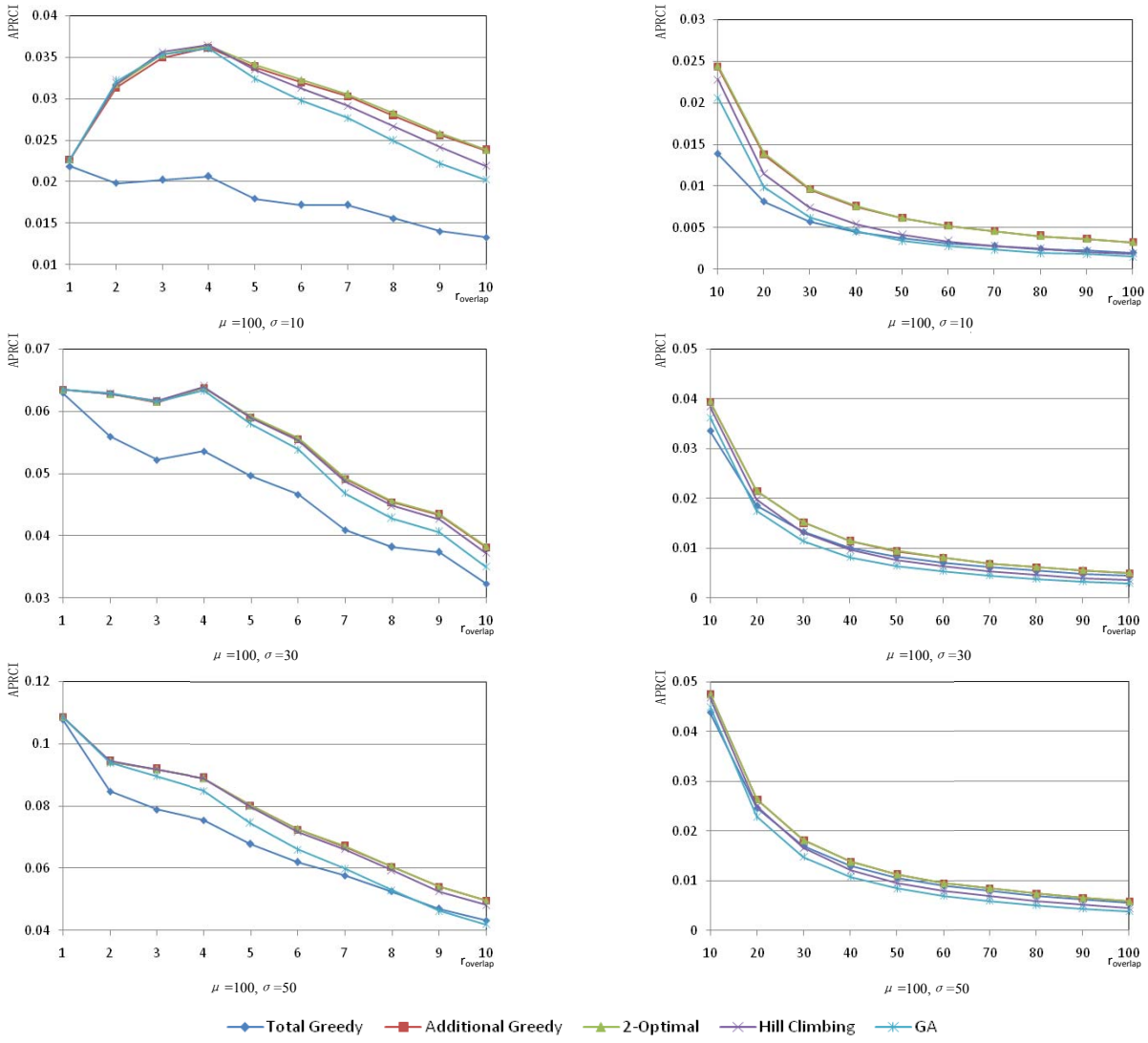

Figure 6. Plots of mean values of APRCIs corresponding to each algorithm against $r_{overlap}$ for various σ when μ=100.

for Hill Climbing Algorithm are much longer than others when the programs and test suites are large. For this reason, they did not compare the average performance of Hill Climbing Algorithm to that of other four algorithms. However, our experiment did not study the fitness landscapes. We just compared the average performance of Hill Climbing Algorithm to that of other four and the results indicated that Hill Climbing Algorithm performs well overall. As for GA, they come to the conclusion that GA is not the best of the five but in most cases, the difference between the performance of GA and that of the Greedy approaches is not significant. But our results show that in most cases, GA performs not so well and the two Greedy algorithms outperform GA overall. One possible explanation of this difference is that in Fig. 5 and Fig. 6, there are some cases in which GA has similar performance with the two Greedy approaches when $r_{overlap}$ is relatively small, and the real programs and test suites they used happened to be these cases, thus they concluded that GA performs as well as the two Greedy approaches. However, a further investigation into the cause of this difference is still needed.

C. *Threats to Validity*

Threats to internal validity are the uncontrolled factors that can also influence the results of our experiment. In this paper, the main threat to internal validity is the possible faults in the code implementation of our experiment. To reduce this threat, we have done a double check on all the codes of our programs.

TABLE IV. T-TEST FOR ADDITHIONAL GREEDY AND 2-OPTIMAL GREEDY

	$r_{overlap}$ Range	Mean Difference Range	Significance Range
μ =10and σ=1	1~10	-0.000063 ~0.000022	0.924~1.000
μ =10and σ=3	1~10	-0.00008 ~0.000043	0.937~1.000
μ =10and σ=5	1~10	-0.000072 ~0.00005	0.937~1.000
μ =25and σ=2.5	1~10	-0.000225 ~0.000030	0.745~1.000
μ =25and σ=7.5	1~10	-0.000105 ~0.000000	0.896~1.000
μ =25and σ=12.5	1~10	-0.000057 ~0.000023	0.942~0.999
μ =25and σ=2.5	5~25	-0.000109 ~-0.000012	0.790~0.927
μ =25and σ=7.5	5~25	-0.000105 ~-0.000007	0.892~0.981
μ =25and σ=12.5	5~25	-0.000057 ~0.000010	0.930~0.987
μ =50and σ=5	1~10	-0.000252 ~-0.00000	0.679~1.000
μ =50and σ=15	1~10	-0.00019 ~-0.00000	0.868~1.000
μ =50and σ=25	1~10	-0.00012 ~-0.00000	0.934~0.999
μ =50and σ=5	5~50	-0.000253 ~-0.000023	0.447~0.899
μ =50and σ=15	5~50	-0.00012 ~-0.000022	0.870~0.945
μ =50and σ=25	5~50	-0.00010 ~-0.000016	0.931~0.969
μ =100and σ=10	1~10	-0.00038 ~-0.00000	0.547~0.998
μ =100and σ=30	1~10	-0.00016 ~0.00000	0.870~1.000
μ =100and σ=50	1~10	-0.00012 ~-0.00000	0.955~1.000
μ =100and σ=10	10~100	-0.000171 ~-0.000021	0.464~0.719
μ =100and σ=30	10~100	-0.00009 ~-0.000009	0.863~0.940
μ =100and σ=50	10~100	-0.00004 ~0.000003	0.951~0.983

Threats to external validity are concerned with the limitations to the generalization of our results. It is possible that our experiment does not well simulate the real cases in practice, which would, to some extent, render the results of our study unconvincing. For example, due to the time budget of our experiment, we set the number of test cases to be covered at a fixed value of 1000. However, other cases, in which the number of test cases is much larger or smaller than 1000, are not simulated in our experiment. To handle this threat, we will conduct more experiments using a wider range of data in the future to augment the generality of our study.

Threats to construct validity occur when the measurement used in the experiment does not well match what the experiment purports to measure. There are two primary threats to construct validity in this study. One is the APRC metric adopted in our experiment since it may not be a perfect surrogate for faults detection. Another threat is that we do not consider the time complexity of these five algorithms when we compare the effectiveness of them.

V. RELATED WORK

In [5], [10], [11], Rothermel et al. first formulized the test case prioritization problem and presented six techniques which were all based on coverage of statement or branches. They also conducted an empirical experiment in which they applied these techniques to various programs and compared the test suites produced by these techniques to random, untreated, and optimal test case orders. The results of empirical studies showed that these techniques can improve the rate of fault detection.

In [7], Li et al. presented the results from an empirical study of the application of five search algorithms to six programs ranging from 374 to 11,148 lines of code. It was the "first paper to study metaheuristic and evolutionary algorithms empirically for test case prioritization for regression testing." The study focused on the comparison of the performance of these five algorithms and showed that Greedy approaches are surprisingly effective while the Genetic Algorithms also perform well.

In [3], Chen et al. conducted a simulation study in which four heuristics were applied to the generated coverage matrices rather than real programs. To compare the performance of the four heuristics, they first defined the concept of *ratio of overlapping* as a measurement for simulation study. The paper provided guidelines for choosing the most appropriate heuristics for test suite reduction.

In [15], Zhang et al. assumed that there is a time budget that does not allow all the test case to be executed and the execution time of each test case is not assumed the same but given a value. In this case, they proposed a technique called integer linear programming to handle time-aware test case prioritization problem. The results showed that this approach achieved a better rate of fault detection than traditional techniques for time-aware test case prioritization.

There were many previous studies focused on comparing the performance of various techniques under different circumstances for test case prioritization. However, they all applied techniques to a relatively small set of real programs and focused on the comparison among candidate techniques. Our work was based on simulation and we also studied other factors that may affect the performance of these techniques.

VI. CONCULSION AND FUTURE WORK

This paper discusses the performance of five typical techniques, Greedy Algorithm, Additional Greedy Algorithm, 2-Optimal Greedy Algorithm, Hill Climbing Algorithm and Genetic Algorithm, in various cases for test case prioritization. It also reports empirical results of a comparison of these five techniques. Since we assume that each test case takes the same time to be accomplished, the only criterion to compare the performance of these five techniques is the APRCI. The main findings of our experiment can be summarized as follows:

1. The performance of each algorithm rises with the increase of σ and the decrease of μ.
2. When $r_{overlap}$=1, these five algorithms have nearly the same performance. When $r_{overlap}$ is extremely large, the effect of each algorithm is really limited and few APRC scores can be improved.
3. When $r_{overlap}$ is moderate, all these algorithms can significantly improve the APRC scores. In most cases, Additional Greedy Algorithm and 2-Optimal Greedy Algorithm perform better than the other three algorithms and they have the best performance when the value of $r_{overlap}$ is around 3. There is no significant difference between the performance of these two algorithms.

Based on the findings of our experiment, we have the following recommendation for different ranges of $r_{overlap}$. When $r_{overlap}$ is close to 1, any of these five algorithms can be chosen. When $r_{overlap}$ is extremely large ($r_{overlap}$>50), there is no need to apply any of these algorithms since the algorithms themselves are time-consuming. When $r_{overlap}$ is moderate, Additional Greedy Algorithm and 2-Optimal Algorithm are the preferable choices.

In this paper, the execution time of each test case is assumed the same while in practice, the execution time of each test case differs a lot and there is usually a time limit for test case execution. So applying these techniques to time-aware test case prioritization [13][15] is a topic for our future work. And another work in the future is to do a quantitative statistical analysis on how various factors influence the performance of each search algorithm.

REFERENCES

[1] G. Antoniol, M.D. Penta, and M. Harman, Search-Based Techniques Applied to Optimization of Project Planning for a Massive Maintenance Project, Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05), pp. 240-249, 2005.

[2] J. E. Baker, Adaptive Selection Methods for Genetic Algorithms, Proceedings of the 1st International Conference on Genetic Algorithms, pp. 101-111, 1985.

[3] T. Y. Chen and M. F. Lau, A Simulation Study on Some Heuristics for Test Suite Reduction, Information and Software Technology, vol. 40, no. 13, pp. 777-787, 1998.

[4] S. Elbaum, A. Malishevsky, and G. Rothermel, Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization, Proceedings of the 23rd International Conference on Software Engineering (ICSE '01), pp.329-338, 2001.

[5] S. Elbaum, A.G. Malishevsky, and G. Rothermel, Test Case Prioritization: A Family of Empirical Studies, IEEE Transaction on Software Engineering, vol. 28, no. 2, pp. 159-182, 2002.

[6] J. M. Kim and A. Porter, A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments, Proceedings of the 24th International Conference on Software Engineering (ICSE '02), pp. 119-129, 2002.

[7] Z. Li, M. Harman, and R. M. Hierons. Search Algorithms for Regression Test Case Prioritization, IEEE Transaction on Software Engineering, vol. 33, no. 4, pp. 225-237, 2007.

[8] Lijun Mei, Zhenyu Zhang, W. K. Chan, T. H. Tse: Test Case Prioritization for Regression Testing of Service-oriented Business Applications, Proceedings of the 18th International Conference on World Wide Web (WWW '09), pp. 901-910, 2009.

[9] Bo Qu, Changhai Nie, Baowen Xu, Xiaofang Zhang: Test Case Prioritization for Black Box Testing. 31st Computer Software and Applications Conference (COMPSAC '07), pp. 465-474, 2007.

[10] G. Rothermel, R. Untch, C. Chu, and M.J. Harrold, Test Case Prioritization: An Empirical Study, Proceedings of the 15th International Conference on Software Maintenance (ICSM '99), pp. 179-188, 1999.

[11] G. Rothermel, R. Untch, C. Chu, and M.J. Harrold, Prioritizing Test Cases for Regression Testing, IEEE Transaction on Software Engineering,vol. 27, no. 10, pp. 929-948, 2001.

[12] A. Srivastava and J. Thiagarajan, Effectively Prioritizing Tests in Development Environment, Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'02), pp. 97-106, 2002.

[13] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, Time-aware Test Suite Prioritization, Proceedings of the 2006 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '06), pp. 1-11, 2006.

[14] W.E. Wong, J.R. Horgan, S. London, and H. Agrawal, A Study of Effective Regression Testing in Practice, Proceedings of the 8th International Symposium Software Reliability Engineering (ISSRE '97), pp. 230-238, 1997.

[15] Lu Zhang, Shan-Shan Hou, Chao Guo, Tao Xie, Hong Mei: Time-aware Test Case Prioritization Using Integer Linear Programming, Proceedings of the 2009ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '09), pp. 213-224, 2009.

[16] Lingming Zhang, Ji Zhou, Dan Hao, Lu Zhang, Hong Mei: Prioritizing JUnit Test Cases in Absence of Coverage Information, Proceedings of the 25st IEEE International Conference on Software Maintenance (ICSM '09), pp. 19-28, 2009.