Computing and Informatics, Vol. 36, 2017, 705–732, doi: 10.4149/cai_2017_3_705

IMPROVED ANNEALING-GENETIC ALGORITHM FOR TEST CASE PRIORITIZATION

Zan Wang, Xiaobin Zhao, Yuguo Zou

School of Computer Software
92 Weijin Rd, Tianjin University
Tianjin, 300072, China
e-mail: {wangzan, zhaoxiaobin, zouyuguo}@tju.edu.cn

Xue Yu*

College of Management and Economic
92 Weijin Rd, Tianjin University
Tianjin, 300072, China
e-mail: yuki@tju.edu.cn

Zhenhua Wang

American Electric Power
700 Morrison Rd
Gahanna, OH, 43230, USA
e-mail: vincent@tju.edu.cn

Abstract. Regression testing, which can improve the quality of software systems, is a useful but time consuming method. Many techniques have been introduced to reduce the time cost of regression testing. Among these techniques, test case prioritization is an effective technique which can reduce the time cost by processing relatively more important test cases at an earlier stage. Previous works have demonstrated that some greedy algorithms are effective for regression test case prioritization. Those algorithms, however, have lower stability and scalability. For

* Corresponding author

this reason, this paper proposes a new regression test case prioritization approach based on the improved Annealing-Genetic algorithm which incorporates Simulated Annealing algorithm and Genetic algorithm to explore a bigger potential solution space for the global optimum. Three Java programs and five C programs were employed to evaluate the performance of the new approach with five former approaches such as Greedy, Additional Greedy, GA, etc. The experimental results showed that the proposed approach has relatively better performance as well as higher stability and scalability than those former approaches.

Keywords: Regression testing, test case prioritization, meta-heuristics, genetic algorithm, simulated annealing algorithm, annealing-genetic algorithm (AG)

Mathematics Subject Classification 2010: 68N30

1 INTRODUCTION

Regression testing is a type of software testing that seeks to uncover new faults in existing software systems after changes have been made. It is a frequent testing activity and often time consuming [1, 2]. Research shows that the time cost of regression testing would account for more than one third of the total time cost of software maintenance [3, 4]. At present, the way to reduce the cost of this work has attracted much attention not only from software engineers but also academic researchers. More and more techniques to pursue lower cost in regression testing have been proposed and these methods can be summarized as three categories which are test case selection techniques [5], test suite minimization techniques [6] and test case prioritization techniques [7]. As for lossless aspect of the testing capability method, the test case prioritization does not discard any test cases and assumes that different test cases have different contributions for the testing goals. Those test cases which are more important for the testing goals will be higher scored and executed earlier. Previous researches have showed that test case prioritization techniques can improve the efficiency of regression testing by the early stopping when goals are achieved. As a result, it reduces testing time and costs [8, 9, 10, 11].

The formal definition of the test case prioritization problem given by Rothermel et al. is widely accepted in literature [11]. They first employ a function $f(T)$ to yield an award value for each ordering T of test cases, and then describe the problem as to find an ordered test suite with the highest award value. Rothermel et al. also discussed five possible improvement goals which testers intended to achieve by the test cases prioritization. They are the rate of fault detection of a test suite, the coverage of coverable code, their confidence in the reliability of the system, the rate at which high risk faults are detected and the likelihood of revealing faults related to specific code changes respectively [11]. To evaluate prioritization techniques in meeting these different goals, however, we need different evaluation criteria. If we focus on the first goal listed above, increasing the rate of fault detection of a test suite, APFD (Average of the Percentage of Faults Detected) should be used which is the widely used and effective prioritization technique evaluation criterion. But if our goal is to increase the coverage of coverable code in the system under test at a faster rate, coverage based evaluation criteria such as APSC (Average Percentage Statement Coverage), APBC (Average Percentage Block Coverage) and APDC (Average Percentage Decision Coverage) should be taken. In this work, we aim at the second goal listed above – the increasing the coverage of coverable code in the system under test at a faster rate and the APSC will be selected as the evaluation criterion to compare the performance of prioritization techniques. The APSC for ordering T' is given as follows [10]:

$$APSC = 1 - \frac{TB_1 + TB_2 + \ldots + TB_m}{nm} + \frac{1}{2n} \tag{1}$$

where we assume that a test suite T containing n test cases that covers a set S including m statements and TB_i is the first test case in the order T' of T that covers statement i.

Heuristic algorithms can be employed to solve the test case prioritization problem. In the previous work, many heuristics based techniques for regression test case prioritization have been proposed. In [11, 12, 13, 14, 15, 16, 17, 18], Rothermel et al. investigated several prioritization techniques, such as total statement/branch coverage prioritization and additional statement/branch coverage prioritization, which try to maximize the rate of fault detection. Li et al. applied various meta-heuristic algorithms for test case prioritization [10]. They compared random algorithm, Hill Climbing algorithm, Genetic Algorithm, Greedy algorithm, Additional Greedy algorithm and two-optimal Greedy algorithm and found that Additional Greedy algorithm can achieve better results than other four algorithms.

T	Statement																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
1	X	X	X	X	X	X	X	X									
2							X	X	X	X	X	X					
3													X	X	X		
4																X	X
5	X	X	X	X	X	X											X

Table 1. A case in which Additional Greedy algorithm will not produce an optimal solution

Despite of the efficiency of Additional Greedy algorithm, some issues remained about reliability and scalability of additional greedy algorithm, especially when the coverage data matrix is sparse. To illustrate this problem, a simple example based on statement coverage is employed and shown in Table 1. If the aim is to increase the coverage of coverable code at a faster rate and the evaluation criterion is APSC, Additional Greedy algorithm may select 1-2-3-4-5 as the "optimal" solution. However, the optimal test case ordering for this example is 5-2-3-4-1 because the APSC value of test suite 5-2-3-4-1 is bigger than that of test suite 1-2-3-4-5 (the APSC value of test suite 5-2-3-4-1 is 72.3529 while the APSC value of test suite 1-2-3-4-5 is 71.1765). Additional Greedy algorithm does not work well in this example mainly because greedy based algorithms are a type of avariciously searching algorithm and expect to find the global optimum by making the locally optimal choice at each stage. Unfortunately, they can only explore a smaller space and have less probability to achieve the optimal solution than global searching algorithms such as Genetic Algorithm (GA). We call this problem as "Local Optima Syndrome (LOS)". However, previous experiments showed that Additional Greedy algorithm had better results than Genetic Algorithm in many common cases. It is necessary to find a better algorithm which can not only explore the global searching space but also have at least equal performance compared with Additional Greedy algorithm.

In this paper, we propose an improved AG algorithm for test case prioritization. The new proposed algorithm is a robust optimizing algorithm which can overcome the shortcomings of Additional Greedy algorithm as well as to achieve good performance as Additional Greedy algorithm by three following ways. Firstly, Metropolis operator of SA will be incorporated into the iterative process to maintain diversity of the searching space of GA to alleviate the premature convergence problem. As a result, the hybrid algorithm will have more chances to achieve better results. Moreover, the optimization reserved strategy will be hired to accelerate the convergence rate by retaining the optimum of each generation. Thirdly, a local search operator will be employed with GA to improve the local search capability of the algorithm. These three improvement ways of GA will not only increase the performance of the algorithm but also accelerate its convergence speed.

The rest of this paper is organized as follows. Section 2 describes related work. Section 3 describes the improved AG algorithm for regression test case prioritization. Then we evaluate the proposed algorithm and compare it with other five algorithms in Section 4. Section 5 concludes and Section 6 gives the future works.

2 RELATED WORK

From the perspective of software testers who wish to meet the goal of regression test case prioritization, previous research can be roughly categorized into two classes, namely those for rate of fault detection and the other for code coverage, respectively.

2.1 Prioritization for Rate of Fault Detection

In the previous work on optimizing the fault detection, researchers focused on the first goal listed in Section 1: increasing the rate of fault detection of a test suite. Rothermel et al. explored some empirical studies of several prioritization techniques [11, 12, 13, 14, 15, 16, 17, 18] to reach this goal. They applied Greedy algorithm and Additional Greedy algorithm with different fault detection rate surrogates. The considered surrogates were: branch-total, branch-additional, statement-total, statement-additional, Fault Exposing Potential (FEP)-total, and FEP-additional. They also conducted several experiments to compare these techniques with no prioritization, random prioritization and optimal prioritization. The results showed that all the proposed techniques get higher APFD values than random or no prioritization, which means these techniques can improve the rate of fault detection.

Although Additional Greedy algorithm has relatively better performance, Zhang et al. found that there is a weakness in the additional strategy. Because of the situation in which a test case covers a statement but does not reveal a fault in the statement, the additional strategy may greatly postpone the detection of those faults covered but not revealed. In contrast, the total strategy does not have this weakness. Therefore, they proposed a test case prioritization approach that unifies the total strategy and the additional strategy [19]. They also performed an empirical study to compare their approach with greedy and additional greedy based approaches. The results demonstrate that the proposed approach can significantly outperform both the total and additional strategies.

2.2 Prioritization for Code Coverage

Research on prioritization for code coverage aimed to increase the coverage of executable code at a faster rate. Li et al. investigated various meta-heuristics methods to test case prioritization for faster coverage [10]. They conducted an experiment to compare random prioritization, Hill Climbing algorithm, Genetic Algorithm, Greedy algorithm, Additional Greedy algorithm and two-optimal Greedy algorithm. Rather than APFD, three coverage based metrics including APBC, APDC and APSC with Siemens suite programs and the program space were hired to evaluate the performance of each technique. The results indicated that Additional Greedy algorithm is the most efficient in general. The studies also showed that the applications of meta-heuristics, especially Genetic Algorithm, are effective for regression test case prioritization.

Maia et al. applied another well-known meta-heuristic, GRASP (Greedy Randomized Adaptive Search Procedures), for test case prioritization and conducted an empirical evaluation to compare GRASP based approach with other four search based algorithms [20]. Their experimental results showed that the proposed approach performed significantly better than Genetic Algorithm, Simulated Annealing algorithm, and similarly to Additional Greedy algorithm in terms of coverage performance.

Previous research of test case prioritization for code coverage mainly focus on greedy algorithms and meta-heuristics. However, seldom work has examined the adaption and flexibility of these algorithms. In addition, greedy algorithms are local search based algorithms and have lower scalability and stability as shown in Section 1. Thus, we need a better test case prioritization technique which can explore the global searching space and has good performance as Additional Greedy algorithm. In this paper, we propose a hybrid algorithm for test case prioritization to achieve better result with high stability and scalability. In addition, this paper will verify some primary findings in the previous research to some extent and analyze the limitation of several widely used heuristic algorithms.

3 IMPROVED AG FOR TEST CASE PRIORITIZATION

Genetic algorithm can achieve the global optimum by exploring a global candidate space rather than obtaining only local optimum for each stage. It has shown effectiveness and robustness for test case prioritization problem [10]. However, there are some insufficiencies. One of the major problems for GA is weaker performance compared with Additional Greedy algorithm. This issue is mainly because the premature convergence to some local optima often occurs with GA. The term of premature convergence means that a population for an optimization problem converges too early and leads to suboptimal results. This problem is mainly due to the loss of diversity of the GA's population, being the cause of a decrease on the quality of solutions. To alleviate this problem and improve the performance of GA, Lin et al. proposed a hybrid algorithm Annealing-Genetic (AG) by combining the local stochastic search from simulated annealing algorithm and the global genetic operations especially the crossover operation from Genetic Algorithm effectively [21]. In this paper, the hybrid algorithm AG will be hired to maintain a certain degree of genetic diversity for better test case prioritization. Moreover, two further strategies will be hired to accelerate the computing process for faster convergence. Firstly, local searching in each iterative evolution will be employed to search the local optimum much faster. And secondly, optimization reserved strategy will be adopted to hold the optimal solution in each step.

To illustrate, consider Figure 1, and Table 2 that represent the computational framework and pseudo codes of the improved AG algorithm. As described in Table 2, the proposed hybrid algorithm has six important parts: individual generation and population initiation, fitness function, Metropolis criterion determination, genetic operators, optimization reserved strategy and stopping criteria. The detailed descriptions of these parts are listed as follows.

3.1 Individual Generation and Population Initialization

Like GA, the new hybrid algorithm evolves a population including candidate solutions (called individuals) toward better individuals by an iterative process, too. Each individual in the population is a candidate solution for the optimization problem. For TCP, one candidate solution can be represented as a string which is comprised of the executing sequence of test cases in a test suite. For example, "5-3-4-2-1" denotes an executing sequence of five test cases in a test suite, where the testing process follows a sequence of fifth, third, fourth, second and first test cases. After generating a fix number of individuals randomly, the algorithm will initialize the population.

```
Pseudo code: Improved Annealing-Genetic
Parameters:
      Initial Temperature – $T_0$
      Cooling Rate – $q$
      Frozen Temperature – $T_{end}$
      Population Size – NIND
      Generation Gap – GGAP
      Crossover Rate – $P_c$
      Mutation Rate – $P_m$
 1    initialize population $P_0$
 2    evaluate individuals in $P_0$
 3    $current\_point$ ← the individual with the highest evaluation value in $P_0$
 4    $solution\_point \leftarrow current\_point$
 5    $current\_population \leftarrow P_0$
 6    $T \leftarrow T_0$
 7    while $(T > T_{end})$ do
 8    begin
 9       $no\_of\_point \leftarrow 0$
10       $candidates \leftarrow \{\}$
11       while $(no\_of\_point <$ NIND$)$ do
12       begin
13          generate $next\_point$ from $current\_point$
14          // Metropolis criterion determination
15          $df \leftarrow f(next\_point) - f(current\_point)$
16          if $\min[1, \exp(df/T)] >$ random$[0, 1)$ then
17             $candidates \leftarrow \{candidates, next\_point\}$
18             $current\_point \leftarrow next\_point$
19             $no\_of\_point \leftarrow no\_of\_point + 1$
20          else
21             pick another point from $current\_population$ as $current\_point$
22       end
23       select from $candidates$
24       crossover
25       mutate
26       reverse
27       re-insert to get new population $P_{new}$
28       evaluate individuals in $P_{new}$
29       $current\_point$ ← the individual with the highest evaluation value in $P_{new}$
30       if $f(current\_point) > f(solution\_point)$ then
31          $solution\_point \leftarrow current\_point$
32       $current\_population \leftarrow P_{new}$
33       $T = q * T$
34    end
```

Table 2. Pseudo code of Improved Annealing-Genetic algorithm

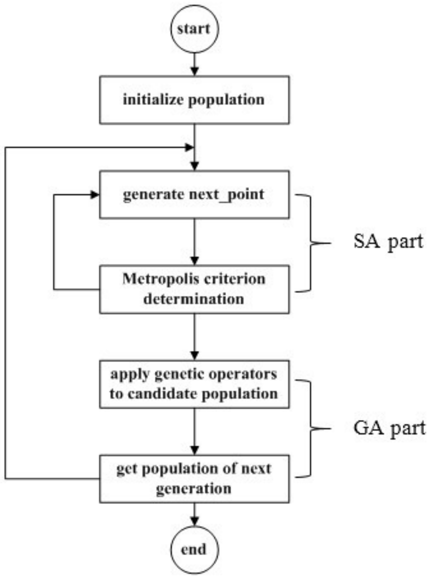

start
initialize population
generate next_point
Metropolis criterion determination
SA part
apply genetic operators to candidate population
get population of next generation
GA part
end

Figure 1. Flow chart of improved AG

3.2 Fitness Function and Evaluation Criteria

After each round of the iterative process, some worst solutions will be replaced by the new breeds. Therefore, each candidate solution needs to be awarded a measure indicator representing the distance from achieving the goal. The measure indicator can be generated by a fitness function and be employed to evaluate how good a candidate solution is. As described in Section 1, there are different evaluation criteria to meet different goals for TCP and different criteria leads different fitness function. Because increasing the coverage of coverable code in the system under test at a faster rate is the goal of this paper, APSC will be hired as the fitness function to evaluate each solution. The fitness function can be defined by Formula (1).

3.3 Metropolis Criterion Determination

As described above, GA has a shortage that it usually converges to some local optimum and has difficulty to reach the global one. This problem is usually because the searching space is not large enough to achieve the optimum and does not have enough variance. To address this problem, the Metropolis criterion of SA will be introduced to accept some worse individuals with a probability to maintain the diversity of the population. The acceptance probability is defined as follows:

$$P = \begin{cases} 1, & df > 0, \\ \exp\left(\frac{df}{T}\right), & df \leq 0 \end{cases} \tag{2}$$

where T is the current temperature, and df is the fitness difference of two individuals.

3.4 Genetic Operators

Like GA, improved AG has similar genetic operators which make the individuals in the population go through a process of evolution. As an improved GA, it has four types of operators: selection, crossover, mutation and local search, respectively. Selection is the stage of a genetic algorithm in which individuals are chosen with a selecting probability from a population for further breeding operators such as crossover and mutation. Crossover is a genetic operator used to produce child solutions from parent individuals which were chosen by the selection operator. Mutation alters one or more gene values in a chromosome to maintain the genetic diversity of the population. In this paper, we redefine the traditional three operators for TCP and incorporate a local search operator before the computational process proceed to the next generation for better performance. Detailed operators are described as follows:

Selection. The roulette wheel selection strategy is employed in this paper to select potentially useful solutions for recombination. Let NIND be the population size and $\mathit{fitness}(i)$ be the fitness value of individual i in the population, its probability of being selected is $P_i = \mathit{fitness}(i)/\sum_{j=1}^{NIND} \mathit{fitness}(j)$.

Crossover. There are two steps of crossover in this paper. Firstly, two points to be selected on the parent strings will exchange each other. Everything between the two points will be swapped between the parent strings and two child individuals will be rendered. Unlike binary strings, a candidate solution for TCP is a sequence string and there may be conflict numbers in the child individuals. After that, an extra process to amend the child individuals will be lunched to deal with the conflict for legal child individuals. For example, there are ten test cases in a test suite. Two executing sequences of the test suite, 1-5-7-2-3-4-9-6-8-10 and 10-2-8-3-6-5-9-1-7-4 are selected for crossover. After exchange the substrings between the third and sixth locus of the two parents in this example, the algorithm will give two child chromosomes, 1-*-8-3-6-5-9-*-*-10 and 10-*-7-2-3-4-9-1-*-*. There are some conflict numbers in the two child individuals and partial mapping method is hired to amend the child individuals. Then the amended child individuals are 1-4-8-3-6-5-9-2-7-10 and 10-6-7-2-3-4-9-1-8-5, respectively.

Mutation. The mutation is performed by exchanging the locations of two selected points randomly.

Local search. The interest on local search operators comes from the fact that they may effectively and quickly explore the basin of attraction of optimal solutions, achieving optimum more accurate and quickly. GA based algorithms, on the contrast, explore the global space by evolving the population for optimum and have low speed for evolution. In this paper, we will incorporate a neighborhood based local search operator – *reverse* – into AG for better individual after crossover and mutation. This operator could improve the local searching ability of Genetic Algorithm and it is evolutionary because it is a single direction operation. That is, only accept the individual whose fitness has increased after reverse operation. Reverse means to select two positions in the test suite and then to reverse the suite between these two positions. The introduction of local searching into AG will not only improve the performance of the algorithm but also compensates the time which AG consumes for exploring a larger space.

3.5 Optimization Reserved Strategy

Re-insert operator constructs the new population using the optimal individuals in the parent population and offspring produced by genetic operators. This operator ensures the realization of the optimization reserved strategy and keeps the population size constant.

3.6 Stopping Criteria

As a hybrid algorithm of SA and GA, the new algorithm will stop when it reaches some stopping conditions. In this paper, the algorithm first assigns an initial temperature and then lowers the temperature in a decreasing rate for each generation. After exceeding the temperature threshold, the algorithm will stop.

4 EMPIRICAL EVALUATION

In order to evaluate the performance of the proposed test case prioritization approach based on improved AG, several empirical experiments are conducted. This section describes those studies, including experiments design, subjects' description, results discussion and some shortcomings.

4.1 Research Questions

We are interested in the following research questions.

[RQ1:] Can improved AG based test case prioritization approach improve the robustness?

[RQ2:] How does the new proposed algorithm compare to some former algorithms in terms of performance?

[RQ3:] How does the new proposed algorithm compare to some former algorithms in terms of the convergence speed?

These three questions concern the robustness, performance and convergence speed of the new proposed algorithm. For RQ1, the improved AG algorithm proposed in this paper is a global optimizing algorithm. It will explore a larger candidate space for the optimum. Individuals who have higher fitness values will survive in the evolve process. Let us review the example were the Additional Greedy algorithm does not work well in Section 1. Because the APSC value of test suite 5-2-3-4-1 is bigger than that of test suite 1-2-3-4-5, test suite 5-2-3-4-1 will be regarded as a better solution and will survive. As a result, improved AG is a robust optimizing algorithm which can explore a larger candidate space and overcome the shortcomings of Additional Greedy algorithm. To answer RQ2 and RQ3, more experiments will be investigated.

4.2 Experiments Setup

4.2.1 Evaluation Target

This experiment is designed to compare the proposed improved AG algorithm based approach with other heuristic algorithm based approaches, including Greedy algorithm, Additional Greedy algorithm, Genetic Algorithm (GA), Simulated Annealing algorithm (SA) and Hill Climbing algorithm (HC), in terms of coverage rate and time performance.

4.2.2 Subjects and Test Suites

In our study, three open sources Java programs (see Table 3) and five C programs (see Table 4) are used to evaluate the improved AG algorithm with some related algorithms.

Java subjects and JUnit test suites. *Jtopas*, *xml-security* and *jmeter* are the three java programs to be used in our study. *Jtopas* is a Java library for the common problem of parsing arbitrary text data. *Xml-security*, supplied by the XML subproject of the well-known open source Apache project, is a component library implementing XML signature and encryption standards. *Jmeter* is a Java desktop application for functional behavior load test and performance measurement. All these programs are assembled with their corresponding JUnit test suites. Each test suite contains several unit test case classes which have one or more test methods in each of them. Because of this feature of JUnit, two types of test case granularity, the test-class level and the test-method level are considered. Each JUnit class is a test case at the test-class level while a test method will be regarded as the test case at the test-method level. Details of the three programs and their corresponding test suites are listed in Table 3. In our study, for stochastic searching algorithms such as GA, SA and HC, each of them will be executed 10 times on every test suite against the three programs.

C subjects and test suites. In order to evaluate the prioritization performance of the proposed approach on C programs, we use four small programs from the

Subject	KLoC	Number of Classes	Number of Methods	Class-Level Tests	Method-Level Tests
*jtopas_v*0	1.83	21	263	7	120
*jtopas_v*1	1.89	19	284	10	126
*jtopas_v*2	2.03	21	302	11	128
*jtopas_v*3	5.36	50	748	18	209
*xml-security_v*0	17.4	167	1 537	11	78
*xml-security_v*1	18.3	179	1 627	15	92
*xml-security_v*2	19.0	180	1 629	15	94
*xml-security_v*3	16.9	145	1 398	13	84
*jmeter_v*0	29.5	319	2 467	25	70
*jmeter_v*1	33.7	334	2 919	26	78
*jmeter_v*2	33.1	319	2 838	29	80
*jmeter_v*3	37.3	373	3 445	33	78
*jmeter_v*4	38.4	380	3 536	33	78
*jmeter_v*5	41.1	389	3 613	37	97

Table 3. Experimental Java subjects

Siemens suite [22] and a relatively large program (*space*). *Tcas* is an aircraft system for collision avoidance. *Tot_info* is a program designed to compute statistics of input data. The *print_tokens* program is a lexical analyzer. *Replace* is used for pattern matching and substitution. *Space*, developed for the European Space Agency, is one of the most common subjects of analysis for evaluating the prioritization techniques. Each of these programs has a test pool designed to test them. Table 4 depicts the five C programs in detail. To generate test suites for these five C programs, we use a manner which is similar to the manner used in [10]. Test suites of *space* are generated to achieve branch coverage in a somewhat minimal fashion – continually picking test cases from the test pool and adding them to the test suite as long as they add branch coverage, until all branches are covered. Test suites of the four small programs are also generated for branch coverage, but not in a very minimal manner – for each branch, we pick up a test case that hit the branch from the test pool randomly. The reason why we do not generate test suites for these small programs in the minimal manner is because that the size of test suites generated in the minimal manner is too small to use meta-heuristic prioritization techniques. We generate 100 test suites for each of the four small programs and 70 test suites for *space* program. Each algorithm is executed 10 times for the four small programs and 1 time for the *space* program, for each test suite.

4.2.3 Coverage Information

As Java is an object-oriented programming language while C is process oriented, the coverage information of Java and C programs is collected at different granularities with different tools.

Subject	LoC	Number of Executable Statements	Test Pool Size	Average Test Suite Size
tcas	174	73	1 608	83
tot_info	568	138	2650	199
print_tokens	726	203	4 130	318
replace	564	273	5 542	401
space	9 564	3 814	13 585	155

Table 4. Experimental C subjects

Coverage information of Java programs. "Cobertura" [23] is used to collect the coverage information of the three Java programs at method level. It reports the coverage information whether each method in the program is executed by each test case or not. Then we write a conversion program in Python to read in the method coverage information from Cobertura and represent the method coverage matrix as output. Each element in the method coverage matrix represents whether a method in which one or more lines of code are executed by a test case or not, where "1" represents executed but "0" means not. Only executable methods will be considered.

In our experiment, collecting the method coverage information using Cobertura is conducted on a Dell desktop with a 3.10 GHz Intel Core i5-2400 CPU and 4 GB physical memory, running Windows 8 and JDK 1.7.0.

Coverage information of C programs. Different from collecting coverage information on Java programs, coverage information of C programs is collected by a tool named "gcov". gcov reports the frequency of each statement in the program executed by each test case. Then we write another conversion tool by Python script. This tool reads in the statement coverage information from gcov and represents the statement coverage matrix as output. Each element in the statement coverage matrix represents whether a statement is executed one or more times by a particular test case or not. Similarly, "1" denotes executed and "0" means not. Those statements that are not executable are not considered.

In our experiment, collecting the statement coverage information by gcov is conducted on a Dell desktop with a 3.10 GHz Intel Core i5-2400 CPU and 4 GB physical memory, running SunOS 5.10 (a.k.a. Solaris 10) and GCC 3.4.3.

4.2.4 Evaluation Measure

An effective evaluation measure is needed to evaluate these test case prioritization approaches. Although some researches on the effectiveness of the coverage report irrelevance between coverage and capability of fault detection, coverage is still an important concern in various testing standards. As our focus in this paper is the second goal listed in Section 1: increasing the coverage of coverable code in the system under test at a faster rate and our coverage information is collected at different granularities, we choose two different but very similar coverage evaluation metrics –

APSC (Average Percentage Statement Coverage) for C programs and APMC (Average Percentage Method Coverage) for Java programs. Higher APSC/APMC values mean faster statement/method coverage rates. Like APSC, APMC can be defined as follows:

Consider a test suite T containing n test cases that covers a set S including m methods. Let TB_i be the first test case in the order T' of T that covers method i. The APMC for ordering T' is given as follows:

$$APMC = 1 - \frac{TB_1 + TB_2 + \ldots + TB_m}{nm} + \frac{1}{2n}. \tag{3}$$

4.2.5 Parameter Settings

All the algorithms including improved AG algorithm, Greedy algorithm, Additional Greedy algorithm, GA, Hill Climbing algorithm and SA algorithm are implemented with MATLAB.

For Simulated Annealing algorithm and improved Annealing-Genetic algorithm, the initial temperature is to be determined in a manner which is similar to the manner used in [21]. For sake of the efficiency of the algorithm, we define the initial acceptance probability of detrimental move to be 0.6. From the Metropolis criterion,

$$P = \exp\left(\frac{df}{T}\right), \quad df \leq 0 \tag{4}$$

we obtain

$$T = df/\ln P = df/\ln 0.6 \cong -2df. \tag{5}$$

The largest detrimental move in the initial population is

$$df = (f_{min} - f_{max})\,/population\_size \tag{6}$$

where f_{max} is the maximum value of the APSC/APMC values of individuals in the initial population and f_{min} is the minimum. Thus, the initial temperature T_0 can be defined as follows:

$$T_0 = \frac{f_{max} - f_{min}}{population\_size/2}. \tag{7}$$

For Genetic Algorithm, the max generation – Maxgen is equal to the number of iterations of SA and improved AG algorithms.

Other parameters of these algorithms are shown in Table 5.

4.3 Results Analysis and Discussion

4.3.1 The improved Annealing-Genetic Algorithm Has Good Performance as Additional Greedy Algorithm for Test Case Prioritization

Figure 2 shows the boxplots of the fitness metrics APMC for all three Java programs at test-class level. Each subfigure indicates the results for one program. Like

Algorithm	SA	GA	Improved AG
Cooling Rate	0.9	–	0.9
Frozen Temperature	0.01	–	0.01
Length of Metropolis Chain	50	–	–
Population Size	–	50	50
Generation Gap	–	0.9	0.9
Crossover Probability	–	0.8	0.8
Mutation Probability	–	0.1	0.1

Table 5. Parameter Settings

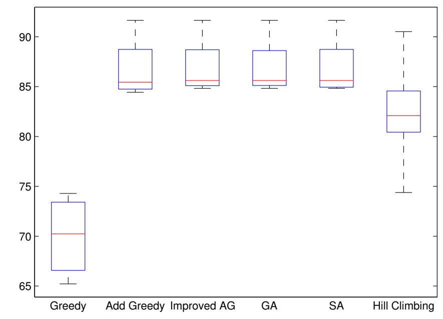
Figure 2, Figure 3 depicts the boxplots of the similar fitness metrics at test-method level. Figure 4 is a summary of boxplots for all three programs. The subfigures in Figures 2, 3 and 4 indicate that the Greedy algorithm is the worst and the Additional Greedy algorithm is slightly better than the others. The improved AG algorithm ranks on the second higher fitness metric value regardless of test case granularity and programs under test.

For C programs, Figure 5 shows the boxplots of the fitness metrics APSC for *tcas*, *tot_info*, *print_tokens* and *replace*, respectively. Figure 6 is a summary of boxplots for all small C programs. Figure 7 presents the corresponding boxplots for *space*. Similar to Java programs, these figures show that the improved AG algorithm has achieved the second best performance for all five C programs. Moreover, with the increase of C program size, the differences between improved AG and other four approaches including Greedy algorithm, Genetic Algorithm, Simulated Annealing algorithm and Hill Climbing algorithm become more significant.

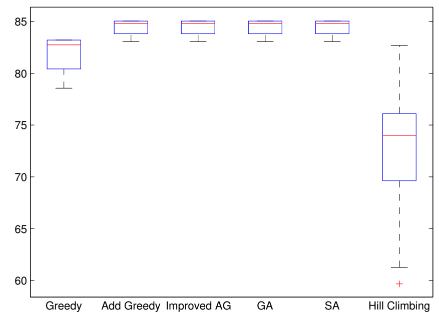
For both Java and C programs, improved AG obtains the best performance when compared to the meta-heuristics based approaches including Genetic Algorithm, Simulated Annealing algorithm and Hill Climbing algorithm.

Figures 2, 3 indicate that although Additional Greedy algorithm achieves the best performance overall, the improved AG algorithm gives a little higher APMC values than Additional Greedy algorithm for *jtopas* at both the test-class and the test-method level. This observation demonstrates that greedy base algorithms may not work well in some cases, but the proposed AG algorithm always works well and has higher stability than five other algorithms.

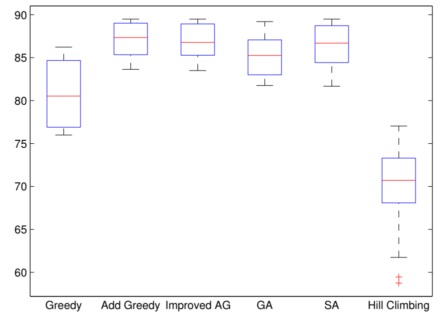
To investigate the statistical significance of the differences between six algorithms, a non-parametric statistical test named the Friedman test [24] rather than ANOVA will be conducted at 0.05 significance level. The testing results exhibited in Table 6 and Table 8 are for Java programs at the test-class/test-method level. The testing results exhibited in Table 10 and Table 12 are for all small C programs and *space* respectively. The "null hypothesis" is that the means of the APMC/APSC values for the six algorithms are equal. The Significance value (p-value) will be calculated to decide acceptance of the "null hypothesis". If the p-value is less than 0.05, the "null hypothesis" should be rejected. Otherwise, the "null hypothesis" should be accepted. The smaller the p-value (< 0.05) is, the stronger the evidence



a) *jtopas*



b) *xml-security*



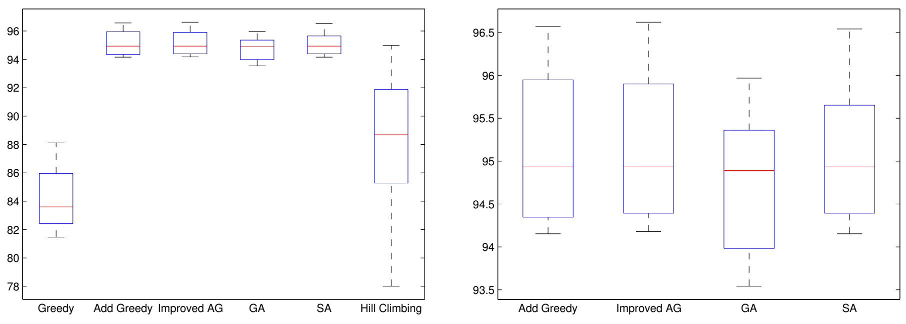
c) *jmeter*

Figure 2. Boxplots of APMC for Java programs at the test-class level, by program and by approach

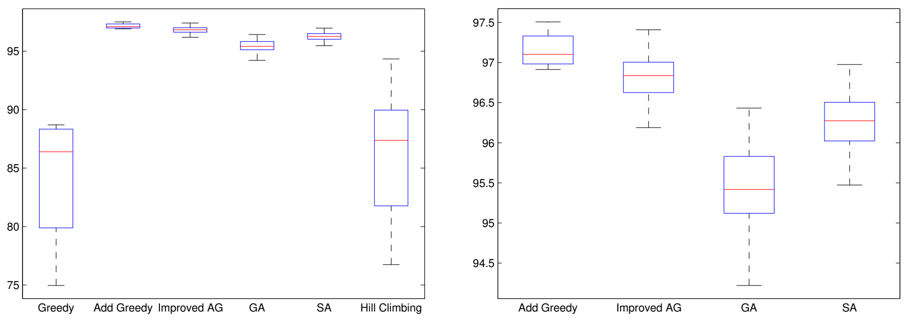
will be against the "null hypothesis". The reason to use Friedman test rather than Fisher's ANOVA (Analysis of Variance) is that the distributions of each data set obtained by different approaches are not normal and the variances are not all equal for all programs.

The results of the Friedman test show that the means of the APMC/APSC values for the six algorithms are not all equal. This means that the performances of these algorithms are significantly different. In order to locate the differences, we performed the LSD (Least Significant Difference) test, which is a method of multiple comparisons. The significance level of LSD test is also set at 0.05. The results are presented in Tables 7, 9 for Java programs and 11, 13 for C programs. If the interval between lower confidence limit and the upper confidence limit does not include zero, we can believe the difference between the two algorithms is significant.

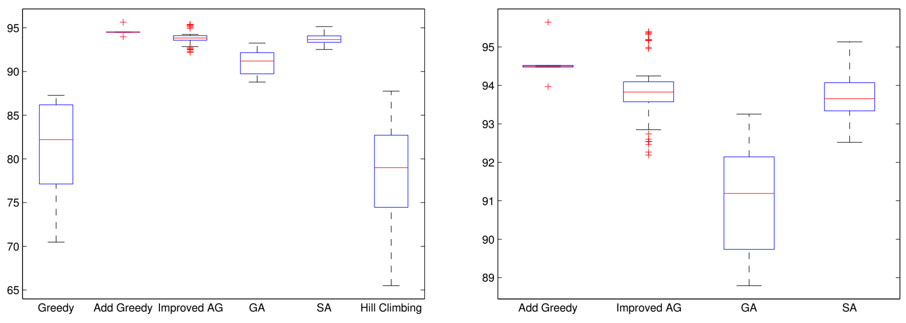
At the test-class level of Java programs, there is no significant difference between the improved AG algorithm and Additional Greedy algorithm. However, at



a) *jtopas*

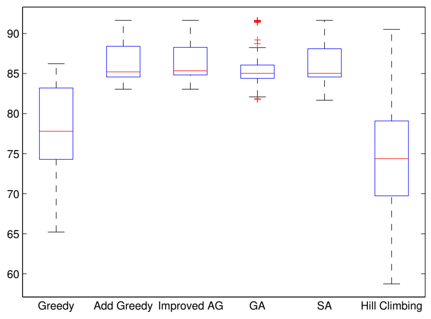


b) *xml-security*

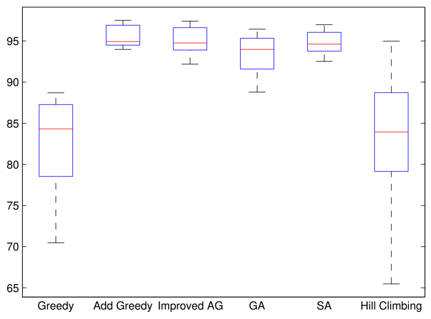


c) *jmeter*

Figure 3. Boxplots of APMC for Java programs at the test-method level, by program and by approach



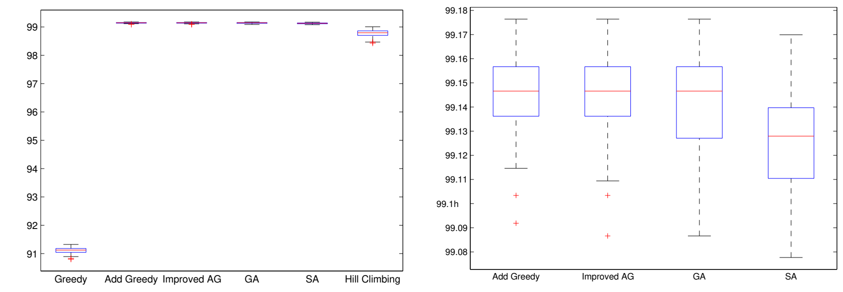
a) *test-class level*



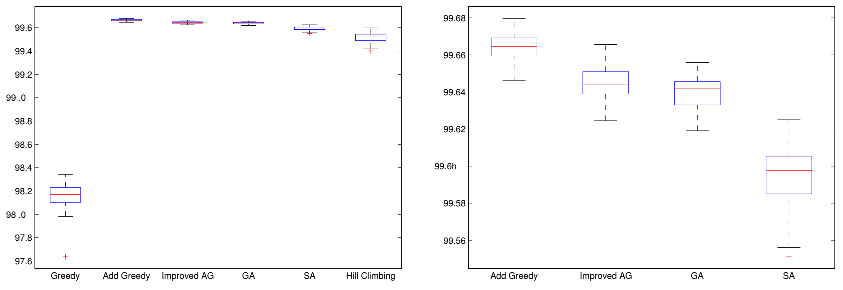
b) *test-method level*

Figure 4. Boxplots of APMC for all three Java programs

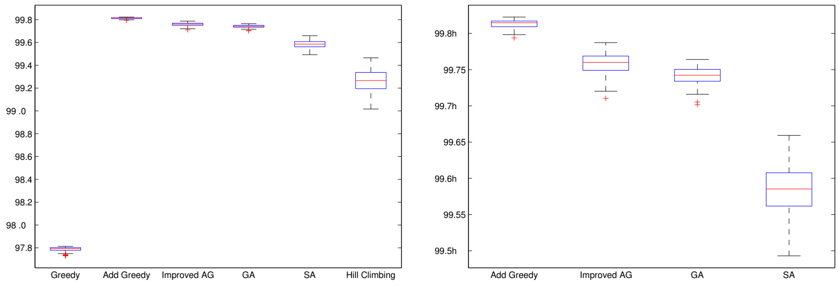
the test-method level, the difference between improved AG and Additional Greedy is significant. Regardless of test case granularity, the improved AG algorithm outperforms Greedy algorithm, Genetic Algorithm, Hill Climbing algorithm and Simulated Annealing algorithm in terms of coverage, although the difference between improved AG and SA is not significant.



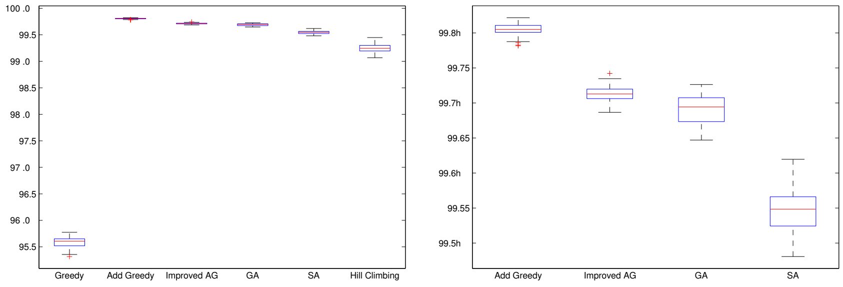
a) *tcas*



b) *tot_info*



c) *print_tokens*



d) *replace*

Figure 5. Boxplots of APSC for the four small C programs by program and by approach

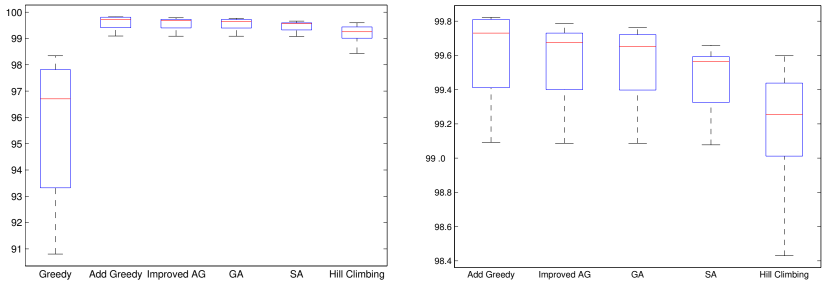


Figure 6. Boxplots of APSC for all small C programs

	Sum of Squares	df	Mean Square	Chi-sq	Significance
Between Groups	1 776.99	5	355.398	580.31	.000
Within Groups	366.51	695	0.527		
Total	2,143.5	839			

Table 6. Friedman test for all three Java programs at the test-class level

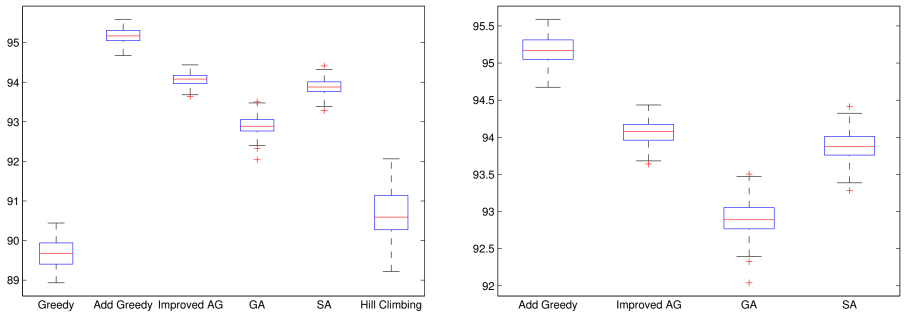


Figure 7. Boxplots of APSC for *space* program

Algorithm (x)	Algorithm (y)	Lower Confidence Limit	Mean Difference (x − y)	Upper Confidence Limit
Greedy	Add Greedy	−3.8425	−3.2464(*)	−2.6504
	Improved AG	−3.5817	−2.9857(*)	−2.3897
	GA	−2.7960	−2.2000(*)	−1.6040
	SA	−3.3067	−2.7107(*)	−2.1147
	Hill Climbing	−0.1675	0.4286	1.0246
Add Greedy	Improved AG	−0.3353	0.2607	0.8567
	GA	0.4504	1.0464(*)	1.6425
	SA	−0.0603	0.5357	1.1317
	Hill Climbing	3.0790	3.6750(*)	4.2710
Improved AG	GA	0.1897	0.7857(*)	1.3817
	SA	−0.3210	0.2750	0.8710
	Hill Climbing	2.8183	3.4143(*)	4.0103
GA	SA	−1.1067	−0.5107	0.0853
	Hill Climbing	2.0325	2.6286(*)	3.2246
SA	Hill Climbing	2.5433	3.1393(*)	3.7353
(*) The mean difference is significant at 0.05 level.				

Table 7. Multiple comparison (LSD) for all three Java programs at the test-class level

A little different from Java programs, improved AG significantly outperforms the other three meta-heuristics based approaches including SA on four small C programs, considering the APSC results.

4.3.2 The Improved AG Algorithm Can Explore the Global Searching Space

Figure 8 a) shows a sample of the optimization process of Genetic Algorithm on Java program *jtopas* at the test-class level, and Figure 8 b) shows the optimization process of the improved AG algorithm with the same test suite. As we can see, GA achieves the final result after about 25 generations while improved AG achieves its

	Sum of Squares	df	Mean Square	Chi-sq	Significance
Between Groups	2 167.5	5	433.499	636.43	.000
Within Groups	216.5	695	0.312		
Total	2 384	839			

Table 8. Friedman test for all three Java programs at the test-method level

Algorithm (x)	Algorithm (y)	Lower Confidence Limit	Mean Difference (x − y)	Upper Confidence Limit
Greedy	Add Greedy	−4.9107	−4.2821(*)	−3.6536
	Improved AG	−4.1393	−3.5107(*)	−2.8821
	GA	−2.3643	−1.7357(*)	−1.1071
	SA	−3.5571	−2.9286(*)	−2.3000
	Hill Climbing	−0.8571	−0.2286	0.4000
Add Greedy	Improved AG	0.1429	0.7714(*)	1.4000
	GA	1.9179	2.5464(*)	3.1750
	SA	0.7250	1.3536(*)	1.9821
	Hill Climbing	3.4250	4.0536(*)	4.6821
Improved AG	GA	1.1464	1.7750(*)	2.4036
	SA	−0.0464	0.5821	1.2107
	Hill Climbing	2.6536	3.2821(*)	3.9107
GA	SA	−1.8214	−1.1929(*)	−0.5643
	Hill Climbing	0.8786	1.5071(*)	2.1357
SA	Hill Climbing	2.0714	2.7000(*)	3.3286
(*) The mean difference is significant at 0.05 level.				

Table 9. Multiple comparison (LSD) for all three Java programs at the test-method level

final result after about 35 generations. However, GA achieves its final APMC result 91.6267 %, but improved AG achieves 91.6533 % finally. This means that population premature appeared in GA after about 25 generations while improved AG can jump out of the local optimization after about 25 generations. Figure 9 a) shows a sample of the optimization process of Genetic Algorithm on C program *tcas*. Figure 9 b) shows the optimization process of improved AG on program *tcas* with the same test suite. In this case, GA also trapped into a local optimal solution, but improved AG jumped out of the trap and got the global optimization.

Improved AG employs the advantages of Simulated Annealing algorithm effectively into Genetic Algorithm by introducing the local stochastic search from SA

	Sum of Squares	df	Mean Square	Chi-sq	Significance
Between Groups	6 678.13	5	1 335.63	1 942.73	.000
Within Groups	196.87	1 995	0.1		
Total	6 875	2 399			

Table 10. Friedman test for all small C programs

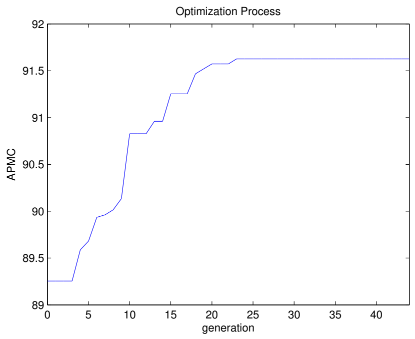
Algorithm (x)	Algorithm (y)	Lower Confidence Limit	Mean Difference (x − y)	Upper Confidence Limit
Greedy	Add Greedy	−5.1923	−4.8187(*)	−4.4452
	Improved AG	−4.3011	−3.9275(*)	−3.5539
	GA	−3.6123	−3.2387(*)	−2.8652
	SA	−2.3886	−2.0150(*)	−1.6414
	Hill Climbing	−1.3736	−1.0000(*)	−0.6264
Add Greedy	Improved AG	0.5177	0.8912(*)	1.2648
	GA	1.2064	1.5800(*)	1.9536
	SA	2.4302	2.8037(*)	3.1773
	Hill Climbing	3.4452	3.8187(*)	4.1923
Improved AG	GA	0.3152	0.6888(*)	1.0623
	SA	1.5389	1.9125(*)	2.2861
	Hill Climbing	2.5539	2.9275(*)	3.3011
GA	SA	0.8502	1.2237(*)	1.5973
	Hill Climbing	1.8652	2.2387(*)	2.6123
SA	Hill Climbing	0.6414	1.0150(*)	1.3886

(*) The mean difference is significant at 0.05 level.

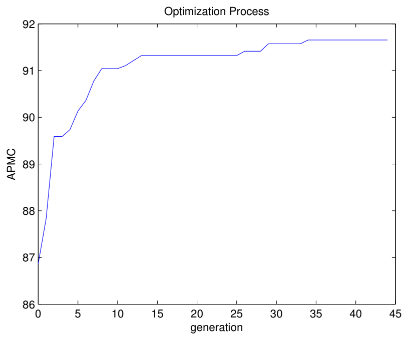
Table 11. Multiple comparison (LSD) for all small C programs

	Sum of Squares	df	Mean Square	Chi-sq	Significance
Between Groups	1 193.86	5	238.771	341.1	.000
Within Groups	31.14	345	0.09		
Total	1 225	419			

Table 12. Friedman test for *space* program



a) optimization process of GA



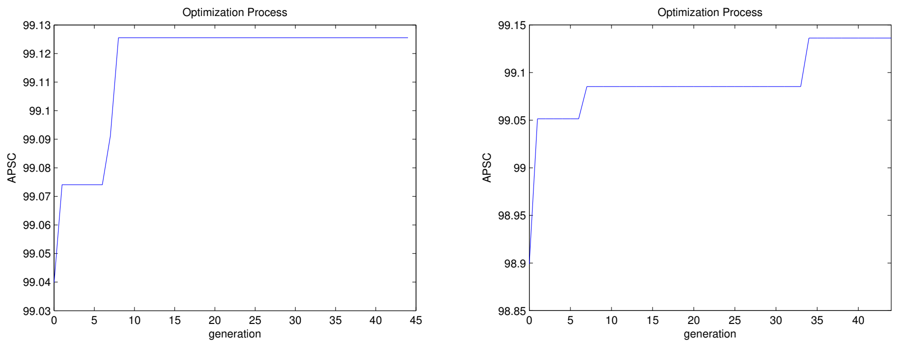
b) optimization process of improved AG

Figure 8. A sample of optimization process of Java program *jtopas* at test-class level

Algorithm (x)	Algorithm (y)	Lower Confidence Limit	Mean Difference (x − y)	Upper Confidence Limit
Greedy	Add Greedy	−5.8012	−4.9000(*)	−3.9988
	Improved AG	−4.6440	−3.7429(*)	−2.8417
	GA	−2.8012	−1.9000(*)	−0.9988
	SA	−3.9583	−3.0571(*)	−2.1560
	Hill Climbing	−1.7012	−0.8000	0.1012
Add Greedy	Improved AG	0.2560	1.1571(*)	2.0583
	GA	2.0988	3.0000(*)	3.9012
	SA	0.9417	1.8429(*)	2.7440
	Hill Climbing	3.1988	4.1000(*)	5.0012
Improved AG	GA	0.9417	1.8429(*)	2.7440
	SA	−0.2154	0.6857	1.5869
	Hill Climbing	2.0417	2.9429(*)	3.8440
GA	SA	−2.0583	−1.1571(*)	−0.2560
	Hill Climbing	0.1988	1.1000(*)	2.0012
SA	Hill Climbing	1.3560	2.2571(*)	3.1583

(*) The mean difference is significant at 0.05 level.

Table 13. Multiple comparison (LSD) for *space* program



a) optimization process of GA b) optimization process of improved AG

Figure 9. A sample of optimization process of C program *tcas*

into GA. This not only maintains the diversity of population to alleviate the premature problem, but also improves the local search capability to accelerate the convergence.

After analyzing the algorithms mentioned above, we can find that improved AG can explore a global searching space and have more probability to achieve the global optimization. Unlike the improved AG, greedy algorithms can only search in a local space and have lower scalability and stability as shown in Table 1, Section 1.

4.3.3 The Improved AG Algorithm Takes a Longer Execution Time

Table 14 and 15 summarize the results of Java programs and C programs respectively. From these two tables, we know that in terms of execution time, greedy algorithms perform more efficiently than meta-heuristics based approaches. The proposed approach performs not so well because it explores a global searching space. The larger the searching space an approach explores, the longer the execution time it takes.

Algorithm	APMC Ranking	Execution Time Ranking
Greedy	5	1
Additional Greedy	1	2
Genetic Algorithm	4	5
Hill Climbing	6	3
Simulated Annealing	3	4
Improved AG	2	6

Table 14. Summary of results for Java programs

Algorithm	APSC Ranking	Execution Time Ranking
Greedy	6	1
Additional Greedy	1	2
Genetic Algorithm	3	5
Hill Climbing	5	3
Simulated Annealing	4	4
Improved AG	2	6

Table 15. Summary of results for C programs

In addition, the experimental results also verify several conclusions which have already been demonstrated in previous works by other people, including Li et al. [10]. They are:

1. Additional Greedy algorithm has the best performance,
2. Hill Climbing algorithm is very unstable, which is easy to yield suboptimal results that are merely locally optimal, but not globally optimal,
3. In the early stage of Genetic Algorithm running, it is likely to cause the population precocity.

In the later stage, the advantage of excellent individuals is not obvious because of consistency in the fitness value, resulting in the entire population evolutionary stagnation.

5 CONCLUSIONS

Regression testing is a frequent and time consuming process. As an important technique for improving the efficiency in using test cases, regression test case prioritization techniques focus on the execution order of test case used in regression testing. As an optimization problem, regression test case prioritization problem can be solved by heuristics. However, as mentioned above, the greedy based algorithms are not stable and scalable when the problem is complex.

To alleviate the problem, this paper proposed a new regression test case prioritization technique based on the improved Annealing-Genetic (AG) algorithm. The improved AG algorithm, firstly, combines Simulated Annealing algorithm and Genetic algorithm to explore a larger searching space, secondly, employs a local search operator to search the local optimum much faster, and thirdly, hires optimization reserved strategy to hold the optimal solution in each iteration. As a result, improved AG has more probability to achieve the better and stable optimum. After introducing the detail of improved AG, an experiment was conducted to evaluate the performance of the proposed approach in terms of coverage and execution time. The Friedman test [24] rather than ANOVA has been launched to compare the results of different algorithms. The test results showed that the performance of the proposed approach is equal to Additional Greedy algorithm and it performs significantly better than Greedy algorithm, Genetic Algorithm and Hill Climbing algorithm for all programs used. Improved AG also outperforms Simulated Annealing algorithm, although the difference between them is not significant in terms of coverage. In short, improved AG can not only avoid population premature of Genetic Algorithm but also can have higher stability and scalability. However, the proposed approach performs not so well in terms of execution time because of the global searching space it explores.

6 FUTURE WORK

We have found two main research directions for the future work. First, the improved AG algorithm will be evaluated with different parameter settings in the future. This will elucidate whether its good performance is due to algorithm itself or parameter settings. Second, some work should be done to improve the proposed approach so that it can have a better performance in terms of execution time. Besides these two directions, we plan to conduct some further experiments with much larger size programs.

7 ACKNOWLEDGEMENTS

This work is partly supported by a project from National Natural Science Foundation of China, with the project number ‘61202030’ and ’71502125’. The authors also thank anonymous reviewers for their constructive comments.

REFERENCES

[1] Jeya Mala, D.—Ruby, S.—Mohan. V.: A Hybrid Test Optimization Framework – Coupling Genetic Algorithm with Local Search Technique. Computing and Informatics, Vol. 29, 2010, No. 1, pp. 133–164.

[2] Rothermel, G.—Harrold, M. J.: Analyzing Regression Test Selection Techniques. IEEE Transactions on Software Engineering, Vol. 22, 1996, No. 8, pp. 529–551, doi: 10.1109/32.536955.

[3] Schach, S.: Software Engineering. Boston, MA, Aksen Associates, 1992.

[4] Leung, H. K. N.—White, L. J.: Insights into Regression Testing. Proceedings of the International Conference on Software Maintenance, 1989, pp. 60–69.

[5] Rothermel, G.—Harrold, M. J.: A Safe, Efficient Regression Test Selection Technique. ACM Transactions on Software Engineering and Methodology, Vol. 6, 1997, No. 2, pp. 173–210, doi: 10.1145/248233.248262.

[6] Chen, T. Y.—Lau, M. F.: On the Divide-and-Conquer Approach Towards Test Suite Reduction. Journal of Information Sciences, Vol. 152, 2003, No. 1, pp. 89–119, doi: 10.1016/s0020-0255(03)00060-4.

[7] Wong, W. E.—Horgan, J. R.—London, S. et al.: A Study of Effective Regression Testing in Practice. Proceedings of IEEE International Symposium on Software Reliability Engineering, 1997, pp. 264–274, doi: 10.1109/issre.1997.630875.

[8] Kim, J. M.—Porter, A.: A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments. Proceedings of the International Conference on Software Engineering, May, 2002, pp. 119–129, doi: 10.1145/581339.581357.

[9] Srikanth, H.—Williams, L.—Osborne, J.: System Test Case Prioritization of New and Regression Test Cases. Proceedings of the International Symposium on Empirical Software Engineering, 2005, pp. 62–71, doi: 10.1109/isese.2005.1541815.

[10] Li, Z.—Harman, M.—Hierons, R. M.: Search Algorithms for Regression Test Case Prioritization. IEEE Transactions on Software Engineering, Vol. 33, 2007, No. 4, pp. 225–237, doi: 10.1109/TSE.2007.38.

[11] Rothermel, G.—Untch, R. H.—Chu, C.—Harold, M. J.: Prioritizing Test Cases for Regression Testing. IEEE Transactions on Software Engineering, Vol. 27, 2001, No. 10, pp. 929–948, doi: 10.1109/32.962562.

[12] Elbaum, S.—Malishevsky, A. G.—Rothermel, G.: Test Case Prioritization: A Family of Empirical Studies. IEEE Transactions on Software Engineering, Vol. 28, 2002, No. 2, pp. 159–182, doi: 10.1109/32.988497.

[13] Rothermel, G.—Untch, R. H.—Chu, C.—Harold, M. J.: Test Case Prioritization: An Empirical Study. Proceedings of the International Conference on Software Maintenance, Oxford, UK, 1999, pp. 179–188, doi: 10.1109/icsm.1999.792604.

[14] Elbaum, S. G.—Malishevsky, A. G.—Rothermel, G.: Prioritizing Test Cases for Regression Testing. Proceedings of the International Symposium on Software Testing and Analysis, Portland, Oregon, USA, 2000, pp. 102–112, doi: 10.1145/347324.348910.

[15] Elbaum, S.—Gable D.—Rothermel, G.: Understanding and Measuring the Sources of Variation in the Prioritization of Regression Test Suites. Proceedings of the Seventh International Software Metrics Symposium, London, England, 2001, pp. 169–179.

[16] Elbaum, S. G.—Malishevsky, A. G.—Rothermel, G.: Incorporating Varying Test Costs and Fault Severities into Test Case Prioritization. Proceedings of the International Conference on Software Engineering, Toronto, Ontario, Canada, 2001, pp. 329–338, doi: 10.1109/icse.2001.919106.

[17] Malishevsky, A.—Rothermel, G.—Elbaum, S.: Modeling the Cost-Benefits Tradeoffs for Regression Testing Techniques. Proceedings of the International Conference on Software Maintenance, Montreal, Canada, 2002, pp. 204–213, doi: 10.1109/icsm.2002.1167767.

[18] Rothermel, G.—Elbaum, S.—Malishevsky, A.—Kallakuri, P.—Davia, B.: The Impact of Test Suite Granularity on the Cost-Effectiveness of Regression Testing. Proceedings of the 24th International Conference on Software Engineering, Orlando, USA, 2002, pp. 130–140, doi: 10.1145/581356.581358.

[19] Maia, C. L. B.—do Carmo, R. A. F.—de Freitas, F. G.—Lima de Campos, G. A.—de Souza, J. T.: Automated Test Case Prioritization with Reactive GRASP. Advances in Software Engineering, 2010, Vol. 2010, Article ID 428521, doi: 10.1155/2010/428521.

[20] Zhang, L. M.—Hao, D.—Zhang, L.—Rothermel, G.—Mei, H.: Bridging the Gap Between the Total and Additional Test-Case Prioritization Strategies. Proceedings of the 2013 International Conference on Software Engineering, 2013, pp. 192–201, doi: 10.1109/icse.2013.6606565.

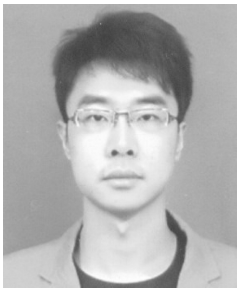
[21] Lin, F.-T.—Kao, C.-Y.—Hsu, C.-C.: Applying the Genetic Approcah to Simulated Annealing in Solving Some NP-Hard Problems. IEEE Transactions on Systems, Man and Cybernetics, Vol. 23, 1993, No. 6, pp. 1752–1767, doi: 10.1109/21.257766.

[22] Hutchins, M.—Foster, H.—Goradia, T.—Ostrand, T.: Experiments of the Effectiveness of Dataflow- and Controlflow-Based Test Adequacy Criteria. Proceedings of the 16th International Conference on Software Engineering, 1994, pp. 191–200.

[23] Cobertura. Availaible on: `http://cobertura.github.io/cobertura/`.

[24] Friedman, M.: The Use of Ranks to Avoid the Assumption of Normality Implicit in the Analysis of Variance. Journal of the American Statistical Association, Vol. 32, pp. 675–701, 1937, doi: 10.1080/01621459.1937.10503522.

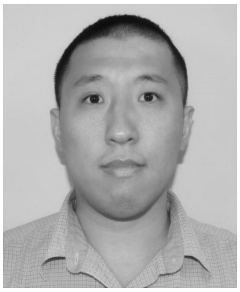
Zan Wang received his B.Sc. degree in applied mathematics in 2000, his M.Sc. degree in computer science in 2004 and Ph.D. degree in information management and information systems in 2009 from Tianjin University, Tianjin, China. Now, he serves as Associate Professor in the Department of Software Engineering of Tianjin University, Tianjin, China. His current research interests include software quality, machine learning, random optimization algorithms and social network.



Xiaobin Zhao received his B.E. degree in software engineering from Tianjin University, Tianjin, China. Currently, he is an M.E. student of Tianjin University. His research interests include software quality, software testing and optimization, and machine learning.

Yuguo Zou received his B.E. degree in software engineering from Tianjin University, Tianjin, China. He is studying in Tianjin University for his Master's degree. He published a paper in an international journal. His research interests include software engineering, software testing and automatic fault localization.

Xue Yu received her B.Sc. degree in computer science in 2000 from Tianjin University, Tianjin, China, her M.Sc. degree in computer science in 2003 from University of Wollongong, Wollongong, N.S.W., Australia, and Ph.D. degree in information management and information systems in 2009 from Tianjin University, Tianjin, China. She is Associate Professor in the Department of Information Management and Information Systems, Tianjin University, Tianjin, China. Her current research interests are optimization algorithms, information filtering, collaborative filtering, recommendation systems and business intelligence.



Zhenhua Wang received his Ph.D. degree from Clemson University, South Carolina, USA. He is currently working at American Electric Power, Ohio, USA. His research interests include power system optimization using genetic algorithms, smart grid, renewable energy and power system stability.