Online Resource for FCS-19229

1. Basic knowledge

For convenience, some fundamental contents are first introduced here.

1.1 Regression test case selection

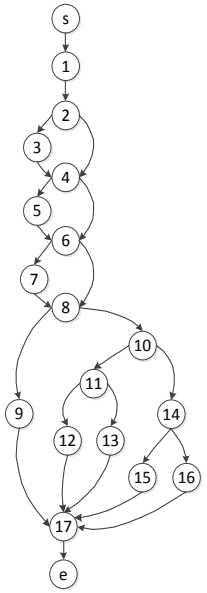
The goal of regression test selection is to select test cases that execute changed code of the program under test or code affected by changes to reduce the test cost, and at the same time, preserve the same fault detection ability as the original test suite. Generally, the RTS problem can be defined as follows [1]:

- Given a previous version of the program *P*, the current version of the program *P'* and a test suite *T* for *P*, select a subset *T'* of *T* and use it to test *P'*.

1.2 Control Flow Diagram

Control Flow Diagram (CFD) is a graphical representation of the program control structure. It is a directed graph G (N, E, s, e), where *N* is called the node-set of graph G and each node in it corresponds to a program statement. *E* is called the set of edges. (n_i, n_j) is called the edge of G, and it indicates that control flow exists from n_i to n_j. Generally, the CFD of each program contains unique entry node s and exit node e. Fig. 1(b) is the CFD of triangle classification program, corresponding to the source program in Fig. 1(a) [2].

```
void triangle(int a,int b,int c)
 {
1  int t;
2   if(a>=b)
3       { t=a;a=b;b=t;}
4   if(a>=c)
5       { t=a;a=c;c=t ;}
6   if(b>=c)
7       {t=b;b=c;c=t}
8    if(a+b<=c)
9      printf( "Not Triangle" );
     else
      {
10       if(a==b)
         {
11             if(b==c)
12               printf( "Equilateral" );
                 else
13               printf( "Isosceles" );
          }
         else
         {
14           if(b==c)
15               printf( "Isosceles" );
             else
16               printf( "Triangle" );
         }
     }
17 return 0;
  }
```



(a) Source program (b) Control Flow Diagram

Fig. 1 Triangle classification program

1.3 Analysis of node types

To study the balance of test cases traversing program branches, we analyze types of program nodes (program statements). The selected nodes and the process of calculating balance are described below.

Sequence, selection and cycle are the three basic structures of the program. In order to reduce the cost, it is essential to select nodes to calculate the balance. From Fig. 1(b), we can see that when a node (corresponding to a statement in Fig. 1(a)) has only one direct successor, the number of test cases traversing the node is the same as traversing its direct successor. That is, test cases traverse the node and its direct successor in a balanced way. Therefore, sequential nodes will not be considered to calculate the balance in our method. For example, n_1 has only one direct successor n_2 in Fig. 1(b). It can be seen that the number of test cases traversing n_1 and n_2 must be equal. Therefore, we ignore the nodes like n_1. However, it is another case for selection or cycle structure. As each branch node includes at least two direct successors, that is, if the number of direct successors of the branch node n_i is expressed as $Next(n_i)$, it can be seen that $Next(n_i) > 1$. In this case, test cases traversing the branch node may traverse several of its direct successors unequally. Thus the number of test cases traversing each of the direct successors should be considered to calculate balance.

Therefore, we take branch nodes into account to calculate the balance. According to the Z-path coverage [3], the cycle structure can be changed into a double-branch selection structure basing on times the cycle body is executed. For convenience, branch nodes and the loop nodes are both referred to as branch nodes in the subsequent part of this paper. As shown in Fig. 1(b), n_2 is a branch node and its two direct successors are n_3 and n_4. Given that there are five test cases and the number of test cases traversing n_3 and n_4 is five, zero respectively. In other words, the five test cases traverse n_3 while none of them traverse n_4. Hence, the imbalance traversing two branches of n_2 should be given attention to getting an adequate test of the program.

2 Evolution selection of regression test cases based on diversity

In this paper, we propose a RTS method to achieve balanced testing of programs. The steps of the proposed method are described as follows.

2.1 Steps of test case evolution selection

Step1 Assign the values of all control parameters used in this algorithm, encode the statements of

the program under test with a series of nodes, choose the target paths and instrument the program;
Step2 Initialize a population and randomly generate gene of each individual;
Step3 All test cases represented by individuals execute the program under test;
Step4 Judge whether the terminal criterion of the algorithm is met, i.e., the maximum generation or path coverage is met. If it is, turn to step 7;
Step5 Calculate the fitness of each individual;
Step6 Perform selection, crossover and mutation operations to generate offspring, turn to step 3;
Step7 Stop the evolution process, decode the optimal solutions, output the desired test cases and their traversed paths.

2.2 Diversity-based test cases selection algorithm

The algorithm of our method is shown in Algorithm 1. The variables are shown in the LETTER.

Algorithm 1 Diversity-based test cases selection algorithm

Input: The original test cases, target paths $v = \{p_1, p_2, \cdots p_{|tp|}\}$ and the maximum generation G
Output: The selected test cases
1:BEGIN
2: $Setprameters()$;
3: $Initialize(x_k)$;
4: $generation \leftarrow 0$;
5: $Do\ \ WHILE(generation \le G \parallel Individual(x_k).r < |tp|)$
6: $Individual(x_k).r \leftarrow 0$;
7: $generation = generation + 1$;
8: $Fitness(x_k)$;
9: $Select(x_k)$;
10: $Cross(x_k)$;
11: $Mulate(x_k)$;
12: $IF((Individual(x_k).r == |tp|) \parallel (generation == G))\ \ THEN$
13: $Find\ \max(Individual(x_k).r)$;
14: $TestCaseSet() \leftarrow x_k$;
15: END IF
16: END WHILE
17: END

2.3 Encoding individuals

Individuals of Genetic Algorithm (GA) are encoded in binary form. The number of encoded genes included in each individual is the same as the number of original test cases. Each gene in the individual corresponds to a different test case. Each gene has a value of 0 or 1 at the locus. If the value is 1, it indicates that the corresponding test case of the gene is included in the individual. Here we call the gene is activated. Otherwise, if it is 0, it indicates that the test case is discarded

by the individual. Thus it will not participate in evolution selection. For example, suppose that there are two individuals x_1 and x_2, and eight test cases from t_1 to t_8 in the original set. Accordingly, eight genes exist in each individual and each gene corresponds to a different test case. The correspondence is shown in line with the arrows in Fig. 2. For t_1, t_3, t_4 and t_6, their corresponding genes in x_1 are all 1, and it indicates that these test cases are activated. In other words, x_1 contains the four test cases. Similarly, there are three test cases in x_2, they are t_1, t_4 and t_7 respectively.

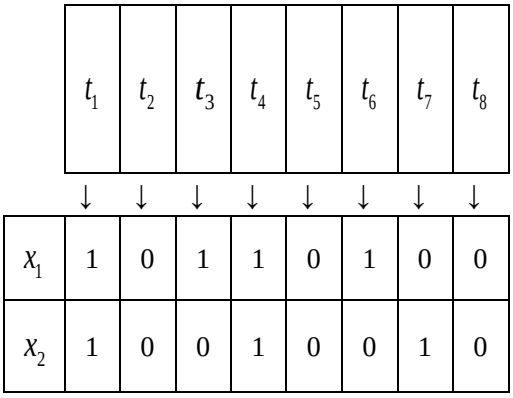


Fig. 2 Genes of individuals and corresponding test cases

Denote GE_g as the gth gene. To randomly encode genes in individuals, we realize it by generating random numbers, as shown in Eq. (1):

$$GE_g = \begin{cases} 0, 0 \le rand < \alpha \\ 1, \alpha \le rand \le 1 \end{cases} \qquad (1)$$

Where $rand$ means a random number and the variable a is used to adjust the number of different gene values. It is obvious that we have transformed the ranges of $rand$ and a into the same range [0,1].

3 Experiments

In order to further verify the effectiveness of the proposed method, in addition to the triangle classification program and interserve program, all the Siemens programs, one Unix program grep and the space program of the European Space Agency [2] are considered to verify the effectiveness of the proposed method.

Different methods may have different capabilities in selecting test cases. Using our method, test cases are selected based on the similarity of execution paths to achieve coverage balance. So we compare our method (marked as DRTS) mainly with similarity-based test case selection methods for experiments, a semi-supervised clustering method (marked as SCRTS) [4] and two methods (marked as BFOS and SFOS) based on the farthest-first ordered sequence for test case

prioritization and selection [5]. The BFOS is a branch coverage-based method and the SFOS is a statement coverage-based method. Besides, we compare the proposed method with the Random method. One reason is that the Random method is a common method. Many existing methods of test case selection are compared with it such as the method in [1]. Second, the Random method used in this paper does not directly select test cases randomly from original test cases. It uses the same individuals representation as the proposed method. Using the Random method, crossover and mutation operations will not be performed, and a test case set represented by an individual will be selected randomly from all individuals. Common parameters and the initial population data of all these methods are the same to guarantee a fair comparison.

3.1 Implementation platform and parameter settings

All programs are written in C language and run in the environment of VC++ 6.0. In all experiments, we use roulette-wheel selection, one-point crossover and one-point mutation. The probabilities of crossover and mutation are set to 0.9 and 0.3 respectively in the experiments. One reason is that other references also set the parameter values of Genetic Algorithm like this. For example, several methods in [2]. Besides, it is found by multiple experiments that when the probabilities of crossover and mutation are set to 0.9 and 0.3, the experimental results of our method are better than other comparison methods, as is seen in section 3.4. For other parameters, we will give their values in the corresponding experiments. In each run, the terminal condition of Genetic Algorithm is to reach the maximum evolution generation or the path coverage is 100%.

Mutation testing is a fault-oriented testing technique. During mutation testing, we can choose the location and type of faults to be seeded, and we can also choose mutation operators based on different levels of testing [6]. As programs with real faults are difficult to find, one common solution is to seed faults, either manually or automatically by applying mutation operators into the programs. We choose the former way to seed faults. We use mutation operators to manually implant faults into the source program to simulate real faults in the software. As the efficiency of weak mutation testing is significantly higher than that of strong mutation testing. We apply the method proposed in [7] to transform the mutant killing problem into a branch coverage problem based on weak mutation testing. Considering the characteristics of the selected programs, we apply 14 types of Method-Level mutation operators in MuClipse to construct mutation branches [7]. As the scale of many of the selected program functions are not large in the experiment, we use the method of manual implantation of mutants. At present, faults seeded automatically by mutation tools has been applied widely, such as Jumble [5] and Muclipse [7], next we will further study using these tools.

We consider different program sizes and the number of target paths when seeding faults. The larger the program size and the more the target paths, the more faults will be seeded. For the triangle classification program, we seeded 91 faults. The mutation operators of Method-Level used in the program include ROR, AOIU, AOIS, LOI, AORB and COI. The used mutation operators, mutants number and test case number of other programs are set as shown in Table 1. Paths and LOC represent the number of target paths and lines of code respectively. Besides, Operators, Mutants and Test cases represent mutation operators, the number of faults and the number of original test cases respectively. Each experimental result is obtained by calculating the average value by experiments for multiple times.

Table 1 The parameter setting of programs

ID	Programs	LOC	functions	LOC	Paths	Operators	Mutants	Test cases
C1	schedule	368	upgrade_process _prio	26	6	ROR,COI, AODS	52	46
C2	interserve	136	interserve	136	20	ROR,COI AORS	168	231
C3	tcas	173	alt_sep_test	31	39	COR,COD	86	266
C4	tot_info	406	tot_info	406	33	AORB,COI, AORS,ROR AODU	257	318
C5	flex	13255	YY_DECL, add_action	1133	103	COD,ASRS, ROR,COI, AODS	582	645
C6	space	6199	fixsgrid	115	16	ROR,COI	223	342
C7	space	6199	fixgramp	90	12	ROR,COI, AORS	134	208
C8	schedule2	308	schedule	35	8	AOIS,LOI	89	138
C9	print_token	192	get_actual_token	19	12	AODS,COD ROR,COI	42	39
C10	print_token2	53	print_token2	53	11	LOR,ROR, COI	152	182
C11	replace	564	makepat	59	22	COR,COI, COD,ROR	224	384
C12	grep	13226	common_op_mat ch_null_string_p	82	18	ASRS,COD, ROR,COI, AORS	364	329

3.2 Questions to be verified and experimental metrics

Our experiment uses the following metrics and aims to explore the following research questions (RQ):

RQ1: Can our method select test cases effectively?

It is verified by reduction rate (RR) in Eq. (2), where $|tc|$ indicates the number of original test cases, while $|tc|'$ represents the number of final selected test cases. RR is the ratio of unselected cases to the number of original cases.

$$RR = \frac{|tc| - |tc|'}{|tc|} \qquad (2)$$

RQ2: Does the parameter α in this method affect the reduction rate and fault detection rate?

It is verified by analyzing experimental results obtained under different values of α. The purpose of discussing RQ2 is to verify the impact of changing α on reduction rate and further the impact on fault detection rate.

RQ3: Can the test cases selected by our method adequately cover target paths in the program under test?

After test cases represented by x_k executing the program, we use the path coverage (PC) $Cov(x_k)$ (See Eq. (4) in the LETTER) that the test cases can achieve to verify the adequate selection of test cases.

RQ4: Can test cases selected by our method traverse the program evenly?

We use the coverage balance (CB) $CB(x_k)$ (See Eq. (2) in the LETTER) to evaluate the balance of test cases traversing the program. The smaller $CB(x_k)$, the more balanced to traverse program branches by selected test cases.

RQ5: Can the proposed method effectively detect faults?

We use the fault detection rate (FD) to evaluate the fault detection ability of the proposed method. The fault detection rate is the ratio of faults that can be detected by selected test cases to those that can be detected by the original test cases.

RQ6: How is the precision and recall of the proposed method for selecting test cases?

Precision is used to indicate the proportion of test cases in the result set that contain fault detection test cases. The recall rate is used to indicate the proportion of selected test cases for fault detection to all test cases for fault detection. A comprehensive measurement $F-measure$ is usually used as a comprehensive evaluation method of precision and recall. It [8] can be expressed as follows:

$$F-measure = 2 * Precision \times Recall / (Precision + Recall) \quad (3)$$

RQ7: How efficient is the proposed method?

In addition to the above metrics, we use the run time of our algorithm to evaluate the efficiency.

3.3 Discussion of experimental parameters

As the triangle classifier program is typical and it has been widely used as a benchmark program [2]. We take it as an example to discuss parameters in this section.

3.3.1 Discussion of the ratio of the number of selected individuals to the number of test cases

The most important attribute for RTS is whether the test suite selected is still effective in terms of fault detection. So when discussing the ratio of the number of selected individuals to the number of test cases, we first consider the fault detection ability at different ratios of our method. In addition, we also consider the efficiency of the algorithm. In the experiments, the number of individuals (NOI) ranges from 5 to 60. By adjusting variable a in Eq. (1), the reduction rate of the test cases in the experiments is set small in order to find the ratio that can achieve a high fault detection rate. The ratio is discussed under seven different experimental conditions. Experimental results are in Table 2 and Table 3.

In Table 2, “data range” represents the value range of the three input data of the triangle classification program. When the data range is [1,128] and the ratio is 1: 1, the fault detection rate of the algorithm is low, which is 52.7%. That is because the selected test cases cannot effectively cover the branches where the faults are located. From Table 2 we can see that when the ratio is 1: 5, the fault detection rate of the algorithm is better than other ratios on the whole. Although the fault detection rate is also high when the ratio is 1: 7, it can be seen from Table 3 that at this time, the algorithm runs for a long time. In other words, the algorithm efficiency with a ratio of 1: 7 is lower than that with a ratio of 1: 5. Especially when there are many test cases and large data range, the run time is longer. Therefore, the ratio of the number of selected individuals to the number of test cases is set to 1: 5 in experiments.

Table 2 Effect of different ratio on fault detection rate

Experimental settings		Ratios of the selected individuals to the number of test cases						
Data range	NOI	1:1	1:2	1:3	1:4	1:5	1:6	1:7
[1,128]	5	52.7	71.1	93	96.3	**96.6**	96.1	95.3
[1,256]	10	74.6	74.9	75	89.5	**93.3**	92.5	92.25
[1,512]	20	75	99.3	99	98.3	99.8	99.8	**100**
[1,1024]	30	73.3	75	99.8	99.5	**100**	**100**	**100**
[1,2048]	40	**100**	**100**	**100**	**100**	**100**	99.5	99.3
[1,4096]	50	98.3	95.8	**100**	**100**	**100**	**100**	**100**
[1,8192]	60	96.8	95.5	95	93.3	94	94.3	**97.8**

As can be seen from Table 3, in addition to several values, we can draw a conclusion. With the expansion of the data range and the increase in the number of test cases and individuals, more and more time is consumed. It can also be seen that as the ratio of the number of individuals to the number of test cases decreases, the run time consumes more and more from the trend in Table 3.

Table 3 Effect of different ratio on the run time

Experimental settings		Ratios of the selected individuals to the number of test cases						
Data range	NOI	1:1	1:2	1:3	1:4	1:5	1:6	1:7
[1,128]	5	0.03	0.03	**0.026**	0.03	0.028	0.052	0.04
[1,256]	10	**0.028**	0.059	**0.028**	0.038	0.05	0.11	0.12
[1,512]	20	**0.02**	0.04	0.061	0.051	0.07	0.089	0.144
[1,1024]	30	**0.05**	0.052	0.069	0.09	0.11	0.13	0.16
[1,2048]	40	**0.072**	0.08	0.14	0.177	0.191	0.22	0.252
[1,4096]	50	**0.12**	0.14	0.194	0.27	0.27	0.387	0.393
[1,8192]	60	**0.087**	0.134	0.2	0.26	0.343	0.32	0.4

3.3.2 Effect analysis of test balance

Experiments are performed with and without coverage balance in the individual fitness. In the case of common parameters and reduction rate are the same, the effects of two different fitness functions of individuals on the fault detection rate, run time and $F-measure$ are discussed under seven different experimental conditions. The two different fitness functions are shown in the Eq. (4) and Eq. (5). $f(x_k)$ indicates that the fitness of x_k is calculated based on the coverage balance and path coverage after the individual x_k running the program. $f'(x_k)$ only considers the path coverage but not the coverage balance. The details of coverage balance $CB(x_k)$ and path coverage $Cov(x_k)$ are shown in Eq. (2) and Eq. (4) respectively in the LETTER.

$$f(x_k)=1-BP(x_k)+Cov(x_k) \quad (4)$$

$$f'(x_k)=Cov(x_k) \quad (5)$$

Experimental results are shown in Table 4. NOI and TCS are denoted as the number of individuals and test case size respectively. It can be seen that when coverage balance is considered,

the values of fault detection rate, run time and $F-measure$ are better. This is because if the balance is considered in the calculation of the individual fitness, the termination condition of GA is reached faster and the program will be more evenly covered by selected test cases. We can conclude that test balance is useful in practice.

Table 4 Effect analysis of test balance

Data range	NOI	TCS	$f(x_k)$			$f'(x_k)$		
			FD/%	time/s	FM	FD/%	time /s	FM
[1,128]	5	25	**100**	**0.031**	**0.93**	96.5	0.036	0.920
[1,256]	10	50	**100**	**0.04**	**0.912**	96.3	0.046	0.900
[1,512]	20	100	**100**	**0.122**	**0.932**	95.3	0.153	0.917
[1,1024]	30	150	**100**	**0.251**	**0.906**	97	0.291	0.893
[1,2048]	40	200	**100**	**0.31**	**0.900**	96.3	0.46	0.888
[1,4096]	50	250	**100**	**0.439**	**0.927**	96.5	0.501	0.908
[1,8192]	60	300	**100**	**0.54**	**0.928**	96.8	0.902	0.914

3.4 Experimental results

For the seven questions to be verified in 3.2, the experimental results are given in this section. As we set the initial population data of all these methods be the same to guarantee a fair comparison, and some program functions do not contain many lines of code. There are overlapping values in the results. The average values of path coverage, test case reduction rate, coverage balance, run time, $F-measure$ and the fault detection rate of the five methods are recorded by multiple experiments.

(1) Reduction rate

For reduction rate, compared with other methods, for DRTS only the "tot_info" program (C4) with a reduction rate of 63.3%, which is slightly lower than 66.7% of the SCRTS method in Table 5. But it is better than the values of BFOS, SFOS and the Random method, and BFOS has a minimum reduction rate of 52.7%. For other programs, the reduction rate of the proposed method is higher than other methods. Besides, though there are several overlaps between SCRTS and SFOS, SFOS shows a small improvement on SCRTS for other programs. The data verify the RQ1, it means that our method can effectively reduce the number of test cases.

Table 5 Comparison of reduction rate and path coverage

ID	Reduction rate %					Path coverage %				
	DRTS	SCRTS	BFOS	SFOS	Random	DRTS	SCRTS	BFOS	SFOS	Random
C1	**46.2**	36.7	40.2	36.7	22.7	**100**	**100**	**100**	**100**	88.4
C2	**74.7**	62.6	63.2	68.7	60.5	**100**	88.9	**100**	**100**	70.5
C3	**66.7**	56.7	60.4	56.7	16.6	**100**	**100**	**100**	**100**	**100**
C4	63.3	**66.7**	52.7	53.4	58.3	**100**	92.7	93.3	95.7	55
C5	**59.8**	41.3	51.8	53.7	50.9	**100**	69.6	95.7	97.7	48.7
C6	**60.7**	46.7	49.8	49.1	48.8	**100**	73.8	97.7	**100**	62.5
C7	**58.9**	44.7	54.6	53.3	50.9	**100**	91.1	**100**	**100**	63.9
C8	**42.5**	33.4	36.7	33.4	27.4	**100**	**100**	**100**	**100**	**100**

C9	**38.5**	35.6	34.2	35.6	22.9	**100**	**100**	97.7	**100**	**100**
C10	**55.2**	48.6	50.6	53.7	31.5	**100**	**100**	96.4	95.3	**100**
C11	**46.1**	31.7	29.9	37.3	25.7	**86.7**	82	84.1	82	64
C12	**57.7**	31.8	33.4	36.1	17.8	**88.9**	68.9	73.6	70.7	52.4

(2) Influence of different a on the reduction rate and the fault detection rate

We use the triangle classification program to verify the impact of a on the reduction rate and the fault detection rate. We conduct the experiments in two cases, one is under the small data range [1,512], and the other is under the large data range [1,8192]. Experimental results are shown in Fig. 3, Fig. 4, Fig. 5 and Fig. 6. In the case of different numbers of original test cases, the influence of different a (in Eq. (1)) on reduction rate and the fault detection rate are discussed. The maximum evolution generation is set to 300. We first select a small range of test data [1,512] in the experiment.

The results of different a on reduction rate are shown in Table 6. Denote $|tc|$ as the number of test cases. It can be seen from Table 6 that the reduction rate (denoted as RR) goes up gradually with the increase of a for different numbers of test cases. Besides, it can also be concluded that the more of initial test cases, the bigger the reduction rate with the same a. For example, when the number of test cases is 50 and a is 0.1, the reduction rate is 5.6%. When a is 0.5 or 0.99, the value is 39.3% and 60.4% respectively. Considering $|tc|$ is 300, a is 0.1, the reduction rate is 8.1%, when a is 0.5 or 0.99, the reduction rate is 47.4% and 88.3% respectively.

Table 6 Influence of different a on reduction rate under small data range

RR \ |*tc*| / *a*	50	100	150	200	300
0.1	5.6	6.5	7.6	7.8	8.1
0.2	15.2	16.6	16.4	17.0	17.8
0.3	26.3	27.2	29.0	28.2	28.4
0.4	32.7	34.9	36.4	36.7	38.8
0.5	39.3	41.4	46.3	45.4	47.4
0.6	48.0	52.0	51.7	52.3	54.4
0.7	51.3	57.1	61.7	62.1	62.2
0.8	56.9	66.2	65.5	68.7	70.4
0.9	57.3	70.5	73.2	77.3	82.6
0.95	58.8	74.0	78.2	79.5	86.3
0.99	60.4	77.7	79.6	80.9	88.3

Table 7 Influence of different a on reduction rate under large data range

RR \ |*tc*| / *a*	50	100	200	300
0.1	6.5	7.8	8.8	8.6
0.2	17.1	17.4	17.9	17.9
0.3	23.3	27.0	26.1	27.2
0.4	32.9	36.9	37.6	38.4
0.5	42.9	44.5	48.6	48.8
0.6	46.8	51.9	54.9	58.7
0.7	53.9	56.9	65.4	66.2
0.8	56.9	62.1	69.3	74.2
0.9	57.1	69.4	78.2	81.5
0.95	57.9	73.6	80.7	86.4
0.99	58.8	74.5	83.0	89.4

To further verify the effect of a on the reduction rate, a large data range of [1,8192] is selected for the experiment, in which the maximum evolution generation is the same as the small data range. Experimental results are shown in Table 7. When $|tc|$ is 100 and a is 0.1, *RR* is 7.8%, when a is 0.5 or 0.99, *RR* is 44.5% and 74.5% respectively. Considering $|tc|$ is 300 and a is 0.1, *RR* is 8.6%. When a is 0.5 or 0.99, *RR* is 48.8% and 89.4% respectively. We can reach the same conclusion as in the case of the small data range.

Fig. 3 and Fig. 4 show the effect of different a on reduction rate intuitively. We can find that the reduction rate converges faster when there are fewer initial test cases. On the contrary, the

reduction rate converges slower when there are more initial test cases. According to Table 6 and Table 7 as well as Fig. 3 and Fig. 4, it can be concluded that a has a direct effect on the reduction rate. The reduction rate will increase with the increase of a under the same population size and test case size. In other words, increasing the a means decrease the size of initial test suite. It indicates that reduction rate of test cases can be controlled by changing a.

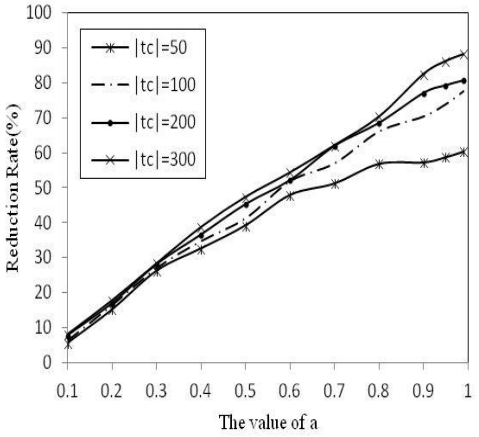


Fig. 3 Effect of a on reduction rate under small data range

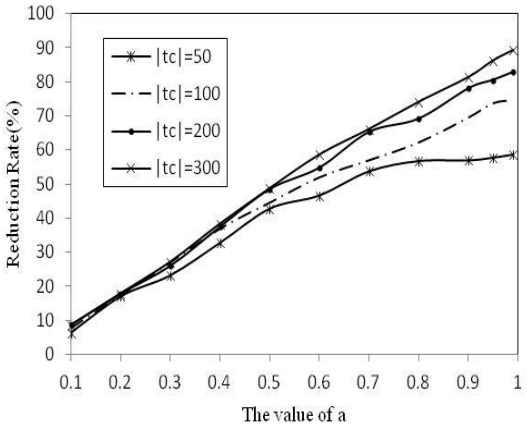


Fig. 4 Effect of a on reduction rate under large data range

From Fig. 3 and Fig. 4, we can see that the reduction rate of test cases can be controlled by adjusting variable a. It is more important to know what the impact of different controlled reduction rates might be in RTS. So we further verified how the fault detection rate of the proposed method will be affected as the reduction rate varies under different a. We use the triangle classification program to verify the impact. In the experiment, a takes values from 0 to 1. In each data range, we have done four sets of comparative experiments for different number of initial test cases. Experimental results are shown in Fig. 5 and Fig. 6.

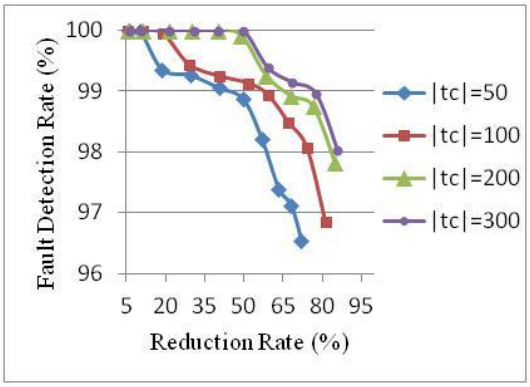


Fig. 5 Effect of different reduction rate on the fault detection under small data range

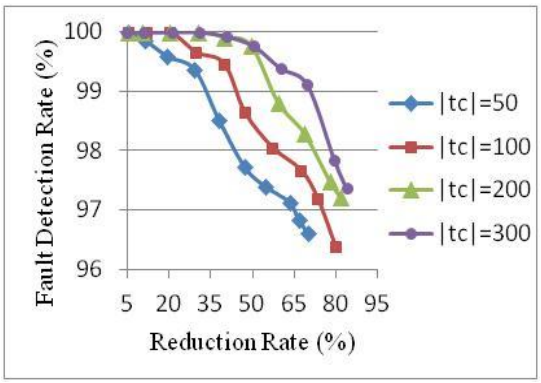


Fig. 6 Effect of different reduction rate on the fault detection under large data range

When the data range is [1,512], the result is shown in Fig. 5. For each number of test cases, it can be seen that the fault detection rate decreases slowly with the increase of reduction rate at the beginning, and then decreases more obviously. This is because deleting redundant test cases and a small number of test cases that can detect faults have little effect on the fault detection rate. Then the fault detection rate will decrease rapidly as the reduction rate further increases, as more and more test cases that can detect faults are reduced. When the data range is [1,8192], the result is

shown in Fig. 6. We can get the same conclusion as the small range. Besides, when the initial number of test cases is large, such as 200 or 300, the fault detection rate is larger than that of 50 or 100. Because there are more redundant test cases in the initial population of the former.

It can be seen from Fig. 3 and Fig. 4 that the reduction rate gradually increases as a increases, that is, a varies in direct proportion to the reduction rate. Experimental results in Fig. 5 and Fig. 6 show that the fault detection rate decreases from slowly to rapidly as the reduction rate increases. In each data range, for different initial test case sizes, it can be seen that when the fault detection rate decreases slowly with the change of reduction rate, there is a suitable range for a. That is to say, for each curve in Fig. 5 and Fig. 6, within the range where the fault detection rate decreases slowly with the reduction rate increases, an appropriate a could be selected from this range. It can also be seen from Fig. 5 and Fig. 6 that for the same program, the appropriate range of a is not the same as the reduction rate varies under different conditions.

In large industrial projects, performing all tests after each change is expensive and time-consuming. To get feedback on these changed code more rapidly, it is crucial to reduce the time [8]. When determining the "best" a, besides reduction rate and the fault detection rate, the run time of algorithms should also be considered. According to the experimental results in Fig. 3, Fig. 4, Fig. 5 and Fig. 6, we can see that there is indeed an appropriate range of a, which can ensure that the fault detection rate does not decrease much when the reduction rate increases. Therefore, we can choose an appropriate a from this range to balance reduction rate, fault detection rate and run time. As the algorithm efficiency is very important for RTS, besides fault detection rate and reduction rate, run time of the proposed method is also considered to determine α in the experiment. The initial conditions of the five methods in the paper are the same. After obtaining the experimental results that can be achieved by the other four methods, the value of α in this paper is gradually changed by experiments so that the selected α can balance the reduction rate, fault detection rate and the run time. The experimental results verify the RQ2.

(3) Path coverage

As is seen in Table 5, in addition to replace (C11) and grep (C12), all target paths in other programs are covered by our method. That is to say, the path coverage for most of the programs is 100% except the replace and grep. Compared with other methods, the values of DRTS are higher, and it answers RQ3. Even for replace and grep, the values of path coverage of DRTS are better than other methods. That's because the optimization goal of our method includes path coverage, good coverage can be achieved by selected test cases. Besides, we can see from Table 5 that the values of the two FOS methods are higher than SCRTS. There is a small difference in path coverage between SFOS and BFOS.

(4) Coverage balance

It can be seen from Table 8 that the test cases selected by our method can achieve better coverage balance than other methods except for print_token (C9). That is, test cases selected by DRTS can more evenly cover the branches of programs, so the RQ4 is verified. As in the evolution of Genetic Algorithm, the coverage balance is considered for selecting test cases to cover different program branches. Besides, SCRTS and SFOS have several overlapping values for some programs. In addition to programs of the overlapping values and tot_info (C4), test cases selected by SFOS can cover program branches more evenly than SCRTS.

Table 8 Comparison of coverage balance and fault detection rate

ID	Coverage balance %					Fault detection rate %				
	DRTS	SCRTS	BFOS	SFOS	Random	DRTS	SCRTS	BFOS	SFOS	Random
C1	**35.4**	37.8	36.9	37.4	56.7	**100**	**100**	**100**	**100**	**100**
C2	**23.2**	51.5	46.5	40.4	87.8	**100**	86.4	**100**	**100**	70.6
C3	**26.7**	32.5	33.4	32.5	46.8	**100**	100	**100**	**100**	74.4
C4	**51.4**	53.8	64.1	60.5	60.2	**86.4**	64.5	81.3	84.4	62.5
C5	**30.7**	62.8	40.8	36.4	47.2	**91.5**	70.2	82.7	86.7	59.4
C6	**41.2**	65	52.9	46.2	71.5	**100**	96.7	96.7	97.3	85.3
C7	**43.7**	56	55.8	51.7	60.2	**100**	95.7	**100**	**100**	82.5
C8	**32.8**	38.4	36.4	38.4	64.7	**100**	**100**	96.4	**100**	56.4
C9	44.5	44.9	**41.8**	43.5	66.7	94.5	**97.4**	90.7	91.5	67.9
C10	**37.7**	42.5	43.2	41.3	59.4	**100**	**100**	**100**	**100**	79.3
C11	**48.8**	55.8	57.6	54.7	62.3	**86.7**	77.4	**86.7**	84.8	62.6
C12	**49.5**	64.5	62.7	64.5	74.9	**92.6**	81.4	82.5	**92.6**	66.9

(5) Fault detection rate

In this group of experiments, the same test case reduction rate is used to verify the fault detection ability of the selected test cases by different methods. From the results of fault detection rate in Table 8, the fault detection rate of SCRTS for print_token (C9) is higher than that of our method. In other cases, our method shows a higher fault detection rate than the other four methods, and the RQ5 is verified. That is because the selected cases by our method can cover the program branches in a balanced way before the algorithm terminates. Besides, the two FOS methods perform better than SCRTS and the Random method on the whole. The statement coverage-based SFOS shows a small improvement on branch coverage-based BFOS.

(6) $F-measure$

From the perspective of the comprehensive factor F-measure, compared with other methods, the $F-measure$ of our method is more stable, as is seen in Fig. 7. Though there are several overlaps between DRTS and BFOS, DRTS shows a small improvement on BFOS for other programs. The values of DRTS are higher than other methods for multiple programs. Besides, combined with fault detection rate in Table 8, it indicates that the precision and recall of our method are better while ensuring the fault detection rate, and the RQ6 is verified.

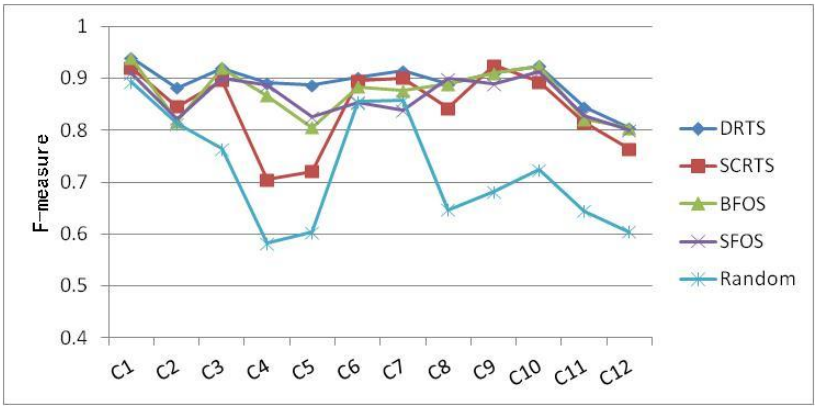
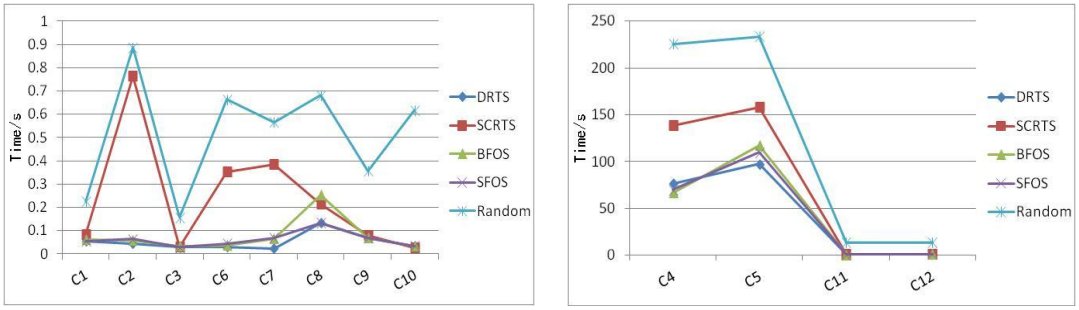


Fig. 7 Comparison of $F-measure$

(7) Run time

From the run time in Fig. 8, for some programs that are less affected by the coverage balance, the efficiency of our method is not obvious, and the run time may even be slightly more than some

of the comparison methods. For example, for the program tot_info (C4), the time of our algorithm is slightly more than BFOS and SFOS, but much less than the SCRTS and the Random methods. For most programs, our method ensures that the termination condition of Genetic Algorithm is reached as soon as possible. This trend is more obvious for complex programs, such as the case of program flex (C5). That is because when the termination conditions of several algorithms are the same, BFOS and SFOS need to obtain an ordered sequence of program entities, then test cases are selected and prioritized based on the farthest-first ordered sequence. SCRTS uses clustering technology. When there are more test cases, it takes a long time. The Random method requires more generations of GA when the required coverage is not reached. Therefore, we can conclude that the efficiency of our method is higher, and the stability of the run time is better than other methods, and the RQ7 is verified.



(a) (b)

Fig. 8 Comparison of run time

References

[1] R. Simone, S. Giuseppe, and A. Giuliano, M. Alessandro. SPIRITuS: A Simple Information Retrieval regression Test Selection approach. *Information and Software Technology*. 2018, 99: 62-80. ISSN 0950-584. doi: 10.1016/j.infsof.2018.03.004.

[2] Y. Zhang. Theories and Methods of Evolutionary Generation of Test Data for Path Coverage[D]. *China University of Mining and Technology*. Xu Zhou. 2012, 25-121.

[3] H. Xia, X. Song, L. Wang. Research of test case auto-generating based on Z path coverage. *Modem Electronics Technique*. 2006, 29(6) :92-94. doi:10.3969/j.issn.1004-373X.2006.06.039.

[4] X. M. Cheng, Q. H. Yang, and T. L. Zhai, et al. Test Case Selection Technique Based on Semi-supervised Clustering Method. *Computer Science*. 2018, 45(1): 249-254. doi: 10.11896/ j.issn.1002-137X.2018.01.044.

[5] C. R. Fang, Z. Y. Chen, K. Wu, and Z. H. Zhao. Similarity-based test case prioritization using ordered sequences of program entities. *Software Qual J*. 2014, 22: 335–361. ISSN 0963-9314. doi: 10.1007/s11219-013-9224-0.

[6] X. Y. Dang, D. W. Gong, X. J. Yao. Feasible Path Generation of Weak Mutation Testing Based on Statistical Analysis. *Chinese Journal of Computers*. 2016, 39(11): 2355-2371. ISSN 0254-4164. doi: 10.11897/SP.J.1016.2016.02355.

[7] G. J. Zhang, D. W. Gong, X. J. Yao. Test Case Generation Based on Mutation Analysis and Set Evolution. *Chinese Journal of Computers*. 2015, 38(11): 2318-2331. ISSN 0254-4164. doi: 10.11897/SP.J.1016.2015.02318.

[8] V. Blondeau, A. Etien, N. Anquetil, S. Cresson, P. Croisy, and S. Ducasse. Test case selection in industry: an analysis of issues related to static approaches. *Software Qual J*. 2017, 25, 1203–1237. ISSN 0963-9314. 10.1007/s11219-016-9328-4.