Improved Evolutionary Generation of Test Data for Multiple Paths in Search-based Software Testing

Ziming Zhu and Xiong Xu
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
University of Chinese Academy of Sciences
Beijing, China
Email: {zhuzm,xux}@ios.ac.cn

Li Jiao
State Key Laboratory of Computer Science
Institute of Software, Chinese Academy of Sciences
Beijing, China
Email: ljiao@ios.ac.cn

***Abstract*—Search-based software testing has achieved great attention recently, but the efficiency is still the bottleneck of it. This paper focuses on improving the efficiency of generating test data for multiple paths. Genetic algorithms are chosen as the heuristic algorithms in search-based software testing in this paper. First, we propose an improved grouping strategy of target paths to balance the load of each calculation resource. This work makes a contribution to the parallel execution in search-based software testing. Then, common constraints of the target paths in the same group are collected to reduce the search space of test data. Symbolic execution technique is used in this phase. Based on the reduced search space, we can accelerate the convergence of search process and improve the efficiency of search-based software testing. Finally, our method is applied to some study cases to compare with other methods.**

***Keywords*—Search-based software testing; multiple paths; genetic algorithms; symbolic execution**

I. Introduction

Software testing is an important method to guarantee the software quality [1]. In order to detect the faults in the software, the best way is to generate test data to cover each executable path so as to make sure that every executable path is in safe. Sometimes, we only need to cover several target paths which are particularly concerned by us in practice.

There are many methods of generating test data which are classified as static and dynamic ones. Static methods do not need to execute the software while dynamic methods need to execute the software under test when generating test data [2].

Search-based software testing (SBST) is one of dynamic methods which converts the software testing into an optimizing problem to generate the test data by using heuristic algorithms. Among these meta-heuristic algorithms, genetic algorithms are the most widely used [3], [4]. The first use of genetic algorithms for generation of test data was in [5]. As to the branch coverage criterion, genetic algorithms were employed to search for test data for covering the target path in [6], [7]. The influences of the genetic parameters in SBST such as mutation probability, crossover probability, and fitness were further investigated in [8]. Except for the branch coverage criterion, condition-decision coverage criterion is also an important measurement in software testing. SBST was employed in this area in [9] in 2001. With the development of SBST, different tools for SBST in popular languages were proposed recently. AUSTIN is an open source SBST tool for C language [10] and Evosuite was proposed for Java [11]. In addition, SBST has been applied to industrial application recent years [12].

Previous SBST methods can cover only one target path during one search process. However, real-world software always contains multiple target paths. It is obvious that generating test data covering multiple paths in one search process can improve the efficiency of SBST. Researches about the multi-objective evolutionary optimization used in software testing have been proposed recent years [13], [14], [15], [16], [17], [18], [19]. However, the multi-objective evolutionary algorithms cannot handle well when the number of objectives increases, see [14], [15]. When a large number of target paths have to be covered in software, it will become a complicated multi-objective optimization problem. To this end, [16] converted the complicated optimization problem into several simpler optimization sub-problems by grouping the target paths.

Since [16] does not take the parallel execution into consideration, the big difference of the number of target paths in each group makes the execution in parallel inefficient. In 2012, [17] improved the method by balancing the number of target paths in each group. Due to the similar number of target paths in each group, SBST can make sure that each calculation resource is in balance. However, [17] only paid attention to the number of target paths in each group. The difficulty of the search process of each group was overlooked. Generally, the more similar the target paths are, the more easily we can generate the corresponding test data. Thus, there are some deficiencies in the method of grouping target paths in [17]. Moreover, since the grouping strategy is based on the similarity between target paths, the common constraints of the target paths in each group can be utilized to cut down the search space of the test data during the search process.

We propose an approach that can improve the efficiency of the evolutionary generation of test data for multiple paths by reducing the search space of the test data. The approach uses symbolic execution tools to collect the common constraints of each group to restrict the initial population and their offspring to the range of the given area.

978-1-5090-4601-0/17/$31.00

We improve the existing grouping strategies according to the difficulty of the search process of each group. More target paths are assigned in the higher similarity group instead of balancing the number of target paths in each group equally.

The rest of the paper is organized as follows. The next section introduces the basic knowledge of SBST with genetic algorithms. The improved grouping strategy is introduced in Section III. Section IV elaborates our approach that uses the common constraints of each group to improve the efficiency of the search process. The case studies are presented in Section V and Section VI concludes the paper.

II. Search-based Software Testing with Genetic Algorithms

In this section, we will introduce how genetic algorithms work in SBST.

A. Genetic Algorithms

SBST uses meta-heuristic algorithms to automate the generation of test data that meet a test adequacy criterion. Genetic algorithms are the most widely used meta-heuristic algorithms in SBST [3], [4].

Genetic algorithms learn from the nature that animals which are not adapted to the environment are eliminated and others survive. So, each test data in the search space currently under consideration is referred to as an "individual". The set of individuals currently under consideration is collectively referred to as the current "population". The main loop of a genetic algorithm can be seen in Fig. 1.

First, we choose several individuals to compose the initial population randomly. Each individual is evaluated by the fitness function which guides the search to promising areas of the search space. The fitness function is problem-specific and needs to be defined for different problems.

Then, individuals are selected to generate offspring by the crossover operator. In the selection phase, the higher fitness value the individuals have, the more probability the individuals will survive. During crossover, elements of each individual are recombined to form two offspring individuals that embody characteristics of their parents.

Subsequently, elements of the newly-created chromosomes are mutated at random, with the aim of diversifying the search into new areas of the search space.

Finally, the next generation of the population is chosen in the "reinsertion" phase, and the new individuals are evaluated for fitness. This cycle continues, until the genetic algorithms find a solution or the resources allocated to the search are exhausted. The detailed description of genetic algorithms can be found in [20].

The fitness function used in this paper is the standard fitness function which is computed by combining the branch distance and approach level. The branch distance is normalized to a value between [0,1].

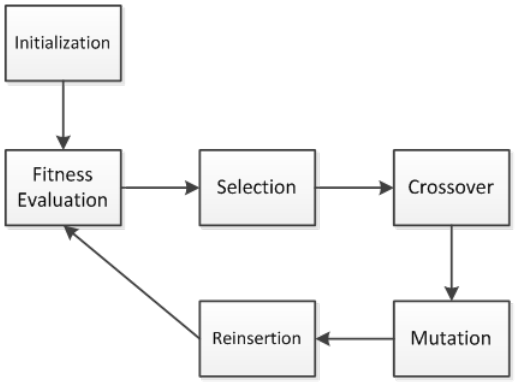

Fig. 1. Overview of the main steps of a genetic algorithm

B. Search-based Software Testing

SBST converts the problem of test data generation for many paths into a multi-objective optimization problem.

We denote the program under test as P, the number of target paths is m, $m \in \mathbb{N}^+$ and the m target paths are represented as $p_1, p_2, \cdots, p_m$. For every program under test, we first get the control flow graph (CFG). A CFG G consists of a finite set N of nodes and a finite set E of edges. An edge e_{ij} connects the nodes n_i and n_j. The nodes in the CFG represent two types of structures. One is basic block which consists of a sequence of consecutive statements, and the other is the condition of a select statement. The path of the program is a sequence of consecutive nodes $p = (n_1, n_2, \cdots, n_k)$ or edges $p = (e_1, e_2, \cdots, e_k)$. So, we denote the nodes which the path p_i traverses as $n_{i1}, n_{i2}, \cdots, n_{i|p_i|}$, where $|p_i|$ is the path length of p_i.

To convert the generation of test data into a multi-objective optimization problem, we regard the m target paths as m objectives in the multi-objective optimization problem. In order to generate the test data to cover target paths, every objective needs a fitness function for evaluating the fitness of the test data. We denote $f_i(x)$ as the fitness function for the target path p_i, where x represents the input data of the program. When the value of the fitness function $f_i(x)$ equals zero, the test data x will traverse the target path p_i. Then, the problem of generating the test data to cover p_i can be converted into searching the optimal solution of minimizing $f_i(x)$.

So, we can formulate the multi-objective optimization problem as:

$$\min\{f_1(x), f_2(x), \cdots, f_m(x)\}.$$

As we can see, if we generate a test data that covers one of the m target paths, the corresponding objective can be eliminated. Then, we can transform the m-objective optimization problem into an $(m-1)$-objective optimization problem. So, with the search proceeding, more and more target paths will be covered, the number of objectives will be smaller and smaller so that the multi-objective optimization problem will become simpler and transform into a one-objective optimization problem in the end.

The process of the generation of test data for covering multiple paths is as above. The key of the search process is the meta-heuristic algorithms. Here, we choose the most widely used algorithms: genetic algorithms in this paper.

In the previous sub-section, we introduced the genetic algorithms briefly. So, in the SBST, we regard all the test data as the individuals in genetic algorithms. In the initialization phase, we choose several test data randomly from the search space. The search space is the value range of the test data. Then, each test data will be evaluated by the fitness function. The selection, crossover and mutation action will be operated according to the fitness value of each individual. Finally, we will generate the optimal solution to cover the target paths with genetic algorithms.

III. The Improved Grouping Strategy

As we can see, the generation of test data for covering multiple paths problem can be converted into multi-objective optimization problem. However, real-world software programs usually have a large number of target paths. When the number of target paths increases, the number of objectives will increase as well. This will cause the optimization problem to be too complex to be solved. To this end, some grouping strategies are proposed to divide target paths into different groups. As a result, the complicated multi-objective problem is converted into several simpler multi-objective sub-problems. However, traditional grouping strategies have its disadvantages. In this section, we point out the drawback of them and proposed our improved grouping strategy.

A. Traditional Grouping Strategy

In general, similar paths usually have similar inputs. We will explain it in Example 1. TABEL I is the source code of Example 1 and Fig. 2 shows its corresponding CFG. The nodes n_0 and n_{10} are the starting and ending nodes in Fig. 2.

Now, we choose four executable paths in the example: $p_1 = (n_0, n_1, n_2, n_{10}), p_2 = (n_0, n_1, n_3, n_5, n_6, n_{10}), p_3 = (n_0, n_1, n_3, n_5, n_7, n_8, n_{10}), p_4 = (n_0, n_1, n_3, n_5, n_7, n_9, n_{10})$ and regard the path p_4 as the basic path. Intuitively, p_3 is the most similar path to p_4 and p_1 is the least similar to it. We generate a test data for each path randomly. The test data $t_1 = (a = 12, b = 5, c = 7, d = 11), t_2 = (a = 1, b = 2, c = 12, d = 8), t_3 = (a = 0, b = 2, c = 3, d = 9)$ and $t_4 = (a = 2, b = 4, c = 4, d = 1)$ are the test data traverse the target path p_1, p_2, p_3 and p_4, respectively. As we can see, the test data for p_3 and p_4 must satisfy the condition that $\{a \leqslant 5 \wedge b \leqslant 5 \wedge c \leqslant 5\}$ while the test data for p_2 and p_4 only need to satisfy the condition that $\{a \leqslant 5 \wedge b \leqslant 5\}$. As for the test data for p_1 and p_4, no common constraint needs to be satisfied in this example. So, the test data for p_3 are the most similar to test data for p_4 while the test data for p_1 are the least similar to them. We can conclude that generally, the more similar two paths are, the more similar two corresponding inputs are. When two paths are similar, a little change in the test data which traverse one path may help us to get the other path's test data.

TABLE I
THE SOURCE CODE OF EXAMPLE 1

```
int exam(int a, int b, int c, int d){
1  if a > 5
2    {cout<<"Target 1";}
3  else if b > 5
4    {cout<<"Target 2";}
5  else if c > 5
6    {count<<"Target 3";}
7  else if d > 5
8    {cout<<"Target 4";}
9  else {count<<"Target 5";}
}
```

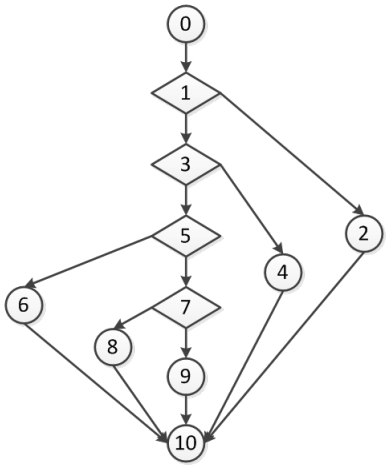

Fig. 2. The CFG of Example 1

Based on the conclusion above, traditional grouping strategies group the target paths according to the similarity in order to improve the efficiency of generation of test data.

For the program under test P, the paths p_i and p_j are denoted as $p_i = (n_{i1}, n_{i2}, \cdots, n_{i|p_i|})$ and $p_j = (n_{j1}, n_{j2}, \cdots, n_{j|p_j|})$, where $|p_i|$ and $|p_j|$ are the path length of p_i and p_j. The nodes $n_{i1}, n_{i2}, \cdots, n_{i|p_i|}$ and $n_{j1}, n_{j2}, \cdots, n_{j|p_j|}$ are the nodes that paths p_i and p_j traverse, respectively. Compare p_j and p_i from their first node. If the nodes are the same, then compare the next until the nodes are different. Find the number of the successively same nodes from the beginning and denote it as $|p_i \cap p_j|$. The similarity between p_i and p_j is defined as the ratio of the number of successively same nodes to the larger path length, and denoted as $s(p_i, p_j)$, whose expression is as follow:

$$s(p_i, p_j) = \frac{|p_i \cap p_j|}{\max\{|p_i|, |p_j|\}}.$$

According to the similarity between each pair of paths, we can construct a similarity matrix S of target paths as follow:

$$S = \begin{pmatrix} s(p_1, p_1) & s(p_1, p_2) & \cdots & s(p_1, p_m) \\ s(p_2, p_1) & s(p_2, p_2) & \cdots & s(p_2, p_m) \\ \vdots & \vdots & \ddots & \vdots \\ s(p_m, p_1) & s(p_m, p_2) & \cdots & s(p_m, p_m) \end{pmatrix},$$

where m is the number of the target paths.

The sum of all elements in the ith row of the matrix above is calculated to reflect the similarity between the path p_i denoted as s_i and the others. To maximize the similarity in the same group between target paths. We choose the biggest s_i's corresponding target path as the basic path of the first group g_1 denoted as g_{b1}. Then, the similarities between g_{b1} and all paths (including g_{b1}) are sorted in a descending order, and the target paths corresponding to the first $|g_1|$ similarities are selected to form g_1 where $|g_1|$ represents the number of target paths in g_1.

With the help of grouping the target paths, the efficiency of the search process is improved. However, with the development of parallel processing, the efficiency of this problem can be improved further by parallel execution. Each multi-objective sub-problem can be solved by different calculation resources separately. Due to the similarity-based grouping strategy, there are always huge differences between different groups. This has caused load imbalance problem in parallel execution. Researchers improved the grouping strategy by balancing the number of target paths in each group [17]. However, the difficulty of generating test data in different groups was ignored. Load imbalance problem still exists in some cases.

B. Our Grouping Strategy

We take the difficulty of generating test data in different groups into consideration and improve the grouping strategy further to balance the load of different calculation resources.

Our grouping strategy is also based on the similarity of target paths. The difference is that we assign more target paths in the early generated groups instead of balancing the number of paths in all groups.

From the conclusion above, we can see that the more similar two paths are, the more similar two corresponding inputs are. When two paths are similar, a little change in the test data which traverse one path may help us to get the other path's test data. That is to say, similar paths' test data have some common information. If we get the test data of one of the similar paths, we also get the common information of the similar paths. With this common information, test data of other similar paths can be obtained more easily. So, we can assume that the more similar the target paths in the same group are, the more easily we can generate the test data for the target paths in this group. According to the grouping strategy in [17], the basic path of the early generated groups is much more similar to other target paths compared to the basic path of the groups generated later. Besides, the early generated groups have the priority to choose the more similar paths to form themselves so that the groups generated later have less paths options. Moreover, the last group has no chance to choose the similar paths, all the remains form it. So, the target paths in the later generated groups usually share few similarities which makes the difficulty of generating the test data for the target paths increase. From the examples in [17], we can see this phenomenon.

TABLE II
THE SOURCE CODE OF THE `INSERT SORT` PROGRAM

```
void insert(int a[6]){
  int i, j;
  int temp;
1 for(i = 1; i < 6; i++){
2   temp = a[i];
    j = i-1;
3   while(j >= 0 && temp < a[j]){
4     a[j+1] = a[j];
      j = j-1;
    }
5   a[j+1] = temp;
  }
}
```

The source code of the `insert sort` program in [17] is provided, as shown in TABEL II, where 1, 2, 3, 4 and 5 are nodes in its CFG. Groups obtained by the grouping strategy in [17] are shown in the Fig. 3. According to Fig. 3, we can see that the number of the four groups is nearly the same ($|g_1| = 10, |g_2| = 10, |g_3| = 10, |g_4| = 9$) due to the grouping method proposed in [17]. Let us consider the similarity between the target paths in these 4 groups. The common prefix-path of the target paths in g_1 is $\langle 1, 2, 4, 3, 5, 1, 2, 4, 3\rangle$, and the prefix-paths of the target paths in the other 3 groups (g_2, g_3, g_4) are $\langle 1, 2, 5, 1, 2, 4, 3\rangle$, $\langle 1, 2, 4, 3, 5, 1, 2\rangle$ and $\langle 1, 2, 5, 1, 2\rangle$, respectively.

As we assumed before, the length of the common prefix-path of the target paths in the early generated groups is longer than that of the groups generated later, as well as the similarity between the target paths in them.

Then, let us see the experimental results of the six programs in Fig. 4. Ltc(s) means the longest time spent among all calculation resources. Td(s) is the difference between the longest and the shortest time consumption. We pay attention to the value of Td(s) in the `make` program labeled by a red circle. In general, the method proposed in [17] in 2012 performs better than it proposed in [16] in 2011. However, the results are just the opposite in the red circle. The reason is that the difficulty of generating test data for target paths is different, but the number of target paths in each group is almost the same in the method in [17].

In the worst case, if the paths in the last group are totally different from each other, the test data of one of the paths of them will not help in generating test data for other paths.

The difficulty of generating test data for target paths in the early generated groups is relatively less than it of the later generated groups. So, we should assign relatively more target paths to the early generated groups instead of balancing the number of paths in all groups. The specific difference between the numbers of target paths in each group should depend on specific programs and specific similarity in each group. Generally, we can first assign a constant to be the difference between the numbers of target paths in each group. Based on the results of our experiments, the biggest number of the paths in the group should not be more than two times of the smallest one. In this circumstance, the results are relatively good.

Group	Paths
g_1	**1,2,4,3,5,1,2,4,3,5,1,2,5,1,2,4,3,4,3,5,1,2,4,3,5,1**; 1,2,4,3,5,1,2,4,3,5,1,2,5,1,2,4,3,4,3,5,1,2,4,3,4,3,5,1; 1,2,4,3,5,1,2,4,3,5,1,2,5,1,2,4,3,4,3,4,3,5,1,2,4,3,4,3,5,1; 1,2,4,3,5,1,2,4,3,5,1,2,4,3,4,3,5,1,2,4,3,4,3,5,1,2,4,3,5,1; 1,2,4,3,5,1,2,4,3,5,1,2,4,3,5,1,2,4,3,5,1,2,4,3,4,3,4,3,4,3,4,3,5,1; 1,2,4,3,5,1,2,4,3,4,3,4,3,5,1,2,5,1,2,4,3,4,3,5,1,2,5,1,2,4,3,4,3,5,1; 1,2,4,3,5,1,2,4,3,4,3,5,1,2,5,1,2,4,3,4,3,4,3,5,1,2,4,3,4,3,4,3,4,3,4,3,5,1; 1,2,4,3,5,1,2,4,3,4,3,5,1,2,4,3,5,1,2,4,3,4,3,4,3,5,1,2,4,3,4,3,4,3,5,1; 1,2,4,3,5,1,2,4,3,4,3,5,1,2,4,3,4,3,4,3,5,1,2,4,3,5,1,2,4,3,5,1; 1,2,4,3,5,1,2,4,3,4,3,5,1,2,4,3,4,3,4,3,5,1,2,4,3,5,1,2,4,3,4,3,4,3,4,3,5,1.
g_2	**1,2,5,1,2,4,3,4,3,5,1,2,4,3,4,3,5,1,2,4,3,5,1,2,5,1**; 1,2,5,1,2,4,3,4,3,5,1,2,4,3,4,3,5,1,2,4,3,5,1,2,4,3,4,3,4,3,4,3,4,3,5,1; 1,2,5,1,2,4,3,4,3,5,1,2,4,3,4,3,5,1,2,4,3,4,3,5,1,2,5,1; 1,2,5,1,2,4,3,4,3,5,1,2,4,3,4,3,4,3,5,1,2,4,3,4,3,4,3,4,3,4,3,4,3,5,1; 1,2,5,1,2,4,3,4,3,5,1,2,4,3,4,3,4,3,5,1,2,5,1,2,5,1; 1,2,5,1,2,4,3,4,3,5,1,2,5,1,2,4,3,4,3,4,3,5,1,2,4,3,4,3,4,3,4,3,4,3,5,1; 1,2,5,1,2,4,3,4,3,5,1,2,5,1,2,4,3,4,3,5,1,2,4,3,5,1; 1,2,5,1,2,4,3,4,3,5,1,2,5,1,2,4,3,4,3,5,1,2,4,3,4,3,4,3,4,3,5,1; 1,2,5,1,2,4,3,5,1,2,4,3,5,1,2,4,3,4,3,4,3,5,1,2,5,1; 1,2,5,1,2,4,3,5,1,2,4,3,4,3,4,3,5,1,2,4,3,5,1,2,4,3,4,3,4,3,5,1.
g_3	**1,2,4,3,5,1,2,5,1,2,4,3,4,3,4,3,5,1,2,4,3,4,3,4,3,5,1,2,5,1**; 1,2,4,3,5,1,2,5,1,2,4,3,4,3,4,3,5,1,2,4,3,4,3,5,1,2,4,3,4,3,4,3,4,3,5,1; 1,2,4,3,5,1,2,5,1,2,4,3,4,3,4,3,5,1,2,5,1,2,4,3,4,3,5,1; 1,2,4,3,5,1,2,5,1,2,4,3,4,3,5,1,2,4,3,4,3,4,3,5,1,2,4,3,4,3,4,3,5,1; 1,2,4,3,5,1,2,5,1,2,4,3,5,1,2,4,3,4,3,4,3,5,1,2,4,3,4,3,5,1; 1,2,4,3,5,1,2,5,1,2,5,1,2,4,3,5,1,2,4,3,4,3,4,3,4,3,5,1; 1,2,4,3,5,1,2,5,1,2,5,1,2,4,3,5,1,2,4,3,4,3,5,1; 1,2,4,3,5,1,2,5,1,2,5,1,2,4,3,4,3,5,1,2,4,3,4,3,5,1; 1,2,4,3,5,1,2,4,3,4,3,5,1,2,4,3,4,3,4,3,5,1,2,4,3,4,3,4,3,5,1,2,5,1; 1,2,4,3,5,1,2,4,3,4,3,5,1,2,4,3,4,3,4,3,5,1,2,5,1,2,4,3,4,3,5,1.
g_4	**1,2,5,1,2,5,1,2,4,3,4,3,5,1,2,4,3,4,3,4,3,5,1,2,4,3,4,3,4,3,4,3,5,1**; 1,2,5,1,2,5,1,2,4,3,4,3,5,1,2,4,3,4,3,4,3,4,3,5,1,2,4,3,4,3,4,3,4,3,4,3,5,1; 1,2,5,1,2,5,1,2,4,3,4,3,5,1,2,4,3,4,3,5,1,2,4,3,4,3,4,3,4,3,5,1; 1,2,5,1,2,5,1,2,4,3,4,3,5,1,2,4,3,4,3,5,1,2,4,3,4,3,5,1; 1,2,5,1,2,5,1,2,4,3,4,3,5,1,2,4,3,5,1,2,4,3,4,3,5,1; 1,2,5,1,2,5,1,2,4,3,5,1,2,4,3,4,3,5,1,2,4,3,4,3,4,3,4,3,5,1; 1,2,5,1,2,5,1,2,5,1,2,4,3,5,1,2,4,3,4,3,5,1; 1,2,5,1,2,4,3,5,1,2,5,1,2,4,3,4,3,4,3,5,1; 1,2,5,1,2,4,3,5,1,2,4,3,5,1,2,4,3,4,3,4,3,4,3,4,3,5,1; 1,2,5,1,2,4,3,5,1,2,4,3,4,3,4,3,5,1,2,4,3,4,3,5,1,2,5,1.

Fig. 3. Groups obtained by methods in [17]

	Ltc(s)		Td(s)	
	Avg.	Vac.	Avg.	Vac.
flex				
Method in Gong et al. (2011)	2.027	0.002	1.055	0.902
Proposed method	2.019	0.002	1.049	0.928
t	1		0.031	
make				
Method in Gong et al. (2011)	1.503	0.005	0.790	0.471
Proposed method	1.338	0.010	1.150	0.067
t	9.705		3.495	
grep				
Method in Gong et al. (2011)	0.522	0.031	0.521	0.031
Proposed method	0.411	0.031	0.365	0.029
t	3.171		4.588	
sed				
Method in Gong et al. (2011)	0.055	0.0003	0.053	0.0003
Proposed method	0.042	0.0003	0.031	0.0001
t	4.333		11	
space				
Method in Gong et al. (2011)	3.685	0.080	3.685	0.080
Proposed method	3.365	0.014	1.747	2.403
t	7.441		8.729	
bash				
Method in Gong et al. (2011)	0.748	0.005	0.748	0.005
Proposed method	0.598	0.014	0.597	0.014
t	7.894		7.947	

Fig. 4. Value of indicators for industrial programs in [17]

An ideal difference between the number of paths in different groups will let all the calculation resources finish their work in almost the same time. The differences between the number of paths in different groups may not be the same. But the cost to get the ideal value is higher because various factors can influence the results. We will continue our work to research the optimal group size in the future.

IV. Constraint-based Search for Test Data Generation

SBST suffers a lot from the large search space of test data. So, we want to reduce the state space of test data by adding constraints to the search process to improve the efficiency of SBST.

A. Integrate the Symbolic Execution Technique

The performance of SBST depends on the size of search space, the diversity of the initial populations, and the effectiveness of the fitness functions. It is inefficient to use evolutionary testing in a large search space without a diversified population or without sufficient guidance. In this paper, we focus on the size of search space. We hope to add some constraints to reduce the size of the search space of test data. Symbolic execution technique is integrated with SBST in our paper. The search space is a subset of the original one after adding the constraints to it. The search process will be definitely more efficient on the constrained search space.

Symbolic execution is another important software testing method and has shown great promise in automatically generating test cases that achieve high code coverage [21], [22], [23]. Symbolic execution uses the symbolic instead of concrete inputs to execute the programs and maintains a path condition (PC). The PC is a path condition represented by a union of constraints that the target path's corresponding test data must satisfy. After symbolic execution, all the executable path will get the corresponding PC. Test data generation is performed by solving the collected constraints of PC, using an off-the-shelf decision procedure or constraint solver. So, if the test data traverse the specific path, they must satisfy the PC of the path.

Since we group the target paths according to the similarity, and the similarity is defined as the ratio of the number of successively same nodes to the larger path length, all the target paths in the same group share the same prefix-path. The PC of the prefix-path is the common constraints that all these target path's corresponding test data must satisfy. Adding these constraints to the search of the multiple paths in the same group will cut down the size of the search space and improve the efficiency of the search process.

The general process of the improved generation of test data for multiple paths proposed in this paper is as follows:

(1) Group the target paths according to similarity. The grouping strategy was described in Section III.
(2) Find the common prefix-path of each group generated in Step 1.

(3) Collect the PC of the prefix-path of each group by the off-the-shelf symbolic execution tools.
(4) Add the constraints to the search space of the test data to reduce the size of it.

B. Relax the Constraints

However, when we utilize the symbolic execution technique to improve the efficiency of our method, we also face the challenges in symbolic execution. Constraint solving continues to be one of the key bottlenecks in symbolic execution, where it often dominates runtime. We must figure out that the main purpose of using symbolic execution technique is to improve the efficiency of the search process in SBST. Expending too much time in constraint solving will make us lose more than gain. So, when the PC of the prefix-paths contains complex constraints or external function call that the constraint solver cannot handle or need to cost a lot of time, we should get a relaxed version of the PC by ignoring the difficult part of them.

The following rules help us to generate the relaxed version of PC:

- Any complex constraint that will cost much time will be relaxed: a new variable will replace the constraints.
- For PC which contains loops, we can set a constant k to limit the number of the loop iterations. That is to say we force all the loops to stop at most after k iterations. Then, the PC we get is the relaxed version of the original PC.

Since the relaxed version of PC relaxes the constraints of the original PC, this operation will not break the condition that the corresponding test data satisfy.

The search space constrained by the unrelaxed constraints is a subset of that constrained by the relaxed ones. The search space constrained by the relaxed constraints is a subset of the original one. The search process on the smallest size of the search space constrained by the unrelaxed constraints can be definitely the fastest. But we should face the problem of how to solve all the consuming complex constraints. So, with the relaxed version of PC, we can improve the efficiency of the search process without facing the challenges of symbolic execution.

V. Case Study

In order to validate the proposed method in this paper, we apply our method to four kinds of sort programs.

We improve the method to generate test data for multiple paths by improving the grouping strategy and employing symbolic execution technique. Separate results on the effect of using 1) the group size balancing, 2) the common constraint extraction, and 3) the combined overall approach, may be expected to be seen. However, the improved grouping strategy can help to balance the load of each calculation resource while symbolic execution technique can improve the efficiency of the search process. These two methods used in our paper improve the traditional method in two aspects. It may not be necessary to do the extra experiments. Due to the space limitation and the reason we mentioned above, we only test the combined overall approach in this paper to compare with other methods.

TABLE III
The average number of iterations required to cover all the target paths in each group

Programs	**Method in This Paper**			
	g_1	g_2	g_3	g_4
Bubble Sort	9.3	21.4	100(92.4%)	100(81.3%)
Insert Sort	8.6	17.2	38.2	59.1
Select Sort	19.0	100(82.8%)	52.3	65.2
Shell Sort	22.1	50.6	100(89.3%)	100(69.8%)
Programs	**Method in [17]**			
	g_1	g_2	g_3	g_4
Bubble Sort	19.7	49.9	100(60.2%)	100(49.4%)
Insert Sort	18.9	100(90.3%)	100(80.0%)	100(39.5%)
Select Sort	30.1	100(57.6%)	100(74.2%)	100(26.1%)
Shell Sort	28.1	100(89.8%)	100(70.4%)	100(30.2%)

Among the methods for generating test data for multiple paths in SBST, the method in [17] performs best to the best of our knowledge. So, we focus on comparing the performance of our method with that of the method in [17] in this section.

These four kinds of sort programs are `bubble sort` program, `insert sort` program, `select sort` program and `shell sort` program. The reason for choosing these four programs is that the test data are arrays, so we can easily get the inherent information of different test data without encoding the input data. Among these four programs, the `insert sort` program was also chosen to verify the performance of different methods of generating test data in [17]. Due to the different experimental environment from [17], we will execute the method proposed by [17] in our experimental environment again in order to compare the performance fairly. The hardware configuration in our experiment is as follows: 2*Intel i5 3230M CPU, 8GB memory and the operating system is 64-bit Windows 7.

As to the parameters in genetic algorithms, the population size is set to 20, the genetic operations are roulette-wheel selection, one-point crossover, and one-point mutation with their probabilities of 0.9 and 0.3, respectively. Besides, the maximum of the number of generations is set to 100 and the input values (the value of each element in the array) were constrained in [0,1000]. In order to compare the load balancing in different calculation resources, the number of calculation resources is set to four in this paper (This number is the same as that in the case study in [17]). That is to say, all the target paths have to be divided in to four groups in our experiments.

We randomly choose 22 paths as target paths for each four programs. Then, the grouping method in Section III is employed to group the target paths. Since the number of target paths is 22 and the number of groups is 4, we assign the difference value between the numbers of target paths in each group to one to make sure that the number of paths in g_1 is not more than two times of the number of paths in g_4. According to the method proposed in our paper, target paths are divided into 4 groups, g_1, g_2, g_3, g_4, which contains 7, 6, 5, 4 paths, respectively. Based on the method proposed in

[17], target paths in each program are divided into 4 groups, g_1, g_2, g_3, g_4, which contains 6, 5, 6, 5 paths, respectively.

After grouping the target paths, the next step is to find the common prefix-path of each group. Because the programs are small and the prefix-paths are short, we can get the common constraints manually. When the industrial programs are tested, program instrumentation can be used to collect the traces of the target paths and find the common prefix-path of each group. Then, the genetic algorithms are employed on the constrained search space with the constraints collected before.

We execute each method 100 times in every programs and compute the average value of the results to draw the graphs as follows. The experimental results for the four programs are shown in Fig. 5, Fig. 6, Fig. 7 and Fig. 8, respectively.

We also collect the information of the experimental results in TABEL III. It is worth noting that the values in TABLE III represent the average number of iterations required to cover all the target paths in each group in different methods. However, there are some values with percent signs in it. These values with percent signs indicate that the method cannot cover all the target paths in this group in 100 iterations (The maximum number of iterations is set to 100 in our experiment). The value of them in parentheses represent the average of the rate of coverage in the 100th iteration. Apparently, when neither of the methods can cover all the target paths in one hundred iterations, the method with larger value performs better.

As shown in Fig. 5, Fig. 6, Fig. 7 and Fig. 8, the results of our proposed method are on the right side, while the graphs on the left side show the rate of the coverage by using the method in [17]. In most cases, our method can get the value of the rate of coverage up to 100% in 100 iterations, while the method in [17] can only cover all the target paths in no more than half of the cases. As we can see in Fig. 5, Fig. 6, Fig. 7 and Fig. 8, the rate of the coverage in each group with our method grows much faster than that with the method in [17]. According to TABLE III, most of the number of iterations required to cover all the target paths in each group with our method is smaller than that with the method in [17]. For example, to cover all the target paths in group g_1, the average iterations required by our proposed method are 9.3, 8.6, 19.0 and 22.1 in the four programs, respectively. When using the method in [17], the corresponding number of iterations are 19.7, 18.9, 30.1 and 28.1, respectively. When neither of the methods can achieve 100% of the coverage in 100 iterations, most of the values of the coverage achieved by our method are much more than that achieved by the method in [17].

These results show that our method outperforms the method in [17] in most cases. We have improved the efficiency of SBST for covering multiple paths obviously by cutting down the search space of genetic algorithms.

Except for the efficiency, we focus on the load balancing between different calculation resources for each group. As we can see in Fig. 5, Fig. 6, Fig. 7 and Fig. 8, in both of the two methods, the rate of coverage of g_1 grows faster than that of the other 3 groups while g_4's value grows the slowest in most situations (An abnormal situation is shown in the `select sort` in Fig. 7, the reason is that the test data for covering the target paths in g_2 are few in the search space. This problem will be mentioned in the next section in our future work). This phenomenon indicates the assumption we proposed in Section III.

The difficulty of generating test data for target paths in the early generated groups is relatively less than that of the later generated groups. So, in our improved grouping strategy, g_1 has the most target paths while the number of target paths in g_4 is the smallest. With the improved grouping strategy in our method, the difference between the numbers of iterations of four groups is smaller than that with the method in [17]. Let's take the results of `insert sort` program as an example. From the information in TABLE III, the largest number of iterations required to cover all the target paths in the same group with our method is 59.1 while the smallest is 8.6. The difference between them is 50.5. However, the smallest number of that with the method in [17] is 18.9 and the other three groups cannot cover all the target paths within 100 iterations. From the tendency of the curve of the other three groups, a large number of additional iterations are needed to cover all the target paths with the method in [17]. So, the difference between the numbers of iterations required to cover all the target paths with the method in [17] is at least 91.1, much more than that of our method. Similar conditions are shown in other three programs in Fig. 5, Fig. 7 and Fig. 8.

It is obvious that our method can balance the load of different calculation resources better than the other method.

From all the results above, we can get the conclusions as follows:

- Our method in this paper can improve the efficiency of SBST for generating test data for multiple paths.
- Our method in this paper can balance the load of different calculation resources better.

VI. CONCLUSION AND FUTURE WORK

With the development of SBST, generating test data for target paths by heuristic algorithms has become more and more important. Generating a set of test data for multiple paths in one search process is meaningful in practice.

In this paper, we propose a novel method to improve the efficiency of generating a set of test data for multiple paths in SBST. Furthermore, we improve the traditional grouping strategy to balance the load of different calculation resources. Symbolic execution technique is utilized for collecting the constraints to reduce the search space of test data. In order to avoid the challenges in symbolic execution, the relaxed version of constraints are proposed to optimize the method. The results of case studies show good performance of our method in both efficiency and load balancing.

For future work, we will apply our method to more programs and seek the way to optimize the group sizes to improve our method. Besides, we will improve the method further by taking the inherent difficulty of covering the target paths into consideration. The inherent difficulty is not equal to the difficulty we mentioned before in this paper. The paths

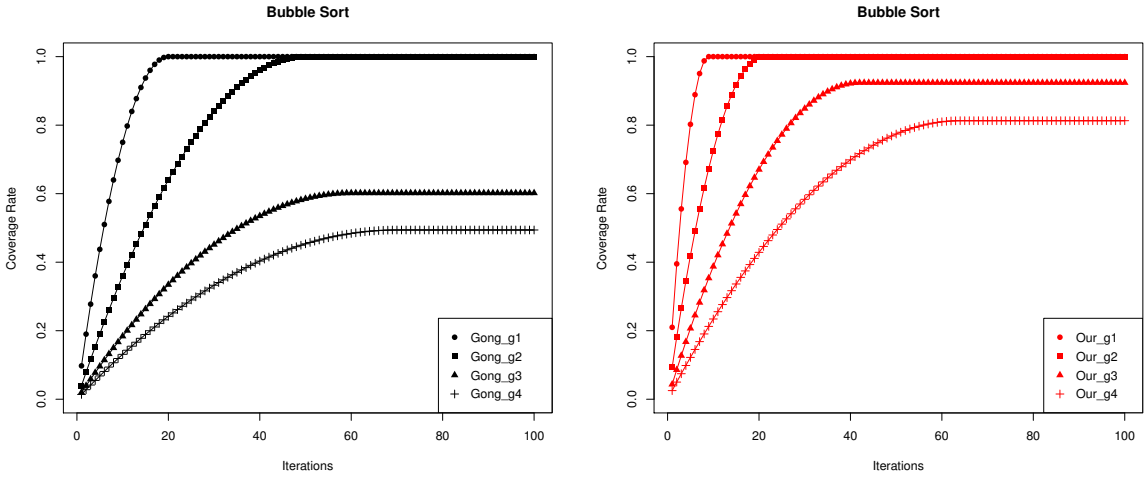

Fig. 5. The average coverage rate against the number of iterations in the `bubble sort` programs

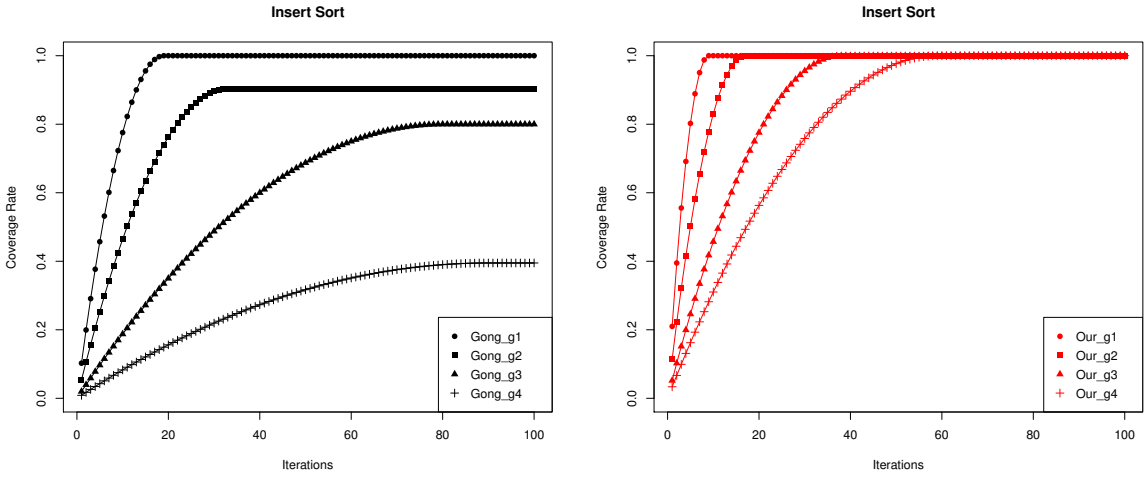

Fig. 6. The average coverage rate against the number of iterations in the `insert sort` programs

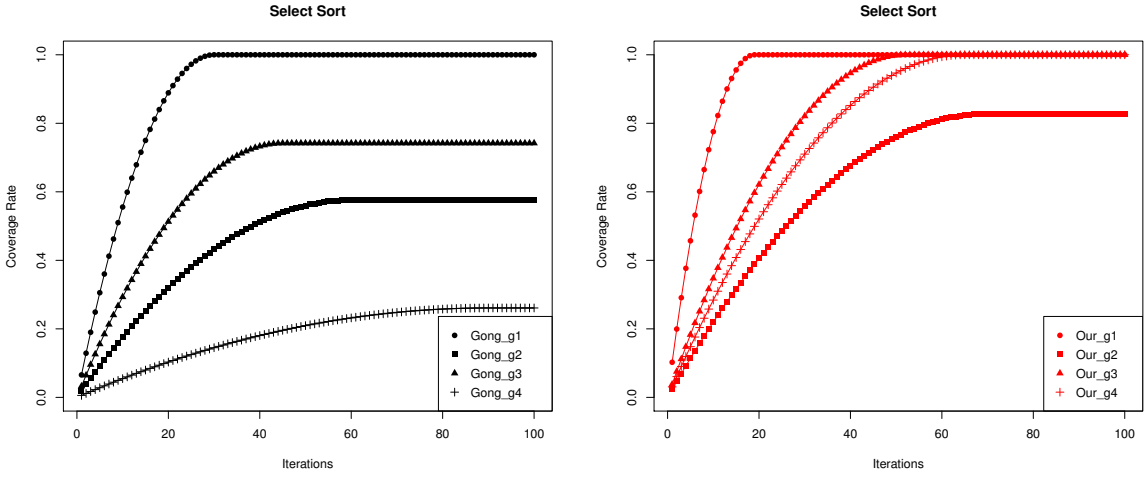

Fig. 7. The average coverage rate against the number of iterations in the `select sort` programs

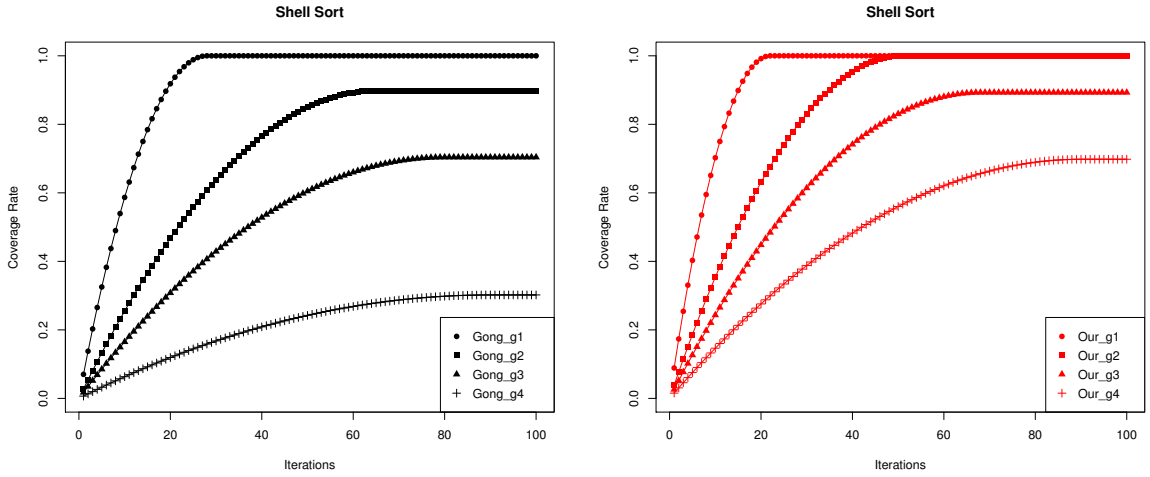

Fig. 8. The average coverage rate against the number of iterations in the `shell sort` programs

with much inherent difficulty refers to the paths that can be covered by few test data. The inherent difficulty is also an important factor which affects the load balancing. More work about the symbolic execution is also needed in the future. The relation between the proportion and/or the number of common constraints and the execution time of the symbolic execution needed in our method should be researched further. We will also focus on some specific encoding methods to encode the ordinary test data so that more inherent information can be shown in SBST and our method can work much better in more areas.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their comments and suggestions. This work is partially supported by the National Science Foundation of China (NSFC) under grant No. 91418206.

REFERENCES

[1] G. J. Myers,"The art of software testing," 1979.
[2] J. H. Shan, J. Wang, and Q. I. Zhi-Chang,"Survey on path-wise automatic generation of test data," Acta Electronica Sinica, 2004, 32(1), pp. 109-113.
[3] A. Watkins, E. M. Hufnagel, "Evolutionary test data generation: a comparison of fitness functions," Software Practice and Experience, 2006, 36(1), pp. 95-116.
[4] J. Miller, M. Reformat, and H. Zhang, "Automatic test data generation using Genetic algorithm and program dependence graphs," Information and Software Technology, 2006, 48(7), pp. 586-605.
[5] S. Xanthakis, C. Ellis, C. Skourlas, A. Le Gall, S. Katsikas, and K. Karapoulios, "Application of genetic algorithms to software testing," In Proceedings of the 5th International Conference on Software Engineering and Applications, 1992, pp. 625-636.
[6] B. F. Jones, H. Sthamer, and D. E. Eyres, "Automatic structural testing using genetic algorithms," Software Engineering Journal, 1996, 11(5), pp. 299-306.
[7] A. Watkins, "The automatic generation of test data using genetic algorithms," in Proceedings of the 4th Software Quality Conference, 1995, vol. 2, pp. 300–309.
[8] H. Sthamer, "The automatic generation of software test data using genetic algorithms," University of Glamorgan, 1996.
[9] C. C. Michael, G. Mcgraw, and M. A. Schatz, "Generating software test data by evolution," IEEE Transactions on Software Engineering, 2001, 27(12), pp. 1085-1110.
[10] K. Lakhotia, M. Harman, and H. Gross, "AUSTIN: An open source tool for search based software testing of C programs," Information and Software Technology, 2013, 55(1), pp. 112–125.
[11] G. Fraser, A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," SIGSOFT/FSE'11, ACM Sigsoft Symposium on the Foundations of Software Engineering, 2011, pp. 416-419.
[12] N. Tillmann, J. De Halleux, T. Xie, "Transferring an automated test generation tool to practice: from pex to fakes and code digger" Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ACM, 2014, pp. 385-396.
[13] M. A. Ahmed, I. Hermadi, "GA-based multiple paths test data generator," Computers and Operations Research, 2008, 35(10), pp. 3107-3124.
[14] Y. Cao, C. H. Hu, S. B. Chen, L. M. Li, "Automatic test data generation for multiple paths and its applications," Computer Engineering and Applications, 2010, 46(27), pp. 32-35.
[15] M. Harman, Y. Jia, Y. Zhang, "Achievements, open problems and challenges for search based software testing," IEEE International Conference on Software Testing, Verification and Validation, 2015, pp. 1-12.
[16] D. Gong, W. Zhang, and Zhang, "Evolutionary generation of test data for multiple paths coverage," Chinese Journal of Electronics, 2011, 19(2), pp. 233-237.
[17] D. Gong, T. Tian, and X. Yao, "Grouping target paths for evolutionary generation of test data in parallel," Journal of Systems and Software, 2012, 85(11), pp. 2531-2540.
[18] R. M. Hierons, M. Li, X. Liu, S. Segura, W. Zhang, "SIP: optimal product selection from feature models using many-objective evolutionary optimization," ACM Transactions on Software Engineering & Methodology, 2016, 25(2):17.
[19] W. Zheng, R. M. Hierons, M. Li, X. H. Liu, V. Vinciotti, "Multi-objective optimisation for regression testing," Information Sciences, 2016, 334, pp. 1-16.
[20] P. Mcminn, "Search-based software testing: past, present and future," in IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, 2011, pp. 153-163.
[21] L. A. Clarke, "A program testing system," ACM'76 Proceedings of the 1976 Annual Conference, 1976, pp. 488-491.
[22] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezze, "Using symbolic execution for verifying safety-critical systems," ACM Sigsoft Software Engineering Notes, 2001, 26(5), pp. 142-151.
[23] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," Tools and Algorithms for the Construction and Analysis of Systems, Springer Berlin Heidelberg, 2003, pp. 553-568.