ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.20, No.12, December 2009, pp.3117−3136
doi: 10.3724/SP.J.1001.2009.00580

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

面向路径覆盖的演化测试用例生成技术*

谢晓园[1,2], 徐宝文[1,2,3+], 史 亮[4], 聂长海[2,3]

[1](东南大学 计算机科学与工程学院,江苏 南京 210096)

[2](计算机软件新技术国家重点实验室(南京大学),江苏 南京 210093)

[3](南京大学 计算机科学与技术系,江苏 南京 210093)

[4](微软中国研发集团,北京 100190)

Genetic Test Case Generation for Path-oriented Testing

XIE Xiao-Yuan[1,2], XU Bao-Wen[1,2,3+], SHI Liang[4], NIE Chang-Hai[2,3]

[1](School of Computer Science and Engineering, Southeast University, Nanjing 210096, China)

[2](State Key Laboratory for Novel Software Technology at Nanjing University, Nanjing 210093, China)

[3](Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China)

[4](Microsoft (China) Company Ltd, Beijing 100190, China)

+ Corresponding author: E-mail: bwxu@seu.edu.cn

Xie XY, Xu BW, Shi L, Nie CH. Genetic test case generation for path-oriented testing. *Journal of Software*, 2009,20(12):3117−3136. http://www.jos.org.cn/1000-9825/580.htm

Abstract: Nowadays many researches have focused on structural ET based on statement and branch coverage and there are few researches on path-oriented ET. To solve this problem, this paper provokes an approach to construct the fitness function for test case generation in path-oriented ET based on the similarity evaluation techniques. First, a basic model for fitness function design is provided. The core of the model is to evaluate the similarity between the execution track and the target path. Accordingly three different algorithms for the similarity evaluation are provided. This model can automatically generate fitness function for each target path. The empirical studies present the superiority of the approach over several other path-oriented testing techniques, especially for the complex paths. Besides, the limitation and the applicable scope of the approach are pointed out.

Key words: software testing; evolutionary testing; path-oriented testing; fitness function design; similarity evaluation

摘 要: 为了解决目前结构性演化测试主要集中于面向语句、分支等覆盖标准,缺乏面向路径覆盖标准的问题,提出了基于相似性度量的适应值函数构造方法,以用于生成覆盖指定路径的测试用例.首先给出适应值函数构造基本模型,即利用测试数据的真实执行轨迹来评估它相对于指定路径的适应值.该模型的核心在于度量执行轨迹与指定路径之间的相似度,为此给出了 3 种不同的相似度度量算法.该模型可以完全自动化地为每一条目标路径构造出特

* Supported by the National Natural Science Foundation of China under Grant Nos.90818027, 60633010 (国家自然科学基金); the National High-Tech Research and Development Plan of China under Grant No.2009AA01Z147 (国家高技术研究发展计划(863)); the National Grand Fundamental Research 973 Program of China under Grant No.2009CB320703 (国家重点基础研究发展规划(973))

Received 2008-06-11; Revised 2008-11-27; Accepted 2009-02-24

定的适应值函数.实验结果表明,相对于其他路径测试方法,该方法在针对复杂路径的情况下具有一定的优势.此外,实验结果还指出了该方法的适用性范围和局限所在.

关键词: 软件测试;演化测试;路径覆盖测试;适应值函数设计;相似性度量

中图法分类号: TP311 文献标识码: A

1 Introduction

Evolutionary Testing (ET) is a promising technique for automatic test case generation. ET uses a kind of meta-heuristic search techniques, Genetic Algorithm (GA), to convert the task of test case generation into an optimization problem[1,2]. It can be utilized in many fields, such as structural testing, functional testing, safety testing, etc[3−8]. In structural ET, each candidate solution in the population could be one single conceivable test case. And the global optima are test cases that satisfy the test target, such as the execution of certain statements or a specified path of interest[9−11]. Nowadays many researches have focused on structural ET based on statement, branch and condition coverage[12]. However, there is a gap on ET based on path-oriented testing.

Path-Oriented testing is an important structural testing technique, aiming at covering the specified target paths or node sequences according to some coverage criteria. As one of the most important complementary criteria of statement or branch coverage, it can be more capable in fault detecting[13]. Nowadays the commonly used techniques in path-oriented testing include manual test case generation, non-heuristic automatic techniques, static analyzing, dynamic methods, etc. However, they all have disadvantages, such as being expensive, inefficient, etc[14−18].

Fortunately ET can be applied in path-oriented testing. Since ET can search for test cases to cover the desired paths intelligently and automatically without any manual work, the process of test case generation becomes quite efficient. Besides, as a meta-heuristic search technique, GA is powerful in the complicated search spaces, which makes ET especially suitable for the coverage of complex paths[19,20].

Therefore, we apply our approach in path-oriented testing using ET, which can generate test cases to cover specified paths efficiently. Commonly the fitness function is a numeric representation of the test target and it should be designed according to the particular situation, in order to provide a proper guidance for the evolutionary search. Thus the fitness function design methods for statement, branch or condition coverage may no longer be suitable for path-oriented ET. In this paper, our approach is used to construct the fitness function for path-oriented ET, based on the similarity evaluation between the execution track and the target path or node sequence. Accordingly we list three different algorithms for the similarity evaluation. The empirical studies present the superiority of our approach over several other path-oriented testing techniques, especially for the complex paths with huge search domain, and we point out the limitation and the applicable scope of this approach. Correspondingly we also put forward several conclusions and some advice for the adaptability of our approach in different situations.

The rest of the paper is organized as follows. Section 2 briefly introduces the structural ET and points out the lack of proper fitness function design methods for path-oriented ET. Section 3 introduces the basic idea of our approach in fitness function design for path-oriented ET. Section 4 provides three algorithms for similarity evaluation. Empirical results and analysis are illustrated to show the advantages and the limitations of our approach in Section 5, by comparing with other commonly used path-oriented testing techniques. Finally in Section 6, we present the conclusion of this paper.

2 Structural Evolutionary Testing

Structural ET is an automatic test case generation technique for the prescribed structural coverage criteria. Up

to now, most researches in structural ET have concentrated on the statement, branch and condition coverage. Depending on the construction of the fitness function, previous work can be divided into three categories: the condition-oriented approaches, the full-coverage-oriented approaches, and the hybrid approaches[21,22].

In the condition-oriented approaches, the test target is divided into partial aims. ET generates test cases to execute all the partial aims one by one, in order to achieve the full coverage. The fitness function computes a static distance for each individual that indicates how far it is away from executing a partial aim in the desired way[2,5,9,11,22]. In the full-coverage-oriented approaches, ET treats all the program structures as a whole target. It tries to generate a test suite with a full coverage directly. Test cases pass through the unexecuted or rarely covered statements. Or branches are assigned with higher fitness values[3,22−25]. And in the hybrid approaches, Wegener, *et al*. combine the above ideas by using both the static distance information and the dynamic coverage information[22].

Some of the approaches above may be adjusted to fit the path coverage criterion, while others cannot be transformed since their basic ideas are not suitable for path-oriented testing.

For condition-oriented approaches, we can apply them in path-oriented testing by extending them to the PC-oriented approach. PC-oriented approach utilizes the path condition (PC) instead of each single branch predication to build the fitness function, following the same rules as condition-oriented approaches[26]. However, this extended approach has several limitations. For example, it is very complicated and expensive to apply PC-oriented approach in the long and complex paths with loops, arrays, sub-procedure calling, etc, because to build a fitness function with such an approach it is usually necessary to combine all the branch predications together. Even though this combination does not strictly require containing only input parameters as the independent variables in many situations, it is strongly recommended to represent the temporal variables with the input parameters in order to avoid the “flag variable problem”[26,27]. Thus, the assistance of data-flow analysis will be needed to investigate the dependency among variables. Consequently, the complexity and the cost could increase due to the features of the loops, arrays, sub-procedure calling, etc.

As for full-coverage-oriented approaches, it is also very difficult to adjust them to fit the path-oriented testing, since its fitness function may easily construct a coarse fitness landscape. For most individuals with low fitness values, it cannot tell which one has higher possibility to execute the unexecuted or rarely executed paths. Consequently, it cannot provide a strong guidance to direct the evolutionary search from low-fitness regions to high-fitness regions. Once encountering some quite complex paths, it can hardly attain a full coverage.

Finally for hybrid approaches, even though Wegener, *et al*. proposed a basic idea to apply these approaches in path-oriented testing, they did not give the concrete formula of fitness function[22]. Actually the idea of combining the static and the dynamic information is good and it may be useful for the statement or the branch coverage. However, it is still not suitable for the path coverage criterion. By collecting all the branch predications in the target path, these approaches may encounter the same problem in PC-oriented approach, that is, the limitations from data-flow analysis. Besides, it may be even harder to balance the weight between multiple predications and the approaching levels, thus harder to construct a fitness function.

Therefore, in this paper we propose a new approach in details for fitness function design in path-oriented ET. This approach adopts the idea of “dividing and conquering”, and alleviates the confusion between the static and the dynamic information by using the dynamic one only. In fact the empirical studies show that with a specific target path, fitness function constructed with our approach is powerful enough to guide ET generating test cases efficiently.

3 Fitness Function Design for Path-Oriented ET

Unlike the previous three approaches, we adopt the basic idea of dividing and conquering from condition-oriented approaches and construct individual fitness function for each target path. Actually the cost of constructing fitness functions for all the target paths in our approach is much lower than the one for all the branching conditions in condition-oriented approaches. Because the conditions in a program usually have various forms, constructing fitness function for each special condition needs a particular algorithm. Thus it could make this task labor-consuming and hard to be automated. Differently our approach for path-oriented testing does not need the particular algorithm for each individual path. It constructs the fitness functions for all the target paths with a general form, which takes the test case and the target path as parameters. Thus, it can be conducted simply and automatically without much cost.

In our approach, we use the distance between a test case and the target to represent the fitness values of the test case. The key issue to evaluate the distance is to determine the similarity between the execution track of the associated test case and the target path. Suppose the target path is a statement sequence like this: $target=\langle p_0,p_1,\ldots,p_{n-1}\rangle$, in which p_i $(0\le i\le n-1)$ denotes the statement or branching condition to be executed. And the execution track of a test case is also a statement sequence like this: $track=\langle t_0,t_1,\ldots,t_{m-1}\rangle$, in which t_i $(0\le i\le m-1)$ stands for the covered statement or condition. The formula for evaluating the distance between a test case and the target is defined as follows:

$$Distance(test,target)=Length(target)-Similarity(track,target) \quad (1)$$

in which, *Length* returns the amount of the statements in target path. In our approach it returns n. And *Similarity* is used to evaluate the similarity between the corresponding execution track and the target path. Its return value is within the range $[0,n]$. The higher value it returns, the more similar is the track to the target path. If the track is different from the target path completely, *Similarity* returns 0. Conversely, if the track matches the target path exactly, *Similarity* returns n.

Consequently the function of *Distance* also has its return value within $[0,n]$. It will return a low value when the similarity between the execution track and the target path is high, which means the associated test case is close to the test target. If its return value is 0, the associated test case is the desired global optimum and can satisfy the target. Oppositely if its return value is n, the corresponding test case cannot satisfy the target at all.

In Eq.(1) the function of *Distance* is minimized during the optimization and can only provide a linear discrimination among a population of test cases. To normalize this function and provide a higher level of differentiation, the result of *Distance* can act as the input of other algebra formulas, such as reciprocal function, Gaussian function, Hamming distance function, and so on Ref.[11], whose output is the final fitness value.

As presented above, the key issue of our approach is constructing the function of *Similarity*, that is, designing algorithms to evaluate the similarity between the execution track of the associated test case and the target path. Therefore in the next section we will provoke three different similarity evaluation algorithms. Each algorithm accepts both *track* and *target* as parameters, and can serve as the function of *Similarity*.

4 Algorithms for Similarity Function

In this section, we propose three algorithms for function *Similarity* design in order to evaluate the similarity between the execution track of the associated test case and the target path.

4.1 Similarity function based on path matching

Algorithm MUNS (matching using uninterrupted node sequence) is a method based on path matching in

similarity evaluation between the execution track and the target path. It requires covering the statements in the target path one by one without any interruption. Generally MUNS tries to find a sub-string of the execution track that can exactly match the longest prefix of the target path, and then returns the length of the prefix as the similarity between the track and the target path. Thus only if part of the execution track could match the whole target path exactly, the similarity between them will be evaluated as the maximum n, and the associated test case will be chosen as the desired solution by ET. The detail of this algorithm is shown in Algorithm 1.

Algorithm 1. MUNS.

input *track*: the execution track $\langle t_0,t_1,\ldots,t_{m-1}\rangle$,
target: the target path $\langle p_0,p_1,\ldots,p_{n-1}\rangle$;
output *similarity*: the similarity between *track* and *target*.
declare *length*: the length of the current prefix of *target*
t: the element of *track*
p: the element of *target*

```
begin
    similarity:=0;
    //first search, if not found, return null
    t_match:=find the first t_i=p_0 in track from t_0;
    while (t_match<>null)
        length:=1;
        t:=next of t_match;
        p:=p_1;
        //evaluate the length of the prefix of target
        //that exactly matches the sub-string of track
        while (t<>null and p<>null)
            if (t=p) then length:=length+1;
                    else break;
            t:=next of t in track;
            p:=next of p in target;
        end while
        if (length>similarity) then similarity:=length;
        //continue searching
        t_match:=find t_i=p_0 in track from the next element of t_match;
    end while
    return similarity;
end MUNS
```

It can be learned from Algorithm 1 that the outer loop searches for the sub-strings of *track* that is started with p_0. And for each sub-string like this, the inner loop finds the prefix of *target* that could exactly match it and evaluates the *length* of the current prefix. At the exit of the outer loop, algorithm returns the maximal *length* as the similarity between *track* and *target*. For instance, *target*=(0,1,2), $track_1$=(0,4,0,1,2), $track_2$=(1,0,0,1,5), then *MUNS* ($track_1$,*target*)=3, *MUNS*($track_2$,*target*)=2.

Suppose the length of *track* is m, the length of *target* is n. From Algorithm 1 we can conclude that in the worst case, the outer loop that searches for t_i matches p_0 in *track* will iterate m times. And the inner loop that investigates the elements in both *track* and *target* will iterate min(m,n) times at most. Therefore, the algorithm MUNS takes $O(m\times\min(m,n))$ time in the worst case.

4.2 Similarity function based on node sequence matching

In some cases, the test target of path-oriented testing is not a consecutive execution trace. Instead, it could be a

sequence of several discrete statements, which just indicates the order of them to be executed[13]. Thus, we only need to generate test cases, which can orderly cover each statement in the target sequence. For example, in the definition-use pair coverage criterion, the target path could be a sequence of definition and using statements. The execution track is required to cover the prescribed definition statement first, and after the execution of several other statements then cover the prescribed using statement.

Aiming at the testing requirement above, we propose the algorithm MDNS (matching using discrete node sequence) to evaluate the similarity between the execution track and the target statement sequence. Different from MUNS, MDNS only requires covering the statements in the target sequence orderly without regarding the interruption.

In MDNS, we need to find the longest prefix of the target sequence, in which the statements could be orderly covered by a sub-string of the execution track. MDNS returns the length of the prefix as the similarity of the execution track and the target sequence. Thus only if part of the execution track could orderly cover all the statements of the target sequence, the similarity between them will be evaluated as the maximum n, and the associated test case will be chosen as the desired solution by ET. Algorithm 2 shows the process of MDNS.

Algorithm 2. MDNS.

input *track*: the execution track $\langle t_0,t_1,\ldots,t_{m-1}\rangle$,
target: the target sequence $\langle p_0,p_1,\ldots,p_{n-1}\rangle$;
output *similarity*: the similarity between *track* and *target*.
declare *length*: the length of the current prefix of *target*
t: the element of *track*
p: the element of *target*

```
begin
    similarity:=0;
    //first search, if not found, return null
    t_match:=find the first t_i=p_0 in track from t_0;
    while (t_match<>null)
        length:=1;
        t:=t_match;
        p:=p_1;
        //evaluate the length of the prefix of target
        //that can be orderly covered by the sub-string of track
        while (t<>null and p<>null)
            t:=find p in track from t;
            if (t<>null) then length:=length+1;
                         else break;
            p:=next of p in target;
        end while
        if (length>similarity) then similarity:=length;
        //continue searching
        t_match:=find t_i=p_0 in track from the next element of t_match;
    end while
    return similarity;
end MDNS
```

It can be learned from Algorithm 2 that the outer loop searches for the sub-strings of *track* that is started with p_0. And for each sub-string like this, the inner loop finds a prefix of *target* that can be covered orderly by the sub-string and evaluates the *length* of the current prefix. In the end, the algorithm will return the maximal *length* as the similarity between *track* and *target*. For instance, *target*=(0,1,2), $track_1$=(4,0,3,1,0,5), $track_2$=(0,0,1,1,2,5), then

$MDNS(track_1,target)=2$, $MDNS(track_2,target)=3$.

Suppose the length of *track* is *m*, and the length of *target* is *n*, similar to the situation in MUNS, in MDNS the outer loop iterates for *m* times at most, and the inner one iterates for min(*m*,*n*) times at most. Thus, the algorithm MDNS also takes $O(m\times\min(m,n))$ time in the worst case.

4.3 Similarity function based on scatter graph method

In this section, we propose an algorithm MSG (matching using scatter graph), which is based on scatter graph technique, to evaluate the similarity between the execution track and the target. Scatter graph was originally applied to detect the DNA sequence in biology research[28]. Recently it has been introduced into software engineering for the identification of similar codes, and has acquired competent effect[29].

In MSG, the statements of the execution track are required to match the statements of target path exactly at the same position. We define the *level* between two strings as the amount of the matched positions, where the two strings have the same elements. Generally, MSG looks for pairs of strings, among which each pair consists of a sub-string of execution track and a sub-string of target path, which match each other at the same position. Thus the *level* for one pair of sub-strings means the amount of the matched position between the two sub-strings. MSG returns the highest *level* among all the pairs of sub-strings as the similarity between the execution track and the target path. Thus only if part of the execution track could match the target path exactly at each position, the similarity between them will be evaluated as the maximum *n*, and the associated test case will be chosen as the desired solution by ET. Algorithm 3 illustrates the corresponding process of algorithm MSG.

Algorithm 3. MSG.

input *track*: the execution track $\langle t_0,t_1,\ldots,t_{m-1}\rangle$,
target: the target path $\langle p_0,p_1,\ldots,p_{n-1}\rangle$;
output *similarity*: the similarity between *track* and *target*.
declare *graph*: the scatter graph
level: the level between two strings
t: the element of *track*
p: the element of *target*

```
begin
    similarity:=0;
    garph:=initialize a scatter graph with track and target;
    //evaluate along the track axis
    t:=t0;
    while (t<>null)
        level:=evaluate level between ⟨p0,…,pn−1⟩ and ⟨t,…,tm−1⟩
        if (level>similarity) then similarity:=level;
        t:=next of t in track;
    end while
    //evaluate along the target axis
    p:=p0;
    while (p<>null)
        level:=evaluate level between ⟨p,…,pn−1⟩ and ⟨t0,…,tm−1⟩
        if (level>similarity) then similarity:=level;
        p:=next of p in target;
    end while
    return similarity;
end MGS
```

From Algorithm 3 we can see that MSG first initializes a scatter graph with *track* and *target*. Then it traverses the whole graph following the diagonal direction, evaluating the *level* for each pair of sub-stings, and finally returns the maximal *level* as the similarity between *track* and *target*.

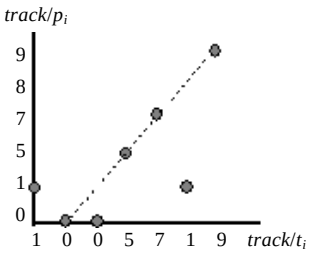


Fig.1 Scatter graph

In MSG, initializing a scatter graph with *track* and *target* means to build a two-dimensional matrix in computer storage, in which the x-axis is assigned with t_i in *track*, while the y-axis is built with p_i in *target*. If the signs of a point in two coordinates are identical, we can mark a solid dot on this point and assign value of 1 to it. For instance, *target*=(0,1,5,7,8,9), *track*=(1,0,0,5,7,1,9), the corresponding scatter graph is shown in Fig.1.

Besides evaluating the *level* between two sub-strings, $sp=\langle p_{start},\ldots,p_{end}\rangle$ and $st=\langle t_{start},\ldots,t_{end}\rangle$, means to count the number of solid points along the direction of diagonal from point (t_{start},p_{start}), until either sub-string first exceeds its ending point. The corresponding process of function EvaluateLevel is shown in Algorithm 4.

For example, for the scatter graph shown in Fig.1, MSG first evaluates the *level* following the diagonal direction along *track* axis, and gets the highest *level* of 4. After that, MSG repeats the process along *target* axis, and gets the highest *level* of 1. Since 4 is the maximum in this case, the *similarity* is finally returned as 4.

Algorithm 4. Function EvaluateLevel.

input *graph*: the initialized scatter graph,
sp: the current sub-string *of target*, $\langle p_{start},\ldots,p_{end}\rangle$,
st: the current sub-string *of track*, $\langle t_{start},\ldots,t_{end}\rangle$;
output *level*: the level between *sp* and *st*.
declare *t*: the element of *st*
p: the element of *sp*

```
begin
    t:=t_start;
    p:=p_start;
    level:=0;
    while (t<>null and p<>null)
        if (graph(p,t)=1) then level:=level+1;
        t:=the next of t in st;
        p:=the next of p in sp;
    end while
    return level;
end EvaluateLevel
```

Suppose the length of *track* is *m*, the length of *target* is *n*, and MSG first builds the scatter graph and takes $O(m\times n)$ time, then it traverses the scatter graph to evaluate the *similarity*, which also takes $O(m\times n)$ time. Thus the algorithm MSG takes $O(m\times n)$ time altogether.

5 Case Study

5.1 Basic experiment settings

In our empirical study, we use three indexes to evaluate the performance of ET, which are P_h, *HAG* and *TTC*. Due to the randomicity of ET, the result of one single experiment cannot represent the performance of ET. Thus we repeat the same experiment independently for many times, and investigate the frequency of the successful experiments. Suppose the total number of experiments is T_0, the size of population is P_{size}, and the maximal

generation of iteration in one experiment is I_{Max}, the three indexes are defined as follows:

- **Hit Percent** (***P***$_h$): the hit percent of ET. $P_h=T_h/T_0$, in which T_h stands for the number of successful experiments that have generated the desired test cases for target paths. If T_0 is large enough, P_h could indicate the probability of success in one experiment approximately. This index is an essential one to evaluate the performance of ET. The higher P_h is, the more effective ET is.
- **Hit Average Generation** (**HAG**): the average generations of ET to obtain the desired test cases, in which H_i means the generation of ET to acquire the desired test cases in the *i*th successful experiment. This index reflects the convergence speed of the evolution. The lower *HAG* is, the faster ET is.

$$HAG=\sum_{i=1}^{T_h} H_i / T_h .$$

- **Total Test Cases** (**TTC**): the approximate average total number of test cases generated in one experiment.

$$TTC=P_{size}\times[P_h\times HAG+(1-P_h)\times I_{\text{Max}}].$$

The lower it is, the lighter ET is.

To evaluate the three indexes of ET, we investigate six typical programs as the testing objectives, which are shown in Table 1. It can be discovered that the objective programs are either unit functions with loop or branching structures or integrated systems. The main reason for choosing these objective programs is due to their variety in structures, which facilitates us to choose either short and simple or deep and complex paths as test targets, hence caters to the requirements of our experimental study.

Table 1 Testing objectives

Programs	Input	Program type	Description
PushDown	*A*[·], *first*, *last*	Unit function with loop and branching structures	Order array *A*[·] within [*first*,*last*] using heap sorting.
BinarySearch	*A*[·], *l*, *target*	Unit function with loop and branching structures	Using binary search to find *target* in *A*[·] whose length is *l*.
SumM	*A*[·], *first*, *last*	Unit function with loop and branching structures	Analyze *A*[·] within [*first*,*last*] to find a consecutive segment, in which the sum of elements is *M*.
NextDay	*month*, *day*, *year*	Integrated system with multi-branch structures	Calculate the next day of the input date with *month*, *day* and *year*.
LineCircle	x_1, y_1, x_2, y_2, *r*, *cx*, *cy*	Integrated system with multi-branch structures	Judge the position relationship between line, which is decided by (x_1,y_1) and (x_2,y_2), and circle, whose center is (*cx*,*cy*) and radius is *r*.
CDPlayer	*A*[·], *first*, *last*	Integrated FSM system with multiple state transitions	A finite state machine used to control the CD-player. The responsive events are stored in array *A*[·] from *first* to *last*.

Additionally, in order to conduct ET for the six testing objective, we need to set up the evolutionary test case generator with several parameters. Certainly we construct the fitness function following our approach described with formula (1) in Section 3, which we adopt the three different similarity evaluation algorithms in Section 4 to implement.

Moreover, we determine the strategy for each step of ET. Generally in ET, there are always varieties of strategies in each step and their combinations form different configurations, which may lead to different performances. In our experiments we select eight common used configurations based on our previous research[30], which are illustrated in Table 2.

In Table 2, we investigate three steps of ET including coding, selection and survival, and for each step we choose two usual strategies. The feature of each configuration, which is not the key issue in this paper, has been studied and discussed in details with the elementary experiments in our previous work[30,31].

Table 2 Different configurations of ET

Config	Coding	Selection	Survival	Config	Coding	Selection	Survival
C_1	Gray	random	unrepeatable	C_5	binary	random	unrepeatable
C_2	Gray	random	fitness	C_6	binary	random	fitness
C_3	Gray	roulette-wheel	unrepeatable	C_7	binary	roulette-wheel	unrepeatable
C_4	Gray	roulette-wheel	fitness	C_8	binary	roulette-wheel	fitness

Finally in our experiments we utilize some benchmark parameters acquired from the previous experiments, which are also not the key point of this paper[30,31]. These parameters include: the mutation possibility P_m (assigned with 0.1), the crossover possibility P_c (assigned with 0.5), the survivability P_s (assigned with 0.5), T_0 (assigned with 300), and I_{Max} (assigned with 100). Besides, for P_{size} we choose different values for different programs. Generally a large population could have a strong ability in searching for the optima and may lead to a high P_h for ET, while a small population may deteriorate the performance of ET. However, if the P_h in all the programs are close to 1 or 0, it will be very hard to compare these results. Therefore in our primary experiments, we found the feasible size for each program to make sure that we could get the comparable results. We assigned P_{size} with 32 for CDPlayer, 16 for LineCircle, 64 for NextDay, and 8 for the other three programs. What is more, for the non-heuristic techniques, the generated test suit is as large as the population in ET.

5.2 Study on the validity and effectiveness

It is known that random testing, as a representation of non-heuristic testing techniques, is widely used in many testing domains. It has several advantages. First, it can be easily applied in many kinds of program structures, without the limitations of loops, arrays, sub-procedure calling, etc. Second, it can be fully automated. However, it also has serious problems. Due to the lack of guidance, the performance of random testing for path-oriented testing could be deteriorated with the increasing complexity of target paths.

On the other hand, path-oriented ET with our fitness function design approach can also be applied in many kinds of program structures without any limitations of loop, array, etc, meanwhile it can achieve the same level of automation as random testing. Moreover, it can conquer the disadvantages of random testing, that is, with a guided search; it could be more efficient for the complicated target paths.

Therefore in this experiment, we will mainly aim at covering the deep and complex paths, in order to present the superiority of our approach. We choose one complex path for each objective program, which is shown in Table3.

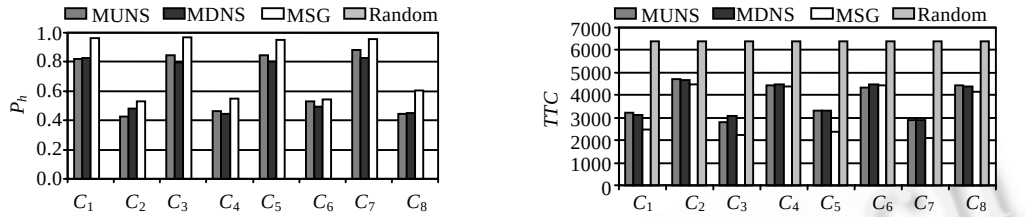
Table 3 Target paths for experiment 1

Programs	Test target
PushDown	Cover a specified path with multiple iterations of loop.
BinarySearch	Cover a specified path with multiple iterations of loop.
SumM	Cover a specified path with multiple iterations of loop.
NextDay	Cover a program path, which can result in the February 28th of a special leap year (the year can be divided by 400).
LineCircle	Cover a program path, which can result in the tangent between circle and line.
CDPlayer	Cover a specified state transition sequence.

The empirical study in this section evaluates the corresponding value of P_h, *HAG* and *TTC* for path-oriented ET using three similarity evaluation algorithms MUNS, MDNS, MSG and random testing to compare their performances. In the experiment, we have acquired a great deal of results. Due to the space, we present part of them from three aspects:

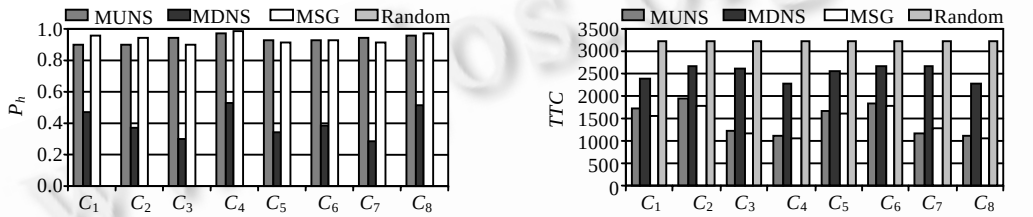
1. For each program, we compare the performance of path-oriented ET using the three algorithms with random testing. From analyzing the empirical results we can educe the following two conclusions:

(i) Path-Oriented ET using the three similarity evaluation algorithms acquires satisfied performance and presents a strong superiority over random testing in most testing objectives. The performance comparisons among the three algorithms and random testing of NextDay and CDPlayer are illustrated in Figs.2 and 3 respectively.



(a) P_h comparison (b) *TTC* comparison

Fig.2 Empirical results in NextDay



(a) P_h comparison (b) *TTC* comparison

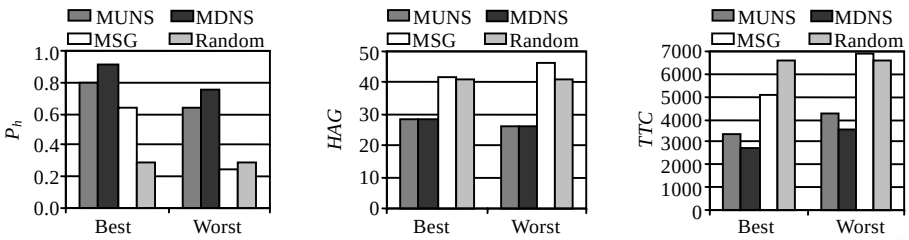
Fig.3 Empirical results in CDPlayer

It is obvious in Fig.2(a) and Fig.3(a) that for all of the eight configurations ET with the three algorithms can obtain satisfied P_h, while random testing can only obtain the P_h of 0. Thus there is no need to illustrate the *HAG* comparison since its value of random testing must be 0. Besides, even though the three algorithms can gain high P_h, the cost of generation is not very high, which can be seen from Fig.2(b) and Fig.3(b).

Actually in NextDay, ET with all the three algorithms can obtain the P_h around 0.8 in one half of the configurations, and around 0.5 in another half. In the best case, which uses C1 and MSG, ET can obtain the P_h of 0.963 with the generation of 2 454 test cases in total, while random testing generates 6 400 test cases without finding any desired ones. Similarly in CDPlayer, ET using C_4 and MSG can acquire the P_h of 0.987 with the generation of 1 053 test cases in all, while random testing cannot even generate one desired test case with the *TTC* of 3 200.

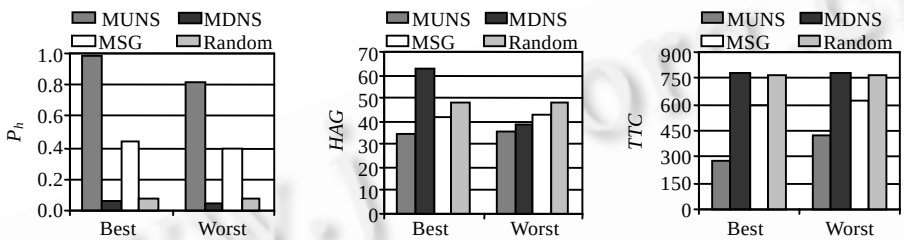
Apart from NextDay and CDPlayer, we have also acquired similar results in PushDown, SumM and LineCircle. Figures 4, 5 and 6 illustrate part of the performance comparison between ET with three similarity evaluation algorithms and random testing in the three object programs. For each program, we pick up the best and the worst situation of ET respectively, in order to display its advantage over random testing.

It can be discovered from Fig.4, Fig.5 and Fig.6 that though different algorithms have different performances, ET can show its superiority in most situations. It is obvious that for the three programs, ET with MUNS can always obtain much higher P_h, lower *HAG* and *TTC* than random testing even in the worst case. Besides, though MDNS and MSG do not perform as well as MUNS, they also show the advantage in most cases: MDNS does not perform well in SumM and LineCircle, but it can obtain the best performance in PushDown. And MSG can acquire satisfied performance in SumM and LineCircle, even though it performs a little worse than random testing in the worst situation of PushDown.



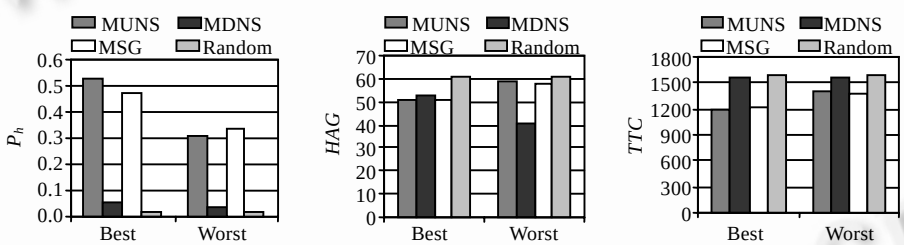
(a) P_h comparison (b) *HAG* comparison (c) *TTC* comparison

Fig.4 Parts of the empirical results in PushDown



(a) P_h comparison (b) *HAG* comparison (c) *TTC* comparison

Fig.5 Parts of the empirical results in SumM



(a) P_h comparison (b) *HAG* comparison (c) *TTC* comparison

Fig.6 Parts of the empirical results in LineCircle

Actually in PushDown, ET using C_8 and MDNS algorithm can obtain the P_h of 0.913 with the generation of only 273 test cases in all, while random testing can only attain the P_h of 0.297, with the cost of generating 659 test cases. Besides using C_8 and MUNS, ET can obtain the P_h of 0.987 with 279 test cases in SumM, and the P_h of 0.527 with 1 187 test cases in LineCircle. However in both of the programs, the P_h of random testing are lower than 0.1.

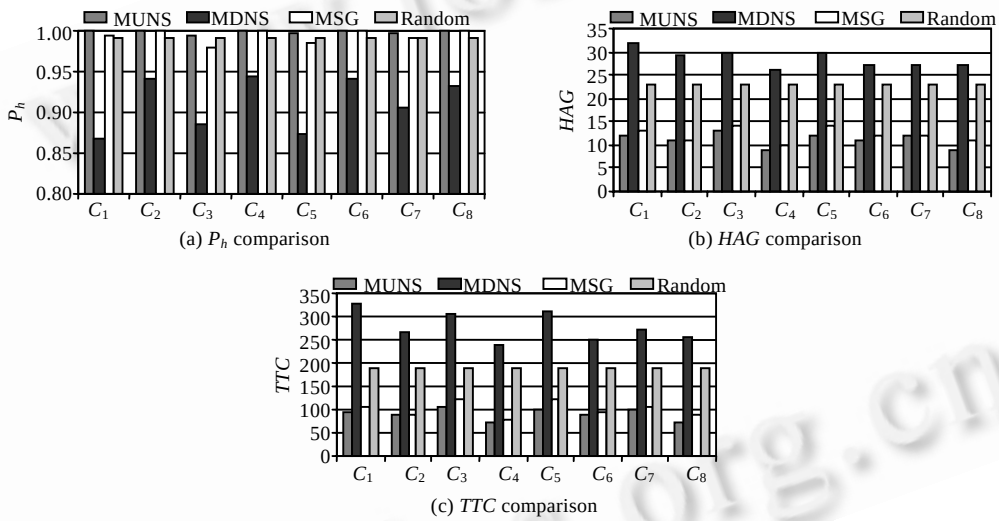
The reason for the superiority of ET with our fitness function design approach can be concluded from the characteristics of the search domain determined by the object programs and target paths. Usually when the search domain is quite complex and huge and the desired test cases are quite sparse in the domain, ET with our fitness function design approach could be more competent than the non-heuristic search, such as random search, for the test case generation. Since our approach can provide a strong guidance for seeking the desired test cases, while random testing can only conduct a blind search without any direction.

Apart from resulting in the complex search domain and causing low efficiency for the non-heuristic techniques, long target paths usually involve the occurrence of loops, arrays, sub-procedure calling, etc., which are roadblocks for most commonly used path-oriented testing techniques, such as static analyzing, dynamic methods, etc. But from our experimental results, we can conclude that ET with our fitness function design approach can easily deal with these situations and also acquire satisfied performance. Additionally since our approach is an automatic technique, it

can be much more effective and practical than the manual methods. In our studies, ET with our fitness function design approach can generate desired test cases within 1 second. However, for the same task it usually takes at least several minutes or more with manual methods, no matter how familiar the test engineer is with the program.

(ii) Even though ET with our fitness function design approach has presented the superiority over random testing in most situations, it does not mean that random testing is useless in all the applications. Our empirical results show that in some situations, random testing can perform as well as our approach. Actually in the situations of simple search domain, large portion of desired test cases, etc., may utilize random testing instead of our approach.

The object program, BinarySearch, is an example of this kind of situations and its empirical results are illustrated in Fig.7. It can be discovered that in this program, for all the eight configurations random testing has acquired a P_h as high as ET with MUNS and MSG, which are higher than 0.95. However, Figs.7(b) and (c) also illustrate that random testing has a higher *HAG* and *TTC* than ET with MUNS and MSG, which indicates that the search for the desired test cases in random testing is slower, thus it needs to generate more test cases in total than the search of ET.



(a) P_h comparison

(b) *HAG* comparison

(c) *TTC* comparison

Fig.7 Empirical results in BinarySearch

In fact, it can be learned from Table 1 that in BinarySearch the target path involves several iterations of loop. To cover the target path only requires a partial-ordered relationship among the object element and several special elements in the array. Obviously there are many test cases satisfying this relationship within the input domain, which makes the search for optima fairly easy. For this kind of programs and target paths, we can use random testing instead of ET with our fitness function. Even though it may be a little slower and may generate more test cases in total to use random testing, it can obtain satisfied results and the application could be much lighter and simpler.

2. It can be discovered from Fig.2 that in NextDay, with different configurations, ET using our fitness function design approach can obtain different performance. Actually there are similar results in the other five object programs. By comparing the performances of the eight configurations for each program we discover that some programs prefer the configurations with lower selection pressure, while others prefer the configurations with higher selection pressure[11,30,31].

In NextDay, for all the three algorithms, configurations of C_1, C_3, C_5 and C_7 obtain a high P_h and a low *TTC*. It can be learned from Table 2 that these four configurations all choose unrepeatable survival scheme, which provides low

selection pressure.

Oppositely in PushDown, for all the three algorithms, configurations of C_2, C_4, C_6 and C_8 acquire a satisfied performance. And these four configurations all choose fitness survival scheme, which provides a high selection pressure for ET. Similarly in BinarySearch, for the algorithms of MDNS and MSG, configurations of C_2, C_4, C_6 and C_8 also perform better than the other four. And for MUNS, since all the configurations can obtain quite high P_h, there is no distinctness among them.

Besides, in the other three programs, LineCircle, SumM, and CDPlayer, even though the best configurations are not consistent with the three algorithms, C_4 and C_8 are the common ones that can obtain a high P_h for all the three algorithms in these programs. It is known that C_4 and C_8 both use the roulette-wheel selection scheme and fitness survival scheme, which make them own the highest selection pressure among these eight configurations.

The reason that different programs prefer different configurations could be concluded from the distribution of the global and local optima within the search domain. If there are a lot of local optima around the global optima within the search domain, it is advised to choose configurations with lower selection pressure, such as C_1, C_3, C_5 and C_7, to avoid the prematurity. Conversely, it is advised to choose configurations with higher selection pressure, such as C_2, C_4, C_6 and C_8, to speed up the search for global optima without worrying about the prematurity. For example in PushDown, BinarySearch, SumM, CDPlayer and LineCircle, there are not many local optima in the search domain. Thus C_2, C_4, C_6 and C_8 can obtain good results.

3. It is known from the discussion above that for different programs the three similarity evaluation algorithms obtain different results. Thus, we compare the performances of the three algorithms in each program. On the purpose of obtaining visible results, we choose C_8 for PushDown, BinarySearch, LineCircle, SumM and CDPlayer, and C_7 for NextDay. The comparison of P_h among the three algorithms in each object program is illustrated in Fig.8.

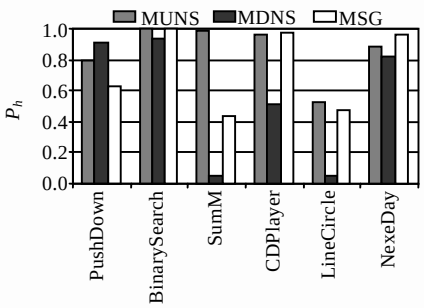


Fig.8 Comparison of P_h of algorithms

It can be discovered from Fig.8 that algorithm MUNS always performs quite well in all the object programs. And in most programs it performs the best among the three algorithms. MSG can also provide satisfied P_h for most programs, but its performance is not as stable as MUNS. It can be seen that MSG performs worse than MUNS in PushDown and SumM. Besides, MDNS performs worse than the other two algorithms in most cases. It can be discovered that MDNS obtains similar performance to the other two algorithms in PushDown, BinarySearch and NextDay, and gains worse performance in SumM, CDPlayer and LineCircle.

Commonly the performance of the algorithm lies on the guidance it provides. From the description of the algorithms in Section 4, we can learn that both MUNS and MSG evaluate the similarity between the execution track and the whole target path. The only difference is that MUNS is concerned with matching the prefix of the target path, while MSG cares about locating at the right position of the target path. Thus these two algorithms can provide similar ability of guidance.

Differently MDNS does not evaluate the similarity between the execution track and the whole target path. Instead, it investigates several discrete node segments of the target path. Thus it cannot provide as strong guidance as the other two algorithms. Actually the performance of MDNS depends on the instruction of the discrete node segments. If the discrete node segments provide enough information about the target path, MDNS may also perform well. In an extreme situation, when the missing nodes in the discrete node segments are all on a DD-path (path

chain), which are surely to be executed orderly, the performance of MDNS will be similar to the other two algorithms. What is important is that MDNS has its own application fields. It is especially suitable for inexactly defined test target, such as definition-use pair coverage, loop coverage and so on, in which the target may not be a whole program path. Instead it could be designated with only a few critical statements.

5.3 Study on the limitation and the applicable scope

The first experimental study reveals the advantages of our methods for the complex target paths. And it also mentioned that our approach can easily deal with some limitation of other commonly used path-oriented testing techniques. These techniques include two major approaches: the static analyzing, and the dynamic methods. Besides, according to Section 2 ET with PC-oriented fitness function is also a technique for path-oriented testing.

Usually these techniques could be complex and expensive to apply and may have difficulty in dealing with arrays, loops and sub-procedure calling etc.

First in static analyzing, for example the symbolic execution, the execution utilizes symbolic values as the program inputs, and then combines all the branch predications together with the path condition. The path condition can only contain input parameters as the independent variables. To generate test cases covering one specific path, symbolic execution just solves the corresponding path condition as solving a constraint system. However, since the inputs are not real values it might be very difficult to acquire such path condition when encountering the loops and the arrays[15,18]. Even though there are several methods aiming to solve this problem, and some researchers have developed testing tools on system level for symbolic execution[32,33], the complexity of using symbolic execution in those situations will still increase greatly. Besides, sometimes the path condition could be very complicated, then solving such constraint system is also a very expensive task[34].

As for dynamic methods, for example the relaxation methods, even though it uses the real values as the inputs, it still requires investigating all the branch predications, analyzing their dependency to the input variables, and then constructing the linear constraint system[13,15,17,18]. Since the relaxation methods are based on the idea of slicing, the features of loops, arrays and sub-procedure calling in a program could also increase the difficulty of their application. For another instance, Zhao has provoked a testing technique especially for string inputs. It constructs objective function with respect to the string predicate which does not meet a given path condition, and minimize the objective function by using a speedy descent search algorithm[35]. Due to the usage of branch condition, this method may also suffers from the above problem. Besides, since it uses the local search technique, its search may not as efficient as GA for complex path.

While in the ET with PC-oriented fitness function, there are similar problems. As mentioned in Section 2, since constructing such fitness function requires the assistance of data-flow analysis to investigate the dependency among variables, the complexity and the cost of its application could increase due to the features of the loops, arrays, sub-procedure calling, etc.

On the other hand, in path-oriented ET with our fitness function design approach, constructing and solving the path conditions are not required, thus they do not need the information of branch predication or the assistance of data-flow analysis. All the necessary information to build the fitness function is the node sequence of the target path. The whole testing process could be fully automated and uniform for any kinds of program paths, and will not be affected by the features of loops, arrays, sub-procedure calling, etc. For such targets, such as the target paths for PushDown, BinarySearch, SumM and CDPlayer in Table 3, our approach could be more practicable and economic, which can also be approved by the experimental results in Section 5.2.

Nonetheless, it can be discovered that if the programs don’t contain the structures of loops, arrays or sub-procedure calling, that is, for a common branch program, the first three testing techniques can actually be

applied well. In this situation, to investigate the dependency among variables could be less complicated, which becomes relatively practicable.

Therefore to compare the performances between ET with our fitness function design approach and the other path-oriented testing techniques above, and to illustrate their limitations and the applicable scopes respectively for common branch programs, we conduct the second experiment. However, since the our approach uses GA as a searching technique, that is, a global searching technique, to make it fair, we choose ET with PC-oriented fitness function as a representative of the other techniques above in the second experiment.

In this experimental study, we use the two branch programs: LineCircle and NextDay as testing objectives. For each program we choose three different paths as testing targets. The length of the paths increases from the first one to the third one. Since the experimental results are quite similar between the two programs, due to the space, we will only present the results of LineCircle as an example. For LineCircle, we choose three paths as the targets, whose information is shown in Table 4. It can be seen that with the increase of the path length, the corresponding path-condition becomes more and more complex.

Table 4 Target paths in LineCircle for experiment 2

ID	Description of test target	Node track of target path	Simplified path condition
P_1	Cover a program path, which can result in situation of "*not_line*"	(start,1,end1)	$(x_1=x_2)$ & $(y_1=y_2)$
P_2	Cover a program path, which can result in situation of "*through_center*"	(start,1,2,3,4,8,end2)	$(x_1=x_2)$ & $(y_1!=y_2)$ & $(abs(a*cx+b*cy+c)<0.001))$
P_3	Cover a program path, which can result in situation of "*not_line*"	(start,1,2,3,5,6,8,9,10,11,end5)	$(x_1!=x_2)$ & $(y_1=y_2)$ & $(abs(a*cx+b*cy+c)>=0.001))$ & $(r-0.1<=abs(a*cx+b*cy+c)/sqrt(a*a+b*b)<=r+0.1)$

Aiming at these three paths, we use our approach and PC-oriented approach repetitively to build the fitness functions, conducting path-oriented ET. To implement our approach, we still use the three different similarity evaluation algorithms, MUNS, MDNS and MSG. The result of P_h is shown in Figs.9(a), (b) and (c). It can be discovered from Fig.9 that:

1. When aiming at P_1, the shortest path, our approach with all the three algorithms can only acquire very low P_h. The reason is that in this situation the path is too short to provide enough coverage information for our approach. Thus the fitness function cannot provide subtle distinction among the candidate solutions or a good guidance during the search.

On the other hand, the PC-oriented approach performs very well in this situation. It can be discovered from Table 4 that for P_1, the path condition is quite simple. In fact, since this short path only contains one branch structure, the path condition contains only one branch predication correspondingly. In this situation, the path coverage has actually already degenerated to branch coverage. Meanwhile, the PC-oriented approach is essentially equal to the condition-oriented approach for branch coverage. Many researches have shown that condition-oriented fitness function can be easily applied for the branch coverage and can acquire satisfied performance[26,36].

Therefore from Fig.9(a) we can conclude that for the extremely short paths with quite simple path conditions, it is better to use the PC-oriented approach to construct fitness function, instead of using our approach. Because in this case, the PC-oriented fitness function could provide more precise guidance. Besides, due to the simplicity of the path condition, the construction of such fitness function will not be much more complex than the condition-oriented approach for branch/node coverage.

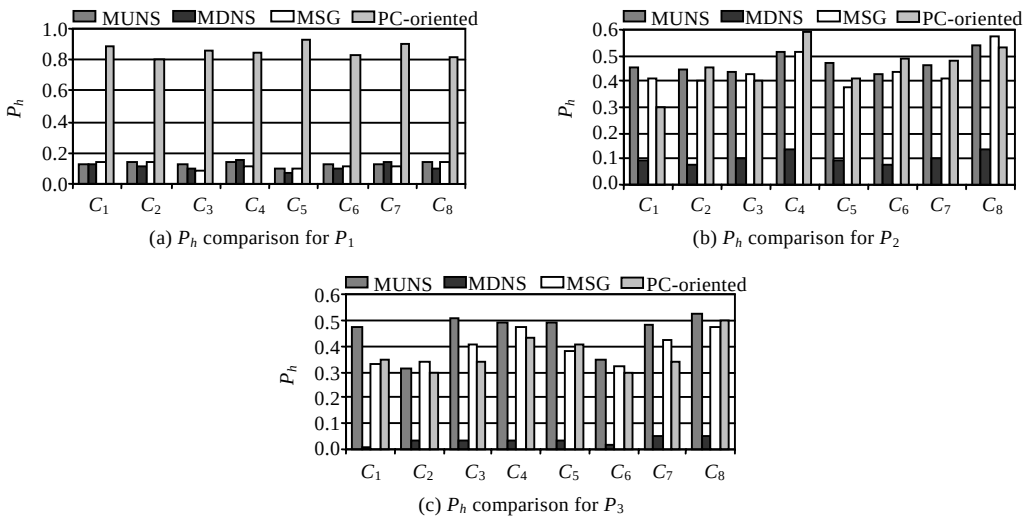
2. With the increasing length of the target paths, the performance of our approach improves, while the performance of PC-oriented approach deteriorates.

When aiming at P_2, the medium-length path, the performance of our approach with the three similarity

evaluation algorithms has been improved evidently compared to P_1. Even though the performance of MDNS is not as good as the other two algorithms, some configurations using MUNS and MSG have achieved higher P_h than the PC-oriented approach. Moreover, when aiming at P_3, the longest path, it can be discovered that for each configuration, our approach can achieve higher P_h than PC-oriented method, either using MUNS or MSG.

The reason for this phenomenon is that with the increasing length of paths, the coverage information becomes more and more precise in guiding the search for our approach, while the path condition becomes more and more complex to build an effective fitness function for PC-oriented approach. Therefore, it can be concluded that for the long paths with quite complex path conditions, it is still better to use our approach instead of the PC-oriented approach to build fitness function for path-oriented ET. Because normally the longer the path is, the more branch predications it contains. In this situation, PC-oriented approach becomes too simple to provide a subtle way to combine all the branch predications together. Thus its effectiveness may decrease with the increasing of the length.

Additionally since PC-oriented approach usually requires the assistance of data-flow analysis, the more complex the path condition is, the more expensive the data flow analysis could be. While for our approach, as mentioned above, the complexity of fitness function constructing will be stable and can be fully automated. Therefore, even Fig.9(b) and (c) show that PC-oriented approach can also achieve acceptable performance, it is less practicable than our approach due to the increasing complexity and cost of the fitness function construction.



(a) P_h comparison for P_1

(b) P_h comparison for P_2

(c) P_h comparison for P_3

Fig.9 Empirical results in LineCircle

5.4 Empirical conclusions

From the analysis above we can summarize the following empirical conclusions:

1. For path-oriented testing with long and complex target paths, which contain loops, arrays, sub-procedure calling, etc., ET with our fitness function design approach can be successfully applied without the confusion between static and dynamic information, or being lack of guidance.

2. Different objective programs prefer different configurations. For the programs with a lot of local optima around the global optima, such as NextDay, it is inclined to use configurations with lower selection pressure in order to avoid the prematurity. In our experiments, such configurations usually involve random selection and unrepeatable survival.

Conversely, for the object programs without many local optima, such as PushDown, BinarySearch, SumM, CDPlayer and LineCircle, it is better to choose a configuration with higher selection pressure to speed up the

convergence. In this paper, configurations with roulette-wheel selection and fitness survival can provide high selection pressure.

3. Among the three similarity evaluation algorithms, MUNS and MSG have better performances than MDNS in most cases. Because MUNS and MSG evaluate the similarity between the execution track and the whole target path, while MDNS investigates several discrete node segments of the target path. Thus MUNS and MSG can provide stronger guidance than MDNS.

4. ET with our fitness function design approach is not the substitution of other path-oriented testing techniques. In some cases, other techniuques are still practical.

For the target paths with loops, arrays, sub-procedure calling, etc, if the target is quite easy to cover, such as the target path in BinarySearch, non-heuristic automatic techniques like random testing can also be applied efficiently, which is much lighter and simpler. While for the very short paths without loops, arrays, sub-procedure calling, etc, static analyzing, dynamic methods, ET with PC-oriented fitness function, etc. may perform better than ET with our fitness function design approach, since the path is too short to provide enough coverage information for our methods.

Actually ET with our fitness function design approach could be an important complementary technique of other commonly used techniques in path-oriented testing. Our approach is designed especially for the complex target paths, for which other techniques usually have trouble in generating the desired test cases. To achieve the full coverage based on the prescribed criterion we can first conduct other proper techniques in order to cover the simple and the short paths, and then apply ET with our fitness function design approach to cover the remains. In this way, the whole process of generation could become cheaper and more efficient.

6 Conclusions

Evolutionary Testing (ET) is an efficient technique of automatic testing and it has many kinds of applications. However, there are few researches on ET in the field of path-oriented testing, which is an important technique in structural testing. Thus, in this paper we provoke our approach to construct the fitness function for path-oriented ET based on the similarity evaluation between the execution track and the target path or node sequence. We provide three different algorithms for the similarity evaluation.

The empirical studies reveal that with proper similarity algorithm and suitable configuration, ET with our fitness function design approach can obtain obvious superiority over other commonly used path-oriented testing techniques. However, it also has limitations. Experimental results show that our approach is more suitable for the long and complicated program paths, while for the very short or simple paths, other techniques may acquire better performance. Generally MUNS and MSG usually have better performance than MDNS. And if the program has a lot of local optima, the configurations with lower selection pressure are more suitable. Otherwise, the ones with higher selection pressure are recommended.

Actually generating test cases for target paths means to search for the sub-domains determined by the paths within the whole input domain. Usually the input domain of a program can be quite large, multi-dimensional, discontinuous and non-linear. And the sub-domain of a certain path may distribute discontinuously and sparsely within the input domain, which makes the blind exhaustive search impractical[19,20]. In this situation, ET that can search for target intelligently and automatically becomes quite helpful. Generally the fitness function, which provides the guidance for the search, is quite important for ET and may influence its performance greatly. In our approach we have provided a proper guidance for ET by evaluating the similarity between the execution track and the target path. Under the guidance, ET could be quite beneficial especially for the long paths and the complex search domain, which are the roadblocks in the non-heuristic techniques and the manual techniques. Besides, since

our approach does not require constructing or solving the path condition, its whole testing process could be fully automated and uniform for any kinds of program paths. Thus it can easily deal with the features of loops, arrays, sub-procedure calling, etc.

Additionally, with the multi-core CPUs being more and more popular the concurrent computation has become a new trend[37,38]. Since ET is quite suitable for the large-scale and concurrent search, we can anticipate that ET will have more widespread applications in the future. Since our approach is based on ET, we can also anticipate that it could be a promising technique for a more practical and efficient path-oriented testing.

References:

[1] Back T. Evolutionary Algorithms in Theory and Practice. London: Oxford University Press, 1996.

[2] Michael C, McGraw G, Schatz M. Generating software test data by evolution. IEEE Trans. on Software Engineering, 2001,27(12): 1085−1110.

[3] Lefticaru R, Ipate F. Functional search-based testing from state machines. In: Proc. of the 1st Int'l Conf. on Software Testing, Verification and Validation. 2008. 525−528.

[4] Tracey N, Clark J, Mander K, McDermid J. An automated framework for structural test-data generation. In: Proc. of the 13th IEEE Conf. on Automated Software Engineering. 1998. 285−288.

[5] Wappler S, Wegener J. Evolutionary unit testing of object-oriented software using strongly-typed genetic programming. In: Proc. of the 8th Genetic and Evolutionary Computation Conf. 2006. 1925−1932.

[6] Wegener J, Sthamer H, Baresel A. Application fields for evolutionary testing. In: Proc. of the 9th European Software Testing Analysis & Review. 2001.

[7] Wegener J, Buhr K, Pohlheim H. Automatic test data generation for structural testing of embedded software systems by evolutionary testing. In: Proc. of the 4th Genetic and Evolutionary Computation Conf. 2002. 1233−1240.

[8] Wegener J, Bühler O. Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system. In: Proc. of the 6th Genetic and Evolutionary Computation Conf. 2004. 1400−1412.

[9] Jones B, Sthamer H, Eyres D. Automatic structural testing using genetic algorithm. Software Engineering Journal, 1996,11(5): 299−306.

[10] Jones B, Eyres D, Sthamer H. A strategy for using genetic algorithms to automate branch and fault-based testing. Computer Journal, 1998,41(2):98−107.

[11] Sthamer H. The automatic generation of software test data using genetic algorithms [Ph.D. Thesis]. Pontyprid: University of Glamorgan, 1996.

[12] Harman M. The current state and future of search based software engineering. In: Proc. of the Future of Software Engineering. 2007. 342−357.

[13] Myers GJ. The Art of Software Testing. 2nd ed., John Wiley & Sons, Inc., 2004.

[14] Ferguson R, Korel B. The chaining approach for software test data generation. ACM Trans. on Software Engineering and Methodology, 1996,5(1):63−86.

[15] King J. Symbolic execution and program testing. Communications of the ACM, 1976,19(7):385−394.

[16] Mueller F, Wegener J. A comparison of static analysis and evolutionary testing for the verification of timing constraints. In: Proc. of the 4th IEEE Real-Time Technology and Applications Symp. 1998. 144−154.

[17] Shan JH, Wang J, Qi ZH. Survey on path-wise automatic generation of test data. Acta Electronica Sinica, 2004,32(1):109−113 (in Chinese with English abstract).

[18] Zhang J, Xu C, Wang XL. Path-Oriented test data generation using symbolic execution and constraint solving techniques. In: Proc. of the 2nd Int'l Conf. on Software Engineering and Formal Methods. 2004. 242−250.

[19] Clark J, Dolado J, Harman M, Hierons R, Jones B, Lumkin M, Mitchell B, Mancoridis S, Rees K, Roper M, Shepperd M. Reformulating software engineering as a search problem. IEE Proc.-Software, 2003,150(3):161−175.

[20] Wegener J, Baresel A, Sthamer H. Suitability of evolutionary algorithms for evolutionary testing. In: Proc. of the 26th Annual Int'l Computer Software and Applications Conf. 2002. 287−289.

[21] Baresel A, Sthamer H, Schmidt M. Fitness function design to improve evolutionary structural testing. In: Proc. of the 4th Genetic and Evolutionary Computation Conf. 2002. 1329−1336.

[22] Harman M. Automated test data generation using search based software engineering. In: Proc. of the 2nd Int'l Workshop on Automation of Software Test. 2007. 2−3.

[23] Pargas R, Harrold M, Peck R. Test-Data generation using genetic algorithm. Software Testing, Verification and Reliability, 1999, 9(4):263−282.

[24] Sthamer H, Wegener J, Baresel A. Using evolutionary testing to improve efficiency and quality in software testing. In: Proc. of the 2nd Asia-Pacific Conf. on Software Testing Analysis and Review. 2002.

[25] Wegener J, Baresel A, Sthamer H. Evolutionary test environment for automatic structural testing, information and software technology special issue on software eng. Using Metaheuristic Innovative Algorithms, 2001,43(14):841−854.

[26] Xu BW, Xie XY, Shi L, Nie CH. Application of genetic algorithms in software testing. In: Proc. of the Advances in Machine Learning Application in Software Engineering. IGI Global, 2007. 287−317.

[27] Harman M, Hu L, Hierons R, Wegener J, Sthamer H, Baresel A, Roper M. Testability transformation. IEEE Trans. on Software Engineering, 2004,30(1):3−16.

[28] Pustell J, Kafatos F. A high speed, high capacity homology matrix: Zooming through SV40 and Polyoma. Nucleid Acids Research, 1994,10(15):42−49.

[29] Ducasse S, Matthias R, Serge D. A language independent approach for detecting duplicated code. In: Proc. of the Int'l Conf. on Software Maintenance. 1999. 109−108.

[30] Shi L, Xu BW, Xie XY. An empirical study of configuration strategies of evolutionary testing. Int'l Journal of Computer Science and Network Security, 2006,6(1):44−49.

[31] Xie XY, Xu BW, Shi L, Nie CH, He YX. A dynamic optimization strategy for evolutionary testing. In: Proc. of the 12th Asia-Pacific Software Engineering Conf. 2005. 568−575.

[32] Anand S, Pǎ sǎ reanu C, Visser W. JPF-SE: A symbolic execution extension to Java PathFinder. In: Proc. of the 13th Int'l Conf. on Tools and Algorithms for the Construction and Analysis of Systems. 2007. 134−138.

[33] Păsăreanu C, Mehlitz P, Bushnell D, Gundy-Burlet K, Lowry M, Person S, Pape M. Combining unit-level symbolic execution and system-level concrete execution for testing NASA software. In: Proc. of the Int'l Symp. on Software Testing and Analysis. 2008. 15−26.

[34] Kannan Y, Sen K, Universal symbolic execution and its application to likely data structure invariant generation. In: Proc. of the Int'l Symp. on Software Testing and Analysis. 2008. 283−294.

[35] Zhao RL. Search-Based automatic path test generation method for character string data. Journal of Computer-Aided Design and Computer Graphics, 2008,20(5):671−677 (in Chinese with English abstract).

[36] Xie XY, Xu L, Xu BW, Nie CH, Shi L. Survey of evolutionary testing. Journal of Frontiers of Computer Science and Technology, 2008,2(5):449−466 (in Chinese with English abstract).

[37] Sutter H. The free lunch is over: A fundamental turn toward concurrency in software. Dr. Dobb's Journal, 2005,30(3). http://www.gotw.ca/publications/concurrency-ddj.htm

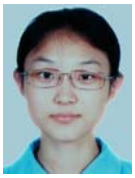
[38] Sutter H, Larus J. Software and the concurrency revolution. ACM Queue, 2005,3(7):56−62.

附中文参考文献:

[17] 单锦辉,王戟,齐治昌.面向路径的测试数据自动生成方法述评.电子学报,2004,32(1):109−113.

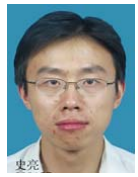
[35] 赵瑞莲.基于搜索的面向路径字符串测试数据自动生成方法.计算机辅助设计与图形学学报,2008,20(5):671−677.

[36] 谢晓园,许蕾,徐宝文,聂长海,史亮.演化测试技术的研究.计算机科学与探索,2008,2(5):449−466.



XIE Xiao-Yuan was born in 1983. She is a Ph.D. candidate at the Computer Software and Theory, Southeast University. Her current research focuses on the Search-based software testing.

XU Bao-Wen was born in 1961. He is a professor and doctoral supervisor at the Southeast University and a CCF senior member. His research areas are program language, software engineering, concurrent and network software.



SHI Liang was born in 1979. He granted doctor degree at Southeast University. He is a software engineer in Microsoft China. His research interests include software analysis and testing.

NIE Chang-Hai was born in 1971. He is an associated professor at the Southeast University. His research areas are software engineering and testing, fuzzy information processing, and neural network.