The Journal of Systems and Software 216 (2024) 112121

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Set evolution based test data generation for killing stubborn mutants[☆]

Changqing Wei [a], Xiangjuan Yao [a,*], Dunwei Gong [b], Huai Liu [c], Xiangying Dang [d]

[a] School of Mathematics, China University of Mining and Technology, Xuzhou 221116, China
[b] College of Automation and Electronic Engineering, Qingdao University of Science and Technology, Qingdao, 266061, China
[c] Department of Computing Technologies, Swinburne University of Technology, Melbourne, Australia
[d] School of Information Engineering (School of Big Data), Xuzhou University of Technology, Xuzhou, 221018, China

ARTICLE INFO

Keywords:
Mutation testing
Set evolution
Stubborn mutants
Test data generation

ABSTRACT

Mutation testing is a fault-based and powerful software testing technique, but the large number of mutations can result in extremely high costs. To reduce the cost of mutation testing, researchers attempt to identify stubborn mutants and generate test data to kill them, in order to achieve the same testing effect. However, existing methods suffer from inaccurate identification of stubborn mutants and low productiveness in generating test data, which will seriously affect the effectiveness and efficiency of mutation testing. Therefore, we propose a new method of generating test data for killing stubborn mutants based on set evolution, namely TDGMSE. We first propose an integrated indicator to identify stubborn mutants. Then, we establish a constrained multi-objective model for generating test data of killing stubborn mutants. Finally, we develop a new genetic algorithm based on set evolution to solve the mathematical model. The results on 14 programs depict that the average false positive (or negative) rate of TDGMSE is decreased about 81.87% (or 32.34%); the success rate of TDGMSE is 99.22%; and the average number of iterations of TDGMSE is 16132.23, which is lowest of all methods. The research highlights several potential research directions for mutation testing.

1. Introduction

Software testing is the main approach for improving the reliability of software products by detecting possible errors (Mantyla et al., 2015). However, it is often difficult, tedious, and time-consuming for testers to perform exhaustive testing.

To support exhaustive testing, mutation testing (Hamlet, 1977) can guide the design of test data that may reveal faults. The key idea of mutation testing is to use artificially or automatically introduced faults (called mutants) to identify the weaknesses in the test suite (i.e., the undetected faults forming the weaknesses of the test suite) and guide test data generation (i.e., the undetected faults forming test targets). Therefore, testers can improve their test suites by designing test data that kill mutants (implying the detection of corresponding faults).

The experience of mutation testing has shown that many mutants are very easy to be killed by almost any test data and the easy-to-kill mutants cannot provide any valuable guidance for generating test data (Ammann et al., 2014; Petrović and Ivanković, 2018). However, actual engineering practice has also shown that there are some mutants that are hard to kill (also known as *stubborn mutants* (Yao et al., 2014)) and provide a significant advantage over the easy-to-kill mutants for generating test data (Dang et al., 2020; Chekam et al., 2021). Interestingly, these stubborn mutants that are often related to special "corner cases" (Chekam et al., 2017), which are research foci for the community of software testing. Some industrial studies (Baker and Habli, 2013; Delgado-Pérez et al., 2018) have also emphasized the importance of using stubborn mutants as the test targets, including a large-scale study with Google developers (Petrović and Ivanković, 2018). Therefore, it is a very meaningful thing to identify stubborn mutants and generate test data to kill them, which can not only improve the error detection ability of test data, but also reduce the cost of testing due to the reduction of mutants

However, generating test data of killing stubborn mutants is a challenging task because testers normally need to perform differential analysis on the paths, constraints, and data states of the program versions (original and mutated versions). To fulfill this challenging task, researchers have proposed a few methods, such as the search domain reduction-based co-evolutionary genetic algorithm (SDRCGA) proposed by Dang et al. (2020) and the dynamic symbolic execution technique (DSE) proposed by Chekam et al. (2021). Although SDRCGA and DSE can effectively generate test data of killing stubborn mutants under certain circumstances, they all have their own shortcomings: (1) SDRCGA only uses a few indicators based on accessibility to identify

☆ Editor: Antonia Bertolino.
* Corresponding author.
E-mail address: yaoxj@cumt.edu.cn (X. Yao).

https://doi.org/10.1016/j.jss.2024.112121
Received 6 June 2023; Received in revised form 8 May 2024; Accepted 1 June 2024
Available online 6 June 2024

stubborn mutants and only generates a test datum that kills a stubborn mutant at a time; (2) DSE searches for stubborn mutants based on the execution of test data generated by other technologies and only uses inefficient symbol execution method to generate test data. In this article, we focus on how to accurately identify stubborn mutants and efficiently generate test data for killing them.

To fulfill this task, we propose a new method of test data generation for killing stubborn mutants based on set evolution, namely TDGMSE. On the one hand, TDGMSE constructs an integrated indicator by introducing infection in identifying stubborn mutants, which improves the accuracy of identifying stubborn mutants. On the other hand, TDGMSE introduces some strategies (such as the spatial similarity and the adjustment operator) for accelerating test data generation to improve the efficiency of generating test data for killing stubborn mutants.

In particular, our work makes the following three contributions:

(1) We propose an enhanced comprehensive indicator for identifying stubborn mutants, which serves as the foundation for identifying stubborn mutants;

(2) We develop a new method of test data generation based on set evolution, which mainly includes (i) a constrained multi-objective mathematical model for generating test data of killing stubborn mutants and (ii) a new genetic algorithm based on set evolution for solving the constructed mathematical model;

(3) The effectiveness of TDGMSE is evaluated through an empirical study based on 14 real-life programs. The experimental results show that TDGMSE can not only accurately identify stubborn mutants, but also improve both the effectiveness and efficiency of generating test data in mutation testing.

The rest of this paper is organized as follows. The related work is introduced in Section 2. Section 3 describes the proposed method, which includes the identification of stubborn mutants and the method of generating a test suite based on set evolution. The design and setting of our empirical study are presented in Section 4. Section 5 gives and analyses the experimental results. Section 6 presents some discussions about our method in this article. Finally, Section 7 provides a summary of the paper and identifies future work.

2. Related work

In this study, the optimization method of the set evolution is used to generate a test suite of killing stubborn mutants to improve the efficiency of mutation testing. Thus, this section introduces the related works, which includes mutation testing, the optimization based on set evolution and test data generation.

2.1. Mutation testing

Mutation testing is one main category of fault-based software testing techniques (Hamlet, 1977) and its complete analysis process is shown in Fig. 1. Given the source program P and a test suite T, we first set some mutation operators and generate all mutants by executing mutation operators on P. Then, we identify equivalent mutants from all mutants. After that, we execute the test data in test suite T on the remaining non-equivalent mutants. If we detect all non-equivalent mutants, we can complete the mutation testing analysis. Otherwise, we need to design additional test data and add them to test suite T for undetected mutants.

In mutation testing, the small grammatical changes are called mutation operators, the modified statements are called the mutated statements, and a new copy of the program with a mutated statement is called a mutant (Wei et al., 2021).

In mutation testing, it is generally necessary to meet three conditions, namely *accessibility*, *infection* (*necessity*) and *propagation* (*sufficiency*), to determine whether a mutant has been killed (Demillo and Offutt, 1991). Given a test datum t, a mutant M and a mutated statement s' from M, we can use three conditions to determine whether

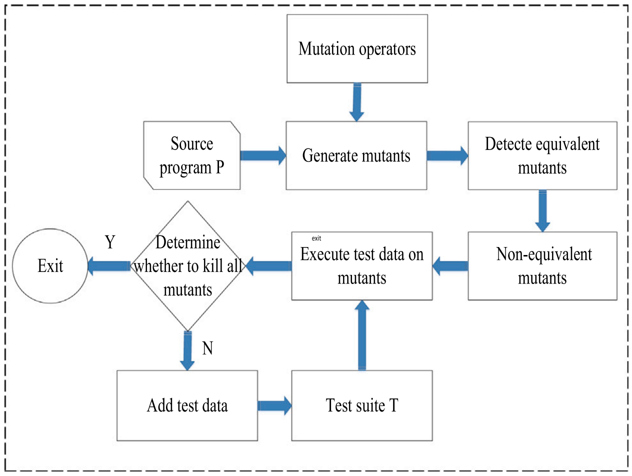


Fig. 1. A complete process of mutation testing.

or not it kills M: (1) $accessibility$ — t must be able to reach the position of s'; (2) $necessity$ — t must change the program state after executing s'; and (3) $sufficiency$ — t must cause the output of the mutant to be different from that of the original program. Furthermore, we can know that M is killed by t when M satisfies three conditions in strong mutation testing, while M is killed by t when M only satisfies *accessibility* and *infection* (*necessity*) in weak mutation testing (Yue and Harman, 2010; Yao et al., 2015).

There are some mutants that cannot be killed by any test data in mutation testing, and these mutants are called *equivalent mutants* (Offutt and Pan, 1997). In addition, there are some *easy-to-kill mutants* (Ammann et al., 2014; Petrović and Ivanković, 2018) that can be almost easily killed by any test data and some *difficult-to-kill mutants* (*stubborn mutants* (Yao et al., 2014)) that can be almost difficultly killed by any test data in mutation testing. However, researchers mainly focus on stubborn mutants that often represent the "key components" of the tested program in generating test data for mutation testing (Yao et al., 2014; Chekam et al., 2017).

At present, researchers have proposed many different hard-to-detect mutant detection techniques. For example, Yao et al. (2014) proposed a method based on execution of test data, which depends on the quantity of test data. Dang et al. (2020) proposed a method by constructing a comprehensive indicator, which depends on indicators based on *accessibility*. Arasteh et al. proposed some methods (Arasteh et al., 2022, 2023; Javad Hosseini et al., 2021), such as the method based on artificial bee colony optimization algorithm (Arasteh et al., 2022), the method based on the discretized and modified forest optimization algorithm (Arasteh et al., 2023) and the error-propagation aware method based on the program control-flow graph and the genetic algorithm (Javad Hosseini et al., 2021). Although the previous methods can detect some hard-to-detect mutants, they still have the inaccurate issue of identifying stubborn mutants, which provides a direction for this study. In mutation testing, *mutation score* (MS) is often used to evaluate the fault detection ability of a test suite and MS refers to the ratio of the killed non-equivalent mutants to all non-equivalent mutants (Dave and Agrawal, 2016). For MS, the larger MS, the stronger the fault detection ability of the test suite.

At present, mutation testing has been widely used in various languages (such as C and Java) and at various levels of software testing (such as unit and integration) (Yue and Harman, 2010). In addition, many mutation testing tools have been developed, some of which have been applied to actual software testing (Namin et al., 2015; Derezinska and Szustek, 2008; Feng et al., 2008).

2.2. Optimization based on set evolution

In recent years, the evolutionary optimization method (EOM) (Korel, 1990; Wegener et al., 2001) has been widely applied to the problem of generating test data in software testing. The main idea of the EOM is as follows: (1) Transform a software testing problem into an optimization one (maximum or minimum); (2) A mathematical model is constructed for the optimization problem (single-objective (Korel, 1990; Wegener et al., 2001) or multi-objective (Harman et al., 2010)); (3) An evolutionary algorithm (such as GA (Korel, 1990; Wegener et al., 2001)) is used to solve the mathematical model and obtain the optimal solution.

However, an evolutionary individual only represents a test datum and only one test datum is generated at a time based on the EOM (Harman et al., 2010; Yue and Harman, 2009), which increases the cost of mutation testing. To improve the efficiency of test data generation for mutation testing, researchers use the optimization method based on set evolution (OMSE) (Yao et al., 2015) to generate a test suite of killing multiple mutants at one time.

The difference between EOM and OMSE in the case of generating test data for mutation testing is given:

Assuming the input of a program P is $X = (x_1, x_2, \cdots, x_d) \in D$ (d is the number of variables in X and D is the input domain of P), and N is the number of mutants. If only one mutant is killed at a time, then the mathematical model of killing N mutants based on EOM is established as follows:

$$\min(\max) \quad f_i(X)$$
$$s.t. \begin{cases} g_i(X) \\ X \in D, \end{cases} \tag{1}$$

where $f_i(X)$ ($i = 1, 2, \cdots, N$) is the objective function of killing the $i\text{-}th$ mutant and $g_i(X)$ ($i = 1, 2, \cdots, N$) is the $i\text{-}th$ constraint function.

However, if the above problem is transformed into a set optimization problem, the mathematical model based on OMSE is:

$$\min(\max) \quad f(\overline{X})$$
$$s.t. \begin{cases} g(\overline{X}) \\ \overline{X} \in 2^D, \end{cases} \tag{2}$$

where $\overline{X} = (X_1, X_2, \cdots, X_N)$ is a set with N test inputs, $f(\overline{X})$ is an objective function of killing N mutants and $g(\overline{X})$ is a constraint function.

To solve Formula (2), researchers use the GA based on set evolution (Yao et al., 2015; Guo et al., 2017). Note that the GA based on set evolution can generate the test suite that kills all mutants in one run. Considering the low-cost advantage of set evolution, this article improves the GA based on set evolution and applies it to the problem of test data generation for killing stubborn mutants.

2.3. Test data generation

In recent years, researchers have proposed many methods of generating test data for mutation testing (Demillo and Offutt, 1991), mainly including the constraint-based method (CBTM) (Offutt, 1988), the dynamic domain reduction method (DDRM) (Offutt et al., 1998), the search-based method (SBM) (Silva et al., 2017), and the hybrid method (HM) (Patrick et al., 2013).

Offutt (1988) proposed a CBTM of generating test data based on mutation testing criteria and the constraint-solving method, but the constraint-solving method greatly limited its applicability. Furthermore, Offutt et al. (1998) proposed a DDRM of generating test data based on backtracking searches for improving the efficiency of test data generation. For SBM, Silva et al. (2017) found that out of 19 papers in the domain of search-based mutation testing, 6 papers used GAs to generate test data under the mutation testing criteria. In addition, Patrick et al. (2013) proposed a HM of generating test data based on mutation analysis and static analysis, but it was ineffective in generating test data that kills stubborn mutants. Although the previous methods could effectively generate test data, they only generated a test datum that killed a mutant at a time, which made the efficiency of generating test data very low and it was very difficult for them to generate test data of killing stubborn mutants.

Different from the previous methods, Shan et al. (2008) proposed an iterative relaxation method of solving the problem of only generating a test datum of killing a mutant at a time, but no automatic approach was developed to select the target path. Matnei Filho and Vergilio (2016) proposed a multi-objective test data generation method for minimizing the number of products in the software product line process (SPLs), but it was ineffective in generating test data that kills stubborn mutants. Similarly, Zhang et al. (2015) proposed a method of generating test data based on mutation analysis and set evolution to solve the problem of only generating one test datum at a time, but it could not be used to generate test data for killing stubborn mutants. In addition, Guo et al. (2017) proposed a dynamic set evolution algorithm for weak mutation testing, but it was unable to handle difficult-to-kill mutants.

Recently, researchers also have proposed some new methods for generating test data. For example, Dang et al. (2021) proposed an enhanced method for mutation detection using fuzzy clustering and multi-population genetic algorithm (FUZGENMUT), but it was also unable to handle stubborn mutants due to the lack of a method for identifying stubborn mutants. Yao et al. (2020) proposed a method of test data generation for weak mutation testing via sorting mutant branches based on their dominance degrees, but it used random sampling when generating test data, which could not effectively generate test data of killing stubborn mutants. In addition, Dang et al. (2020) proposed the search domain reduction-based co-evolutionary genetic algorithm (SDRCGA) and Chekam et al. (2021) proposed the dynamic symbolic execution technique (DSE) for C programs, but SDRCGA and DSE suffered from the inaccurate identification of stubborn mutants and the problem of generating only a test datum for killing a stubborn mutant at a time. If these two problems could be solved, it would greatly improve the efficiency of mutation testing, so this article focused on solving these two problems.

In summary, the above methods has low performance in generating test data of killing stubborn mutants. Unlike existing methods, we construct an integrated indicator to accurately identify stubborn mutants and develop an improved genetic algorithm based on set evolution under a constrained multi-objective mathematical model for generating test data of killing stubborn mutants.

3. The proposed method

The main aim of this study is to improve the performance of generating test data that kill stubborn mutants.

3.1. Overall process

We propose to construct an integrated indicator for identifying stubborn mutants and use a method of generating test data for killing these stubborn mutants based on set evolution. Such an approach is implemented in a so-called TDGMSE framework, which is depicted in Fig. 2.

As observed from Fig. 2, TDGMSE is composed of two main modules: (1) Identification of stubborn mutants; (2) Test data generation based on set evolution.

For module (1) (Section 3.2), we propose a method of constructing an integrated indicator for identifying stubborn mutants, which mainly includes the acquisition of indicators based on accessibility (Section 3.2.1), the acquisition of indicators based on infection (Section 3.2.2), and the construction of the integrated indicator for identifying stubborn mutants (Section 3.2.3).

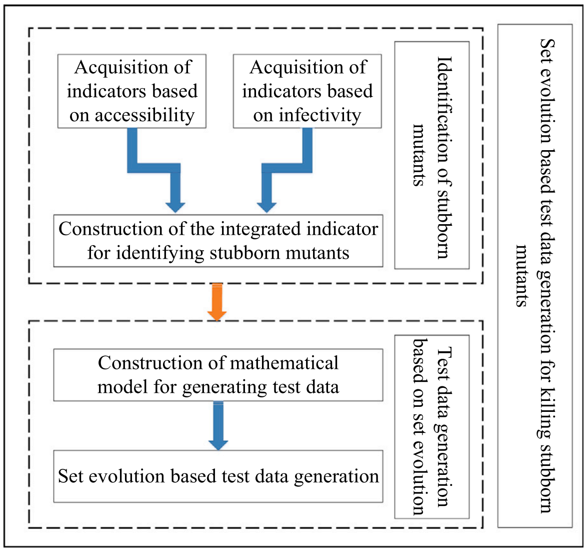
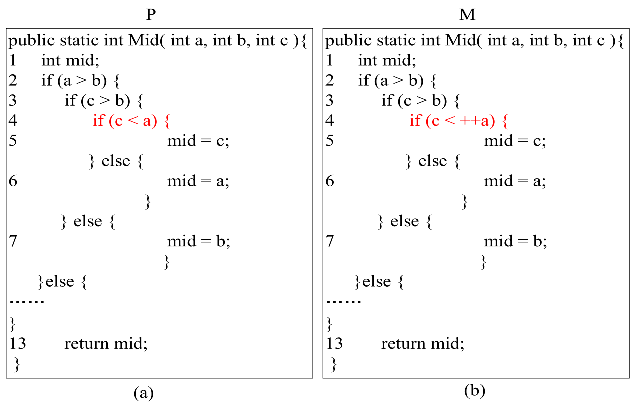

Acquisition of indicators based on accessibility
Acquisition of indicators based on infectivity
Identification of stubborn mutants
Construction of the integrated indicator for identifying stubborn mutants
Construction of mathematical model for generating test data
Set evolution based test data generation
Test data generation based on set evolution
Set evolution based test data generation for killing stubborn mutants

Fig. 2. Framework of our method.



P

```
public static int Mid( int a, int b, int c ){
1   int mid;
2   if (a > b) {
3       if (c > b) {
4           if (c < a) {
5                   mid = c;
            } else {
6                   mid = a;
                }
        } else {
7                   mid = b;
                }
    }else {
……
}
13      return mid;
 }
```

(a)

M

```
public static int Mid( int a, int b, int c ){
1   int mid;
2   if (a > b) {
3       if (c > b) {
4           if (c < ++a) {
5                   mid = c;
            } else {
6                   mid = a;
                }
        } else {
7                   mid = b;
                }
    }else {
……
}
13      return mid;
 }
```

(b)

Fig. 3. Java program for the middle value of three integers.

In module (2) (Section 3.3), the mathematical model for generating test data is first established (Section 3.3.1). Then, a method of generating a test suite with set evolution (Section 3.3.2) is used to generate test data for killing stubborn mutants.

Below, we will illustrate each specific step with the Example 1 from these two aspects.

Example 1. P (Fig. 3(a)) is a simple program to find the middle value of three integers. The input of P is X=(a, b, c), and the input domain of X is $[-10, 10]^3$. Fig. 3(b) shows an example of mutant by implementing mutation operation on statement 4. We totally obtain 11 mutants by implementing mutations on statements 3 and 4, as listed in Table 1.

3.2. Identification of stubborn mutants

To identify stubborn mutants, researchers have proposed some methods. For example, Yao et al. (2014) proposed a dynamic execution method based on branch coverage, Chekam et al. (2021) gave a dynamic execution method based on test method and Dang et al. (2020) provided a static analysis method based on *accessibility*. Although the above three methods can identify stubborn mutants, the method of Dang et al. (2020) has a lower cost than the other two methods since it can identify stubborn mutants without executing the source program and the mutants. Unfortunately, their method only considers

Table 1
Mutants corresponding to statement 4 in P.

Mutants	Original statements	Mutated statements	Mutation operators
M_1	$if(c>b)$	$if(++c>b)$	AOIS
M_2	$if(c>b)$	$if(c-->b)$	
M_3	$if(c>b)$	$if(c++>b)$	
M_4	$if(c>b)$	$if(c-->b)$	
M_5	$if(c>b)$	$if(c>++b)$	
M_6	$if(c>b)$	$if(c>b++)$	
M_7	$if(c>b)$	$if(c>--b)$	
M_8	$if(c>b)$	$if(c>b--)$	
M_9	$if(c<a)$	$if(++c<a)$	
M_{10}	$if(c<a)$	$if(--c<a)$	
M_{11}	$if(c<a)$	$if(c++<a)$	
M_{12}	$if(c<a)$	$if(c--<a)$	
M_{13}	$if(c<a)$	$if(c<++a)$	
M_{14}	$if(c<a)$	$if(c<--a)$	
M_{15}	$if(c<a)$	$if(c<a++)$	
M_{16}	$if(c<a)$	$if(c<a--)$	
M_{17}	$if(c>b)$	$if(c>=b)$	ROR
M_{18}	$if(c>b)$	$if(c<b)$	
M_{19}	$if(c>b)$	$if(c<=b)$	
M_{20}	$if(c>b)$	$if(c==b)$	
M_{21}	$if(c>b)$	$if(c!=b)$	
M_{22}	$if(c>b)$	$if(!(c>b))$	COI
M_{23}	$if(c<a)$	$if(!(c<a))$	
M_{24}	$if(c>b)$	$if(c>\sim b)$	LOI
M_{25}	$if(c>b)$	$if(\sim c>b)$	
M_{26}	$if(c<a)$	$if(c<\sim a)$	
M_{27}	$if(c<a)$	$if(\sim c<a)$	

accessibility as the indicator, which seriously affects the accuracy of identifying stubborn mutants. Therefore, this article comprehensively considers *accessibility* and *infection* to identify stubborn mutants.

3.2.1. Acquisition of indicators based on accessibility

Accessibility, as the first necessary condition determining whether a mutant can be killed, will help us obtain an important criterion for determining whether a mutant is stubborn or not. One of the most important reasons is that the mutated statement in a stubborn mutant should have difficulty being reached by test data.

This article considers the execution probability and variable complexity (Dang et al., 2020) involve paths as indicators to measure *accessibility*.

Assuming that a test input of a program P is $X=(x_1, x_2, \cdots, x_d)$ (d is the number of variables), s is a statement of P, s' is a mutated statement of s and L paths (denoted as $\rho_1, \rho_2, \cdots, \rho_L$) cover s'.

- **Execution probability.** Assume that ρ_r ($r=1,2,\cdots,L$) contains c_r branch statements and $Pro(s_i^{br})$ (as shown in Table 2 (Dang et al., 2020)) is the execution probability of the *i-th* branch statement s_i^{br} ($i=1,2,\cdots,c_r$), the execution probability of ρ_r is defined as:

$$EP(\rho_r)=\prod_{i=1}^{c_r} Pro(s_i^{br}) \tag{3}$$

- **Variable complexity.** Assume that d_{ρ_r} is the number of variables in ρ_r ($r=1,2,\cdots,L$), the variable complexity $VC(\rho_r)$ of ρ_r is defined as:

$$VC(\rho_r)=\frac{d_{\rho_r}}{d} \tag{4}$$

Note that the more input variables involved in a path that reaches a mutated statement, the more difficult it is to generate test data by this path to kill this mutant because of the large search domain.

Based on Formula and (4), the accessibility indicator for ρ_r is defined as:

$$\sigma_r=\omega_1\cdot(1-EP(\rho_r))+\omega_2\cdot VC(\rho_r), \tag{5}$$

Table 2
The execution probability of branch statements.

Branch statements	Execution probability
$Pro(\varepsilon_1 < \varepsilon_2)$	1/2
$Pro(\varepsilon_1 <= \varepsilon_2)$	1/2
$Pro(\varepsilon_1 > \varepsilon_2)$	1/2
$Pro(\varepsilon_1 >= \varepsilon_2)$	1/2
$Pro(\varepsilon_1 == \varepsilon_2)$	q
$Pro(\varepsilon_1 != \varepsilon_2)$	1 - q
$Pro(\mu_1 \&\& \mu_2)$	$Pro(\mu_1)^* Pro(\mu_2)$
$Pro(\mu_1 \parallel \mu_2)$	$Pro(\mu_1) + Pro(\mu_2) - Pro(\mu_1) * Pro(\mu_2)$
$Pro(\neg\mu_1)$	$1 - Pro(\mu_1)$

Notes: ε_i $(i = 1, 2)$ is variable, constant or array element, μ_i $(i = 1, 2)$ is the branch statement composed of ε_1 and ε_2, q = 1/16.

where ω_i $(i = 1, 2)$ is the weight coefficient and $\omega_1 > \omega_2 > 0$. Note that we can set $\omega_1 = 0.6$ and $\omega_2 = 0.4$ according to Dang et al. (2020).

Furthermore, assume that M is the mutant that contains s', a comprehensive indicator of M is defined as:

$$A(M) = \min\{\sigma_r | r = 1, 2, \dots, L\}, \tag{6}$$

where σ_r is the $r\text{-}th$ accessibility based indicator and $A(M) \in [0, 1]$. For $A(M)$, the smaller $A(M)$, the easier it is to kill M.

The following illustrates the specific steps of calculating $A(M)$ based on the mutant M (as shown in Fig. 3(b)) from Example 1.

(1) Calculate the execution probability based on Formula . We first obtain two paths that can reach the mutated statement 4 from M, namely $\rho_1 = (1, 2, 3, 4, 5, 13)$ and $\rho_2 = (1, 2, 3, 4, 6, 13)$. After that, we can calculate $EP(\rho_1)$ and $EP(\rho_2)$ based on Formula and Table 2. For example, we first obtain that P_1 contains three branch statements (statements 3, 4 and 5) and the execution probability of each branch statement is 1/2 based on Table 2. After that, we calculate $EP(\rho_1)$ based on Formula as follows:

$$EP(\rho_1) = \frac{1}{2} \times \frac{1}{2} \times \frac{1}{2} \approx 0.13$$

Similarly, we obtain $EP(\rho_2) \approx 0.13$ and the vector (0.13, 0.13).

(2) Obtain the variable complexity of M based on Formula (4). We first obtain that the inputs of M contained in ρ_1 and ρ_2 are both (a, b, c). After that, we get $VC(\rho_1) = 1$ and $\text{VC}(\rho_2) = 1$ based on Formula (4).

(3) Calculate $A(M)$ based on steps (1) and (2). The comprehensive indicator $\sigma_1 = 0.6 \times (1 - 0.13) + 0.4 \times 1 \approx 0.92$ ($\sigma_2 \approx 0.92$) is obtained by Formula (5) when $\omega_1 = 0.6$ and $\omega_2 = 0.4$. After that, $A(M)$=0.92 is obtained by Formula (6).

3.2.2. Acquisition of indicators based on infection

Although *accessibility* is provided to determine the difficulty of killing mutants in Section 3.2.1, it is inaccurate. For example, the mutants in the same location have the same stubbornness based on *accessibility*, but their stubbornness is indeed different. Note that *infection* describes the state of the program, under which a mutant can change the state of the program (Demillo and Offutt, 1991).

Therefore, this article identifies stubborn mutants by adding *infection* on the basis of *accessibility*. The reason for adding *infection* mainly considers the following two factors: (1) Determining stubborn mutants based on *accessibility* cannot consider the difficulty level between mutants at the same location; (2) According to the experience of testers, we obtain that: the more difficult it is to infect mutants, the more difficult it is to kill them.

This article mainly considers the degree of change in the statement, the mutated statement type (MST), and the mutation operator type (MOT). The reasons for choosing these three important factors are as follows:

(1) The difficulty of killing different mutants varies depending on the size of the change in statement (Yao et al., 2014). For a statement

Table 3
Infection levels under MST and MOT.

Infection types	Representations	Infection levels
MST	s_{fc}	0.3
	s_c	0.25
	s_l	0.25
	s_a	0.15
	s_r	0.05
MOT	o_{one}	0.05
	o_{two}	0.35
	$o_{>two}$	0.6

s and its mutated statement s', the smaller the change in s, the harder it is to be infected.

(2) In mutation testing, modifying a statement may not necessarily result in a change in program state, but the likelihood of a change in program state may depend on MST and MOT. For a assignment statement s_a and the conditional statement s_c, s_a is easier to be infected than s_c when implementing the same mutation operator on them (Yao et al., 2014). In addition, the mutation operator ABS produces few stubborn mutants while the mutation operator LCR can obtain many stubborn mutants (Yao et al., 2014; Yue and Harman, 2010), which indicates that different mutation operators may have different abilities to infect program states.

After giving three important factors, we provide three infection indicators to measure them.

First of all, we use statement similarity to measure the degree of change in the statement. Given a program P, we obtain a mutant M of P. For a statement s of P and a mutated statement s' of M, the greater the similarity between s and s' (the smaller the change in s), the harder it is to be infected. Therefore, the statement similarity is defined as:

- **Statement similarity.** Given a statement s for P and a mutated statement s' for M, the statement similarity $Se(M)$ between s and s' is defined as:

$$Se(M) = \frac{|s \bigcap s'|}{|s \bigcup s'|}, \tag{7}$$

where $|s \bigcap s'|$ is the intersection of the same parts between s and s', and $|s \bigcup s'|$ is the union of different parts of s and s'. For $Se(M) \in [0, 1]$, the smaller $Se(M)$, the easier it is for the mutant to be killed.

Note that we use the Jaccard index[1] for calculating $Se(M)$ and the Jaccard index is used to compare the difference and similarity between two samples in natural language processing (Wei et al., 2021). The larger the value of the Jaccard index, the higher the similarity between two samples. The specific steps for calculating $Se(M)$ are as follows: (1) Calculate the TF matrix (or vector) of s and s' based on $CountVector$ in the $Sklearn$ library; (2) Use $Numpy$ to calculate the intersection and union between s and s'; (3) Calculate the $Se(M)$ according to Formula (7).

Then, we use the infection level of MST and the infection level of MOT to measure MST and MOT. Below, we will provide their definitions in sequence.

For all mutated statements, the infection level varies among different mutated statements in mutation testing. Given a program P, we can obtain the types of all statements from P based on static analysis, mainly including the calling function statement s_{fc}, the conditional statement s_c, the loop statement s_l, the assignment statement s_a and the return statement s_r. Furthermore, we can sort these statements based on the experience of the testers, that is, $s_{fc} > s_c = s_l > s_a > s_r$. Therefore, the infection level of MST is defined as:

[1] https://cuiqingcai.com/6101.html

- **Infection level of MST.** For P, we can obtain a set $\Omega=\{s_{fc}, s_c, s_l, s_a, s_r\}$ (column 2 in Table 3) of MSTs for P. For M and its mutated statement $M_{s'}$, if $M_{s'}$ belongs to the i-th (i = 1, 2, ⋯, 5) MST in Ω, the infection level of MST is $\xi_i(M_{s'})$ and its value is given as shown in column 3 from Table 3. From Table 3, we obtain that the larger $\xi_i(M_{s'})$, the harder it is to be infected.

After that, we give the infection level of MOT. Considering that different mutation operators produce mutants with different infection levels, we classify the mutation operators based on the number of variables directly affected by the mutation operator contained in the mutated statement and obtain three categories: the action of one variable (o_{one}), the action of two variables (o_{two}), and the action of two or more variables ($o_{>two}$). Furthermore, we can sort these operators based on experimental experience, that is, $o_{>two} > o_{two} > o_{one}$. Therefore, the infection level of MOT is defined as:

- **Infection level of MOT.** Given a set $\Omega_o=\{o_{one}, o_{two}, o_{>two}\}$ (column 2 in Table 3) of MOTs for P and a mutation operator M_o for M, if M_o belongs to the j-th (j = 1, 2, 3) MOT in Ω_o, the infection level of MOT is $\zeta_j(M_o)$ and its value is given as shown in column 3 from Table 3. From Table 3, we obtain that the larger $\zeta_j(M_o)$, the more difficult it is to be infected.

Note that the values in Table 3 are obtained using the following method: (1) Obtain each statement type (or mutation operator type) based on the tester's experience and arrange them in order; (2) Assign the value size to each statement type (or mutation operator type) in a certain proportion based on sorting when considering all statement types (or mutation operator types) as unit 1. For example, for MST, we first obtain sorting $s_{fc}>s_c=s_l>s_a>s_r$. Furthermore, we can obtain the values for each statement type based on the ratio of 6:5:5:3:1, that is 0.30 (6/20), 0.25 (5/20), 0.25 (5/20), 0.15 (3/20), and 0.05 (1/20). The value of MOT is similar to that of MST, and we can obtain the final result as shown in Table 3.

Based on statement similarity, the infection level of MST and the infection level of MOT, we will provide a comprehensive indicator.

Given an arbitrary statement s of P, a mutated statement $M_{s'}$ of s in M, a mutation operator M_o for M, a set $\Omega = \{s_{fc}, s_c, s_l, s_a, s_r\}$ (as shown in Table 3) for P and a set $\Omega_o=\{o_{one}, o_{two}, o_{>two}\}$ (as shown in Table 3) for P, if $M_{s'}$ belongs to the i-th (i = 1, 2, ⋯, 5) MST in Ω and M_o belongs to the j-th (j = 1, 2, 3) MOT in Ω_o, the comprehensive infection based indicator of M is defined as:

$$I(M) = \alpha_1 \cdot Se(M) + \alpha_2 \cdot (\xi_i(M_{s'}) + \zeta_j(M_o)), \tag{8}$$

where α_k (k = 1, 2) is the weight coefficient. Note that since the three infection factors are equally important and $\xi_i(M_{s'})+\zeta_j(M_o)\in(0, 1)$, we set α_k=1/2 (k = 1, 2) in this article.

For $I(M)$, the larger $I(M)$, the harder it is to be infected (the harder it is to be killed).

The following illustrates the specific steps for calculating $I(M)$ based on the mutant M (as shown in Fig. 3(b)) from Example 1.

(1) Calculate $Se(M)$ (Formula (7)) according to Jaccard index. For the original statement s ("if(c<a)" from P) and the mutated statement s' ("if(c<++a)" from M), we first obtain the vector TF=[[1 1 0 1 1 1 1 1], [1 1 2 1 1 1 1 1]] after giving the character vector ["(", ")", "+", "<", "a", "c", "f", "i"] from s and s' based on $CountVector$ in the $Sklearn$ library. Then, we employ $Numpy$ to get $|s \bigcap s'| = 7$ and $|s \bigcup s'| = 9$ with the vector TF. Finally, we use Formula (7) to calculate $Se(M)$ and obtain $Se(M) \approx 0.78$.

(2) Obtain $\xi_i(M_{s'})$ ($i\in$\{1, 2, ⋯, 5\}) for M and $\zeta_j(M_o)$ ($j\in$\{1, 2, 3\}) for M based on Table 3. For the mutated statement s' ("if(c<++a)" from M), we can obtain $\xi_2(M_{s'}) = 0.25$ according to Table 3. Furthermore, we can get $\zeta_1(M_o) = 0.05$ according to Table 3 when the mutation operator M_o for M is o_{one}.

(3) Calculate $I(M)$ based on Formula . Given $\alpha_i = 1/2$ (i = 1, 2), the specific process of calculating $I(M)$ is as follows:

$$I(M) = \frac{1}{2} \times 0.78 + \frac{1}{2} \times (0.25 + 0.05) = 0.54.$$

3.2.3. Construction of the integrated indicator for identifying stubborn mutants

Given $A(M)$ and $I(M)$, the integrated indicator $S(M)$ is defined as:

$$S(M) = \beta_1 \cdot A(M) + \beta_2 \cdot I(M), \tag{9}$$

where β_i ($i = 1, 2$) is the weight. Note that β_1 and β_2 are set as follows:

$$\begin{cases} \beta_1 = 0.1 < \beta_2 = 0.9 & \text{A(M)} > 0.5 \\ \beta_1 = \beta_2 = 0.5 & \text{A(M)} \le 0.5 \end{cases}$$

For $S(M)\in[0, 1]$, the larger $S(M)$, the harder it is for M to kill.

Note that the main considerations for setting β_1 and β_2: (1) When $A(M) > 0.5$, $I(M)$ can play a larger role than $A(M)$ in deleting those mutants that are easy to infect; (2) When $A(M)\le0.5$, $A(M)$ and $I(M)$ are considered to identify stubborn mutants not recognized by $A(M)$.

After giving $S(M)$, we set the threshold value η for $S(M)$ to select the stubborn mutants. For a mutant M, we treat M with $S(M) > \eta$ as a stubborn mutant. $\eta = 1/2$ is more appropriate in this article.

The following illustrates the specific steps for calculating $S(M)$ based on the mutant M (as shown in Fig. 3(b)) from Example 1.

(1) Calculate $S(M)$ according to Formula . $A(M) = 0.92$ and $I(M) = 0.36$ are obtained by Formulas (6) and . After that, $S(M)$ is calculated by Formula when $\beta_1 = 0.1$ and $\beta_2 = 0.9$ as follows:

$$S(M) = 0.1 \times 0.92 + 0.9 \times 0.54 \approx 0.58.$$

(2) After obtaining $S(M)$, we select stubborn mutants based on η. For example, when $\eta = 1/2$ and $S(M) \approx 0.58 > 1/2$, M is a stubborn mutant.

3.3. Test data generation based on set evolution

After identifying stubborn mutants, this section presents a method of generating a test suite that kills stubborn mutants. Although several methods (Section 2.2) have been proposed to generate test data for killing stubborn mutants, they often require repeated execution of source programs and mutants, resulting in higher costs for mutation testing. Therefore, we propose a set evolution based approach that aims to generate a test suite that kills all stubborn mutants at once.

3.3.1. Construction of mathematical model for generating test data

Suppose that $X = \{x_1, x_2, \dots, x_d\} \in D$ is the input of program P, where D is the input domain. Given the set of stubborn mutants M_{stub}, we intend to generate a test suite that can kill all stubborn mutants in M_{stub}. If only one mutant is killed at a time, then the mathematical model of killing the i-th mutant based on EOM (Section 2.2) is established as follows:

$$\min(\max) \quad f_i(X)$$
$$s.t. \begin{cases} g_i(X) \\ X \in D, \end{cases}$$

where $f_i(X)$ is the i-th target function and $g_i(X)$ is the i-th constraint function (i = 1, 2, ⋯, $|M_{stub}|$).

For multiple stubborn mutants, we repeat multiple calculations when generating test data based on EOM (Section 2.2), which leads to a high time cost. Therefore, we present a mathematical model for generating a test suite of killing all stubborn mutants.

The appropriate objective function is crucial for the mathematical model. In this article, we mainly consider two aspects: (1) From the adequacy of a test suite. If a test suite is more sufficient, it kills more stubborn mutants; (2) Considering the difference between the mutant and the source program (Yao and Gong, 2012). We evaluate the quality of the test suite from path coverage. Below, we provide two objective functions from these two aspects in sequence.

The test suite is denoted as $\overline{X}=\{X_1, X_2, \cdots, X_n\}$, where $X_i\in D$ is the i-th test datum and n is the number of test data in $\overline{X}$ (i = 1, 2, ⋯, n).

- **Mutation score.** To evaluate the quality of $\overline{X}$, we use mutation score (Section 2.1) as the first objective function $f_{total}(\overline{X})$, which is defined as:

$$f_{total}(\overline{X}) = \frac{N_{kill}(\overline{X})}{|M_{stub}|}, \tag{10}$$

where $N_{kill}(\overline{X})$ is the number of stubborn mutants killed by $\overline{X}$. $f_{total}(\overline{X}) \in [0, 1]$, and the larger $f_{total}(\overline{X})$, the better $\overline{X}$.

- **Path coverage.** Assuming that S_{stub} is the set of mutated statements for M_{stub}. We first select a set of paths that are easily traversed and contain all mutated statements in S_{stub}, denoted as $Path(M_{stub})$. Then, the second objective function is defined as:

$$f_{path}(\overline{X}) = \frac{|Path(\overline{X})|}{|Path(M_{stub})|}, \tag{11}$$

where $|Path(\overline{X})|$ is the number of the paths covered by $\overline{X}$ in $Path(M_{stub})$. Obviously, $f_{path}(\overline{X}) \in [0, 1]$. The larger $f_{path}(\overline{X})$, the more target paths covered (the more stubborn mutants killed by the test suite). Specifically, the generated test suite killed all stubborn mutants when $f_{path}(\overline{X}) = 1$.

Although we generate a test suite $\overline{X}$ based on Formula (10) and (11), $\overline{X}$ has high redundancy. Therefore, we provide two constraints to reduce the redundancy of $\overline{X}$.

Considering the differences between different test data in $\overline{X}$, we provide a method to measure the spatial similarity of $\overline{X}$, and use it to constrain $\overline{X}$.

- **Spatial similarity.** Given $X_i = (x_i^1, x_i^2, \cdots, x_i^d)$ $(i = 1, 2, \cdots, n)$ and $X_j = (x_j^1, x_j^2, \cdots, x_j^d)$ $(j = 1, 2, \cdots, n)$ from $\overline{X}$, the spatial similarity $S(X_i, X_j)$ between X_i and X_j is defined as:

$$S(X_i, X_j) = \frac{\sum_{t=1}^{d} \frac{1}{1+|x_i^t - x_j^t|}}{d}, \tag{12}$$

where $S(X_i, X_j) \in [0, 1]$. The larger $S(X_i, X_j)$, the more compact the spatial distribution between X_i and X_j.
Based on Formula (12), the spatial similarity of $\overline{X}$ is defined as:

$$SS(\overline{X}) = \max\{S(X_i, X_j) | i, j = 1, \ldots, n, i \neq j\}, \tag{13}$$

The larger $SS(\overline{X})$, the more concentrated the distribution of $\overline{X}$.

Furthermore, we hope that each test datum in $\overline{X}$ can kill a stubborn mutant and each test datum corresponds to each stubborn mutant one by one. For this purpose, we use the path matching degree, which is defined as:

- **Path matching degree.** Assuming that $Path_{target}$ is a target path from $Path(M_{stub})$ and $Path(X_i)$ is a actual path traversed by X_i from $\overline{X}$, we obtain that $N_{Path(X_i) \cap Path_{target}}$ is the number of identical nodes in $Path(X_i)$ and $Path_{target}$. After that, the path matching degree $MR(X_i)$ for X_i is defined as:

$$MR(X_i) = \frac{N_{Path(X_i) \cap Path_{target}}}{|Path_{target}|}, \tag{14}$$

where $i = 1, 2, \cdots, |M_{stub}|$. $MR(X_i) \in [0, 1]$, and if $Path_{target}$ and $Path(X_i)$ completely match, then $MR(X_i) = 1$.
Based on Formula (14), the matching rate $MR(\overline{X})$ of $\overline{X}$ is defined as:

$$MR(\overline{X}) = \frac{\sum_{i=1}^{|M_{stub}|} MR(X_i)}{|M_{stub}|}, \tag{15}$$

where $X_i \in \overline{X}$. For $MR(\overline{X}) \in [0, 1]$, the larger $MR(\overline{X})$, the higher the degree of matching. In particular, X_i covers a path (one-to-one correspondence) when $MR(\overline{X}) = 1$.

Based on Formula (13) and (15), we obtain the constraint conditions:

$$\begin{cases} SS(\overline{X}) \leq \delta \\ MR(\overline{X}) = 1 \\ \overline{X} \in 2^D, \end{cases} \tag{16}$$

where δ is a threshold and $\delta = 0.5$ according to Dang et al. (2020).

Finally, we provide a constrained multi-objective model of generating $\overline{X}$ based on Formula (10), (11) and (16):

$$\max \quad \{f_{total}(\overline{X}), f_{path}(\overline{X})\}$$
$$s.t. \begin{cases} SS(\overline{X}) \leq \delta \\ MR(\overline{X}) = 1 \\ \overline{X} \in 2^D, \end{cases} \tag{17}$$

In order to effectively solve the mathematical model, we will next propose a new GA based on set evolution.

3.3.2. *Generation of a test suite with set evolution*

To solve Formula (17), we propose a new GA based on set evolution. Different from the existing methods, this article adds spatial similarity, catastrophe operator and adjustment operator in generating a test suite, which aims to improve the efficiency of generating test data. Before providing the proposed method, we first provide some preparations, which mainly include the following eight components.

- **Representation of individuals.** Given a test suite $\overline{X}$ of a program P, we use binary code to represent $\overline{X}$ in our algorithm and treat it as a vector for easy implementation, which easily represents individuals in many machine languages.

Traditional GAs based on set evolution (Yao et al., 2015; Guo et al., 2017) use random method to obtain initial population, which reduces population diversity. Therefore, we introduce the spatial similarity (Section 3.3.1) to increase the diversity of the initial population.

- **Population initialization.** Given population size N_{size}, an initial population can be obtained: (1) Randomly generate a test suite $\overline{X}$ from the input field of program P; (2) Use Formula (13) to determine their spatial similarity based on $\overline{X}$. If $SS(\overline{X}) \leq \delta$, using binary code instead of $\overline{X}$ to obtain a new individual; otherwise, go to step (1); (3) Repeat steps (1) and (2) until N_{size} is reached to output the initial population.

After providing the initialization population, we need to construct an appropriate fitness function based on Formula (17) to evaluate the performance of individuals.

- **The fitness function.** To solve Formula (17), we convert it into an unconstrained single-objective optimization model by transforming two constraints from Formula (16) into two penalties ($C_1(\overline{X}) = \max\{0, SS(\overline{X})$ - $\delta\}$ and $C_2(\overline{X}) = \max\{0, MR(\overline{X})$ - $1\}$) for modifying the objective function $F(\overline{X}) = \max\{f_{total}(\overline{X}), f_{path}(\overline{X})\}$. After that, the fitness function $Fit(\overline{X})$ is defined as:

$$Fit(\overline{X}) = \lambda_1 \cdot F(\overline{X}) + \lambda_2 \cdot C_1(\overline{X}) + \lambda_3 \cdot C_2(\overline{X}), \tag{18}$$

where $\lambda_i = 1$ $(i = 1, 2, 3)$ is a weight and $\lambda_i = 1$. For $Fit(\overline{X}) \in [0, 1]$, the larger $Fit(\overline{X})$, the more stubborn mutants are killed.

Considering that the tournament selection strategy (TSS) (Zhang and Zhan, 2009) has higher accuracy and faster convergence speed, this article gives a selection operator based on TSS.

- **Selection operator.** The specific steps of the selection operator are as follows: (1) Select $N_{size}/2$ individuals from the population each time; (2) Calculate the fitness value of each individual of the $N_{size}/2$ individuals, and select the optimal individual to enter the offspring population; (3) Repeat (1) and (2) N_{size} times to obtain a new population.

A random crossover with blindness and randomness leads to poor results. Therefore, we give a set crossover operator based on competition between parents and offspring (Ye et al., 2009) to avoid the destruction of excellent individuals.

- **Set crossover operator.** Given m parent individuals, the specific steps for the set crossover operator are as follows: (1) Implement the crossover between different parent individuals to obtain A_m^2; (2) Implement the crossover between parent individuals and their complementary gene strings ($2m$) to obtain $A_m^2 + 3m$ individuals (m parents and $A_m^2 + 2m$ offspring); (3) m individuals with higher fitness values from $A_m^2 + 3m$ individuals are retained as the new population. Note that A_m^2 is $\frac{m!}{(m-2)!}$ here.

Considering that the individual with high fitness do not mutate in the entire large region but seek optimization in the smaller neighboring region of the individual, we give a set mutation operator based on Formula (18) to retain genes with high weight values (genes that have a significant impact on individual fitness) in the individuals and only mutate genes with low weight values.

- **Set mutation operator.** Given the gene length L of an individual and the weight φ_i $(i = 1, 2, \cdots, L)$ of the i-th gene, ϕ_i is defined as:

$$\varphi_i = \frac{|\overline{Fit_i(1)} - \overline{Fit_i(0)}|}{1 + \max\{|\overline{Fit_i(1)}|, |\overline{Fit_i(0)}|\}}, \tag{19}$$

where $\overline{Fit_i(1)}$ ($\overline{Fit_i(0)}$) represents the average fitness when the i-th gene is 1 or 0, and $\varphi_i \geq 0$. Note that $\overline{Fit_i(1)} = \frac{|Fit_i(1)|}{|Fit_i(1)|+|Fit_i(0)|}$ ($\overline{Fit_i(0)} = \frac{|Fit_i(0)|}{|Fit_i(1)|+|Fit_i(0)|}$) and $|Fit_i(1)|$ ($|Fit_i(0)|$) is obtained by Formula (18).
Furthermore, we set the threshold ξ to 0.5, and perform a mutation operation on the gene locus when $\varphi_i \leq \xi$ $(i = 1, 2, \cdots, L)$.

Catastrophe (Yu and Li, 2001) can increase population diversity while maintaining the same population size. Therefore, we introduce a catastrophe operator during the evolution process to avoid premature convergence by increasing population diversity.

- **Catastrophe operator.** The specific steps of the catastrophe operator are as follows: (1) Calculate $SS(\overline{X})$ ($SS(\overline{X}) \in [0, 1]$) for $\overline{X}$; (2) Use the random function $Random()$ to generate a random number R; (3) Verify whether $R \leq SS(\overline{X})$. If so, add new individuals near the optimal individual to the population; otherwise, keep the population unchanged.

To reduce the cost of set evolution, this article proposes a adjustment operator for accelerating test data generation.

- **Adjustment operator.** Given the evolution adjustment condition Δ, if Δ is met, the adjustment operator is performed: (1) Change the position of each test datum to obtain a new individual—the test data of killing different stubborn mutants from the original individual will be sequentially placed in the front position (the gene length of them is denoted as L_f), and other test data (the gene length of them is denoted as L_b) will be sequentially placed in the back position to obtain a new individual (each test datum corresponds one-to-one with each stubborn mutant); (2) Replace the original best individual with the new individual ($L_f + L_b$) and fix L_f of the best individual to form a new population, while only perform crossover and mutation operations for L_b of each individual in the new population. Note that Δ in this article is $N_{kill} > 0$, where N_{kill} is the number of test data for killing stubborn mutants in the best individual.

After giving some previous preparations, we will give a new algorithm of generating a test suite for killing all stubborn mutants, as described in Algorithm 1.

Algorithm 1 SE-TDG

```
Input: The set $M_{stub}$ of the stubborn mutants
Output: The test suite $\overline{X}$
 1: Set parameters
 2: $g \longleftarrow 0$
 3: Initialize Population $P(0)$
 4: while $g < N_{max}$ do
 5:     Calculate the fitness $Fit$ from formula (11)
 6:     Get the best individual with tournament selection operator
 7:     Get $N_{kill}$ based on the binary execution
 8:     Obtain the test suite $\overline{X} = \{X_1, X_2, \cdots, X_{|M_{stub}|}\}$ by the best individual
 9:     if $N_{kill} > 0$ then
10:         for $i = 1 \rightarrow |M_{stub}|$ do
11:             if $X_i$ kills the $i$-th stubborn mutant then
12:                 Implement adjustment operator for $\overline{X}$
13:                 Obtain test suite $\overline{X}' = \{X_i, X_1, \cdots, X_{i-1}, X_{i+1}, \cdots, X_{|M_{stub}|}\}$
14:             end if
15:         end for
16:         $\overline{X} = \overline{X}'$
17:         Replace the individual corresponding to $\overline{X}$ in $P(g)$
18:     end if
19:     for $i = N_{kill} + 1 \rightarrow |M_{stub}|$ do
20:         Implement set crossover operator for $P(g)$
21:         Implement set mutation operator for $P(g)$
22:     end for
23:     if $Fit == 1$ then
24:         Output the test suite $\overline{X}$
25:         Break
26:     else
27:         $g \longleftarrow g + 1$
28:         if $R \leq SS(\overline{X})$ then
29:             for $i = N_{kill} + 1 \rightarrow |M_{stub}|$ do
30:                 Implement catastrophe operator for $P(g)$
31:             end for
32:         end if
33:         Update and obtain the population $P(g)$
34:     end if
35: end while
```

In Algorithm 1, we first set some parameters, such as the population size N_{size} and the maximum number of iterations N_{max} (Line 1). Based on this, we initialize the population $P(0)$ when the iterations g is 0 (Lines 2~3). When $g < N_{max}$, we perform steps (1)~(5) (Lines 4~35); otherwise, exit the loop to output the $\overline{X}$ (Line 35). Steps (1)~(5) are as follows: (1) Calculate fitness Fit from formula (11), get the best individual with tournament selection operator, get N_{kill} based on the binary execution, and obtain the test suite $\overline{X}$ =$\{X_1, X_2, \cdots, X_{|M_{stub}|}\}$ by the best individual (Lines 5~8). (2) Determine whether $N_{kill} > 0$, and if so, traverse M_{stub} and sequentially determine whether X_i kills the ith stubborn mutant. If so, obtain a new test suite $\overline{X}'$ =$\{X_i, X_1, \cdots, X_{i-1}, X_{i+1}, \cdots, X_{|M_{stub}|}\}$ after implementing adjustment operator for $\overline{X}$, and let $\overline{X} = \overline{X}'$ and replace the individual corresponding to $\overline{X}$ in $P(g)$ to go to step (3) until there are no remaining stubborn mutants. (Lines 9~18). Otherwise, directly go to step (3) (Line 9). (3) Perform crossover and mutation operators only for the unadjusted portion from $i = N_{kill}+1$ to $|M_{stub}|$ (Lines 19~22). (4) Judge whether $Fit == 1$. If $Fit == 1$, output $\overline{X}$ (Lines 23~26); otherwise, determine whether $R \leq SS(\overline{X})$, and if so, obtain the population $P(g)$ when g is added by 1 after implementing catastrophe operator only for the unadjusted portion from $i = N_{kill} + 1$ to $|M_{stub}|$ (Lines 27~34); otherwise, directly obtain the population $P(g)$ when g is added by 1 (Line 33~34). (5) Repeat step (1)~(4) until the stop criteria ($g \geq N_{max}$) are met, and output $\overline{X}$ (Lines 1~35).

Note that we have implemented Algorithm 1 ($SE-TDG$) on mutation testing tool $MuClipse$ 1.3 and $Python$ 3.7.4 (64-bit). A general scheme of the tool mainly includes the following two steps: (1) Obtain a set of stubborn mutants based on the proposed integrated indicator (relying on tools $MuClipse$ 1.3 and $Python$ 3.7.4); (2) Generate a test

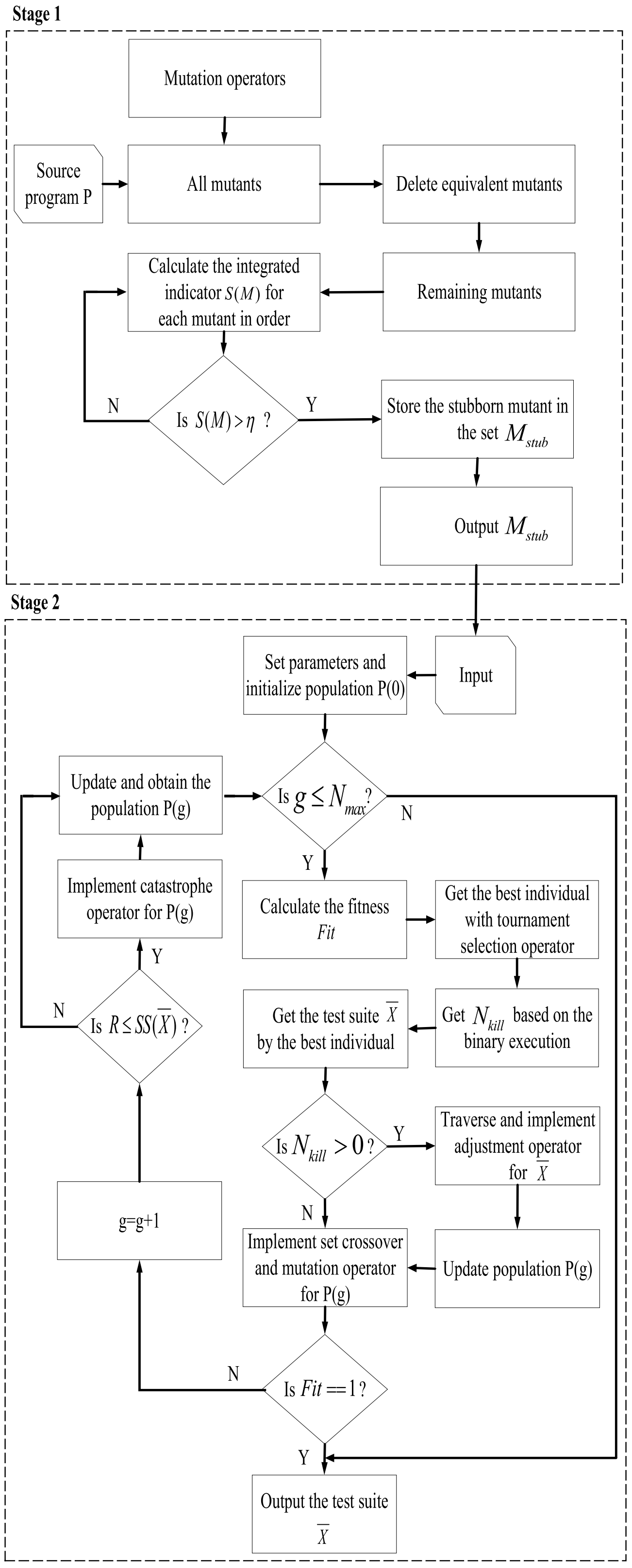

Stage 1
Mutation operators
Source program P
All mutants
Delete equivalent mutants
Remaining mutants
Calculate the integrated indicator $S(M)$ for each mutant in order
Is $S(M)>\eta$?
N
Y
Store the stubborn mutant in the set M_{stub}
Output M_{stub}
Stage 2
Input
Set parameters and initialize population P(0)
Update and obtain the population P(g)
Is $g\le N_{max}$?
N
Y
Implement catastrophe operator for P(g)
Calculate the fitness Fit
Get the best individual with tournament selection operator
Is $R\le SS(\overline{X})$?
Get the test suite $\overline{X}$ by the best individual
Get N_{kill} based on the binary execution
Is $N_{kill}>0$?
Traverse and implement adjustment operator for $\overline{X}$
g=g+1
Implement set crossover and mutation operator for P(g)
Update population P(g)
Is $Fit==1$?
Output the test suite $\overline{X}$

Fig. 4. A flow chart providing information with the different components of the approach, and valuable information about integration.

suite of killing all stubborn mutants with Algorithm 1 (dependent on tool *MuClipse* 1.3). The techniques involved in this tool mainly include the proposed technique for identifying stubborn mutants and the developed test suite generation technique (as shown in Algorithm 1). For the interaction involved in this article, we need to configure *PyDev*[2] to call execution in *MuClipse* 1.3 when executing the python programming language. To better understand the entire process of implementing our framework, we have provided a flow chart with two stages, as shown in Fig. 4. In addition, both the code and the results are available by the database *Mutation-data*.[3]

The following illustrates the specific steps for generating $\overline{X}$ based on Example 1.

(1) According to Formula and $\eta = 1/2$, we get a set M_{stub} containing three stubborn mutants $\{M_{24}, M_{26}, M_{27}\}$ from Table 1. Note that we obtain $S(M_{24}) \approx 0.58$, $S(M_{26}) \approx 0.62$, and $S(M_{27}) \approx 0.62$, all of which are greater than $1/2$.

(2) After providing M_{stub}, we use Algorithm 1 to generate $\overline{X}$ for killing all stubborn mutants in M_{stub}. The specific steps of generating $\overline{X}$ are shown in steps (3) to (8).

(3) Before using Algorithm 1, we set some parameters, such as $N_{max} = 3000$, the gene length L of each variable is 8, the number of the variables (a, b and c) is 3, and the input domain for program P is $[-10, 10]^3$. Note that we set $N_{size} = 3$ here for the convenience of operation.

(4) We obtain the g-th generation population $P(g)$ based on binary encoding when $g = 0$ and $g < N_{max}$, we go to step (5); otherwise, exit the loop to output the result. For example, we obtain $\{\overline{X}_1, \overline{X}_2, \overline{X}_3\}$ by binary decoding $P(0)$, where $\overline{X}_1$ ={(8, 4, −9), (5, −9, −2), (0, 2, −3)}, $\overline{X}_2$ ={(2, −9, 8), (−1, −1, 3), (−3, 0, −9)}, $\overline{X}_3$ ={(2, 0, −9), (2, −6, −5), (7, 0, −1)}. Note that we can obtain $SS(\overline{X}_1) = 0.25$, $SS(\overline{X}_2) \approx 0.30$, and $SS(\overline{X}_3) \approx 0.16$ based on Formula (13).

(5) Calculate the fitness value by Formula (18) based on $\{\overline{X}_1, \overline{X}_2, \overline{X}_3\}$ and get the test suite $\overline{X}_3$ ={(2, 0, −9), (2, −6, −5), (7, 0, −1)} corresponding to the best individual based on the tournament selection operator. Note that we obtain three paths for reaching the mutated statement 3 from M_{24} and the mutated statement 4 from M_{26} and M_{27}, that is, $path_1 = (1, 2, 3, 7, 13)$ for M_{24} and $path_i = (1, 2, 3, 4, 6, 13)$ ($i = 2, 3$) for M_{26} (M_{27}).

(6) Obtain N_{kill} by the execution after binary decoding and determine whether $N_{kill} > 0$. If so, traverse M_{stub} and sequentially determine whether ith test datum kills the ith stubborn mutant. If so, obtain a new test suite $\overline{X}'$ ={X_i, X_1, ⋯, X_{i-1}, X_{i+1}, ⋯, $X_{|M_{stub}|}$} after implementing adjustment operator for $\overline{X}$, and let $\overline{X} = \overline{X}'$ and replace the individual corresponding to $\overline{X}$ in $P(g)$ to go to step (7) until there are no remaining stubborn mutants. For example, we get the test datum (2, −6, −5) of killing M_{26} by executing $\overline{X}_3$ ={(2, 0, −9), (2, −6, −5), (7, 0, −1)} obtained by binary decoding and $N_{kill} = 1 > 0$. Then, we will adjust test suite $\overline{X}_3$ according to the strategy and get a new suite $\overline{X}_3'$ ={(2, −6, −5), (2, 0, −9), (7, 0, −1)}. Finally, we replace the individual for $\overline{X}_3'$ with the individual for $\overline{X}_3$ in $P(g)$ to go to step (7) until there are no remaining stubborn mutants.

(7) Perform crossover and mutation operations for $P(g)$ from $i = N_{kill} + 1$ to $|M_{stub}|$ (the size of the set M_{stub}). Here, we use $\overline{X}_3'$ to illustrate the specific steps (similar to $\overline{X}_3'$ for others). For example, we easily get $N_{kill} = 1$ and $|M_{stub}| = 3$. Then, we only implement crossover operator and mutation operator on the test data $X_2 = (2, 0, -9)$ and $X_3 = (7, 0, -1)$ (except of $X_1 = ((2, -6, -5))$. For crossover operator, we obtain the two best individuals (2, 0, −9) and (7, 0, −1) from eight individuals by implementing the set crossover operator and get a final result $\overline{X}_3^{crossover}$ ={(2, −6, −5), (2, 1, −9), (7, −1, −1)}}. Similar to $\overline{X}_3'$, we get {{(8, 4, −9), (5, −5, −2), (0, 0, −3)}, {(2, −9, 8), (−1, 0, 3), (−3, −1, −9)}, {(2, −6, −5), (2, 1, −9), (7, −1, −1)}}. Afterwards, we implement set mutation operator on $\overline{X}_3^{crossover}$ to obtain a final result $\overline{X}_3^{mutation}$ ={(2, −6, −5), (2, −1, −9), (7, 1, −1)}}. Similar to $\overline{X}_3^{crossover}$,

[2] http://www.pydev.org/updates
[3] https://github.com/wcq1536113693/Mutation-data

we get {{(8, 4, −9), (5, −6, −2), (0, 1, −3)}, {(2, −9, 8), (−1, 1, 3), (−3, 1, −9)}, {(2, −6, −5), (2, −1, −9), (7, 1, −1)}}.

(8) Judge whether $Fit == 1$. If so, we output $\overline{X}$; otherwise, determine whether $R \le SS(\overline{X}_i)$ (where $R \in [0, 1)$ is a random number and i = 1, 2, 3) is met. If so, implement catastrophe operator and obtain the population P(g) when g is added by 1; otherwise, directly obtain the population P(g) when g is added by 1. For example, when $R \approx 0.65$, $SS(\overline{X}_1) = 0.25 < R$, $SS(\overline{X}_2) \approx 0.30 < R$ and $SS(\overline{X}_3) \approx 0.16 < R$.

(9) Repeat step (4)~(8) until the stop criteria ($g \ge N_{max}$ or $Fit ==$ 1) are met, and output the test suite $\overline{X}$. For example, when g=3, we can get $\overline{X}$ ={(2, −6, −5), (2, −6, −1), (7, 0, 1)} and then the obtained fitness $Fit == 1$ based on $\overline{X}$. So, $\overline{X}$ is the final test suite of killing all stubborn mutants.

4. Empirical study

We conduct an empirical study to evaluate the effectiveness and efficiency of the proposed method. The experimental settings are presented in the following.

4.1. Research questions

Our experiments are designed to answer the following three research questions:

RQ1: Can TDGMSE improve the rationality and accuracy for the identification of stubborn mutants?

RQ2: How is the performance of TDGMSE in generating test data for killing stubborn mutants?

- RQ2.1: Can TDGMSE improve the effectiveness of generating test data for killing stubborn mutants?
- RQ2.2: Can TDGMSE improve the efficiency of generating test data for killing stubborn mutants?

4.2. Variables and measures

4.2.1. Independent variables

The independent variable refers to the method of generating test data. Essentially, this article chooses the proposed method of generating test data, namely TDGMSE. In addition, the other five basic technologies (Random, SGA (Yao and Gong, 2014), MGA (Yao and Gong, 2014), SDRCGA (Dang et al., 2020), TCGSE (Zhang et al., 2015), FUZGENMUT (Dang et al., 2021), and TDGSMBD (Yao et al., 2020)) are selected as comparisons for our method.

Random, as the name states, is to randomly sample the input space to generate a large number of test data for killing non-equivalent mutants. Random is often used as the most fundamental comparison method in previous studies (Liu and Li, 1987).

SGA (Yao and Gong, 2014) is a single population genetic algorithm that is used to generate test data, mainly including roulette wheel selection, single-point crossover, and single-point mutation. MGA (Yao and Gong, 2014) is an extended method of SGA, which divides a population into several isolated sub-populations, evolves within these subpopulations, and allows individuals to migrate from one subpopulation to another.

SDRCGA (Dang et al., 2020) is a search domain reduction-based coevolutionary genetic algorithm of generating test data, which is used to generate test data of killing stubborn mutants. The main idea of SDRCGA is as follows: (1) Providing a method for identifying stubborn mutants based on accessibility; (2) Transforming the problem of generating test data to kill mutants into a single objective optimization problem with unique constraints; (3) Generating test data using co-evolutionary genetic algorithm.

TCGSE (Zhang et al., 2015) is a test data generation method based on mutation analysis and set evolution and the specific steps of TCGSE are as follows: (1) Construct mutation branches and integrate all mutation branches into the original program to form a new program; (2) Establish a new mathematical model for the test case generation problem; (3) The set evolutionary optimization method is used to solve the above model.

FUZGENMUT (Dang et al., 2021) makes use of fuzzy clustering and multi-population genetic algorithm (MGA) to improve the effectiveness and efficiency of mutation testing from two perspectives. The main steps of FUZGENMUT are as follows: (1) Use fuzzy clustering to classify mutants into different clusters; (2) Construct a fitness function and transform the test data generation problem into an optimization problem; (3) Apply MGA to generate test data to kill mutants in different clusters in parallel.

TDGSMBD (Yao et al., 2020) is a method of generating test data based on dominance degrees of the mutant branches, which enhances the fault-detection capability of test data. The main steps of TDGSMBD are as follows: (1) Transform the problem of weak mutation testing into the problem of covering the mutant branches; (2) Obtain non-dominant mutant branches after analyzing the dominance relationship of mutant branches; (3) Prioritize all non dominant mutant branches in descending order based on their degree of dominance and generate test data in an orderly manner by sequentially selecting mutant branches.

4.2.2. Dependent variables

The dependent variables in this study are mainly metrics used to measure the technical performance of the study.

For RQ1, we evaluate the performance of TDGMSE from both rationality and accuracy. To evaluate the rationality of TDGMSE, we first provide an important indicator $DE_k(M)$ (Dang et al., 2020) to evaluate the stubbornness of a mutant M. The calculation of $DE_k(M)$ is based on a certain amount of test data that are randomly selected from the input domain of the tested program. More specifically, $DE_k(M)$ can be calculated as follows:

$$DE_k(M) = \frac{T^k_{stub}(M)}{T_{total}} \times 100\%, \tag{20}$$

where T_{total} denotes total number of random test data, and $T^k_{stub}(M)$ is the number of test data that kill M. For $DE_k(M)$, the smaller $DE_k(M)$, the harder it is to kill M. Note that $DE_k(M)$ remains stable when the number of test data reaches a certain value. In this study, following previous work (Dang et al., 2020), we consider those mutants with $DE_k(M) < 25\%$ as stubborn mutants.

Then, we use the pearson correlation coefficient (PCC) (Dang et al., 2020) between $DE_k(M)$ and $S(M)$ or $A(M)$ to quantitatively measure the rationality of our method, as compared against SDRCGA (Dang et al., 2020). Note that $PCC \in [-1, 1]$ is used to measure the correlation (linear correlation) between two variables X and Y (Dang et al., 2020), $S(M)$ (as defined in Section 3.2.3) is the indicator for identifying stubborn mutants from TDGMSE and $A(M)$ (as defined in Section 3.2.1) is the indicator for identifying stubborn mutants from SDRCGA. For a method, if its PCC is closer to −1 according to reference, we can consider it more reasonable.

To evaluate the accuracy of TDGMSE, we compare TDGMSE ($S(M)$) with SDRCGA ($A(M)$) (Dang et al., 2020) based on FPR and FNR (Song and Kwon, 2006). False positive rate (FPR) and false negative rate (FNR) have been commonly used in different areas to evaluate the performance of a method. Specifically, FPR is calculated as:

$$FPR = \frac{FP}{FP + TN} \times 100\%, \tag{21}$$

where FP is the number of stubborn mutants incorrectly identified by the method, and TN is the number of non stubborn mutants correctly identified by this method. For FPR, the smaller the value of FPR, the more accurate the identification of stubborn mutants.

Similarly, FNR is calculated as follows:

$$FNR = \frac{FN}{TP + FN} \times 100\%, \tag{22}$$

Table 4
Subject programs and mutants.

ID	Program	LOCs	Methods	Mutants		Description
				Total	Equivalent	
J1	TrashAndTakeOut	30	2	111	29	Basic arithmetic operations
J2	Mid	26	1	115	18	Return the middle value of three integers
J3	FourBalls	28	1	213	49	Relative weight of four spheres
J4	Triangle	36	1	325	40	Return the type of a triangle with three integer inputs
J5	Cal	46	2	314	43	Calculate the days between two dates in the same year
J6	VendingMachine	120	4	433	42	Simulate a vending machine which vends a small number of products
J7	Precision	301	21	785	34	Utilities for comparing numbers
J8	NumberUtils	636	47	1406	212	Provides extra functionality for Java Number classes
J9	Dfp	1702	113	2133	258	Decimal floating point library for Java
J10	FastMath	2311	100	6486	976	Faster, more accurate, portable alternative to Math and StrictMath for large scale computation
J11	ArrayUtils	3258	319	7551	795	Operations on arrays, primitive arrays (like int[]) and primitive wrapper arrays
J12	Jdkmath	11 540	173	31 576	3265	Alternative to Math and StrictMath
J13	Numbers	20 205	340	64 955	5023	Number types (complex, quaternion, fraction) and utilities (arrays, combinatorics)
J14	Colt	31 407	3106	93 985	12 749	Project for high performance scientific and technical computing
	Sum.	71 646	4230	210 388	23 533	

where TP is the number of stubborn mutants correctly identified by this method, and FN is the number of non-stubborn mutants incorrectly identified by the method. For FNR, the smaller the value of FNR, the more accurate the identification of stubborn mutants.

For RQ2.1, we use the mutation score MS_{stub} and the success rate of generating test data (R_{succ}) to evaluate the effectiveness of our method. Intuitively speaking, if a method has higher MS_{stub} and higher R_{succ} under the same condition, we say it is more effective. MS_{stub} is calculated as follows:

$$MS_{stub} = \frac{N_{stub}^{kill}}{N_{stub}} \times 100\%, \tag{23}$$

where N_{stub}^{kill} is the number of stubborn mutants killed by a test suite and N_{stub} is the total number of stubborn mutants. Intuitively, the higher MS_{stub}, the stronger the fault detection ability of the test suite.

R_{succ} is calculated as follows:

$$R_{succ} = \frac{N_{succ}}{N_{total}} \times 100\%, \tag{24}$$

where R_{succ} is the number of successfully generating test data for killing stubborn mutants, and N_{total} is the total number of runs. For R_{succ}, the higher the value of R_{succ}, the more test data of killing stubborn mutants are successfully generated.

For RQ2.2, we use the number of the required iterations for killing stubborn mutants (N_{iter}) and the execution time of generating test data ($Time$) to evaluate the efficiency of a test data generation method. Intuitively speaking, if a method has fewer iterations (N_{iter}) and less $Time$ under the same condition, we say it is more efficient in generating test data.

4.3. Object programs and mutant generation

We select 14 Java programs from different application fields as our experimental objects. Table 4 lists their basic information. Among them, five (J1 to J5) are small programs that implement basic scientific computing (Zhang et al., 2015, 2010); The other nine programs (J6 to J14) are some popular utilities (Yao et al., 2020; Fraser and Zeller, 2012), which can be downloaded from the website: http://commons.apache.org. We use a popular mutation testing tool, $MuClipse$ (Ma et al., 2005), which is the MuJava plugin on Eclipse (Chen and Gu, 2012), to generate mutants for all 14 Java programs. In generating mutants, we use 15 method-level mutation operators in $MuClipse$. Columns 5 to 6 of Table 4 provide information on mutants.

4.4. Experimental procedure

4.4.1. Experimental step

Our experiments are conducted on a machine with Intel(R) Core(TM) i5-3230M CPU @ 2.60 GHz, 4.00 GB ROM, Windows 8 Operating System, Eclipse SDK 4.2.2 with $MuClipse1.3$ and Python 3.7.4(64-bit). $MuClipse$ is an Eclipse plug-in for MuJava that can automatically generate and execute mutants based on test data.

Table 5
The parameters used by TDGMSE and other four techniques.

Parameters	Parameter values	Parameter description
N_{size}	10	Population size
p_m	0.9	Mutation probability
p_c	0.3	Crossover probability
N_{best}	5	Number of excellent individuals
$N_{sub-size}$	3	Sub-population size
N_{max}	3000	Maximum number of iterations
δ	0.5	Threshold of spatial similarity

Before conducting the experiment, we first execute all 15 method-level mutation operators of $MuClipse$ for each tested program in Table 4 and generate the corresponding mutants for each program. Then, we use the same method as in Yao et al. (2020) to remove the equivalent mutants from all mutants: (1) $MuClipse$ (Ma et al., 2005) is firstly used to run all mutants on the test suites generated by Randoop[4] (Pacheco and Ernst, 2007) and obtain live mutants; (2) Artificial analysis is used to determine whether they are equivalent or not for live mutants. Note that the reason for using manual analysis is that not all live mutants are equivalent mutants after executing all the test suites obtained by Randoop, which requires manual analysis to further identify equivalent mutants to ensure the accuracy of the results.

After that, we provide the specific experimental process as follows:

For RQ1, we apply TDGMSE and SDRCGA (Dang et al., 2020) to identify stubborn mutants. On the one hand, we can prove that TDGMSE is more reasonable by comparing PCC (Dang et al., 2020) based on $DE_k(M)$. To obtain PCC, we perform the following steps: (1) $DE_k^S(M)$ ($DE_k^A(M)$) and $S(M)$ ($A(M)$) from TDGMSE (SDRCGA) are applied to 14 tested programs, and each indicator is tested 20 times for each program to obtain the average of each indicator; (2) PCC is automatically obtained by the statistical tool SPSS. On the other hand, we compare TDGMSE ($S(M)$) with SDRCGA ($A(M)$) from FPR (FNR) (Song and Kwon, 2006) to prove that TDGMSE is more accurate and the FPR (FNR) (Song and Kwon, 2006) is applied to 14 tested programs, and each indicator is tested 20 times for each program to obtain the average of each indicator.

For RQ2.1, all eight baseline techniques (Random, SGA (Yao and Gong, 2014), MGA (Yao and Gong, 2014), SDRCGA (Dang et al., 2020), TCGSE (Zhang et al., 2015), FUZGENMUT (Dang et al., 2021), TDGSMBD (Yao et al., 2020) and TDGMSE) are applied to 14 tested

4 https://randoop.github.io/randoop/

programs, and each technique is tested 20 times for each program to obtain the average of the experimental results. In addition, we conduct a T-test and ANOVA on the results (based on SPSS) to determine whether there are significant differences between TDGMSE and the other five baseline technologies. Note that the seven methods (SGA, MGA, SDRCGA, TCGSE, FUZGENMUT, TDGSMBD and TDGMSE) in the experiment need to set some parameters (as shown in Table 5) to help the smooth progress of the experiment.

For RQ2.2, all six baseline techniques are applied to obtain $Time$, while Random method cannot be considered in the experiments for N_{iter}. Each program will be tested by each technique for 20 times to obtain the average values of N_{iter} and $Time$.

4.4.2. Evaluation procedure and metrics

For RQ1, we apply TDGMSE and SDRCGA (Dang et al., 2020) for identifying stubborn mutants. We first use PCC (in Section 4.4.1) (Dang et al., 2020) to demonstrate whether TDGMSE is more reasonable than SDRCGA. Then, we use FPR (FNR) (Song and Kwon, 2006) to prove that TDGMSE is more accurate and FPR (FNR) is applied to 14 tested programs, and each indicator is tested 20 times for each program to obtain the average of each indicator.

In order to answer RQ2, we set up two sub-questions (RQ2.1 and RQ2.2) to compare the performance of the proposed method from different perspectives.

For RQ2.1, we compare the performance of TDGMSE with other seven benchmark techniques (Random, SGA (Yao and Gong, 2014), MGA (Yao and Gong, 2014), SDRCGA (Dang et al., 2020), TCGSE (Zhang et al., 2015), FUZGENMUT (Dang et al., 2021), and TDGSMBD (Yao et al., 2020)) in effectively generating test data of killing stubborn mutants from MS_{stub} and R_{succ}. We apply the six techniques to 14 tested programs, and each technique for each tested program is repeated 20 times to get a mean value of MS_{stub} (or R_{succ}). In addition, the statistical tool SPSS is used for T-tests of TDGMSE and other four benchmark methods to determine whether there is a significant difference in MS_{stub}.

For RQ2.2, we evaluate the performance of TDGMSE in the efficiency of test data generation from N_{iter} and $Time$. We apply our method and the other five benchmarks (SGA, MGA, SDRCGA, TCGSE, and FUZGENMUT) to 14 tested programs, and each tested program can be repeated 20 times to get the mean value of N_{iter}. Furthermore, we evaluate the performance of TDGMSE by comparing the other seven benchmark techniques (Random, SGA, MGA, SDRCGA, TCGSE, FUZGENMUT and TDGSMBD) in terms of the time cost in this article, and each technique on each tested program is repeated 20 times to get the mean value of $Time$.

4.5. Threats to validity

4.5.1. Threat to internal validity

The main threat to internal effectiveness is related to the implementation of the proposed method, which requires a certain degree of programming work. For example, the results obtained based on indicators of stubborn mutants are influenced by equivalent mutants, and we propose a semi-automatic method to eliminate equivalent mutants for reducing the impact of equivalent mutants on stubborn mutant indicators. In addition, the source code that implements all test data generation techniques has been cross-checked by different personnel to minimize programming errors.

4.5.2. Threat to external validity

The main threat to external effectiveness is related to the generalization of our results. In this study, although we have selected projects of different scales and fields, we still cannot say that the conclusions drawn can be generalized to any type of project. The production process of mutants is also limited to certain selected methods, which may also affect the universality of research. In addition, we have obtained better experimental results based on the values of some weights (such as δ and ω_i ($i = 1, 2$)), but we still cannot say that the conclusions are valid for any type of project.

4.5.3. Threat to construct validity

The threat of construct validity is related to measurement. The indicators used in this article, such as the number of iterations and execution time, are very simple and have been widely used in many studies.

4.5.4. Threat to conclusion validity

Considering the randomness of certain settings (such as thresholds) in the experiment, we implement at least 20 test data generation strategies for each tested program to obtain the average of the experimental results. Based on the collected data showing a very small standard deviation, we believe that the results are statistically reliable.

5. Experimental results

5.1. Answer to RQ1: Rationality and accuracy of identifying stubborn mutants

Table 6 shows the distribution of mutants. In Table 6, columns 3 and 4 provide the number of stubborn mutants and their proportion in the non-equivalent mutants, i.e., a total of 186855 stubborn mutants accounting for 15.44% of all non-equivalent mutants. To calculate $DE_k^A(M)$ (or $DE_k^S(M)$), we sample T_s test data from the test input domain of the tested program (as shown in column 5 of Table 6), such as J1∼J4 sample 5000 test data and J5∼J14 sample 10000 test data. Considering that the average value of samples is an unbiased estimate of the overall average based on statistics, we use the average value of $DE_k^A(M)$ (or $DE_k^S(M)$) and the average value of $A(M)$ from SDRCGA (or $S(M)$ from TDGMSE) to reflect its central tendency. From columns 6∼9 in Table 6, we obtain the following results: (1) The average value of $DE_k^S(M)$ from TDGMSE for each tested program is less than 20% and the average value of $S(M)$ from TDGMSE for each tested program is higher than 0.55; (2) The average value of $DE_k^S(M)$ from TDGMSE for 14 tested programs is 9.84% less than 12.98% (The average value of $DE_k^A(M)$ from SDRCGA), which indicates that the stubborn mutants obtained by TDGMSE are killed by less test data than the stubborn mutants obtained by SDRCGA under the same test suite (The less test data of killing a mutant in the test suite, the more stubborn the mutant becomes).

In addition, the Pearson analysis on SPSS is used to obtain PCCs for TDGMSE and SDRCGA based on columns 6∼9 in Table 6 and the results are shown in Table 7. Note that N_{sample} is the total number of the tested programs in Table 7. From Table 7, we obtain the following results: (1) The PCC for TDGMSE is −0.92 (closer to −1)<−0.82 (SDRCGA), which indicates TDGMSE has a stronger negative correlation than SDRCGA; (2) The p values (bilateral) for TDGMSE and SDRCGA are both 0.00<0.05, which implies the significant differences in statistical results. From the above, we can obtain that TDGMSE is more reasonable than SDRCGA (The more reasonable the indicator is, the more effective it is in identifying stubborn mutants).

Table 8 mainly presents the comparison results of FPR and FNR between the $S(M)$ and the $A(M)$ (Dang et al., 2020). From Table 8, we obtain the following results: (1) The average value of FPR (column 4) from $S(M)$ for each of the 14 tested programs is less than the average value of FPR (column 2) from $A(M)$, and the average value of FNR (column 5) from $S(M)$ for each of the 14 tested programs is less than the average value of FNR (column 3) from $A(M)$; 2) The average value of FPR (FNR) for $S(M)$ is 8.40% (10.33%) less than the average value of FPR (FNR) for $A(M)$, which also indicates that $S(M)$ is more accuracy in identifying stubborn mutants.

In summary, TDGMSE can improve the rationality and accuracy of identifying stubborn mutants.

Table 6
Distribution of the mutants.

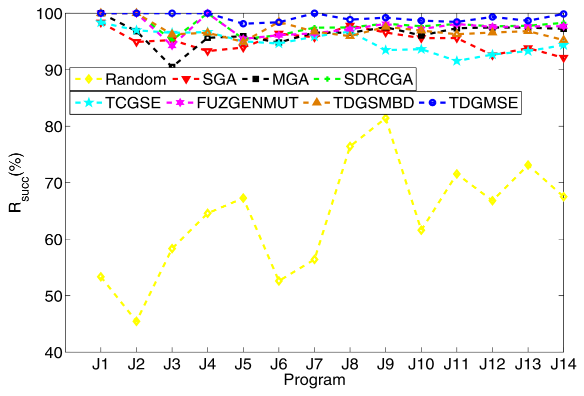
ID	N_{non}	Stubborn mutants		T_s	SDRCGA (Dang et al., 2020)		TDGMSE	
		Number	Ratio		$DE_k^A(M)$	$A(M)$	$DE_k^S(M)$	$S(M)$
J1	82	3	3.66%	5000	7.34%	0.72	4.34%	0.71
J2	97	11	11.34%	5000	16.34%	0.63	14.34%	0.61
J3	164	21	12.8%	5000	9.94%	0.79	7.94%	0.70
J4	285	23	8.07%	5000	19.94%	0.69	12.94%	0.65
J5	271	21	7.75%	10 000	9.47%	0.80	6.47%	0.71
J6	391	77	19.69%	10 000	5.57%	0.84	3.57%	0.76
J7	751	117	15.58%	10 000	11.47%	0.67	9.47%	0.63
J8	1194	318	26.63%	10 000	15.13%	0.75	6.13%	0.71
J9	1875	323	17.23%	10 000	11.57%	0.72	14.57%	0.67
J10	5510	1041	18.89%	10 000	18.37%	0.61	15.37%	0.58
J11	6756	1353	20.03%	10 000	9.23%	0.77	5.23%	0.71
J12	28 311	5243	18.52%	10 000	21.49%	0.60	18.49%	0.57
J13	59 932	6135	10.24%	10 000	16.75%	0.62	14.75%	0.61
J14	81 236	14 157	17.43%	10 000	9.17%	0.78	4.17%	0.72
Sum.	186 855	28 843	15.44%	120 000	Ave.12.98%	0.71	9.84%	0.67

Table 7
PCCs of TDGMSE and SDRCGA (Dang et al., 2020).

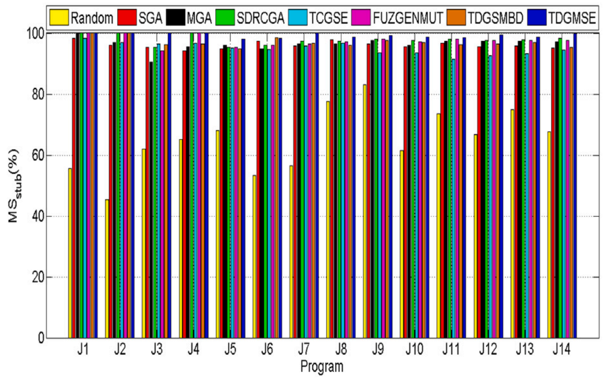
N_{sample}	SDRCGA (Dang et al., 2020)		TDGMSE	
	PCC	p-value	PCC	p-value
14	−0.82	0.00	−0.92	0.00

Table 8
The comparison results for TDGMSE and SDRCGA (Dang et al., 2020).

ID	SDRCGA		TDGMSE	
	FPR (%)	FNR (%)	FPR (%)	FNR (%)
J1	2.70	70.00	1.30	50.00
J2	8.11	52.17	3.66	38.46
J3	12.38	52.54	3.65	36.00
J4	4.98	57.81	2.37	46.88
J5	6.37	61.19	2.12	51.52
J6	15.63	41.75	3.62	24.14
J7	9.54	31.47	2.72	21.26
J8	16.31	17.38	2.20	9.94
J9	15.42	16.74	1.36	9.31
J10	19.10	12.97	0.70	4.81
J11	14.94	12.92	0.95	5.31
J12	8.79	4.90	0.48	1.63
J13	4.10	6.92	0.20	1.58
J14	5.23	8.38	0.76	1.69
Ave.	10.26	31.94	1.86	21.61



(a)



(b)

Fig. 5. The comparison results of effectiveness for TDGMSE and other seven baseline techniques.

5.2. Answer to RQ2: Performance of TDGMSE

5.2.1. Answer to RQ2.1: Effectiveness of TDGMSE

Fig. 5 presents the comparison results of effectiveness for TDGMSE and the other seven baseline techniques (Random, SGA, MGA, SDRCGA, TCGSE, FUZGENMUT, and TDGSMBD) in terms of the average value of R_{succ} and the average value of MS_{stub}.

From Fig. 5, we observe that: (1) From Fig. 5(a), the R_{succ} (blue dotted line) of TDGMSE for each of 14 subject programs is higher than those of the other seven baseline techniques and Random performs worst in all programs; (2) From Fig. 5(b), the MS_{stub} (blue cylinder) obtained by TDGMSE for each of 14 subject programs is higher than those of the other seven baseline techniques, while Random performs worst and SDRCGA is second only to TDGMSE in all programs.

Furthermore, we present some statistics of MS_{stub} for the eight methods, as shown in Table 9. we can see that: (1) The maximum value of MS_{stub} for TDGMSE and the maximum value of MS_{stub} for SDRCGA are both 100% and the minimum value of MS_{stub} for TDGMSE is 98.14%, which is higher than the seven benchmark methods. (2) The average MS_{stub} of TDGMSE is 99.26%, which is also higher than the other seven methods, while Random has the lowest mean value of MS_{stub}. (3) The standard deviation and variance of TDGMSE are 0.71 and 0.51, respectively, lower than those of the other seven methods. In a word, these statistical indicators reinforce that TDGMSE is better than the other seven methods in terms of the effectiveness of generating test data to kill stubborn mutants.

Based on MS_{stub}, we also use SPSS for T-test (T-test is a method used to test the degree of difference between two means of a small sample) to verify whether TDGMSE is more significant than the other seven baseline techniques and the T-test results are shown in Table 10.

Table 9
Statistics of the T-test on MS_{stub} (%) for TDGMSE and other seven methods.

Methods	Samples (No.)	Minimum	Maximum	Sum	Mean value	Standard deviation	Variance
Random	14	45.45	82.97	911.08	65.08	10.31	106.31
SGA	14	94.30	98.35	1344.91	96.07	1.15	1.33
MGA	14	90.48	100.00	1349.86	96.42	2.09	4.37
SDRCGA	14	95.24	100.00	1369.35	97.81	1.52	2.32
TCGSE	14	91.54	98.35	1329.66	94.98	1.94	3.77
FUZGENMUT	14	94.24	100.00	1365.71	97.55	1.70	2.89
TDGSMBD	14	94.91	100.00	1358.23	97.02	1.53	2.36
TDGMSE	14	98.14	100.00	1389.62	99.26	0.71	0.51

Notes: Standard deviation (or variance) describes the degree of clustering of data points around the mean. The smaller the standard deviation (or variance), the more concentrative the distribution of observed values (the better the test data generated by a method).

Table 10
T-test results of comparing TDGMSE and other seven methods.

Comparison methods	Paired difference				p-value (Bilateral)
	Mean value	Standard deviation	95% confidence interval		
			Lower limit	Upper limit	
TDGMSE vs. Random	34.18	10.62	28.05	40.31	0.00
TDGMSE vs. SGA	3.19	1.45	2.35	4.03	0.00
TDGMSE vs. MGA	2.84	2.23	1.55	4.13	0.00
TDGMSE vs. SDRCGA	1.45	1.32	0.69	2.21	0.00
TDGMSE vs. TCGSE	4.28	1.63	3.34	5.22	0.00
TDGMSE vs. FUZGENMUT	1.71	1.59	0.79	2.62	0.00
TDGMSE vs. TDGSMBD	2.24	1.48	1.39	3.09	0.00

Notes: The smaller mean value (standard deviation) of difference, the greater the difference between the comparison methods. The smaller the 95% confidence interval, the more significant the difference between comparison methods. p-value < 0.05 indicates that the mean difference between two paired groups is statistically significant.

Table 11
Statistics of the ANOVA test on MS_{stub} (%) for TDGMSE and other seven methods.

	Mean value	Standard deviation	Variance
Random	65.08	10.31	106.31
SGA	96.07	1.15	1.33
MGA	96.42	2.09	4.37
SDRCGA	97.81	1.52	2.32
TCGSE	94.98	1.94	3.77
FUZGENMUT	97.55	1.70	2.89
TDGSMBD	97.02	1.53	2.36
TDGMSE	99.26	0.71	0.51
Total	93.02	20.97	123.85

The p-value in the last column of Table 10 is less than 0.05 for each pair of comparison (such as TDGMSE vs. Random), which imply the significant difference between TDGMSE and the other seven benchmark methods. In other words, the average MS_{stub} obtained by TDGMSE is significantly higher than the other seven methods in the case of the same sample.

In addition, we also use SPSS for the ANOVA test to verify whether TDGMSE is more significant than the other seven baseline techniques and the obtained results are shown in Table 11. From Table 11, we can obtain the mean value, variance, and standard deviation of the results obtained during 10 executions of each method. Table 12 shows the results of the ANOVA test, and the values of p and F represent the significance of the results. From Table 12, we can obtain that the ratio of f is 116.76 and the p-value is $0.00 < 0.05$, which the average MS_{stub} obtained by TDGMSE is significantly higher than the other seven methods in the case of the same sample.

In summary, TDGMSE can generate test data of killing stubborn mutants with a higher success rate and a higher fault-detection capability than those of the other seven benchmark methods under the same condition. This means that TDGMSE has much higher effectiveness than the other seven baseline techniques.

Table 12
The results of the ANOVA test on MS_{stub} (%) for TDGMSE and other seven methods.

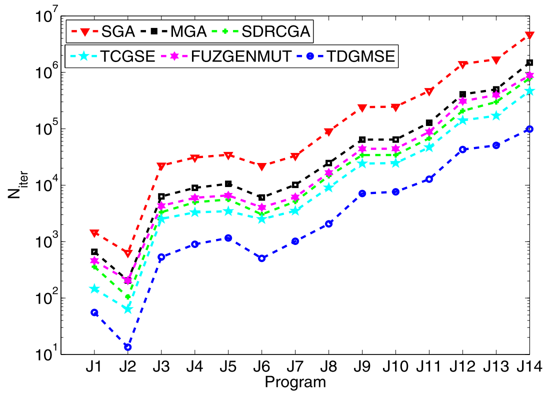
Source	Sum of squares	*df*	Mean square	
Between-treatments	12 653.48	7.00	1807.64	$F = 116.76$
Within-treatments	1610.04	104.00	15.48	
Total	14 263.52	111.00		

5.2.2. *Answer to RQ2.2: Efficiency of TDGMSE*

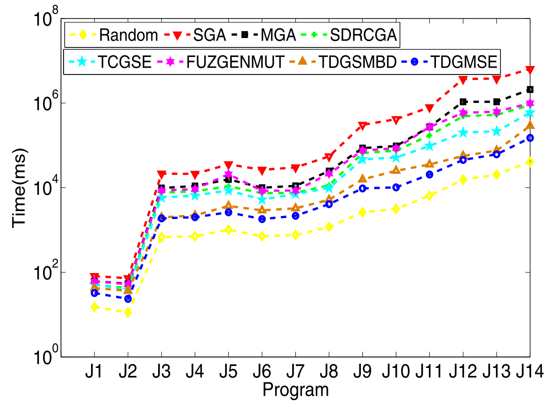
Fig. 6 summarizes the results of efficiency for TDGMSE and the other seven baseline techniques in terms of the average number of iterations (N_{iter}) and the average execution time of generating test data ($Time$).

From Fig. 6, we observe that: (1) From Fig. 6(a), the N_{iter} (blue dotted line) obtained by TDGMSE for each of the 14 subject programs is lower than those of the other five baseline techniques (SGA, MGA, SDRCGA, TCGSE and FUZGENMUT), but SGA performs worst and has very little difference from MGA in all programs; (2) From Fig. 6(b), TDGMSE (blue dotted line) is superior to the other six benchmark techniques in their average execution time and the advantage of TDGMSE in time-cost will become more apparent with the continuous increase of the size of the tested programs. For all 14 subject programs, TDGMSE spends less time generating test data on each program than the other six benchmark techniques except for Random. The reason why Random executed quickly is that it does not require evolutionary iterations after generating data, while TDGMSE (include SGA/MGA/SDRCGA/TCGSE/FUZGENMUT) requires iterations to find test data meeting the requirements. Although the difference in execution time between SDRCGA and TCGSE is small, they are still superior to SGA and HGA (red and black dotted line).

In summary, the time cost of TDGMSE is much lower than that of the six methods (SGA, MGA, SDRCGA, TCGSE, FUZGENMUT, and TDGSMBD), and comparable to that of Random (in the same order of magnitude). Although TDGMSE is not optimal in terms of time cost, it



(a)



(b)

Fig. 6. The comparison results of efficiency for TDGMSE and other seven baseline techniques.

is still the best in other aspects, such as TDGMSE's highest capability of killing stubborn mutants (RQ2.2).

6. Discussion

This article focuses on a heuristic method for generating test data that kill stubborn mutants. To further demonstrate the performance of our proposed method, we will discuss in detail the convergence and stability of the proposed method.

6.1. Convergence of the proposed method

Convergence is one of the evaluation criteria for a heuristic algorithm. For a heuristic algorithm, if the results can quickly converge to the optimal solution during in generating test data of killing stubborn mutants, then the convergence of this method is better.

We apply our method and other five evolutionary algorithms to each tested program for calculating the fitness values obtained by each evolutionary algorithm during 3000 iterations. Fig. 7 shows the convergence of each evolutionary algorithm in generating test data of killing stubborn mutants. From Fig. 7, we can obtain that: (1) For J1 and J2, our method converges to 1.00 with less than 300 iterations, while the other five methods require more than 300 iterations to converge to 1.00; (2) Our method converges to 1.00 with less than 900 iterations from J3 to J6, while the other five methods require more than 900 iterations to converge to 1.00; (3) From J7 to J14, our method converges to 1.00 (or close to 1.00) with less than 1500 iterations, while the other five methods require more than 1500 iterations to converge to 1.00 (or close to 1.00). Therefore, our method has better performance than other five evolutionary algorithms in terms of convergence.

6.2. Stability of the proposed method

Considering that the initial population is randomly generated and these algorithms use random operators, the results obtained by each heuristic method may vary in different executions. Therefore, evaluating a heuristic algorithm should consider its stability. For a heuristic algorithm, if the difference between the final fitness values in different executions is not significant, then the algorithm will be stable. We will confirm stability by calculating the standard deviation of different fitness values.

For a algorithm, the size of the standard deviation is inversely proportional to its stability, that is, the smaller the standard deviation, the more stable it. For this purpose, we executed each algorithm 10 times (including 3000 iterations each time) and obtained the final results as shown in Figs. 8 and 9. Fig. 8 illustrates the fitness values of the obtained results from 10 executions of each evolutionary algorithm and Fig. 9 shows the standard deviation for calculating these results. From Figs. 8 and 9, we obtain that: (1) Compared with other methods, the difference between the final fitness values (Fig. 8) in different executions for our method is not significant; (2) Based on the results of these 10 executions, the standard deviation (Fig. 9) of the proposed method is smaller than the other five evolutionary algorithms for each benchmark program, which indicates that the proposed method has better stability in generating test data that kills stubborn mutants.

7. Conclusions and future work

Mutation testing can provide valuable guidance for generating test data that detect various types of faults, but it is also a very expensive technique. To reduce the cost of mutation testing, researchers attempt to identify stubborn mutants and generate test data to kill them. However, existing methods suffer from the inaccurate identification of stubborn mutants and the problem of generating only one test datum at a time, which will seriously affect the quality of stubborn mutants and the efficiency of generating test data for killing stubborn mutants.

To solve the existing problems, we propose a set evolution based method for generating a test suite that kills stubborn mutants. On the one hand, we propose an integrated indicator based on the accessibility and infection, which can accurately identify stubborn mutants. On the other hand, we construct a constrained multi-objective mathematical model and develop a new genetic algorithm based on set evolution to solve the mathematical model, thereby improving the efficiency of generating test data for killing stubborn mutants. We apply our method and other benchmark techniques to 14 tested programs. The experimental results show that the integrated indicator formed by combining accessibility and infection significantly improves the accuracy of identifying stubborn mutants; the new constrained multi-objective mathematical model and the new genetic algorithm based on set evolution significantly improve the effectiveness and efficiency of generating test data that kill stubborn mutants.

Although this research shows encouraging results, it is still necessary to further optimize our method. First of all, the values of some weights (such as δ and ω_i (i = 1, 2)) and indicators also need to be designed by some better methods, which can improve the generalization ability of the proposed method. Then, we will consider other solutions due to the use of a simple single objective mathematical model transformation method in solving multi-objective mathematical models. In the end, we can consider the identification of equivalent mutants as another research work in future.

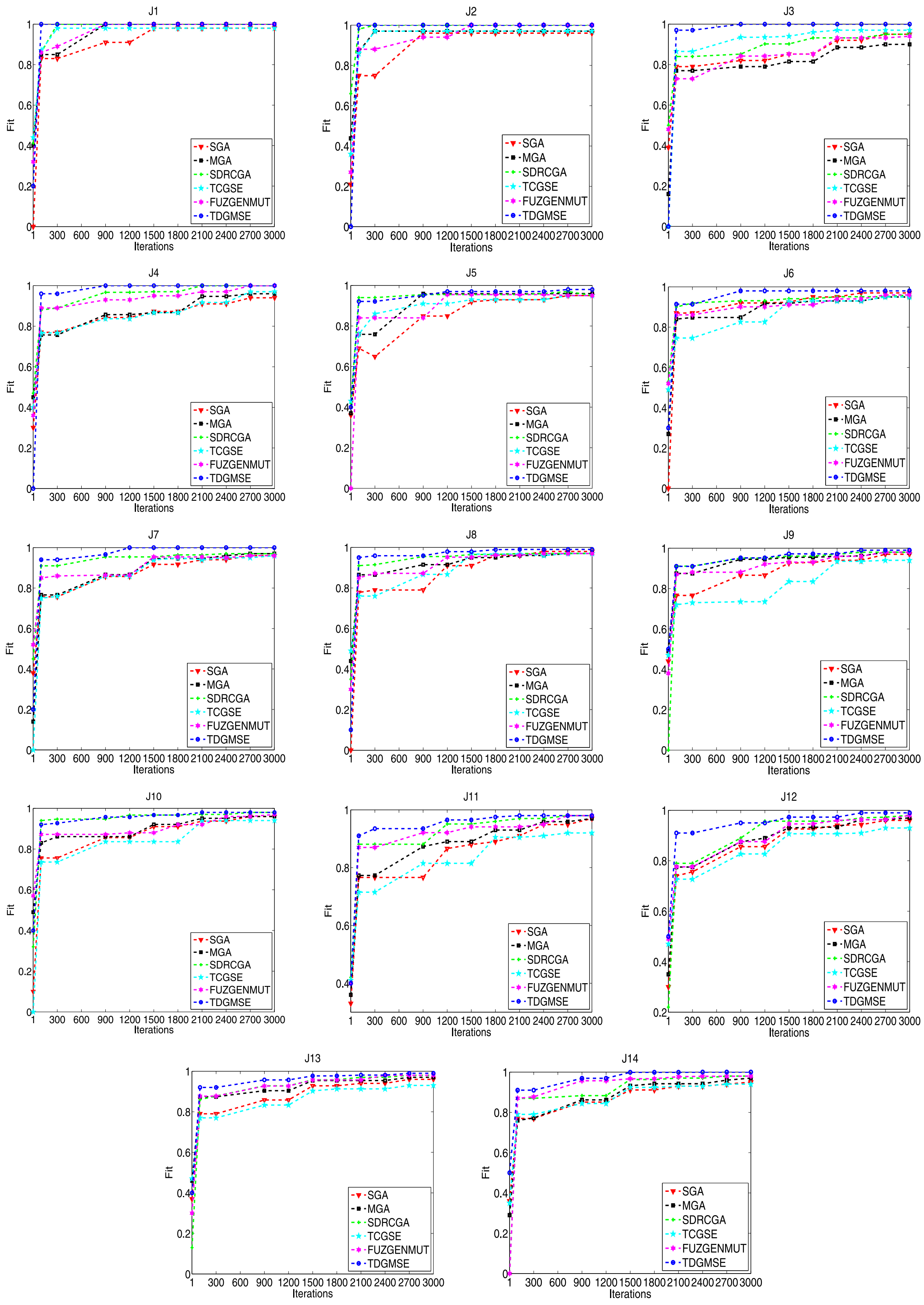


Fig. 7. Convergence of each evolutionary algorithm in generating test data of killing stubborn mutants.

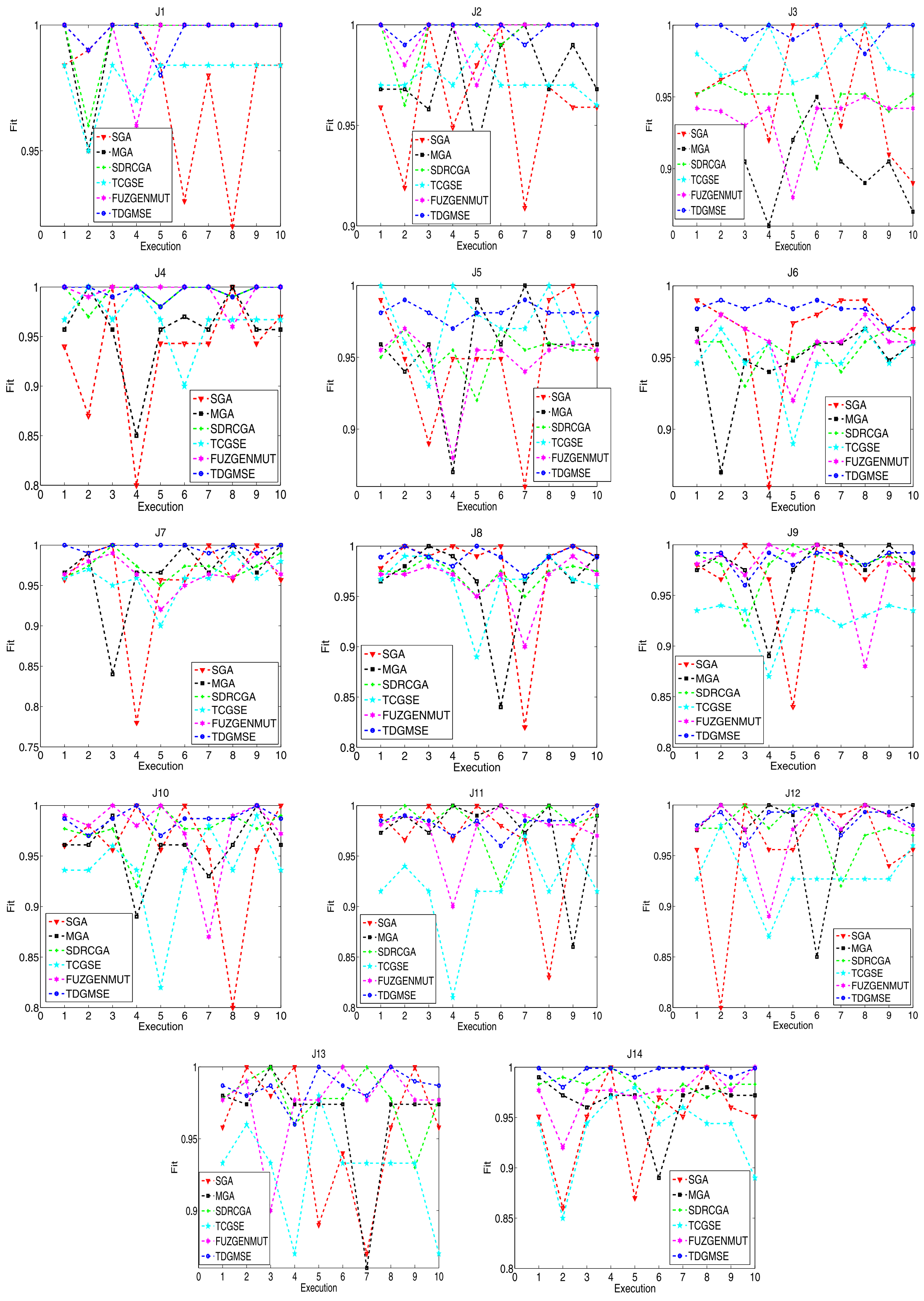


Fig. 8. The fitness values of the obtained results from 10 executions of each evolutionary algorithm.

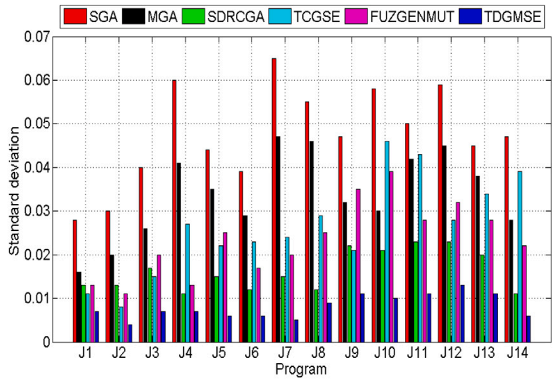


Fig. 9. Standard deviation among the fitness values of the obtained results from 10 executions of each evolutionary algorithm.

CRediT authorship contribution statement

Changqing Wei: Writing – review & editing, Writing – original draft, Software, Methodology, Formal analysis, Conceptualization. **Xiangjuan Yao:** Writing – review & editing, Supervision, Funding acquisition, Conceptualization. **Dunwei Gong:** Supervision, Methodology, Funding acquisition, Conceptualization. **Huai Liu:** Writing – review & editing, Supervision, Methodology, Conceptualization. **Xiangying Dang:** Supervision, Methodology, Conceptualization.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

Acknowledgments

This work was supported by National Natural Science Foundation of China (No. 62373357 and 42230704), the Xuzhou Science and Technology Plan Project under Grant (No. KC21007) and Suzhou municipal science and technology plan project (No. SYG202129 and SYG202351), and the Major Project of Natural Science Research of the Jiangsu Higher Education Institutions of China under Grant 21KJA520006.

References

Ammann, P., Márcio, E.D., Offutt, J., 2014. Establishing theoretical minimal sets of mutants. In: Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation. ICST'14, pp. 21–30.
Arasteh, B., Gharehchopogh, F.S., Gunes, P., Kiani, F., Torkamanian-Afshar, M., 2023. A novel metaheuristic based method for software mutation test using the discretized and modified forrest optimization algorithm. J. Electron. Test. 39 (3), 347–370.
Arasteh, B., Imanzadeh, P., Arasteh, K., Gharehchopogh, F.S., Zarei, B., 2022. A source-code aware method for software mutation testing using artificial bee colony algorithm. J. Electron. Test. 38 (3), 289–302.
Baker, R., Habli, I., 2013. An empirical evaluation of mutation testing for improving the test quality of safety-critical software. IEEE Trans. Softw. Eng. 39 (6), 787–805.
Chekam, T.T., Papadakis, M., Cordy, M., Traon, Y.L., 2021. Killing stubborn mutants with symbolic execution. Trans. Softw. Eng. Methodol. (TOSEM) 30 (2), 1–23.
Chekam, T.T., Papadakis, M., Traon, Y.L., Harman, M., 2017. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In: 2017 IEEE/ACM 39th International Conference on Software Engineering. ICSE, pp. 597–608.
Chen, X., Gu, Q., 2012. Mutation testing: Principal, optimization and application. J. Front. Comput. Sci. Technol. 6 (12), 19.
Dang, X., Gong, D., Yao, X., Tian, T., Liu, H., 2021. Enhancement of mutation testing via fuzzy clustering and multi-population genetic algorithm. IEEE Trans. Softw. Eng. PP (99), 1–17.
Dang, X., Yao, X., Gong, D., Tian, T., 2020. Efficiently generating test data to kill stubborn mutants by dynamically reducing the search domain. IEEE Trans. Reliab. 69 (1), 334–348.
Dave, M., Agrawal, R., 2016. Mutation Testing and Test Data Generation Approaches: A Review. Springer, Singapore, pp. 373–382.
Delgado-Pérez, P., Habli, I., Gregory, S., Alexander, R., Clark, J., Medina-Bulo, I., 2018. Evaluation of mutation testing in a nuclear industry case study. IEEE Trans. Reliab. 67 (4), 1406–1419.
Demillo, R.A., Offutt, A.J., 1991. Constraint-based automatic test data generation. IEEE Trans. Softw. Eng. 17 (9), 900–910.
Derezinska, A., Szustek, A., 2008. Tool-supported advanced mutation approach for verification of c# programs. In: 2008 Third International Conference on Dependability of Computer Systems DepCoS-RELCOMEX. pp. 261–268.
Feng, X., Marr, S., O'Callaghan, T., 2008. ESTP: An experimental software testing platform. In: Testing: Academic and Industrial Conference-Practice and Research Techniques (Taic Part 2008). pp. 59–63.
Fraser, G., Zeller, A., 2012. Mutation-driven generation of unit tests and oracles. IEEE Trans. Softw. Eng. 38 (2), 278–292.
Guo, H., Wang, W., Shang, Y., Zhao, R., 2017. Weak mutation test case set generation based on dynamic set evolution algorithm. Comput. Appl. 37 (9), 1–7.
Hamlet, R.G., 1977. Testing programs with the aid of a compiler. IEEE Trans. Softw. Eng. SE-3 (4), 279–290.
Harman, M., Kim, S.G., Lakhotia, K., Mcminn, P., Yoo, S., 2010. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. In: Third International Conference on Software Testing. pp. 182–191.
Javad Hosseini, S.M., Arasteh, B., Isazadeh, A., Mohsenzadeh, M., Mirzarezaee, M., 2021. An error-propagation aware method to reduce the software mutation cost using genetic algorithm. Data Technol. Appl. 55 (1), 118–148.
Korel, B., 1990. Automated software test data generation. IEEE Trans. Softw. Eng. 16 (8), 870–879.
Liu, S., Li, Y., 1987. An auotmatic data geneartion test method. Comput. Eng. 26–32.
Ma, Y., Offutt, J., Yong, R.K., 2005. MuJava: an automated class mutation system. Softw. Test. Verif. Reliab. 15 (2), 97–133.
Mantyla, M.V., Adams, B., Khomh, F., Engstrom, E., Petersen, K., 2015. On rapid releases and software testing: a case study and a semi-systematic literature review. Empir. Softw. Eng. 20 (5), 1384–1425.
Matnei Filho, R.A., Vergilio, S.R., 2016. A multi-objective test data generation approach for mutation testing of feature models. J. Softw. Eng. Res. Dev. 4 (1), 1–29.
Namin, A.S., Xue, X., Rosas, O., Sharma, P., 2015. MuRanker: a mutant ranking tool. Softw. Test. Verif. Reliab. 25 (5–7), 572–604.
Offutt, A.J., 1988. Automatic Test Data Generation (Ph.D. thesis). Georgia Institute of Technology.
Offutt, A.J., Jin, Z., Pan, J., 1998. The dynamic domain reduction procedure for test data generation: Design and algorithms. Softw. - Pract. Exp. 29 (2), 167–193.
Offutt, A.J., Pan, J., 1997. Automatically detecting equivalent mutants and infeasible paths. Softw. Test. Verif. Reliab. 7 (3), 165–192.
Pacheco, C., Ernst, M.D., 2007. Randoop: feedback-directed random testing for java. In: Companion To the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications. pp. 1–2.
Patrick, M.T., Alexander, R., Oriol, M., Clark, J.A., 2013. Using mutation analysis to evolve subdomains for random testing. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops. pp. 53–62.
Petrović, G., Ivanković, M., 2018. State of mutation testing at google. In: 2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track. ICSE-SEIP, pp. 163–171.
Shan, J., Gao, Y., Liu, M., Liu, J., et al., 2008. A new method for automatic generation of mutation test data. J. Comput. Sci. 31 (6), 1025–1034.
Silva, R.A., Senger de Souza, S.R., Lopes de Souza, P.S., 2017. A systematic review on search based mutation testing. Inf. Softw. Technol. 81, 19–35.
Song, J.S., Kwon, Y.J., 2006. An RTSD system against various attacks for low false positive rate based on patterns of attacker's behaviors. IEICE Trans. Inf. Syst. E89-D (10), 2637–2643.
Wegener, J., Baresel, A., Sthamer, H., 2001. Evolutionary test environment for automatic structural testing. Inf. Softw. Technol. 43 (14), 841–854.
Wei, C., Yao, X., Gong, D., Liu, H., 2021. Spectral clustering based mutant reduction for mutation testing. Inf. Softw. Technol. 132, 106502.
Yao, X., Gong, D., 2012. Mutation testing based on comparison of paths. Acta Electron. Sin. 40 (1), 103–107.
Yao, X., Gong, D., 2014. Genetic algorithm-based test data generation for multiple paths via individual sharing. Comput. Intell. Neurosci. 29 (3), 1–13.
Yao, X., Gong, D., Zhang, G., 2015. Constrained multi-objective test data generation based on set evolution. IET Softw. 9 (4), 103–108.

Yao, X., Harman, M., Jia, Y., 2014. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In: ICSE 2014: 36th International Conference on Software Engineering. ICSE, 31 May-7 June 2014, Hyderabad, India, pp. 919–930.

Yao, X., Zhang, G., Pan, F., Gong, D., Wei, C., 2020. Orderly generation of test data via sorting mutant branches based on their dominance degrees for weak mutation testing. IEEE Trans. Softw. Eng. PP (99), 1–17.

Ye, J., Zhang, Y., Ruan, Y., 2009. A new genetic algorithm based on improved crossover and self-identify high mutation operators. J. Fuzhou Univ. Nat. Sci. Ed. 37 (6), 808–811.

Yu, W., Li, R., 2001. A catastrophe-based parallel genetic algorithm. Comput. Eng. 27 (7), 3.

Yue, J., Harman, M., 2009. Higher order mutation testing. Inf. Softw. Technol. 51 (10), 1379–1393.

Yue, J., Harman, M., 2010. An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. 37 (5), 649–678.

Zhang, G., Gong, D., Yao, X., 2015. Test case generation method based on mutation analysis and set evolution. J. Comput. Sci. 38 (11), 1–14.

Zhang, L., Hou, S., Hu, J., Xie, T., Mei, H., 2010. Is operator-based mutant selection superior to random mutant selection? In: 2010 ACM/IEEE 32nd International Conference on Software Engineering. vol. 1, pp. 435–444.

Zhang, C., Zhan, Z., 2009. Comparisons of selection strategy in genetic algorithm. Comput. Eng. Des. 30 (23), 5471–5474.

Changqing Wei received the master's degree in Applied Mathematics from China University of Mining and Technology in 2020. Now he is a Ph.D. student in the School of Mathematics, China University of Mining and Technology. His main research interests include search-based software testing.

Xiangjuan Yao received the Ph.D. degree in control theory and control engineering from China University of Mining and Technology in 2011. She is a professor in the School of Mathematics, China University of Mining and Technology. Her main research interests include intelligence optimization and search based software testing.

Dunwei Gong received the Ph.D. degree in control theory and control engineering from China University of Mining and Technology, in 1999. He is a Professor with the College of Automation and Electronic Engineering, Qingdao University of Science and Technology. His main research interests include intelligence optimization and control.

Huai Liu is a Lecturer in Department of Computing Technologies at Swinburne University of Technology, Melbourne, Australia. He has worked as a Lecturer at Victoria University and a Research Fellow at RMIT University. He received the Beng in physioelectronic technology and MEng in communications and information systems, both from Nankai University, China, and the Ph.D. degree in software engineering from the Swinburne University of Technology, Australia. Prior to working in Higher Education he worked as an engineer in the IT industry. His current research interests include software testing, cloud computing, and end-user software engineering.

Xiangying Dang received the MS degree in computer science from the School of Information Engineering, Jiangnan University, Wuxi, China, in 2008. She is currently working toward the Ph.D. degree in the School of Information and Control Engineering, China University of Mining and Technology, Xuzhou, China. Her main research interests include software engineering based search and mutation testing and analysis.