Information and Software Technology 000 (2016) 1–17

Contents lists available at ScienceDirect

Information and Software Technology

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

# A path-aware approach to mutant reduction in mutation testing

Chang-ai Sun[a,*], Feifei Xue[a], Huai Liu[b], Xiangyu Zhang[c]

[a] School of Computer and Communication Engineering, University of Science and Technology Beijing, China
[b] Australia-India Research Centre for Automation Software Engineering, RMIT University, Melbourne, Australia
[c] Department of Computer Science, Purdue University, West Lafayette, IN, USA

ARTICLE INFO

*Article history:*
Received 1 August 2015
Revised 15 February 2016
Accepted 22 February 2016
Available online xxx

*Keywords:*
Mutation testing
Selective mutation testing
Control flow
Path depth

ABSTRACT

*Context*: Mutation testing, which systematically generates a set of mutants by seeding various faults into the base program under test, is a popular technique for evaluating the effectiveness of a testing method. However, it normally requires the execution of a large amount of mutants and thus incurs a high cost.

*Objective*: A common way to decrease the cost of mutation testing is mutant reduction, which selects a subset of representative mutants. In this paper, we propose a new mutant reduction approach from the perspective of program structure.

*Method*: Our approach attempts to explore path information of the program under test, and select mutants that are as diverse as possible with respect to the paths they cover. We define two path-aware heuristic rules, namely module-depth and loop-depth rules, and combine them with statement- and operator-based mutation selection to develop four mutant reduction strategies.

*Results*: We evaluated the cost-effectiveness of our mutant reduction strategies against random mutant selection on 11 real-life C programs with varying sizes and sampling ratios. Our empirical studies show that two of our mutant reduction strategies, which primarily rely on the path-aware heuristic rules, are more effective and systematic than pure random mutant selection strategy in terms of selecting more representative mutants. In addition, among all four strategies, the one giving loop-depth the highest priority has the highest effectiveness.

*Conclusion*: In general, our path-aware approach can reduce the number of mutants without jeopardizing its effectiveness, and thus significantly enhance the overall cost-effectiveness of mutation testing. Our approach is particularly useful for the mutation testing on large-scale complex programs that normally involve a huge amount of mutants with diverse fault characteristics.

## 1. Introduction

Mutation testing, basically a fault-based software testing technique [1,2], was originally proposed to measure the adequacy of a given test suite and help design new test cases to improve the quality of the test suite. It has been used for different purposes, such as the generation of test cases and oracles [3], fault localization [4], etc. Fig. 1 shows the principle of mutation testing. Given a base program, different variants, namely mutants, can be generated by seeding various faults through mutation operators. Once a test case shows different behaviors between a mutant and the base program, the mutant is said to be killed by the test case (in other words, the related fault is detected). Apparently, a test suite is regarded as effective if it can kill as many mutants as possible (i.e. large mutation scores). A number of studies [5–7] have shown that compared with manually fault-seeded programs, automatically generated mutants are more similar to the real-life faulty program. Thus, mutation testing has been acknowledged as an effective technique for evaluating the fault-detection capability of a testing method.

However, the real-world application of mutation testing is hindered by some drawbacks, such as the existence of equivalent mutants, lack of appropriate automated tools, etc. One major drawback is the high cost: Due to the large number of mutation operators and possible locations to apply these operators into the program, a huge volume of mutants are generated to guarantee that various faults are covered as many as possible. The execution of all these mutants is quite time-consuming, and the test result verification on mutants is a non-trivial task.

* Corresponding author. Tel.: +861062332931; fax: +861062332873.
*E-mail addresses:* casun@ustb.edu.cn, changai_sun2002@hotmail.com (C.-a. Sun), 729045626@qq.com (F. Xue), huai.liu@rmit.edu.au (H. Liu), xyzhang@cs.purdue.edu (X. Zhang).

http://dx.doi.org/10.1016/j.infsof.2016.02.006
0950-5849/© 2016 Elsevier B.V. All rights reserved.

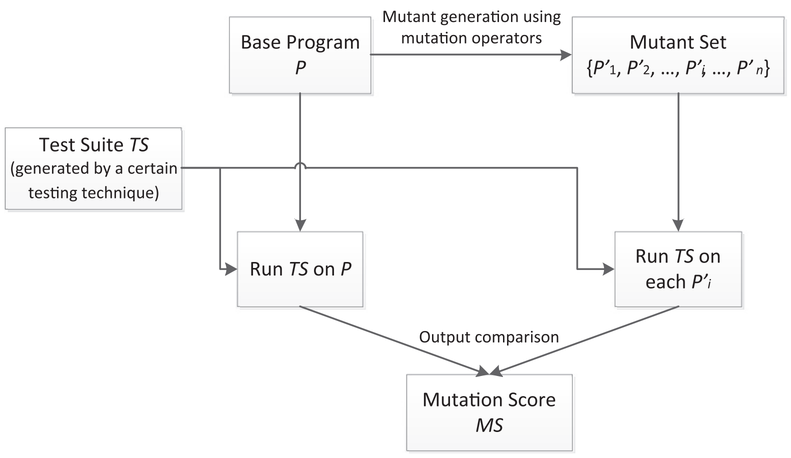

Base Program
P
Mutant generation using mutation operators
Mutant Set
{P'1, P'2, ..., P'i, ..., P'n}
Test Suite TS
(generated by a certain testing technique)
Run TS on P
Run TS on each P'i
Output comparison
Mutation Score
MS

**Fig. 1.** Principle of mutation testing.

Some efforts have been made to decrease the cost of mutation testing by reducing the number of mutants. Mathur and Wong [8] proposed random mutant selection, which is simple and efficient in execution. However, random selection may discard some mutants that are difficult to be killed, and thus affect the quality of test suite that is designed based on the selected mutants. A more systematic approach called operator-based mutant selection [9] was proposed to select a subset of mutants based on certain (not all) mutation operators. Nevertheless, some recent studies [10] showed that the operator-based strategy is actually not superior to random selection.

In this paper, we propose a new mutant reduction approach. Instead of mutation operators, we conjecture that the fault characteristics (in particular, how different a fault is to be detected) are more related to the location of the fault, especially how deep the fault location is in terms of program paths. Therefore, we explore the mutant reduction based on the program structure. In particular, our work makes the following four contributions:

(I) A path-aware approach to mutant reduction is proposed, which explores mutant reduction from the perspective of the path depth in the program under test;
(II) We present four heuristic rules for mutant reduction, two of which are path-aware, one statement-based, and one operator-based;
(III) Four mutant reduction strategies are developed with different priorities among the heuristic rules; and
(IV) The effectiveness of the mutant reduction strategies are evaluated through an empirical study based on 11 real-life programs. It is shown that two strategies giving higher priorities to path-aware rules are superior to random mutant selection, and are more effective than the other two giving higher priorities to statement- or operator-based rules.

The rest of this paper is organized as follows. In Section 2, we introduce the underlying concepts and techniques. In Section 3, we describe our heuristic rules and the mutant reduction strategies. We present the design and setting of our empirical study in Section 4, and discuss the experimental results in Section 5. The work related to our study is discussed in Section 6. Finally, we conclude the paper in Section 7.

## 2. Preliminaries and terminology

In this section, we introduce the basic concepts and preliminaries that will be used by path-aware mutant reduction technique. All concepts are illustrated using an example program implementing heap sort, as shown in Fig. 2. The function call diagram of the program is given in Fig. 3.

Practically, a program is often composed of a number of modules, such as functions in C programs. We distinguish these modules into *caller* and *callee* based on the invoking relationship among them [11].

**Definition 1.** If module $m$ directly invokes module $n$, module $m$ is termed as the caller and module $n$ the callee. The invoking relation is represented as $m \rightarrow n$.

**Definition 2.** $Callers(m)$ refers to the set of direct callers of module $m$, that is, $Callers(m) = \{x | x \rightarrow m\}$.

For example, in the heap sort program (Figs. 2 and 3), $f_1 \rightarrow f_2$, that is, between modules $f_1$ and $f_2$, $f_1$ is the caller, while $f_2$ is the callee. According to Fig. 3, we can get the following:

- $Callers(f_1) = \emptyset$.
- $Callers(f_2) = \{f_1\}$.
- $Callers(f_3) = \{f_1\}$.
- $Callers(f_4) = \{f_2\}$.
- $Callers(f_5) = \{f_2, f_4\}$.

We now define the module depth $MD(m_i)$ of a module $m_i$ based on the invoking relation among modules.

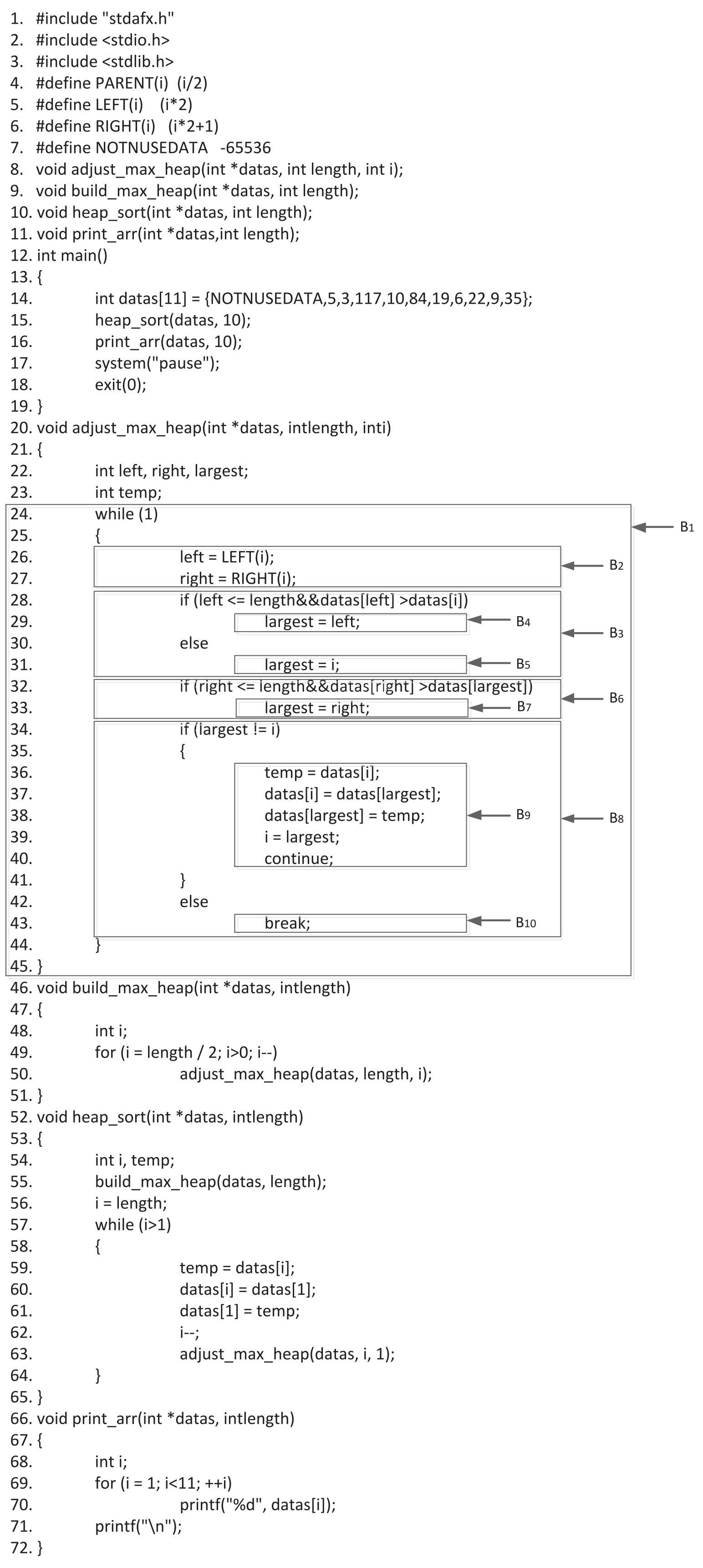
**Definition 3.** For a module $m_i$,

$$MD(m_i) = \begin{cases} 0; & Callers(m_i) = \emptyset \\ \max\left(MD(m_j | m_j \in Callers(m_i))\right) + 1; & Callers(m_i) \neq \emptyset \end{cases}$$

For the heap sort program, we have the following calculations:

- Since $Callers(f_1) = \emptyset$, $MD(f_1) = 0$.
- Since $Callers(f_2) = \{f_1\}$, $MD(f_2) = \max(MD(f_1)) + 1 = 0 + 1 = 1$.
- Since $Callers(f_3) = \{f_1\}$, $MD(f_3) = \max(MD(f_1)) + 1 = 0 + 1 = 1$.
- Since $Callers(f_4) = \{f_2\}$, $MD(f_4) = \max(MD(f_2)) + 1 = 1 + 1 = 2$.
- Since $Callers(f_5) = \{f_2, f_4\}$, $MD(f_5) = \max(MD(f_2), MD(f_4)) + 1 = \max(1, 2) + 1 = 2 + 1 = 3$.

Note that there may exist recursive calls among modules, which in turn results in a cycle in the function call diagram. In this situation, we can break the cycle from the back edges in recursive calls, as discussed in [11].



```
#include "stdafx.h"
#include <stdio.h>
#include <stdlib.h>
#define PARENT(i)  (i/2)
#define LEFT(i)    (i*2)
#define RIGHT(i)   (i*2+1)
#define NOTNUSEDATA   -65536
void adjust_max_heap(int *datas, int length, int i);
void build_max_heap(int *datas, int length);
void heap_sort(int *datas, int length);
void print_arr(int *datas,int length);
int main()
{
        int datas[11] = {NOTNUSEDATA,5,3,117,10,84,19,6,22,9,35};
        heap_sort(datas, 10);
        print_arr(datas, 10);
        system("pause");
        exit(0);
}
void adjust_max_heap(int *datas, intlength, inti)
{
        int left, right, largest;
        int temp;
        while (1)
        {
                left = LEFT(i);
                right = RIGHT(i);
                if (left <= length&&datas[left] >datas[i])
                        largest = left;
                else
                        largest = i;
                if (right <= length&&datas[right] >datas[largest])
                        largest = right;
                if (largest != i)
                {
                        temp = datas[i];
                        datas[i] = datas[largest];
                        datas[largest] = temp;
                        i = largest;
                        continue;
                }
                else
                        break;
        }
}
void build_max_heap(int *datas, intlength)
{
        int i;
        for (i = length / 2; i>0; i--)
                adjust_max_heap(datas, length, i);
}
void heap_sort(int *datas, intlength)
{
        int i, temp;
        build_max_heap(datas, length);
        i = length;
        while (i>1)
        {
                temp = datas[i];
                datas[i] = datas[1];
                datas[1] = temp;
                i--;
                adjust_max_heap(datas, i, 1);
        }
}
void print_arr(int *datas, intlength)
{
        int i;
        for (i = 1; i<11; ++i)
                printf("%d", datas[i]);
        printf("\n");
}
```

**Fig. 2.** Heap sort program.

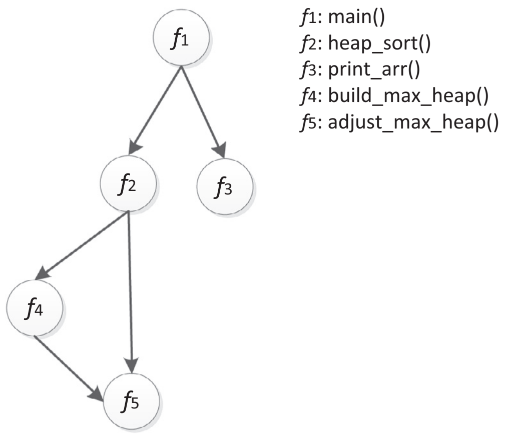


**Fig. 3.** Function call diagram of heap sort program.

A module is often composed of a number of hierarchical blocks, which can be defined based on the program dependence graph (PDG) [12,13]. The nodes in the PDG include statements and predicate expressions, and edges include data dependencies or control dependencies. We further group these nodes into *BasicBlock*, *OptionBlock*, and *LoopBlock*.

**Definition 4.** *BasicBlock* is defined as a segment of continuous executable statements:

$$BasicBlock = \left\{ Statement_{i,\cdots,j} \middle| \left(i \le j\right) \wedge \left(\neg n.\left((n < i) \wedge (Statement_n \prec Statement_i)\right)\right) \wedge \left(\neg m.\left((m > j) \wedge (Statement_j \prec Statement_m)\right)\right)\right\},$$

where $Statement_i$ represents the $i$th logic statement in the program; $Statement_{i,\ \ldots,\ j}$ refers to the segment composed by program lines from $i$ to $j$; $Statement_x \prec Statement_y$ means that $Statement_y$ will be executed if and only if $Statement_x$ is executed.

**Definition 5.** *OptionBlock* selects a region $Region_i$ for execution under predicate expression $\varphi$:

$OptionBlock = (\varphi) \prec Region_i,$

where $\varphi$ is a predicate expression whose value is evaluated to be False or True; $Region_i$ refers to a control dependence region which is a *BasicBlock*, *OptionBlock*, and *LoopBlock*, or their composite; $Region_i$ is executed only if $\varphi$ is evaluated to be True.

**Definition 6.** *LoopBlock* repeats the execution of a region $Region_i$ when the loop predicate expression $\varphi$ is satisfied:

$LoopBlock = ((\varphi) \prec Region_i)^+,$

where $\varphi$ is a predicate expression whose value is evaluated to be False or True; $Region_i$ refers to a control dependence region which is a *BasicBlock*, *OptionBlock*, and *LoopBlock*, or their composite; $Region_i$ is executed only if predicate expression $\varphi$ is evaluated to be True.

As an illustration, let us look at the function "adjust_max_heap()" ($f_5$ in Fig. 3), it is divided into ten blocks ($B_1, B_2, \ldots, B_{10}$). Block $B_9$, which refers to the segment composed of Statements 36–40, is a *BasicBlock*. This is because Statements 37, 38, 39, and 40 will be executed if and only if Statements 36, 37, 38, and 39 are executed, respectively. $B_4$ is an *OptionBlock* because Block $B_4$ will be executed only if the condition "left <= length && datas[left] > datas[i]" is true. $B_1$ is a *LoopBlock* because its control dependence regions (such as $B_2$, $B_3$, etc.) are repeatedly executed inside the while loop.

We classify relationship between two continuous blocks into *SubBlock* and *NextBlock*.

**Definition 7.** Given two blocks $Block_i$ and $Block_j$, $Block_j$ is said to be the SubBlock of $Block_i$ (denoted as $Block_j \in SubBlock(Block_i)$) if and only if $Block_j$ is control dependent on $Block_i$.

From Fig. 2, we can observe that blocks $B_2$, $B_3$, $B_6$, and $B_8$ are control dependent on $B_1$, that is, $B_2$, $B_3$, $B_6$, and $B_8$ are the *SubBlocks* of $B_1$.

**Definition 8.** Given two blocks $Block_i$ and $Block_j$, $Block_j$ is said to be the NextBlock of $Block_i$ (denoted as $Block_j = NextBlock(Block_i)$) if and only if $Block_i$ is immediately post-dominated by $Block_j$ .

In the running example, $B_3$ will be executed immediately after $B_2$ is executed. Thus, we can say that $B_3$ is the *NextBlock* of $B_2$.

**Definition 9.** Given a block $Block_i$, we define its hierarchy $Hierarchy(Block_i)$ by means of the set of parent blocks and predecessor blocks, namely
$\{Block_j | Block_i \in SubBlock(Block_j) \vee Block_i = NextBlock(Block_j)\}$.

As discussed above, we have $B_2$, $B_3 \in SubBlock(B_1)$ and $B_3 = NextBlock(B_2)$. Then, according to Definition 9, we can get the following:

- $Hierarchy(B_1) = \emptyset$.
- $Hierarchy(B_2) = \{B_1\}$.
- $Hierarchy(B_3) = \{B_1, B_2\}$.

We now define the loop/branch depth $LD(b_i)$ of block $b_i$ based on the relations among blocks.

**Definition 10.** Given a block $b_i$,

$$LD(b_i) = \begin{cases} 0; & Hierarchy(b_i) = \emptyset \\ LD(b_j); & Hierarchy(b_i) \neq \emptyset \wedge b_i = NextBlock(b_j) \\ LD(b_j) + 1; & Hierarchy(b_i) \neq \emptyset \wedge b_i \in SubBlock(b_j) \end{cases}$$

For the heap sort program, we can have the following according to Definition 10:

- $LD(B_1) = 0$, because $Hierarchy(B_1) = \emptyset$.
- $LD(B_2) = 1$, because $B_2 \in SubBlock(B_1)$ and $LD(B_1) = 0$.
- $LD(B_3) = 1$, because either ($B_3 \in SubBlock(B_1)$ and $LD(B_1) = 0$) or ($B_3 = NextBlock(B_2)$ and $LD(B_2) = 1$).

## 3. Path-aware mutant reduction approach

There are two basic intuitions behind our path-aware approach, namely depth and diversity.

(1) Depth
According to the code coverage theory, if a test case $tc$ executes a certain control dependence depth $S_p$ of a program path, it should also execute all the depths $S_q \le S_p$ on the same path. Suppose that two mutants $M_u$ and $M'_u$ are associated with different faults on $S_p$ and $S_q$, respectively. It is obvious that if $tc$ can cover the faulty statement of $M_u$, it will definitely also cover the faulty statement of $M'_u$, but not vice versa. In other words, $M_u$ is superior to $M'_u$ with respect to the chance of executing faulty statements. If only one mutant can be used, we should select $M_u$ but not $M'_u$. In this sense, we should give higher priorities to the mutants whose associated faults are in the deeper locations.

(2) Diversity
A program fault is associated with various characteristics, such as fault location, fault type, etc. These fault characteristics, in turn, affect the behaviors of a program. Previous studies [14] have shown that many mutants can have similar or even equivalent behaviors. It is quite challenging to select a subset of mutants that are as diverse as possible in terms of their behaviors and that can be considered as representative of the set of all possible mutants. In our approach, the diversity among selected mutants is achieved by considering different heuristic rules.

### 3.1. Heuristic rules

Traditional mutation analysis is defined as a 5-tuple [15]: $T =< P, S, D, L, A >$, where $P$ is the base program, $S$ is a specification, $D$ is a domain of interest, $L = (l_1, l_2, \ldots, l_i, \ldots, l_n)$ is a $n$-tuple of locations in $P$, and $A = (A_1, A_2, \ldots, A_i, \ldots, A_n)$ is an alternative set associated with location $L$. Sun et al. [16] proposed a distribution-award mutation analysis, which extends the tradition mutation analysis to a 6-tuple $E =< P, S, D, L, A, p_r >$, where $p_r$ is the occurrence probability of each alternative $a_{i,j}$ of $A_i$, and $0 \le j \le |A_i|$, where $A_i$ is a set of possible alternatives (or faults simulated by applicable mutation operators), and $|A_i|$ denotes the number of applicable mutation operators.

In our approach, we extend $L$ in the tuple $T$ to $L =< Loc, MD, LD >$, where $Loc = (l_1, l_2, \ldots, l_i, \ldots, l_n)$ contains the faults locations, $MD(l_i)$ represents the module depth of $l_i$, and $LD(l_i)$ refers to the loop/branch depth of $l_i$. $MD(l_i)$ and $LD(l_i)$ are defined in Section 2. In other words, besides the fault locations (which are normally represented by the statement numbers for the seeded faults), we also make use of the depth values, which, in turn, are derived from the program path structure. In this sense, our approach is developed based on a "path-aware" notion.

We now define four heuristic rules for mutant selection as follows.

**Rule 1** (module depth)**.** Assume the initial mutant set $Mutant_{md} = \emptyset$. Given a certain module depth $S_m$, for a mutant $v_i$, if $MD(l_i) > S_m$, we add $v_i$ into $Mutant_{md}$. Finally, we have $Mutant_{md} = \{v_i | MD(l_i) > S_m\}$.

**Rule 2** (loop/branch depth)**.** Assume the initial mutant set $Mutant_{ld} = \emptyset$. Given a certain loop/branch depth $S_l$, for a mutant $v_i$ , if $LD(l_i) > S_l$, we add $v_i$ into $Mutant_{ld}$. Finally, we have $Mutant_{ld} = \{v_i | LD(l_i) > S_l\}$.

**Rule 3** (statement selection)**.** Assume the initial mutant set $Mutant_{stm} = \emptyset$ and the fault location set $L = \emptyset$. For a mutant $v_i$, if $Location(v_i) \notin L$ (where $Location(v_i)$ denotes the location of the fault in $v_i$), we add $v_i$ into $Mutant_{stm}$. $L$ is further updated $L = L \cup \{Location(v_i)\}$. Such a process is repeated until $L = Loc$ ($Loc$ is defined above).

**Rule 4** (operator selection)**.** Assume the initial mutant set $Mutant_{op} = \emptyset$ and a set of operators $OP = \{op_1, op_2, \ldots, op_k\}$. For a mutant $v_i$, if $Type(v_i) \in OP$ (where $Type(v_i)$ represents the operator type of $v_i$), we add $v_i$ into $Mutant_{op}$. Finally, we have $Mutant_{op} = \{v_i | Type(l_i) \in OP\}$.

Among the above four rules, Rules 1 and 2 are directly based on the intuitions of our path-aware approach. According to Rule 3, we aim to cover different statements as many as possible; however, it does not well reflect the "depth" intuition discussed above, so we do not consider it as one path-aware heuristic rule. Rule 4 is actually the traditional operator-based mutant selection strategy [9].

### 3.2. Mutant reduction strategies

In order to better achieve the "diversity" intuition, we propose the combinatorial usage of the above four heuristic rules in mutant reduction. We can have various strategies based on different orders of priorities in using these rules. In this study, we focus on the following four typical strategies.

*md-ld-op* **strategy**
According to the depth first search algorithm, we firstly select a set of mutants $Mutant_{md}$ based on the "module depth" heuristic rule. Then we extend $Mutant_{md}$ to $Mutant_{md-ld}$ using the "loop/branch depth" rule. Finally, we apply the "operator selection" rule and get $Mutant_{md-ld-op}$ based on $Mutant_{md-ld}$.

*ld-md-op* **strategy**
The selection process of the *ld-md-op* strategy is very similar to that of the *md-ld-op* strategy, except that the priority order between "module depth" and "loop/branch depth" is swapped. The mutant set corresponding to the *ld-md-op* strategy is named as $Mutant_{ld-md-op}$.

*stm-ld-md* **strategy**
In this strategy, the "statement selection" rule has the highest priority, that is, we need to first ensure that each statement is covered by selected mutants at least once according to the breadth first search algorithm. Then we select the mutants based on "loop/branch depth" and " module depth" rules, and finally, get the mutant sets $Mutant_{stm-ld-md}$.

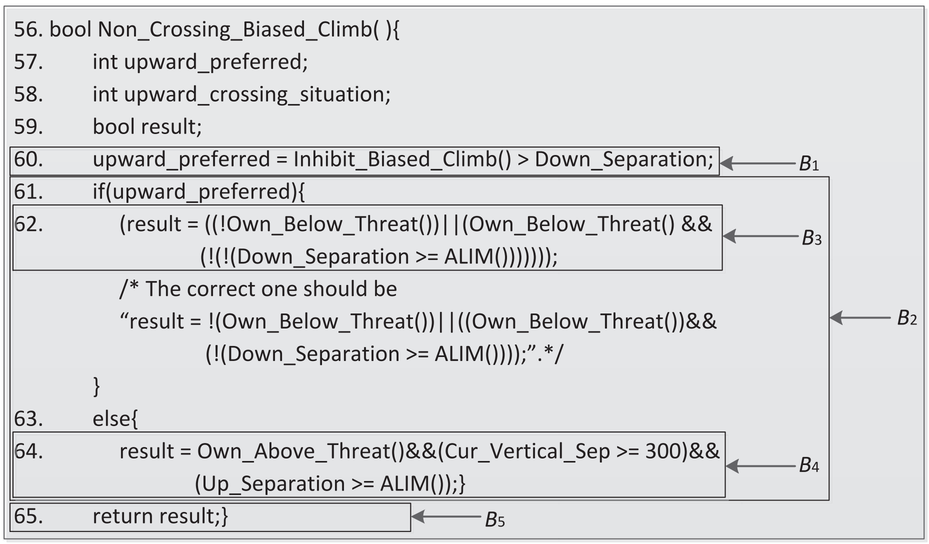
*op-ld-md* **strategy**
The selection process of the *op-ld-md* strategy is very similar to that of the *md-ld-op* strategy, except that the first applied rule is "operator selection" not "statement selection". The mutant set corresponding to the *op-ld-md* strategy is named as $Mutant_{op-ld-md}$.

Algorithm 1 describes the sketched procedure of *md-ld-op* strategy. In Step I (Lines 1–3): the algorithm first sets the sampling ratio, initializes mutant sets to null, and set mutant number counter to 0; In Step II (Lines 6–13): it selects mutants into $Mutant_{md}$ based on the "module depth" heuristic rule; In Step III (Lines 5–21): if the number of selected mutants is smaller than the specified reduction rate, it appends more mutants based on the "loop/branch depth" heuristic rule; otherwise, it returns the selected mutant set, and terminates; In Step IV (Lines 4–29): if the number of selected mutants is still smaller than the specified reduction rate, it appends more mutants based on the "operator selection" heuristic rule; otherwise, it returns the selected mutant set, and terminates; In Step V (Lines 30–35): if the number of selected mutants does not satisfy the sampling ratio, more mutants are randomly selected and appended; otherwise, it outputs the selected mutant set, and terminates.

Similar algorithms can be obtained for other strategies. It should be also noted that there exist other possible strategies, such as *md-ld-stm, stm-md-ld*, and *op-md-ld*, based on different orderings of heuristic rules. Our investigation showed that the most important rule in a mutant reduction strategy is the one with the highest priority. Therefore, we only studied the above four strategies, each of which gives the highest priority to every of the four heuristic rules, respectively.

### 3.3. Illustration

For ease of understanding, we illustrate our proposed heuristic rules and mutant reduction strategies based on a real-life program, namely `tcas`, which was also used in our empirical study, as to be introduced in Section 4. Fig. 4 shows a mutant of `tcas`,



```
56. bool Non_Crossing_Biased_Climb( ){
57.       int upward_preferred;
58.       int upward_crossing_situation;
59.       bool result;
60.       upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
61.       if(upward_preferred){
62.           (result = ((!Own_Below_Threat())||(Own_Below_Threat() &&
                    (!(!(Down_Separation >= ALIM())))));
          /* The correct one should be
          "result = !(Own_Below_Threat())||((Own_Below_Threat())&&
                    (!(Down_Separation >= ALIM())));".*/
          }
63.       else{
64.           result = Own_Above_Threat()&&(Cur_Vertical_Sep >= 300)&&
                    (Up_Separation >= ALIM());}
65.       return result;}
```

**Fig. 4.** A mutant of tcas.

**Table 1**
A summary of a set of 10 mutants in tcas program.

| Mutant no. | Mutation operator | Description | Location (Line no.) | Module depth (MD) | Loop/Branch depth (LD) |
|---|---|---|---|---|---|
| 331 | OLNG | '(!(Down_Separation > = ALIM()))' ⇒ '!(!(Down_Separation > = ALIM())))' | 67 | 2 | 2 |
| 425 | ORRN | ' > =' ⇒ ' < ' | 71 | 2 | 2 |
| 908 | CRCR | 'Inhibit_Biased_Climb() > Down_Separation' ⇒ 'Inhibit_Biased_Climb() > 0' | 81 | 2 | 1 |
| 1565 | CRCR | 'Own_Tracked_Alt < Other_Tracked_Alt' ⇒ '0 < Other_Tracked_Alt' | 96 | 3 | 1 |
| 1655 | CCCR | 'alt_sep=1' ⇒ 'alt_sep = 600' | 123 | 1 | 4 |
| 1784 | CRCR | 'else if (need_upward_RA)' ⇒ 'else if (0)' | 122 | 1 | 3 |
| 1913 | OLLN | '&&' ⇒ '\|\|' | 118 | 1 | 2 |
| 2065 | SRSR | 'alt_sep=2;' ⇒ 'return alt_sep;' | 125 | 1 | 5 |
| 2081 | SSDL | 'alt_sep=2;' ⇒ ';' | 125 | 1 | 6 |
| 3427 | SSDL | 'Two_of_Three_Reports_Valid=atoi(argv[3]);' ⇒ ';' | 147 | 0 | 1 |

where the associated fault is located in Line 62 in the function *Non_Crossing_Biased_Climb( )*.

We can calculate the loop/branch depth for all five blocks (denoted by $B_1, B_2, \ldots, B_5$) in the function *Non_Crossing_Biased_Climb( )*, according to the definition of loop/branch depth (Definition 11) given in Section 2. $B_1$ is a *BasicBlock* that has an empty set of parent blocks and preceding blocks, that is, $Hierarchy(B_1) = \emptyset$; therefore, $LD(B_1) = 0$. Given that $B_2 = NextBlock(B_1)$ and $B_5 = NextBlock(B_2)$, we can calculate $LD(B_5) = LD(B_2) = LD(B_1) = 0$. Since $B_3 \in SubBlock(B_2)$ and $B_4 \in SubBlock(B_2)$, we can have $LD(B_4) = LD(B_3) = LD(B_2) + 1 = 1$.

We can also calculate the module depth of different functions in tcas, as shown in Fig. 5. Based on Definition 3 given in Section 2, we can get $MD(f_1) = 0$, $MD(f_2) = 1$, $MD(f_3) = 1$, $MD(f_4) = 2$, and $MD(f_6) = 2$. Then, $MD(f_5)$ can be calculated as $\max(MD(f_3), MD(f_4), MD(f_6)) + 1 = 3$. Similarly, $MD(f_8) = 3$, $MD(f_9) = 3$, and $MD(f_7) = 3$.

For the mutant given in Fig. 4, since its associated fault is located in $B_4$ of $f_4$, its corresponding module depth and loop/branch depth are 2 (that is, $MD(f_4)$) and 1 (that is, $LD(B_3)$), respectively.

To further illustrate our heuristic rules and mutant reduction strategies, we randomly choose ten mutants for tcas, as summarized in Table 1. In the table, the "Mutant No" column shows the unique identity of the mutant, the "Mutation Operator" column indicates which operator is applied, the "Description" column gives detailed operation on the mutant, the "Location (Line No)" column shows the location of fault in terms of the line number, the "Module Depth (MD)" column and the "Loop/Branch Depth (LD)" column show the module depth and loop/branch depth of the mutant, respectively. Details of how to generate mutants will be given in the next Section 4.

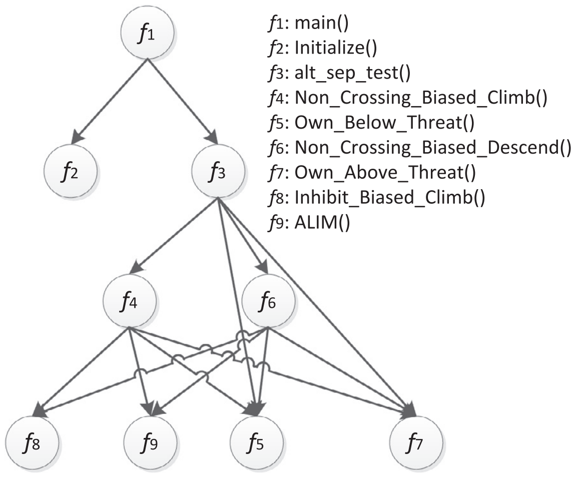


**Fig. 5.** Function call diagram of tcas program.

Given the set of these ten mutants, our heuristic rules work as follows:

- For the Rule 1 – *module depth*: We assume the initial mutant set $Mutant_{md} = \emptyset$, and set the module depth $S_m = 1$. For example, Mutant #331 has the module depth of 2, which is larger than $S_m$, so it is added into $Mutant_{md}$. However, the module

**Algorithm 1** *md-ld-op* mutant reduction strategy.

```
1: Set a real number $0 < x < 100$ to decide $x$ of $Mcount$ mutants will be selected, where $Mcount$ refers to the number of all generated mutants
2: Initialize an integer $Count = 0$
3: Initialize mutant sets $Mutant_{md} = Mutant_{md-ld} = Mutant_{md-ld-op} = \emptyset$
4: while $Mutant_{md-ld-op}$ have not covered all operator types in $OP$ do
5:   for all $ld = S_{l-max}, S_{l-max} - 1, \ldots, 1$ ($ld$ means the loop/branch depth, while $S_{l-max}$ refers to the maximum loop/branch depth of the program under test) do
6:     for all $md = S_{m-max}, S_{m-max} - 1, \ldots, 1$ ($md$ means the module depth, while $S_{m-max}$ refers to the maximum module depth of the program under test) do
7:       if $Count < Mcount \times x\%$ then
8:         Add the mutant on $md$ into $Mutant_{md}$
9:         Increment $Count$ by 1
10:      else
11:        Output $Mutant_{md-ld-op} = Mutant_{md}$ and terminate
12:      end if
13:    end for
14:    $Mutant_{md-ld} = Mutant_{md}$
15:    if $Count < Mcount \times x\%$ then
16:      Add the mutant on $ld$ into $Mutant_{md-ld}$
17:      Increment $Count$ by 1
18:    else
19:      Output $Mutant_{md-ld-op} = Mutant_{md-ld}$ and terminate
20:    end if
21:  end for
22:  $Mutant_{md-ld-op} = Mutant_{md-ld}$
23:  if $Count < Mcount \times x\%$ then
24:    Add the mutant on $ld$ into $Mutant_{md-ld-op}$
25:    Increment $Count$ by 1
26:  else
27:    Output $Mutant_{md-ld-op}$ and terminate
28:  end if
29: end while
30: while $Count < Mcount \times x\%$ do
31:   Randomly select a mutant and add it into $Mutant_{md-ld-op}$
32:   Increment $Count$ by 1
33: end while
34: Output $Mutant_{md-ld-op} = Mutant_{md-ld}$ and terminate
```

depth of Mutant #3427 is $0 < S_m$, so we will not select it into $Mutant_{md}$. After applying Rule 1 to all mutants in Table 1, we have $Mutant_{md} = \{331, 425, 908, 1565\}$.

- For the *Rule 2 – loop/branch depth*: We assume the initial mutant set $Mutant_{ld} = \emptyset$, and set the module depth $S_l = 3$. For example, Mutant #1655 has loop/branch depth of 4, which is larger than $S_l$, so it is added into $Mutant_{ld}$. However, the loop/branch depth of Mutant #331 is $2 < S_l$, so it cannot be selected into $Mutant_{ld}$. After applying Rule 2 to all mutants in Table 1, we finally have $Mutant_{ld} = \{1655, 2065, 2081\}$.
- For the *Rule 3 – statement selection*: We assume the initial mutant set $Mutant_{stm} = \emptyset$ and the fault location set $L = \emptyset$. Suppose that the fault location of Mutant #2065 (that is, Line 125) is not in $L$. It will be added into $Mutant_{stm}$ and $L = \{125\}$. Then, Mutant #2081 is associated with a fault location (that is, Line 125), which is already in $L$, so it cannot be selected into $Mutant_{stm}$. Such a process is repeated until $L = Loc$, and we finally have $Mutant_{stm} = \{331, 425, 908, 1565, 1655, 1784, 1913, 2065, 3427\}$ and $L = \{67, 71, 81, 96, 123, 122, 118, 125, 147\}$.
- For the *Rule 4 – operator selection*: We assume the initial mutant set $Mutant_{op} = \emptyset$, and the operator set is $OP = \{CRCR, SSDL\}$. For example, Mutant #908 is associated with the mutation operator of type CRCR, which is in $OP$, so it is added into $Mutant_{op}$. After applying Rule 4 to all mutants in Table 1, we finally have $Mutant_{op} = \{908, 1565, 1784, 2081, 3427\}$.

In the following, we illustrate how to utilize the *md-ld-op* strategy to select a subset of mutants from those listed in Table 1. Assume that we set the sampling ratio as 50%, $S_m = 2$, $S_l = 3$, and $OP = \{CRCR, SSDL\}$. We first construct $Mutant_{md}$ based on the module depth: Only Mutant #1565 is associated with a module depth (3) larger than $S_m$, so we have $Mutant_{md} = \{1565\}$. We then extend $Mutant_{md}$ to $Mutant_{md-ld}$: There are three mutants (Mutants #1655, #2065, and #2081) that satisfying the loop/branch depth rule (that is, they are associated with the loop/branch depth larger than $S_l$; as a result, $Mutant_{md-ld} = \{1565, 1655, 2065, 2081\}$. Since the size of $Mutant_{md-ld}$ (that is, 4) is smaller than $50\% \times 10 = 5$, we need to apply the operator selection rule to select another mutant (say, Mutant #1784) to further extending $Mutant_{md-ld}$ to $Mutant_{md-ld-op} = \{1565, 1784, 1655, 2065, 2081\}$. It should be noted that multiple mutants may satisfy a certain rule, and in this study, we randomly choose some of them to obtain the required number of mutants. The other three mutant reduction strategies work in a similar way, except the different order of applied heuristic rules.

## 4. Empirical study

We have conducted an empirical study to validate the applicability and effectiveness of the four mutant reduction strategies, and compare cost-effectiveness of our mutant reduction strategies against random mutant selection. The design and settings of the study are introduced in this section.

### 4.1. Research questions

Our empirical study was designed to answer the following research questions:

**RQ1** What is the ranking among the four mutant reduction strategies with regard to the effectiveness of mutant reduction?
As discussed above, each mutant reduction strategy gives the highest priority to each of the four heuristic rules. The ranking among strategies will also imply the ranking among the heuristic rules, and thus help us decide which rule is the best one and thus should be paid the most attention to.

**RQ2** Is the proposed path-aware mutant reduction approach more effective than the random selection technique?
Random selection is not only a popular mutant reduction technique, but has also been justified to have comparable effectiveness to the more systematic operator-based approach [10]. Therefore, if our path-aware approach is shown to be superior to random selection, it is very likely to be quite effective in mutant reduction.

**RQ3** Can the path-aware approach help reduce the number of mutants without jeopardizing the effectiveness of mutation testing?
Suppose that a mutant reduction strategy selects a subset of mutants $Mu_{reduced}$ from the whole set of mutants $Mu_{all}$. Given any test suite $TS$ that can kill all mutants in $Mu_{reduced}$, if $TS$ can also kill all mutants in $Mu_{all}$, we can say that the mutants in $Mu_{reduced}$ is sufficiently representative of those in $Mu_{all}$, and thus the effectiveness of mutation testing is not jeopardized at all. Apparently, the more mutants $TS$ can kill in $Mu_{all}$, the better the mutation reduction strategy is.

**Table 2**
Object programs.

| Object program | Basic functionality | LOC | Number of mutants | | Size of |
|---|---|---|---|---|---|
| | | | Generated by Proteum | Removed equivalent mutants | test pool |
| `print_tokens` | Lexical analyzer | 483 | 5044 | 448 | 4130 |
| `print_tokens2` | Lxical analyzer | 402 | 4689 | 513 | 4115 |
| `replace` | Search and replace tool | 516 | 10,135 | 723 | 5542 |
| `schedule` | Priority scheduler | 299 | 1884 | 101 | 2650 |
| `schedule2` | Priority scheduler | 297 | 2716 | 128 | 2710 |
| `tcas` | Collision avoidance system | 138 | 2616 | 127 | 1052 |
| `tot_info` | Statistics for matrix | 346 | 4625 | 213 | 1608 |
| `bubble` | Sorting algorithm | 24 | 183 | 2 | 75 |
| `minmax` | Max and min values | 33 | 152 | 5 | 60 |
| `nextdate` | Date of next day | 83 | 501 | 40 | 377 |
| `triangle` | Triangle type | 23 | 191 | 9 | 188 |

### 4.2. Variables and measures

#### 4.2.1. Independent variables

The independent variable in our study is the mutant reduction strategy. All four strategies proposed in Section 3 were chosen and evaluated in the experiments. In addition, we selected the random mutant selection as the baseline technique for comparison.

#### 4.2.2. Dependent variables

We used the *mutation score* as the metric for RQ1, RQ2, and RQ3. Suppose that there are $N_{all}$ mutants, among which there are $N_{eq}$ equivalent mutants (the mutants that show exactly the same behaviors as the base program given any possible input). If a test suite $TS$ can kill $N_k$ mutants ($N_k \leq N_{all} - N_{eq}$), the mutation score of $TS$ is defined as

$$mutation\ score = \frac{N_k}{N_{all} - N_{eq}} \times 100\%. \quad (1)$$

In our experiment, for a given subset of mutants $Mu_{reduced}$ selected by a mutant reduction strategy, we constructed a test suite $TS$ that can achieve 100% mutation score on $Mu_{reduced}$. Then we applied $TS$ to test all mutants $Mu_{all}$, and measured $TS$'s mutation score $MS_{all}$ on $Mu_{all}$. The ideal case is $MS_{all} = 100$, which means that $Mu_{reduced}$ is sufficiently representative of $Mu_{all}$. Obviously, the higher $MS_{all}$ is, the more effective a mutant reduction strategy is.

### 4.3. Object programs

We selected two sets of programs as the objects for our empirical study. One set contains the famous seven Siemens programs [17], namely `print_tokens`, `print_tokens2`, `replace`, `schedule`, `schedule2`, `tcas`, and `tot_info`, which we downloaded from software-artifact infrastructure repository [18]. The other set consists of four scientific programs, namely `bubble`, `minmax`, `nextdate`, and `triangle`. All object programs are written in the C language. The first three columns in Table 2 summarize the basic information of these programs.

### 4.4. Mutant generation

We made use of Proteum [19], a C program mutation tool, to generate mutants for each object program. Though there are totally 108 mutation operators in Proteum, only 50 of them were suitable to our object programs. Some generated mutants were syntactically incorrect and thus incurred compiling errors, so they were eliminated. In our study, we only focused on the mutants whose associated faults are in single locations.

To exclude those equivalent mutants, we first ran all test cases (their generation process is reported in the next Section 4.5) on the mutants generated by Proteum. For those mutants that could not be killed by any test case, we then manually checked whether they are equivalent ones or not. To reduce uncertainty in the experiment, each identified equivalent mutant was cross-checked by different individuals. Finally, we excluded those confirmed equivalent mutants from our experiments.

Recently, a simple yet effective technique, namely trivial compiler equivalence (TCE), has been proposed for detecting equivalent mutants by comparing the object code of each mutant with that of the base program. We have tried to use the TCE technique to identify the equivalent mutants in our experiments. However, we found that the TCE technique was not applicable to our empirical studies. The mutation tool we used (i.e., Proteum) generates meta-mutants from which we cannot access object code for each single mutant. The unavailability of object code significantly restricted the applicability of the TCE technique for the generated mutants. Furthermore, the current mutation tool for TCE (namely Milu) supports 19 mutation operators, while Proteum supports 108 operators, among which, 50 were used in our study. In other words, it is very difficult, if not impossible, to use the current TCE technique to precisely identify the equivalent mutants from the mutants generated by Proteum in our experiments.

Columns 4 and 5 in Table 2 give the numbers of all generated mutants and removed equivalent mutants for each object program. From the table, we can observe that the number of mutants used for each program is 4596, 4176, 9412, 1783, 2588, 2489, 4412, 181, 147, 461, and 182, respectively.

### 4.5. Test case generation

Each of the seven Siemens programs is associated with a test pool composed of thousands of test cases [17]. In the original work [17], each test pool was constructed through two steps. First, the category-partition method [20] was applied to generate some initial test cases. Then, the initial test pool was extended by adding more test cases based on dataflow and controlflow coverage testing techniques [21] such that each exercisable element including “branch”, “predicate”, and “define and use association (DU)” was covered by at least 30 different test cases (that is, the coverage for statement, branch, predicate, and *DU* is 100%). In other words, each test pool was designed to achieve 100% coverage for the *DU*, *predicate*, and *branch* criteria. In our study, we made use of the same test pools for the Siemens programs. Similar steps were taken to construct the test pools for the other four scientific programs, that is, initial test cases generated based on functional specifications plus extra test cases constructed according to various coverage criteria. The number of test cases in the test pool for each object program is given in the last column of Table 2.

**Table 3**
Bonferroni mean separation tests for comparing five strategies on all mutants for each object program.

(a) `print_tokens`

| Group | | | | Strategy |
|---|---|---|---|---|
| A | | | | ld-md-op |
| | B | C | | md-ld-op |
| | | C | D | random |
| | | C | D | op-ld-md |
| | | | D | stm-ld-md |

(b) `print_tokens2`

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| A | | | md-ld-op |
| | B | | random |
| | B | C | stm-ld-md |
| | | C | op-ld-md |

(c) `replace`

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| | B | | md-ld-op |
| | B | C | random |
| | B | C | op-ld-md |
| | | C | stm-ld-md |

(d) `schedule`

| Group | | Strategy |
|---|---|---|
| A | | ld-md-op |
| | B | md-ld-op |
| | B | random |
| | B | stm-ld-md |
| | B | op-ld-md |

(e) `schedule2`

| Group | | | | Strategy |
|---|---|---|---|---|
| A | | | | ld-md-op |
| | B | | | md-ld-op |
| | | C | | random |
| | | C | D | stm-ld-md |
| | | | D | op-ld-md |

(f) `tcas`

| Group | | | | Strategy |
|---|---|---|---|---|
| A | | | | ld-md-op |
| | B | | | md-ld-op |
| | B | C | | random |
| | | C | D | stm-ld-md |
| | | | D | op-ld-md |

(g) `tot_info`

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| A | B | | md-ld-op |
| A | B | C | random |
| | B | C | stm-ld-md |
| | | C | op-ld-md |

(h) `bubble`

| Group | | Strategy |
|---|---|---|
| A | | ld-md-op |
| | B | md-ld-op |
| | B | stm-ld-md |
| | B | op-ld-md |
| | B | random |

(i) `minmax`

| Group | | Strategy |
|---|---|---|
| A | | ld-md-op |
| A | | md-ld-op |
| | B | stm-ld-md |
| | B | random |
| | B | op-ld-md |

(j) `nextdate`

| Group | | Strategy |
|---|---|---|
| A | | ld-md-op |
| A | | md-ld-op |
| A | B | random |
| | B | stm-ld-md |
| | B | op-ld-md |

(k) `triangle`

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| | B | | md-ld-op |
| | | C | random |
| | | C | stm-ld-md |
| | | C | op-ld-md |

## 4.6. *Experiment design*

The basic procedure of our experiment is as follows.

(a) Select an object program, which is associated with a set of all non-equivalent mutants, $Mu_{all}$;
(b) Set a real number $0 < x < 100$, where $x\%$ represents the *sampling ratio* of mutants;
(c) Apply one mutant reduction strategy (*md-ld-op, ld-md-op, stm-ld-md, op-ld-md*, or random selection) to select $x\%$ of all the mutants, that is, to get a subset of mutants, $Mu_{reduced}$.
(d) Construct a test suite *TS* using random testing technique, that is, randomly select test cases from the test pool for the object program until all mutants in $Mu_{reduced}$ have been killed.
(e) Apply *TS* to test all the mutants in $Mu_{all}$ and calculate the mutation score $MS_{all}$.

In our experiment, the value of $x$ in Step (b) was set as 1, 2, 3, 4, 5, 10, 20, 50, and 80.

## 4.7. *Threats to validity*

### 4.7.1. *Internal validity*

The major threat to internal validity is with the implementations of the mutant reduction strategies, which required a moderate amount of programming work. All the source code has been cross-checked by different individuals. We are confident that the experiments were conducted correctly.

### 4.7.2. *External validity*

The threat to external validity concerns about to what extent we can generalize our results. In this pilot study, we have selected 11 object programs, whose sizes are small. Though consistent results were shown in all these objects, we cannot say that our conclusions are applicable to any type of programs, especially those with a large scale. In the future, it is necessary to improve the external validity by comprehensively evaluating the proposed path-aware approach on various large-sized programs. In addition, though tens of mutation operators have been used in our study, it was very difficult, if not impossible, to cover all types of faults.

### 4.7.3. *Construct validity*

The threat to construct validity is related to the measurement. Our main metric, mutation score, is the most popular metric in the context of mutation testing, and has been used in almost all related studies.

### 4.7.4. *Conclusion validity*

In our experiment, we have examined the effectiveness of each mutant reduction strategy with nine different sampling ratios on 11 object programs. Thus, we have sufficient experimental data to help us draw conclusions with high confidence. In addition, we also used statistical testing to verify the statistical significance of our results.

## 5. Experimental results

## 5.1. *Results on all mutants*

Our experimental results are presented in Figs. 6 and 7. Fig. 6 shows the comparison of performance (measured by $MS_{all}$) on all mutants (excluding those not killed by any test case in the test pool) of each object program among the five mutant reduction strategies (*md-ld-op, ld-md-op, stm-ld-md, op-ld-md*, and random selection); while Fig. 7 shows the performance comparison on each individual sampling ratio (that is, $x\%$). In the figures, the box plot is used to display the distribution of experimental data. The upper, middle, and lower lines of each box represent the third quartile, median, and first quartile values of $MS_{all}$, respectively. The top and bottom whiskers refer to the maximum and minimum values, respectively. The mean value of $MS_{all}$ for each strategy is denoted by the round dot.

From Figs. 6 and 7, we can observe that the *ld-md-op* strategy always has the best performance in all cases. The *md-ld-op* strategy provides the second best performance; while it is difficult to distinguish the other three strategies.

We further conducted a statistical analysis to verify the statistical significance of our results. Bonferroni means separation tests were run to rank the five strategies in each case (that is, each object program or each sampling ratio). After the tests, all strategies were classified into different groups, as shown in Tables 3 and 4. If

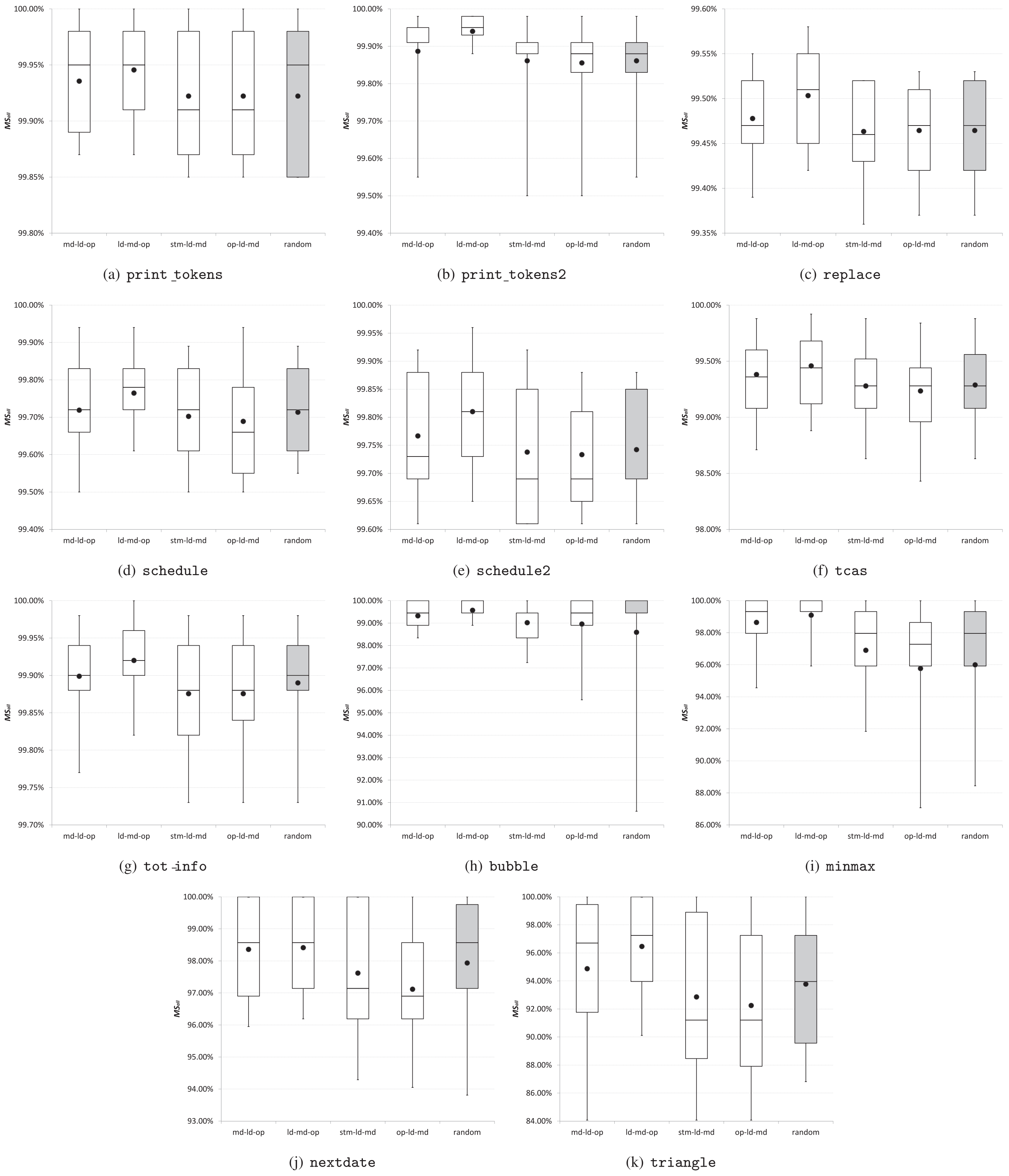
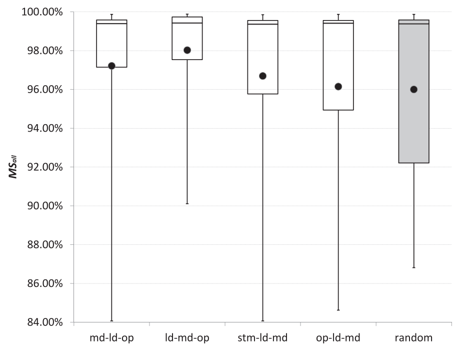
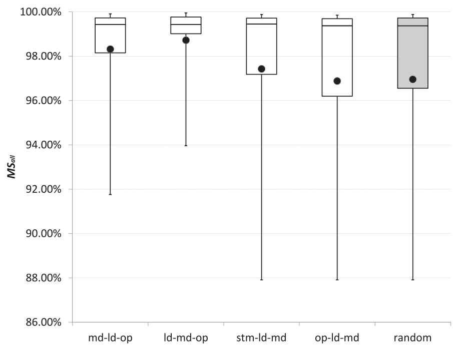


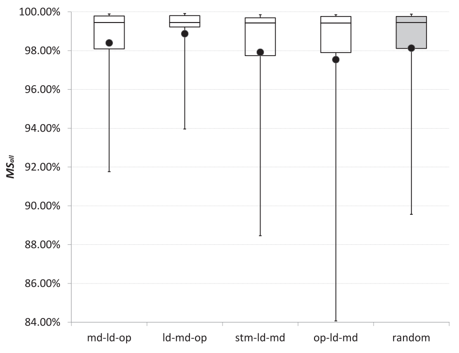
**Fig. 6.** Comparison of $MS_{all}$ on all mutants among five strategies on each object program.



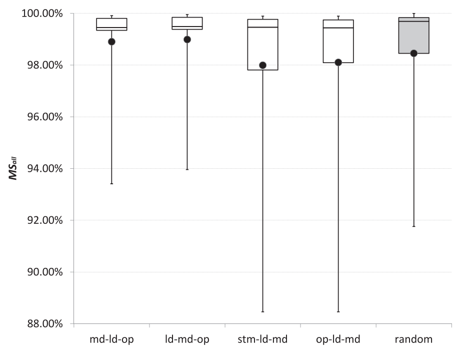
(a) 1%



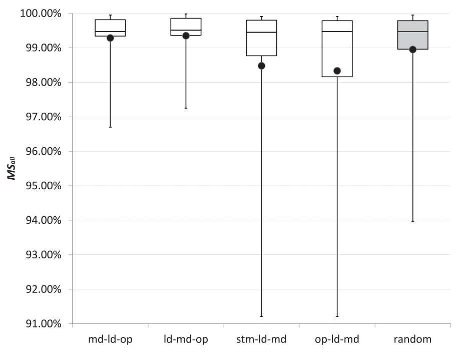
(b) 2%



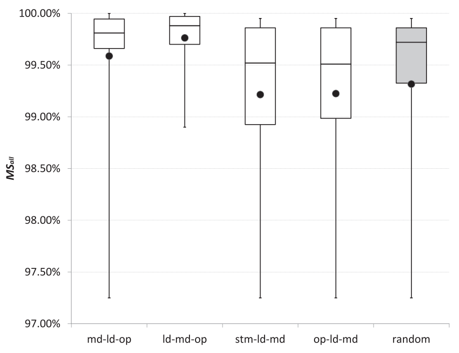
(c) 3%



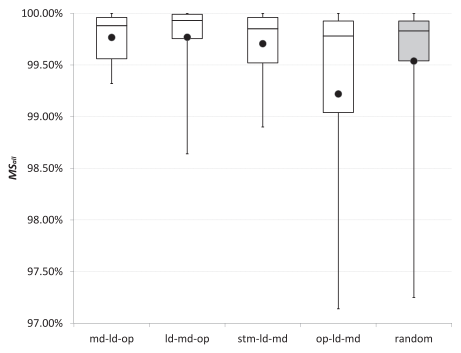
(d) 4%



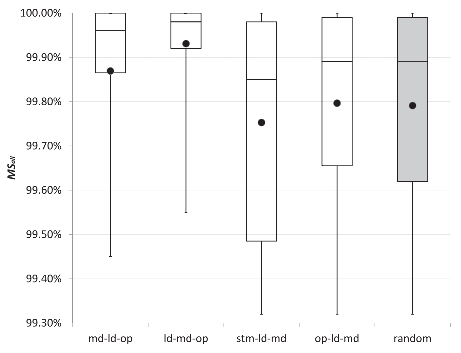
(e) 5%



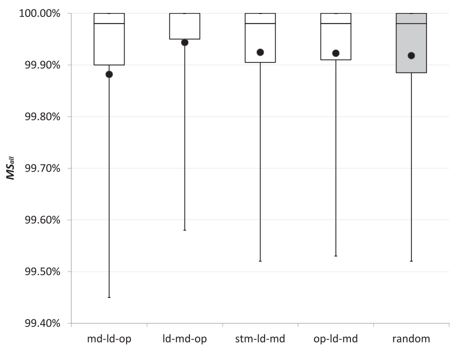
(f) 10%



(g) 20%



(h) 50%



(i) 80%

**Fig. 7.** Comparison of $MS_{all}$ on all mutants among five strategies on each sampling ratio.

**Table 4**
Bonferroni mean separation tests for comparing five strategies on all mutants for each sampling ratio.

| (a) 1% | | | (b) 2% | | | (c) 3% | | | (d) 4% | | | (e) 5% | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Group | | Strategy | Group | | Strategy | Group | | Strategy | Group | | Strategy | Group | | Strategy |
| A | | ld-md-op | A | | ld-md-op | A | | ld-md-op | A | | ld-md-op | A | | ld-md-op |
| A | B | md-ld-op | A | | md-ld-op | A | | md-ld-op | A | | md-ld-op | A | | md-ld-op |
| | B | stm-ld-md | | B | stm-ld-md | | B | random | | B | random | | B | random |
| | B | op-ld-md | | B | random | | B | stm-ld-md | | B | op-ld-md | | B | stm-ld-md |
| | B | random | | B | op-ld-md | | B | op-ld-md | | B | stm-ld-md | | B | op-ld-md |

| (f) 10% | | | (g) 20% | | | (h) 50% | | | (i) 80% | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Group | | Strategy | Group | | Strategy | Group | | Strategy | Group | | Strategy |
| A | | ld-md-op | A | | ld-md-op | A | | ld-md-op | A | | ld-md-op |
| A | B | md-ld-op | A | | md-ld-op | A | B | md-ld-op | A | B | md-ld-op |
| | B | random | A | B | stm-ld-md | | B | op-ld-md | A | B | op-ld-md |
| | B | op-ld-md | | B | random | | B | random | | B | stm-ld-md |
| | B | stm-ld-md | | B | op-ld-md | | B | stm-ld-md | | B | random |

**Table 5**
The number of stubborn mutants for each object program.

| Object program | Number of stubborn mutants |
|---|---|
| `print_tokens` | 7 |
| `print_tokens2` | 19 |
| `replace` | 59 |
| `schedule` | 8 |
| `schedule2` | 10 |
| `tcas` | 34 |
| `tot_info` | 12 |
| `bubble` | 17 |
| `minmax` | 17 |
| `nextdate` | 26 |
| `triangle` | 29 |

some strategies are in the same group, it implies that performance difference among them is not statistically significant. On the other hand, two strategies being in different groups means that their performances are significantly different. Note that one strategy can be classified into multiple groups. For example, in Table 4(a), the *ld-md-op* strategy only belongs to Group A, the *stm-ld-md, op-ld-md*, and random strategies belong to Group B, but the md-ld-op strategy belongs to both Groups A and B. This grouping means that we cannot statistically distinguish the *ld-md-op* and *md-ld-op* strategies (Group A); nor can we statistically distinguish the *md-ld-op, stm-ld-md, op-ld-md*, and random strategies (Group B); however, the *ld-md-op* strategy is statistically different from the *stm-ld-md, op-ld-md*, and random strategies (Group A vs. Group B). It can be observed that in most cases, the *ld-md-op* strategy significantly outperformed the three non-path-aware techniques (*stm-ld-md, op-ld-md*, and random). Though the performance of the *md-ld-op* strategy is normally better than those of the three non-path-aware techniques, their performance difference was not statistically significant in some cases (such as for the `bubble` program and the 10% sampling ratio).

## 5.2. Results on “stubborn” mutants

We observed from Figs. 6 and 7 that all strategies had very high mutation scores. In other words, many of the mutants used in our study are actually easy to be killed by most of test cases. Such “easy-to-kill” mutants are not very useful, as they cannot provide much information in distinguishing the quality of test cases and the effectiveness of different testing techniques, and also may introduce noise to the results. Therefore, it is more interesting to focus on those “stubborn” mutants, which are associated with low failure ratios (in other words, only a small part of test cases in the pool can kill the mutants). To do that, we conducted an extended experiment. In the experiment, we first removed those easy-to-kill mutants that can be killed by the random strategy with the sample ratio of 1%, and the remaining mutants are so-called stubborn mutants. We then investigated the effectiveness of our approach on the stubborn mutants. Table 5 gives the number of “stubborn” mutants for each object program.

The experimental results on the stubborn mutants are summarized in Figs. 8 and 9. The corresponding statistical analysis is reported in Tables 6 and 7. We made the similar observations on the stubborn mutants: the *ld-md-op* strategy always performed the best among all five techniques, and its performance was significantly better than other techniques in most cases; the *md-ld-op* strategy had the second best performance, but in some cases, it could not significantly outperform the three non-path-aware techniques (*stm-ld-md, op-ld-md*, and random); we could not statistically distinguish the performance of the three non-path-aware techniques.

## 5.3. Answers to research questions

Based on our experimental results shown in the previous two sections, we can answer our research questions as follows:

**Answer to RQ1** Among all four proposed strategies, *ld-md-op* was by far the best strategy, followed by *md-ld-op*. We cannot statistically distinguish *stm-ld-md* and *op-ld-md*, the performances of which were not significantly different from that of random strategy. The ranking among the strategies implies that the loop/branch depth was the best heuristic rule for mutant reduction. The module depth, which is a bit coarser than the loop/branch depth, was also a good mutant reduction rule.

**Answer to RQ2** *ld-md-op* and *md-ld-op* are the two strategies that give the highest priorities to the path-aware heuristic rules (that is, Rule 1 – module depth and Rule 2 – loop/branch depth), and thus can be considered to be the path-aware mutant reduction techniques. Both of them showed significantly higher effectiveness than the random selection technique. However, the other two strategies, which give the highest priorities to the statement selection and operator selection rules, respectively, even could not outperform the random strategy, which is consistent with previous studies on comparison between operator-based and random selection techniques [10].

**Answer to RQ3** Though no strategy could achieve the ideal case (that is, $MS_{all} = 100\%$ for all situations), the two path-aware mutant reduction techniques, namely *ld-md-op* and *md-ld-op*, normally delivered fairly high values of $MS_{all}$. In other words, these two techniques were able to select a subset of representative mutants from all mutants such that the effectiveness of mutation testing would not be jeopardized too much. Compared with them, the other two proposed strategies, namely *stm-ld-md* and *op-ld-md*, could only deliver a performance comparable to that of random selection strategy, which implies that they are not very effective in selecting representative mutants.

## 6. Related work

A lot of research has been conducted on mutation testing from various perspectives [22]. In this section, we focus on the work related to the cost reduction for mutation testing.

One major way to decrease the cost of mutation testing is mutant reduction. A straightforward approach for reducing the number of mutants is to randomly select part of mutants [23]. Mathur and Wong [8] systematically investigated the random mutant selection strategy, and gave some guidelines on the appropriate sampling size for random selection. More recently, Zhang at al. [10] proposed the so-called double random selection strategy, and justified that their method is fairly effective.

A more systematic mutant reduction approach is to select mutants based on part of mutation operators, which was first proposed by Mathur [9]. Offutt et al. [24] conducted a series of experiments and explicitly suggested that some mutation operators could be discarded in mutation testing without jeopardizing the effectiveness too much. Wong and Mathur [25] investigated all the 22 mutation operators in the ancient mutation testing tool Mothra, and identified two typical operators, based on which highly representative mutants could be selected. Offutt et al. [26] conducted more experiments on FORTRAN 77 programs, and pointed out five mutation operators that were sufficient by themselves to generate a subset of mutants with similar effectiveness to all possible mutants. Namin et al. [27] considered the mutation operator selection as a statistical problem, and used the rigorous statistical

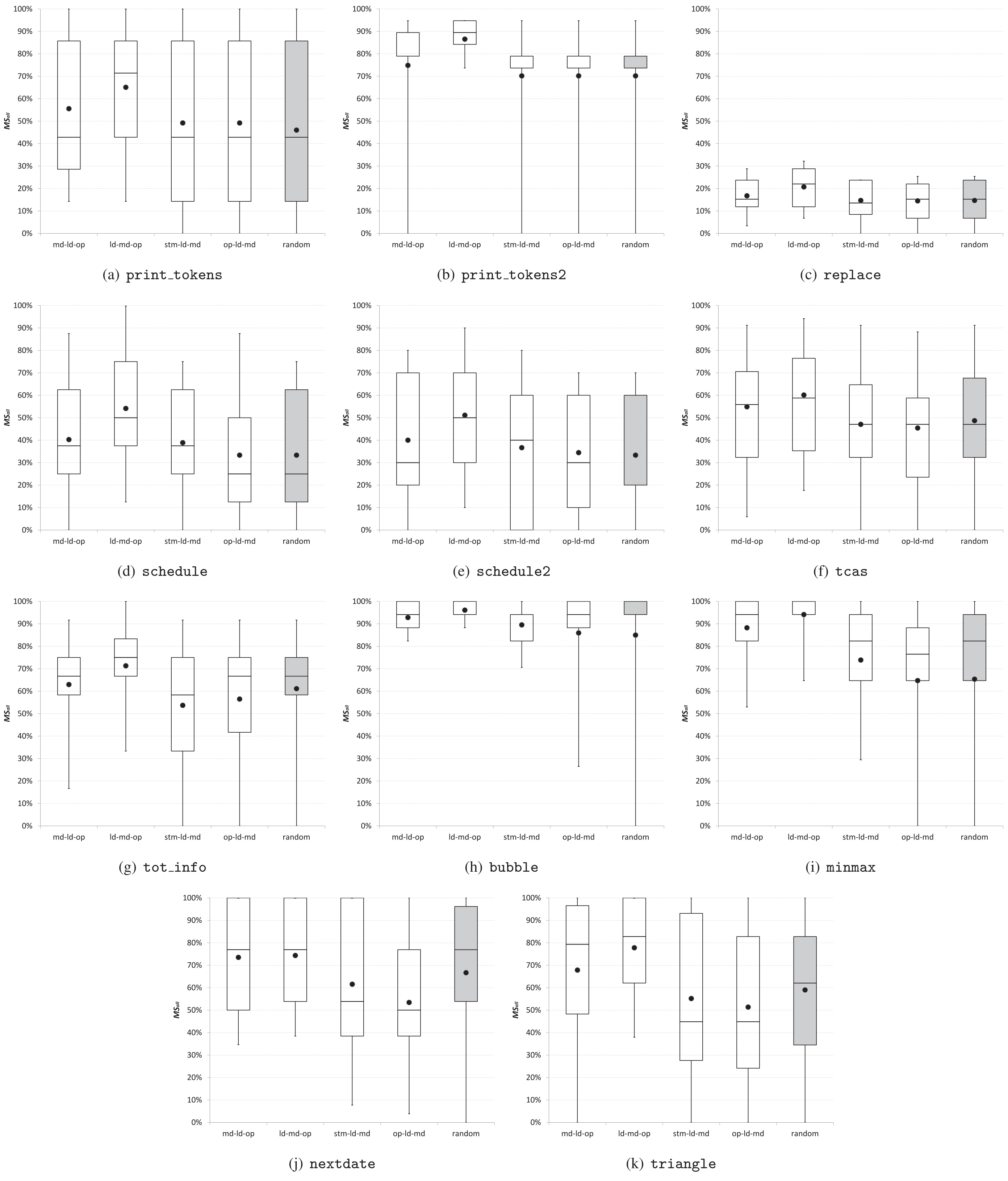


**Fig. 8.** Comparison of $MS_{all}$ on stubborn mutants among five strategies on each object program.

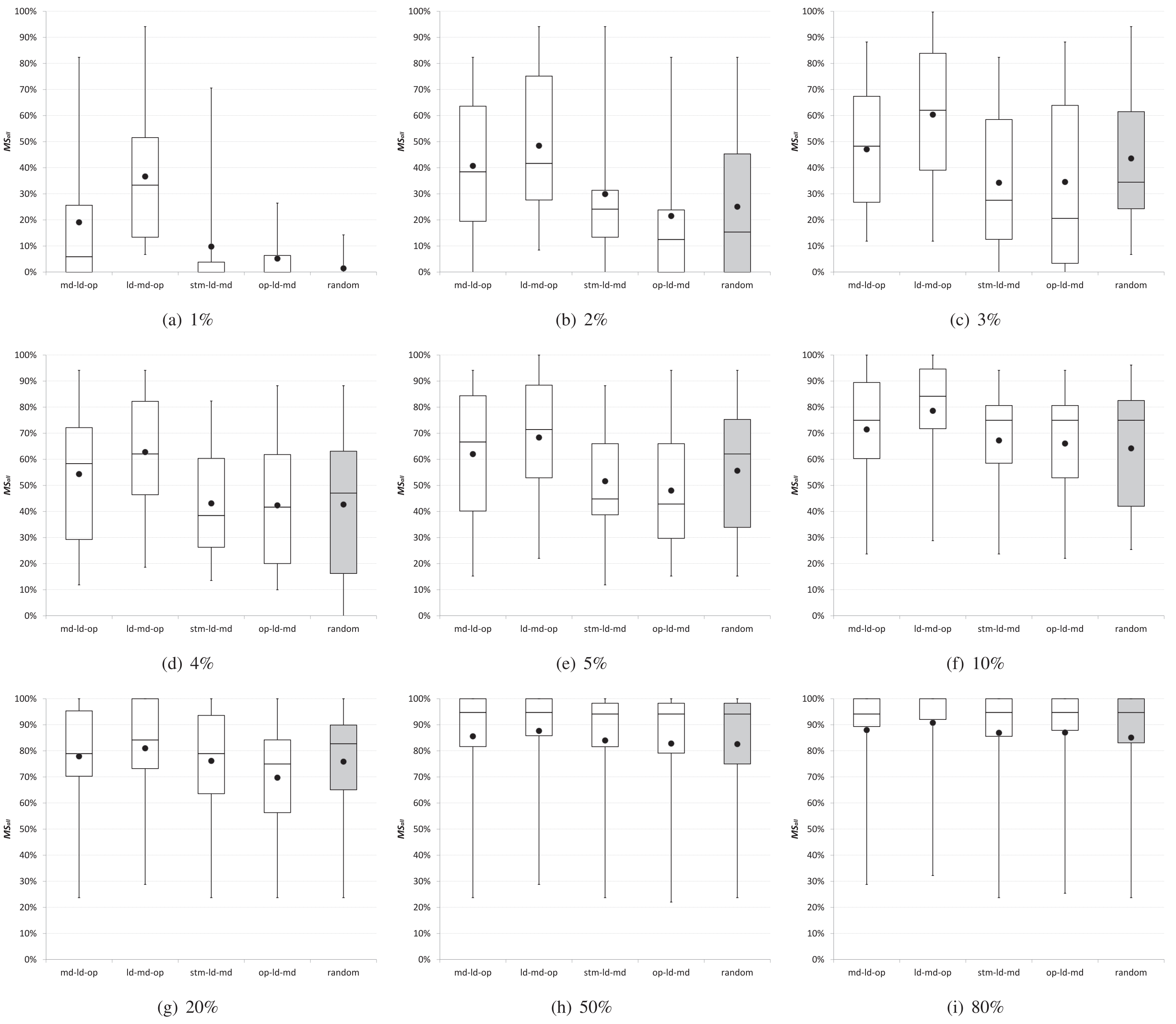


**Fig. 9.** Comparison of $MS_{all}$ on stubborn mutants among five strategies on each sampling ratio.

analysis to identify 28 typical mutation operators from all the 108 mutation operators in Proteum. Vincenzi et al. [28] proposed a family of incremental techniques to select part of mutation operators for unit and integration testing. These techniques could significantly reduce the number of mutants without jeopardizing the mutation scores. Just and Schweiggert [29] investigated three typical mutation operators for Java programs, and found that the existence of "redundant mutants" may affect both the efficiency and the quality of mutation testing. They also gave some guidelines on how to remove these redundant mutants and thus to improve the cost-effectiveness of mutation testing. Delamaro et al. [30] investigated the effectiveness of one-op mutation (that is, the mutation technique with only one powerful operator), and observed that the statement deletion operator (SDL) might be "the most cost-effective" mutation operators for C programs.

Some research has also been conducted to compare the random selection and operator-based selection. Zhang et al. [10] reported an experiment based on the seven Siemens programs, which demonstrated that random mutant selection could be equally effective as operator-based selection given the same number of mutants are selected. Gligoric and Zhang [31] investigated operator-based mutant selection for concurrent programs, and their results showed that selection based on parallel mutation operators could effectively reduce the number of mutants and outperformed random selection. Zhang et al. [32] studied the combination of the random selection and operator-based mutant selection, and observed that the random selection guided by certain program elements could be more effective than pure operator-based selection.

The use of high order mutation is a relatively new approach to mutant reduction. Jia and Harman [33] introduced the concept of "subsuming" high order mutant, which is harder to kill than any first order mutant used for constructing it. Polo et al. [34] has focused on the two order mutants on the research. They proposed three algorithms to combine the first-order mutants into second-order mutants. Papadakis et al. [35] conducted an empirical study to compare the first and second order mutation techniques, and observed that though the first order mutation is normally more effective than the second order mutation, the cost of the second

**Table 6**
Bonferroni mean separation tests for comparing five strategies on stubborn mutants for each object program.

(a) `print_tokens`

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| | B | | md-ld-op |
| | B | C | op-ld-md |
| | B | C | stm-ld-md |
| | | C | random |

(b) `print_tokens2`

| Group | | Strategy |
|---|---|---|
| A | | ld-md-op |
| A | | md-ld-op |
| | B | random |
| | B | stm-ld-md |
| | B | op-ld-md |

(c) `replace`

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| | B | | md-ld-op |
| | B | C | stm-ld-md |
| | B | C | random |
| | | C | op-ld-md |

(d) `schedule`

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| | B | | md-ld-op |
| | B | C | stm-ld-md |
| | | C | random |
| | | C | op-ld-md |

(e) `schedule2`

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| | B | | md-ld-op |
| | B | C | stm-ld-md |
| | B | C | op-ld-md |
| | | C | random |

(f) `tcas`

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| | B | | md-ld-op |
| | | C | random |
| | | C | stm-ld-md |
| | | C | op-ld-md |

(g) `tot_info`

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| | B | | md-ld-op |
| | B | C | random |
| | B | C | op-ld-md |
| | | C | stm-ld-md |

(h) `bubble`

| Group | | Strategy |
|---|---|---|
| A | | ld-md-op |
| | B | md-ld-op |
| | B | stm-ld-md |
| | B | op-ld-md |
| | B | random |

(i) `minmax`

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| A | | | md-ld-op |
| | B | | stm-ld-md |
| | | C | random |
| | | C | op-ld-md |

(j) `nextdate`

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| A | | | md-ld-op |
| A | B | | random |
| | B | C | stm-ld-md |
| | | C | op-ld-md |

(k) `triangle`

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| | B | | md-ld-op |
| | | C | random |
| | | C | stm-ld-md |
| | | C | op-ld-md |

**Table 7**
Bonferroni mean separation tests for comparing five strategies on stubborn mutants for each sampling ratio.

(a) 1%

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| | B | | md-ld-op |
| | | C | stm-ld-md |
| | | C | op-ld-md |
| | | C | random |

(b) 2%

| Group | | | | Strategy |
|---|---|---|---|---|
| A | | | | ld-md-op |
| | B | | | md-ld-op |
| | | C | | stm-ld-md |
| | | C | D | random |
| | | | D | op-ld-md |

(c) 3%

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| | B | | md-ld-op |
| | B | | random |
| | | C | stm-ld-md |
| | | C | op-ld-md |

(d) 4%

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| | B | | md-ld-op |
| | | C | stm-ld-m |
| | | C | random |
| | | C | op-ld-md |

(e) 5%

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| | B | | md-ld-op |
| | | C | random |
| | | C | stm-ld-m |
| | | C | op-ld-md |

(f) 10%

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| | B | | md-ld-op |
| | B | C | stm-ld-md |
| | | C | op-ld-md |
| | | C | random |

(g) 20%

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| | B | | md-ld-op |
| | B | C | stm-ld-md |
| | B | C | random |
| | | C | op-ld-md |

(h) 50%

| Group | | | Strategy |
|---|---|---|---|
| A | | | ld-md-op |
| | B | | md-ld-op |
| | B | | stm-ld-md |
| | B | C | op-ld-md |
| | | C | random |

(i) 80%

| Group | | Strategy |
|---|---|---|
| A | | ld-md-op |
| | B | md-ld-op |
| | B | op-ld-md |
| | B | stm-ld-md |
| | B | random |

order mutation is lower especially due to the much smaller number of equivalent mutants. Harman et al. [36] studied the efficiency and effectiveness of strong subsuming high order mutants (SSHOM). It was shown that the SSHOM technique could significantly reduce the number of mutants (up to 45% fewer than first order mutation), and some easy-to-kill first order mutants could be combined to construct “stubborn” SSHOMs, which would be very useful in mutation testing. There also exist other types of mutant reduction strategies in the literature, such as the clustering algorithm based selection [37] and the domain reduction technique [38].

The existence of equivalent mutants is also a critical factor incurring the high cost of mutation testing: It is often time-consuming and labor-intensive to decide whether a mutant is equivalent to the base program. Offutt and Craft [39] proposed a family of algorithms for identifying equivalent mutants based on data flow analysis and compiler optimization. Hierons et al. [40] used the program slicing technique to help detect the equivalent mutants as well as to directly decrease the number of equivalent mutants to be generated. Yao et al. [41] conducted a series of experiments, and found that some mutation operators tend to produce many equivalent mutants, while others can generate a lot of “stubborn” mutants (that is, those non-equivalent but hard-to-kill mutants). They suggested that mutation operators should be prioritized according to the equivalence and stubbornness. Papadakis et al. [42] investigated the effectiveness of mutant classification techniques in isolating equivalent mutants. It was observed that mutant classification could have high effectiveness when low-quality test suites were used. Kintis et al. [43] made use of second order mutation to isolate the first order equivalent mutants.

Papadakis et al. [44] introduced a novel technique for detecting equivalent mutants, namely trivial compiler equivalence (TCE), and conducted a large-sacle empirical study to evaluate the effectiveness of TCE. It was found that TCE was able to remove 28% of all generated mutants, with 7% being equivalent mutants and 21% duplicated mutants (that is, those mutants showing the same failure behaviors as others).

Another way to reduce the cost of mutation testing is to optimize the execution time. Delamaro et al. [19] made use of instrumentation in compilers to reduce the mutant generation and compiling time. Untch et al. [45] proposed a schema-based approach, which can generate one single meta-mutant that contains all mutants. Ma et al. [46] further combined the schema-based approach with byte-code translation technique, which allowed the mutants to be directly executed and thus saved the compiling time. A novel regression mutation testing technique called ReMT [47] was proposed to improve the efficiency of mutation testing during software evolution process. Zhang et al. [48] developed another technique called FaMT that minimizes and prioritizes test cases for each mutant such that the mutation testing results could be collected more quickly. In some mutation testing tools [49,50], coverage information was explored to prevent redundant executions of some mutants, and thus to decrease the overall time of mutation testing.

Papadakis and Malevris [51] also applied the information about the program paths in mutation testing. They proposed a path based approach for generating test cases that are effective in killing mutants. Different from their work, our study makes use of the path information in the selection of a subset of representative mutants. Though both studies can ultimately improve the cost-effectiveness of mutation testing, they address different problems from different perspectives.

## 7. Conclusion

Mutation testing is a popularly used technique for evaluating the effectiveness of a testing method. However, it incurs a high cost mainly due to the large amount of mutants it generates and executes. Some approaches have been proposed to reduce the cost of mutation testing by selecting a subset of mutants, such as random selection and operator-based selection. In this paper, different from all previous work, we proposed a new mutant reduction approach, from the perspective of the path depth in the program under test. We presented two path-aware heuristic rules as well as the statement and operator selection rules, and developed four mutant reduction strategies that are associated with different priority orders of the selection rules. The effectiveness of the new strategies were evaluated via an empirical study on 11 real-world C programs. It was observed that two strategies, which give highest priorities to the path-aware rules, could select representative mutants that were more effective and precise than those selected in a random manner. It was also shown that these two strategies outperformed the other two strategies, which mainly rely on statement or operator selection rules. In brief, the proposed path-aware approach is a better technique in mutant reduction than the traditional random and/or operator-based selection approaches. Therefore, the work presented in this paper advances mutation testing by further reducing the number for mutants without jeopardizing its fault detection effectiveness.

In this pilot study, we only examined two path-aware heuristic rules and four mutant reduction strategies. It is important to continue the work to investigate more heuristic rules and design more precise reduction strategies. The mutant reduction is particularly useful if the program under test is large in scale and complex in structure (which implies the corresponding mutation testing will be very costly). Thus, future work must be conducted to evaluate the applicability and effectiveness of our path-aware mutant reduction approach on more large-scale programs from various application domains.

## Acknowledgment

This research is supported by the National Natural Science Foundation of China under Grant No. 61370061, the Beijing Natural Science Foundation of China (Grant No. 4162040), the Beijing Municipal Training Program for Excellent Talents under Grant No. 2012D009006000002, and the Fundamental Research Funds for the Central Universities under Grant No. FRF-SD-12-015A. Thanks to the anonymous reviewers who provided useful suggestions on earlier version of this paper. All correspondence should be addressed to both Chang-ai Sun and Huai Liu.

## References

[1] R.G. Hamlet, Testing programs with the aid of a compiler, IEEE Trans. Softw. Eng. 3 (4) (1977) 279–290.
[2] R.A. DeMillo, R.J. Lipton, F.G. Sayward, Hints on test data selection: help for the practicing programmer, IEEE Comput. 11 (4) (1978) 34–41.
[3] G. Fraser, A. Zeller, Mutation-driven generation of unit tests and oracles, in: Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA 2010), 2010, pp. 147–158.
[4] M. Papadakis, Y. Le Traon, Metallaxis-FL: Mutation-based fault localization, Softw. Test., Verific. Reliab 25 (5-7) (2015) 605–628.
[5] A.J. Offutt, R.H. Untch, Mutation 2000: uniting the orthogonal, in: Mutation Testing for the New Century, Kluwer Academic Publishers, 2001, pp. 34–44.
[6] J.H. Andrews, L.C. Briand, Y. Labiche, Is mutation an appropriate tool for testing experiments? in: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), 2005, pp. 402–411.
[7] R. Just, D. Jalali, L. Inozemtseva, M.D. Ernst, R. Holmes, G. Fraser, Are mutants a valid substitute for real faults in software testing? in: Proceedings of the Symposium on the Foundations of Software Engineering (FSE 2014), 2014, pp. 654–665.
[8] A.P. Mathur, W.E. Wong, An empirical comparison of data flow and mutation-based test adequacy criteria, Softw. Test., Verific. Reliab. 4 (1) (1993) 9–31.
[9] A.P. Mathur, Performance, effectiveness, and reliability issues in software testing, in: Proceedings of the 5th International Computer, Software and Applications Conference (COMPSAC 1991), 1991, pp. 604–605.
[10] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, H. Mei, Is operator-based mutant selection superior to random mutant selection? in: Proceedings of the 32nd International Conference on Software Engineering (ICSE 2010), 2010, pp. 435–444.
[11] C. Sun, C. Liu, M. Jin, Effective wove algorithm for software structure graph, J. Beijing Univ. Aeronaut. Astronaut. 26 (6) (2000) 705–709.
[12] J. Ferrante, K.J. Ottenstein, J.D. Warren, The program dependence graph and its use in optimization, ACM Trans. Program. Lang. Syst. 9 (3) (1987) 319–349.
[13] Z. Lin, X. Zhang, Deriving input syntactic structure from execution, in: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2008), 2008, pp. 83–93.
[14] H. Liu, F.-C. Kuo, T.Y. Chen, Comparison of adaptive random testing and random testing under various testing and debugging scenarios, Softw.: Pract. Exp. 42 (8) (2012) 1055–1074.
[15] L. Morell, A theory of fault-based testing, IEEE Trans. Softw. Eng. 16 (8) (1990) 844–857.
[16] C. Sun, G. Wang, MujavaX: a distribution-aware mutation generation system for java, J. Comput. Res. Dev. 51 (4) (2014) 874–881.
[17] M. Hutchins, H. Foster, T. Goradia, T. Ostrand, Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria, in: Proceedings of the 16th International Conference on Software Engineering (ICSE 1994), 1994, pp. 191–200.
[18] H. Do, S.G. Elbaum, G. Rothermel, Supporting controlled experimentation with testing techniques: an infrastructure and its potential impact, Empir. Softw. Eng.: Int. J. 10 (4) (2005) 405–435.
[19] M.E. Delamaro, J.C. Maldonado, Proteum — a tool for the assessment of test adequacy for C programs, in: Proceedings of the Conference on Performability in Computing Systems (PCS 1996), 1996, pp. 79–95.
[20] T.J. Ostrand, M.J. Balcer, The category-partition method for specifying and generating fuctional tests, Communn. ACM 31 (6) (1988) 676–686.
[21] H. Zhu, P.A.V. Hall, J.H.R. May, Software unit test coverage and adequacy, ACM Comput. Surv. 29 (4) (1997) 366–427.
[22] Y. Jia, M. Harman, An analysis and survey of the development of mutation testing, IEEE Trans. Softw. Eng. 37 (5) (2011) 649–678.
[23] A.T. Acree Jr., On Mutation, Georgia Institute of Technology, 1980 Ph.D. thesis.
[24] A.J. Offutt, G. Rothermel, C. Zapf, An experimental evaluation of selective mutation, in: Proceedings of the 15th International Conference on Software Engineering (ICSE 1993), 1993, pp. 100–107.
[25] E.W. Wong, A.P. Mathur, Reducing the cost of mutation testing: an empirical study, J. Syst. Softw. 31 (3) (1995) 185–196.

[26] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, C. Zapf, An experimental determination of sufficient mutant operators, ACM Trans. Softw. Eng. Methodol. 5 (2) (1996) 99–118.
[27] A.S. Namin, J.H. Andrews, D.J. Murdoch, Sufficient mutation operators for measuring test effectiveness, in: Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), 2008, pp. 351–360.
[28] A.M.R. Vincenzi, J.C. Maldonado, E.F. Barbosa, M.E. Delamaro, Unit and integration testing strategies for C programs using mutation, Softw. Test., Verific. Reliab. 11 (3) (2001) 249–268.
[29] R. Just, F. Schweiggert, Higher accuracy and lower run time: efficient mutation analysis using non-redundant mutation operators, Softw. Test., Verific. Reliab 25 (5-7) (2015) 490–507.
[30] M.E. Delamaro, L. Deng, V.H.S. Durelli, N. Li, J. Offutt, Experimental evaluation of SDL and one-op mutation for C, in: Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation (ICST 2014), 2014, pp. 203–212.
[31] M. Gligoric, L. Zhang, Selective mutation testing for concurrent code, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013), 2013, pp. 224–234.
[32] L. Zhang, M. Gligoric, D. Marinov, S. Khurshid, Operator-based and random mutant selection: better together, in: Proceedings of the 28th Automated Software Engineering (ASE 2013), 2013, pp. 92–102.
[33] Y. Jia, M. Harman, Constructing subtle faults using higher order mutation testing, in: Proceedings of the 8th International Working Conference on Source Code Analysis and Manipulation (SCAM 2008), 2008, pp. 249–258.
[34] M. Polo, M. Piattini, I. Garcia-Rodriguez, Decreasing the cost of mutation testing with second-order mutants, Softw. Test., Verific. Reliab. 19 (2) (2009) 111–131.
[35] M. Papadakis, N. Malevris, An empirical evaluation of the first and second order mutation testing strategies, in: Proceedings of the 3rd International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), 2010, pp. 90–99.
[36] M. Harman, Y. Jia, P. Reales Mateo, M. Polo, Angels and monsters: an empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation, in: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE 2014), 2014, pp. 397–408.
[37] S. Hussain, Mutation Clustering, King's College London, 2008 Ph.D. thesis.
[38] C. Ji, Z. Chen, B. Xu, Z. Zhao, A novel method of mutation clustering based on domain analysis, in: Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE 2009), 2009, pp. 1–3.
[39] A.J. Offutt, W.M. Craft, Using compiler optimization techniques to detect equivalent mutants, Softw. Test., Verific. Reliab. 4 (3) (1994) 131–154.
[40] R. Hierons, M. Harman, S. Danicic, Using program slicing to assist in the detection of equivalent mutants, Softw. Test., Verific. Reliab. 9 (4) (1999) 233–262.
[41] X. Yao, M. Harman, Y. Jia, A study of equivalent and stubborn mutation operators using human analysis of equivalence, in: Proceedings of the 36th International Conference on Software Engineering (ICSE 2014), 2014, pp. 919–930.
[42] M. Papadakis, M. Delamaro, Y. Le Traon, Mitigating the effects of equivalent mutants with mutant classification strategies, Sci. Comput. Program. 95, Part 3 (2014) 298–319.
[43] M. Kintis, M. Papadakis, N. Malevris, Employing second-order mutation for isolating first-order equivalent mutants, Softw. Test., Verific. Reliab. 25 (5-7) (2015) 508–535.
[44] M. Papadakis, Y. Jia, M. Harman, Y. Le Traon, Trivial compiler equivalence: a large scale empirical study of a simple, fast and effective equivalent mutant detection technique, in: Proceedings of the 37th International Conference on Software Engineering (ICSE 2015), 2015, pp. 936–946.
[45] R.H. Untch, A.J. Offutt, M.J. Harrold, Mutation analysis using mutant schemata, in: Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA 1993), 1993, pp. 139–148.
[46] Y.S. Ma, A.J. Offutt, Y.R. Kwon, MuJava: an automated class mutation system, Softw. Test., Verific. Reliab. 15 (2) (2005) 97–133.
[47] L. Zhang, D. Marinov, L. Zhang, S. Khurshid, Regression mutation testing, in: Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012), 2012, pp. 331–341.
[48] L. Zhang, D. Marinov, S. Khurshid, Faster mutation testing inspired by test prioritization and reduction, in: Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013), 2013, pp. 235–245.
[49] D. Schuler, A. Zeller, Javalanche: efficient mutation testing for Java, in: Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering (ESEC/FSE2009), 2009, pp. 297–298.
[50] R. Just, F. Schweiggert, G.M. Kapfhammer, MAJOR: an efficient and extensible tool for mutation analysis in a Java compiler, in: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011), 2011, pp. 612–615.
[51] M. Papadakis, N. Malevris, Mutation based test case generation via a path selection strategy, Inf. Softw. Technol. 54 (9) (2012) 915–932.