A New Mutant Generation Algorithm Based on Basic Path Coverage for Mutant Reduction

Xu Qin[1,2](✉), Shaoying Liu[1], and Zhang Tao[2]

[1] Faculty of Computer and Information Sciences, Hosei University, Tokyo, Japan
qin.xu.6d@stu.hosei.ac.jp
[2] School of Software Northwestern, Polytechnical University, Xi'an, China

Abstract. Mutation testing is a fault-based testing technique that can be used to measure the adequacy of a test set, but its application usually incurs a high cost due to the necessity of generating and executing a great number of mutants. How to reduce the cost still remains a challenge for research. In this paper, we present a new mutant generation algorithm based on a basic path coverage that can help reduce mutants. The algorithm is characterized by implementing a basic path segments identification criterion for determining appropriate program points at which faults are inserted and a mutant generation priority criterion for selecting proper mutant operators to make a fault for insertion. We discuss the algorithm by analysing how the two criteria are realized based on analysing the control flow graph of the program and applying effective mutation operators on the appropriate statements in the relevant path segments. We also present an automated mutation testing tool that supports the proposed approach, and a small experiment to evaluate our tool by comparing it with a traditional mutation testing method on six programs. The result of the experiment suggests that the proposed method can significantly reduce the number of mutants and improve the efficiency of mutation testing.

Keywords: Mutation testing · Path coverage · Mutant reduction

1 Introduction

Mutation testing [1] is a effectively white-box testing technique which can be used to evaluate and improve test suite's adequacy, and predict the possible faults present in our system. As a testing technique, mutation testing can truly reveal various flaws of software. But in industry the mutation testing technique has not been widely applied [2,3]. The main reason is that mutation testing is so time-consuming (a large number of mutants and long execution time).

In recent years, many researches have been carried out in order to apply mutation testing in practical application, including mutant random selection [10], high-order mutant [12], mutant clustering methods [13], selective mutation operators [11], mutant detection optimization [18], mutant compilation optimization [19], and parallel execution of mutants [20].

Although the above mutant reduction methods have been used, there is little work on combining the mutation testing with the conventional path coverage testing. In our research, an algorithm for generating mutants based on the basic path coverage is proposed. We first introduce the criteria for identifying the path segments where faults need to be inserted and the mutant generation priority criteria for producing mutants. We then present a mutant generation algorithm under the constraint of these criteria. The main idea is to insert simple syntactic changes on each basic path of the given program to produce mutants. Combining the traditional path testing coverage with the mutation testing can not only assess the efficiency of the ability of test suite for detecting some possible faults, but also achieve basic path coverage which can effectively improve the effectiveness of given test case set.

It is difficult to do mutation testing without a software tool. So we present an a mutation testing tool that supports the proposed approach, which can generates and executes the mutants automatically. We design three components for our tool: the mutant generator, the mutant viewer, and the test executor. The mutant generator component generates mutants automatically by using the effective mutation operators. It takes Java files as input and a set of mutants as output. The mutant viewer lists the information of our generated mutants. And it also show the original code and the code of each mutant, which can help us to know which part was changed. The test executor run the test case set on generated mutants and shows the test result by analyzing the mutation score of our test case set. All this three components provide GUI for testers to use. We have carried out an experiment on the tool for validating the effectiveness of our proposed approach. From the result we can see that our method can significantly improve the efficiency of mutation testing by reducing mutants but also with high mutation score.

Here lists our contributions:

- Proposing a new mutant generation algorithm by combing the mutation testing and basic path coverage testing for reducing generated mutants.
- Designing and implementing a mutation testing tool to validate our proposed method.

The rest of the paper has the following organization. Section 2 introduces the related work on reducing the mutation testing cost. Section 3 gives some definitions and discuses the implementation process of our proposed mutant generation algorithm based on basic path coverage. Section 4 designs and implements a mutation testing tool to support the proposed approach. Section 5 presents an empirical research to evaluate the efficiency of our proposed method. The conclusion of this paper and the future direction of our research are shown in Sect. 6.

2 Background on Mutation Testing and Program Control Flow Graph

In this section, we introduce some basic concepts used in our discussions throughout this paper. They are known as mutation testing, program control flow graph, and independent path.

2.1 Mutation Testing

The mutation testing is a fault-based software testing technique with two steps. The first step is to use a specific mutation operator to simulate a particular type of error and implant the error into the source program. The second step is to explore the source program with a given test set, and assess the adequacy of the test set by the mutation score.

The traditional mutation testing process is described below, Fig. 1 graphically shows a traditional mutation process.

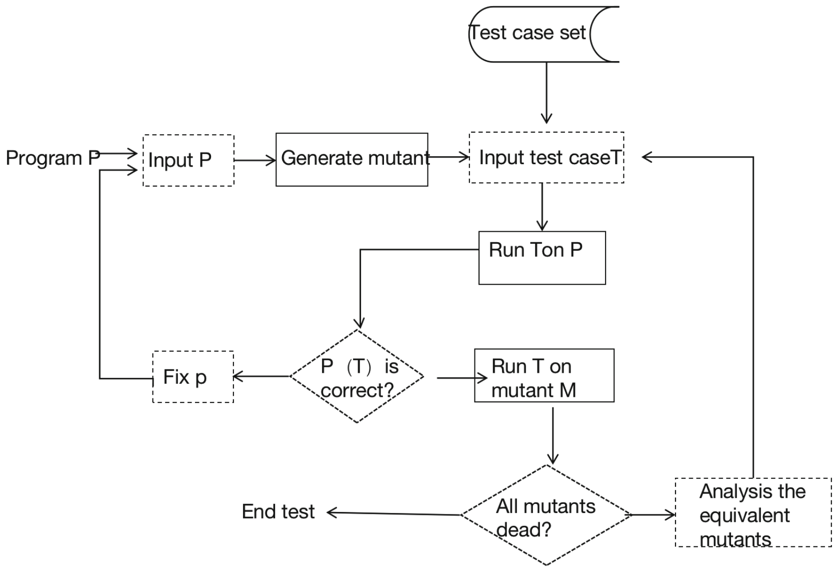


Fig. 1. Traditional mutation testing process

- Input the original program P and use the mutation operators to generate mutants M.
- Input the given test cases and then run the test case on P to get a expected result.

- Run the test case T on each generated mutant. And then compare the result with expected result. If the two is different, we can say the mutant is killed, else the mutant is marked as being alive.
- After running all test cases on mutants, we compute a mutation score. The mutation score is the ratio of killed mutants as a percentage of all mutants. The higher mutation score is, the more mutants are killed. So we need to improve the mutation score to 1.00, suggesting that we killed all the generated mutants. If the test case set can kill all the mutants, we can say it is enough for our testing.
- If the mutation score meets the requirements, the mutation testing ends. Otherwise, the tester need to add more test cases to kill the live mutants. Then we repeat adding new test cases, run the test case and compute a new mutation score until we get a high mutation score.

2.2 Program Control Flow Graph

Program control flow graph is an abstract representation of a process or program. In computer science, a control-flow graph lists all the control flows and shows all the possible path when we execute the program. It was first proposed by Frances. E. Allen in 1970 [5]. The program control flow graph is defined as follows:

A control flow graph, can be represented by CFG = (B, E, nentry, nexit) where B is a node (represents some statements of program P) in the graph and E is edges in CFG [5]. If there is a directed edge goes from node b_i to node b_j, we can say the ordered pair (b_i, b_j) of nodes is an edge. Nentry and nexit are the entry and exit node of the program, respectively. It has a unique starting node START and a unique ending node STOP. There can be at most two direct successors for each node in the CFG. For a node with two direct successors, its outgoing edge has the attribute 'T' or 'F', for a node with only one direct successor, its outgoing edge has the default attribute 'T'. And any node N in the CFG has a path from START to N to STOP.

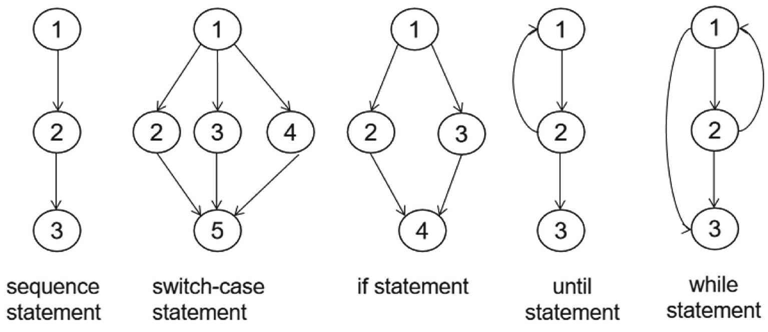


Fig. 2. The CFG of the basic program structures

A node is a linear sequence of program statements, which may have some predecessor nodes and many successor nodes. A START node doesn't have predecessor node and a STOP node doesnt have successor node.

The CFG of the basic program structures are shown in Fig. 2: each statements in our program under test is represented by a node in CFG. Then, we can get all the independent paths (basic paths) of our program by tracing the flow. An independent path or basic path means that at least one new processing statement or a newly determined program path is introduced compared to other independent paths. If we go thorough each basic path, we can say that we have executed all the statement in our program at least once and each basic path has been judged to 'TRUE' or 'FALSE'.

3 Our Proposed Mutant Generation Algorithm

In this part, we propose a new mutant generation algorithm, which aims to generate a less mutants that can simulate software defects. Based on the selective mutation technique, this algorithm uses mutation points as research objects and selects appropriate program path segments and target sentences for mutation on each basic path. The proposed algorithm combines mutation testing and basic path coverage testing.

3.1 Preliminary

In order to facilitate the description of the algorithm, we first give the following definitions:

Definition 1 (Immediate predecessor node and successor node). A immediate predecessor node n_h of a node n_i in is a node that satisfy $P_G(n_i) = n_h|(n_h, n_i) \in E$. ($P_G(n_i)$ means the immediate predecessor node of node n_i; E is a set of directed edges in G.)

A immediate successor node n_j of a node n_i in is a node that satisfy $S_G(n_i) = n_j|(n_i, n_j) \in E$. ($S_G(n_i)$ means the immediate successor node of node n_i.)

Definition 2 (Leaf node, sequence node and selection node). A leaf node n_i is a node that satisfy $S_G(n_i) = \emptyset$ and $\exists! P_G(n_i)$. It means a leaf node only have one input edge and no output edge.

A sequence node n_i is a node that satisfy $\exists! P_G(n_i)$ and $\exists! S_G(n_i)$. It means a sequence node only has one input edge and one output edge.

A selection node n_i is a node containing a condition. It means a selection node have more than one output edge. Note that compound Boolean expressions generate at least two predicate node in control flow graph.

Definition 3 (Sequence path-segment, Unique path-segment). A sequence path-segment $sp = (n_1, n_2, ...n_n)$ is a path segment that the first node is a selection node and the other nodes n_i of sp is either a sequence node or a leaf node.

A unique path-segment is a path-segment that satisfy $ups_i = (N_i, E_i)|E_i \notin \forall ups_j$. It means that any edges of a *ups* is unique from other unique path-segments.

3.2 Recognition of Target Path Segments to Be Mutated

In order to generate mutants, we need to find the target mutation path segments and then select the target mutation sentence in these segments to generate the mutant using the appropriate mutation operator. Here, we propose a identification criterion for path segments to be inserted into error. The purpose is to select the unique path segments in the basic path for error insertion, ensure the basic path coverage, and make the mutation of each path as independent as possible. And also a mutation sentence selection priority criteria is proposed to select the node with high importance as the target sentence, which choose the sentence that has a greater impact on the execution result of each path in the basic path. For each criterion, a simple example will be given.

Fault Insertion Path Segments Identification Rule 1 (R1)
If there exists a leaf node n_i in the control flow graph, then trace back to its immediate predecessor node n_h, if n_h is a sequence node, then continue to trace back to its immediate predecessor node until we meet a non sequence node n_a, and mark this sequence-path segment $sp = (n_a, ..., n_h, n_i)$ as a path segment to be inserted into fault.(a fault is a simple syntactic change).

The following figure Fig. 3 illustrates the application of R1: node 5 is a leaf node and trace back to its immediate predecessor node 4 and continue to trace back to the non sequence node 2. Then marks the sequence-path segment $sp = (n_2, n_4, n_5)$ as a path segment for fault insertion.

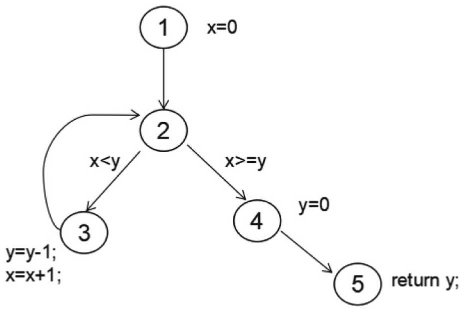


Fig. 3. A example for identification rule 1

Fault Insertion Path Segments Identification Rule 2 (R2)
Find all the unique path-segment ups $ups_i = (N_i, E_i)|E_i \notin \forall ups_j$ in each basic paths in the control flow graph CFG and mark this ups as a path segment. If

there is a loop structure in the program, the basic path only includes no loop and one loop.

Figure 4 illustrates the application of the rule R2: we can find all the basic paths, path1:1-7; path2:1-2-3-6-7; path3:1-2-4-6-7; path4:1-2-4-5-6-7, and then find the unique path-segment ups1:1-7; ups2:2-3-6; ups3:4-6; ups4:-4-5-6. These unique path-segments are path segments suitable for fault insertion.

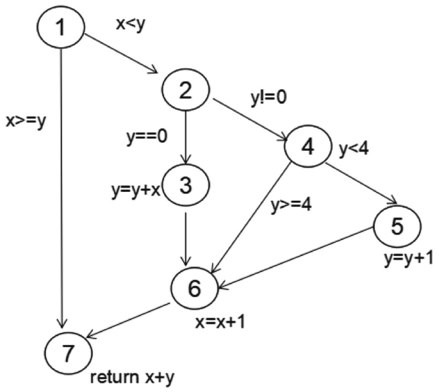


Fig. 4. A example for identification rule 2

3.3 Recognition of Target Path Segments to Be Mutated

The statements in a program are not purely independent, they have a certain relationship. Each statement has different effects on the execution of different paths. Using the relationship between them, selecting important sentences for mutation testing to generate mutants can reduce testing cost and improve testing efficiency.

This part proposes a criterion for selecting target sentences to be mutated based on our mutation sentence selection priority criteria. Different from the traditional mutation test method, we first analyze the sentence corresponding to each node in the basic path, and select the node with high importance as the target sentence. First establish the evaluation index, then set the sentence priority, and finally propose our mutation sentence selection priority criteria to generate mutants for the target mutation path segments in the previous section.

Evaluation Index: First, consider whether the node where the statement is located will cause a change in the execution path. Based on our previous definition of different nodes in the path. Different nodes have different effects on each path. Among them, the sequence node (expression statement) has the least impact on the execution path. It only affects the execution results of this path and has no effect on other paths. The second is the single node (return value statement), which mainly affects the return value. Finally, the selection node (loop statements, conditional statements) have the greatest impact on the program, it may cause the execution path to change.

Next consider the number of variables in the statement. In general, more variables have a greater impact on program execution. So it makes sense to consider how many variables the target statement contains.

Set the Statement Priority: Our goal is to select the statements that will affect the execution results of this path but have less impact on the execution results of other paths. Through the analysis of the above evaluation indicators, we can know that evaluation index 1 is negatively correlated with our goal, so our priority setting should be: sequence node >single node >select node. Evaluation index 2 is positively correlated with our goal, so the priority should be: statements include with more variables >statements with less variables.

Then propose our mutation sentence selection priority criteria:

Mutation Sentence Selection Priority Criteria 1 (P1)
For each path-segment of path-segments set $ps = (ps_1, ps_2, ..., ps_n)$, if there is a sequence node n_i at the ps_i, we insert the fault at the statement of n_i. And if $n_i = (s_1, s_2, ..., s_n)$ (s means statement), we insert a fault at the first statement.

As shown in Fig. 4: for the path segment ps1:2-3-6, using P1 we find a single node 3, and use the appropriate mutation operator AOM to generate the mutant ,the statement ‘y = y + x’ of the node3 will be mutated to ‘y = y − x’.

Mutation Sentence Selection Priority Criteria 2 (P2)
For each path-segment of path-segments set $ps = (ps_1, ps_2, ..., ps_n)$, if there is no sequence node and there is a selection node (predicate operation) n_i at the ps_i, use the appropriate mutation operator to insert the simple syntactic change (fault) at the selection node n_i.

As shown in Fig. 4: for the path segment ps2:4-6, using P2 we find a selection node 4, and generate the mutant ‘if($y \leq 4$)’ for the node 4 statement ‘if($y < 4$)’ using the appropriate mutation operator CBM.

Mutation Sentence Selection Priority Criteria 3 (P3)
If there is no sequence node and no selection node in the ps_i, inserted fault in the first statement of remaining nodes which are suitable to be inserted fault.

3.4 The Mutant Generation Algorithm

Applying the Fault Insertion basic path segments Identification Criterion for deter- mining appropriate program segments at which faults are inserted and a Mutation Sentence Selection Priority Criterion for selecting proper statements to make a fault for insertion above, we propose an algorithm whose input is a program and the output is a mutant set.

The algorithm is shown as follows: first, draw the control flow graph CFG of the original program. Then, using the Fault Insertion basic path segments Identification Criterion to determine appropriate program points at which faults are

Algorithm 1. Basic Path Coverage based on Mutant Generation Algorithm

```
Input:
    Original program, $P$;
Output:
    Mutant set, $M$;
1: Function mutant-generation(program p) {
2: Draw CFG for each functions in original program p;
3: Set FPS=∅;
4: for each $CFG_i \in CFG$ do
5:     initialize $FPS_i = \emptyset$ ;
6:     if $\exists SPinCFG_i$ then
7:         add sp into $FPS_i$, $FPS_i \rightarrow FPS_i + sp$;
8:     end if
9:     for each $CFG_i inCFG$ do
10:        find all the unique path-segment ups;
11:        add ups into $FPS_i$, $FPS_i \rightarrow FPS_i + ups$;
12:    end for
13: end for
14: add $FPS_i$ into $FPS$, $FPS = (FPS_1, FPS_2, ..., FPS_n)$;
15: for each $ps$ in $EPS$ do
16:     Set M=∅;
17:     if $\exists sequencenode$ then
18:         generate mutant $m_i$ from the sequence node statement $s_i$ for fault insertion;
19:         add $m_i$ into $M$, $M \rightarrow M + m_i$;
20:     else if $\exists selectionnode$ then
21:         generate mutant $m_j$ from the selection node statement $s_j$ for fault insertion;
22:         add $m_j$ into $M$, $M \rightarrow M + m_j$;
23:     else
24:         generate mutant $m_k$ from the selection node statement $s_k$ for fault insertion;
25:         add $m_k$ into $M$, $M \rightarrow M + m_k$;
26:     end if
27: end for
28: Return M;
29: }
```

inserted by analyzing the program control flow graph(CFG) and find the appropriate fault insertion path segments in the CFG. Finally, using the Mutation Sentence Selection Priority Criterion to generate a mutant at the appropriate statement of the path segments we marked above. The steps of this algorithm are shown as follows:

Step 1. For the original program p, we divide it into some program modules or functions $p = (f_1, f_2, ..., f_n)$ and draw a CFG for each module or function $CFG = (CFG_1, CFG_2, ..., CFG_n)$.

Step 2. Do analysis for CFG_i, find the appropriate fault insertion basic path-segments FPS_i in the CFG.

- First Initialize FPS_i to empty, if the CFG have leaf node, apply the Fault Insertion basic path segments Identification Rule 1 (R1) to find a sequence path-segment sp, add sp into FPS_i;
- Then apply the rule 2 to find unique path-segments *ups* on the CFG, add each *ups* into FPS_i;
- Then we can get a fault insertion path-segment set $FPS_i = (sp_1, ups_1, ups_2, ..., ups_n)$.

Step 3. Repeat step 2 for each CFG_i to get the total fault insertion path-segment set $FPS = (FPS_1, FPS_2, ..., FPS_n)$ for the original program.

Step 4. Do analysis for each path-segment ps_i in the fault insertion path-segment set FPS using the Mutation Sentence Selection Priority Criterion above and get a mutants set M_i.

- apply P1, generate mutant for a sequence node statement using appropriate mutation operator. For example, MAO, BOC, VMC;
- apply P2, generate mutant for a predicate node statement;
- apply P3, generate mutant for the first statement of remaining node that can be inserted into fault.

Step 5. For each path-segment ps_i in the fault insertion path-segments set FPS, Repeat the above step (4) to get the mutants set $M = M_1, M_2, ..., M_n$ of the original program P.

4 Tool Design and Implementation

Our proposed algorithm is aiming at reducing the cost of mutation testing by generating less but effectively mutants. In order to validate the efficiency and effectiveness (accuracy of mutation score), we implement an automated mutation testing tool to support our algorithm. It takes the program and test case set as input, does the mutation testing automatically, and finally produces a analysis report to show the test result.

The main functions of this automated mutation testing tool are using effective mutation operators to generate mutants (mutant generation), executing the given test case set on the mutants (mutant execution), showing the result and report of mutation testing (result analysis). Figure 5 shows the overall structure of our mutation testing tool. It is composed of 3 components.

The mutant generator generates mutants by using the effective mutation operators. It generate mutants for selected java files. The GUI for the mutant generator can help us to choose which project and which files under test. The mutant viewer component lists the detailed information for each generated mutant including operatorType, lineNumber, description and so on. And it also shows the code of original program and mutant which help us to know which statement of program under test is mutated and design test cases to kill the generated mutants. The test executor runs the test case set on generated mutants and reports the testing result by computing the mutation score of given test case set.

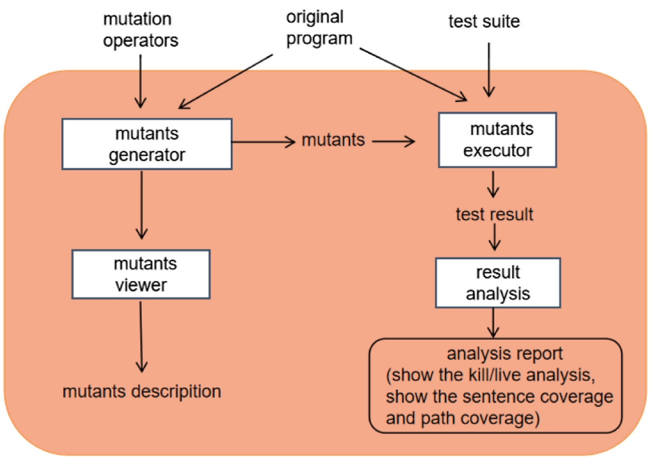


Fig. 5. The overall structure of the automated mutation testing tool.

4.1 Mutants Generator

According to the above design, we implement the tool using Java Swing and the user interface are shown as follows. Figure 6 shows how the Mutants Generator works. We can select the project and a set of Java files under test to create mutants, view the description of applied mutation operators, and press the Generate button to prompt the tool to generate mutants. The Mutants Viewer panel will show the information for each mutant after generation.

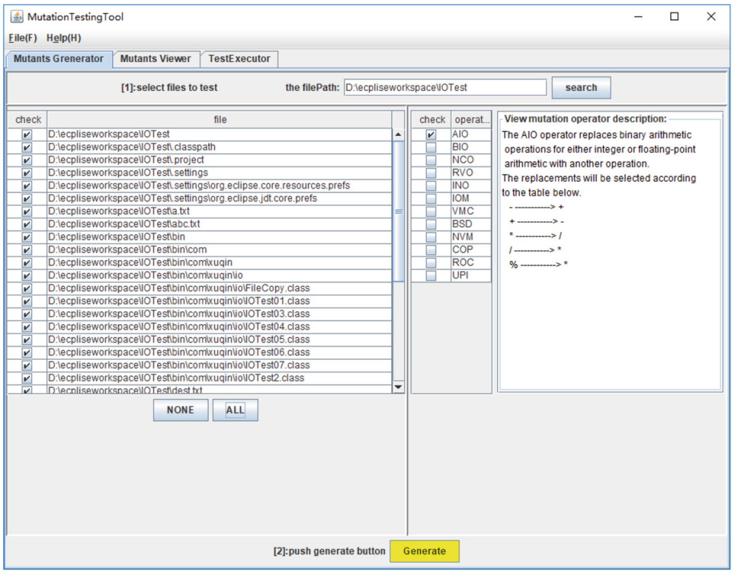
The specific steps are as follows:

- Step1, push the ‘search’ button to select the java project to be tested.
- Step2, view mutation operators description.
- Step3, select the Java files to test and the user can push the ‘ALL’ button to choose all the files listed.
- Step4, push the ‘Generate’ button to generate mutants for the selected java files.

4.2 Mutants Viewer

When generating a mutant, we produce a mutant description, including the operator type, className, description, lineNumber. This description details the alternate operations applied in each fault insertion statement we marked before. Using this, the tester can easily position the mutation statement and the mutation operator type which is beneficial to the later mutant viewing and result statistics.

The Mutants Viewer pane in Fig. 7 lists all the generated mutants and shows some detailed description of each mutant. It help us to analyze mutants by displaying the information of each mutant and which statement of given program



the filePath: D:\ecpliseworkspace\IOTest

check	file
✓	D:\ecpliseworkspace\IOTest
✓	D:\ecpliseworkspace\IOTest\.classpath
✓	D:\ecpliseworkspace\IOTest\.project
✓	D:\ecpliseworkspace\IOTest\.settings
✓	D:\ecpliseworkspace\IOTest\.settings\org.eclipse.core.resources.prefs
✓	D:\ecpliseworkspace\IOTest\.settings\org.eclipse.jdt.core.prefs
✓	D:\ecpliseworkspace\IOTest\a.txt
✓	D:\ecpliseworkspace\IOTest\abc.txt
✓	D:\ecpliseworkspace\IOTest\bin
✓	D:\ecpliseworkspace\IOTest\bin\com
✓	D:\ecpliseworkspace\IOTest\bin\com\xuqin
✓	D:\ecpliseworkspace\IOTest\bin\com\xuqin\io
✓	D:\ecpliseworkspace\IOTest\bin\com\xuqin\io\FileCopy.class
✓	D:\ecpliseworkspace\IOTest\bin\com\xuqin\io\IOTest01.class
✓	D:\ecpliseworkspace\IOTest\bin\com\xuqin\io\IOTest03.class
✓	D:\ecpliseworkspace\IOTest\bin\com\xuqin\io\IOTest04.class
✓	D:\ecpliseworkspace\IOTest\bin\com\xuqin\io\IOTest05.class
✓	D:\ecpliseworkspace\IOTest\bin\com\xuqin\io\IOTest06.class
✓	D:\ecpliseworkspace\IOTest\bin\com\xuqin\io\IOTest07.class
✓	D:\ecpliseworkspace\IOTest\bin\com\xuqin\io\IOTest2.class
✓	D:\ecpliseworkspace\IOTest\dest.txt

check	operat...
✓	AIO
	BIO
	NCO
	RVO
	INO
	IOM
	VMC
	BSD
	NVM
	COP
	ROC
	UPI

View mutation operator description:

```
The AIO operator replaces binary arithmetic
 operations for either integer or floating-point
 arithmetic with another operation.
The replacements will be selected according
to the table below.
  - -----------> +
  + -----------> -
  * -----------> /
  / -----------> *
  % -----------> *
```

Fig. 6. The Mutants Generator GUI.

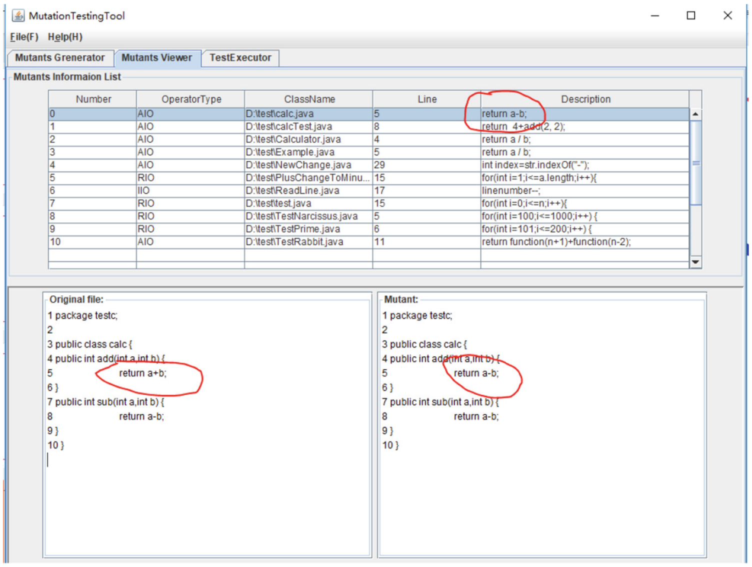
is changed by the mutant. It is divided into two parts. The upper part is a mutants list which shows a brief descriptions of the each mutant including a operatorType, className, description and lineNumber. The lower part shows the original code and the mutant. By choosing a mutant in the mutants list of the upper part, the lower part will show the original java file and the mutant, which helps testers to know which statement is mutated, design test cases to kill mutants which are difficult to kill.

4.3 Test Executor

Figure 8 shows the GUI of the Test Executor pane. It runs the test case set on mutants and reports the testing result by analysing the mutation score. Lower left part shows the number of mutants generated by each operator. The lower right part shows the results of mutation testing and the number of live mutants and dead mutants. The test case can be created or provided by testers. The specific step is as follow:

First, the testers need to select the class and the test case set and then push the 'Run' button to run the test cases on the mutants.

When the test is finished, the OP Number panel will show the mutants number of each operator and the mutants result panel will show the mutation testing result including the mutation score and the number of killed mutants, live mutants and the total mutants. Also, the tester can export the result into a

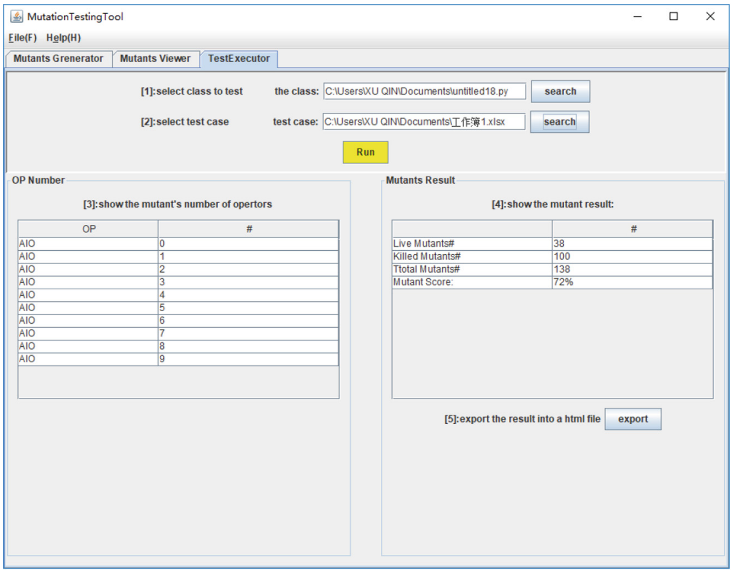


Number	OperatorType	ClassName	Line	Description
0	AIO	D:\test\calc.java	5	return a-b;
1	AIO	D:\test\calcTest.java	8	return 4+add(2, 2);
2	AIO	D:\test\Calculator.java	4	return a / b;
3	AIO	D:\test\Example.java	5	return a / b;
4	AIO	D:\test\NewChange.java	29	int index=str.indexOf("-");
5	RIO	D:\test\PlusChangeToMinu...	15	for(int i=1;i<=a.length;i++){
6	IIO	D:\test\ReadLine.java	17	linenumber--;
7	RIO	D:\test\test.java	15	for(int i=0;i<=n;i++){
8	RIO	D:\test\TestNarcissus.java	5	for(int i=100;i<=1000;i++) {
9	RIO	D:\test\TestPrime.java	6	for(int i=101;i<=200;i++) {
10	AIO	D:\test\TestRabbit.java	11	return function(n+1)+function(n-2);

Original file:
```
1 package testc;
2
3 public class calc {
4 public int add(int a,int b) {
5               return a+b;
6 }
7 public int sub(int a,int b) {
8               return a-b;
9 }
10 }
```

Mutant:
```
1 package testc;
2
3 public class calc {
4 public int add(int a,int b) {
5               return a-b;
6 }
7 public int sub(int a,int b) {
8               return a-b;
9 }
10 }
```

Fig. 7. The Mutants Viewer GUI.



[1]:select class to test   the class: C:\Users\XU QIN\Documents\untitled18.py

[2]:select test case   test case: C:\Users\XU QIN\Documents\工作簿1.xlsx

[3]:show the mutant's number of opertors

OP	#
AIO	0
AIO	1
AIO	2
AIO	3
AIO	4
AIO	5
AIO	6
AIO	7
AIO	8
AIO	9

[4]:show the mutant result:

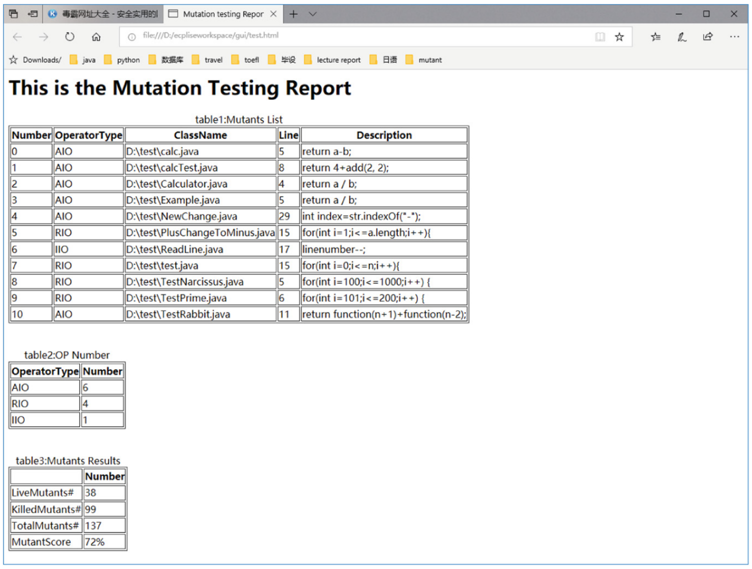
	#
Live Mutants#	38
Killed Mutants#	100
Ttotal Mutants#	138
Mutant Score:	72%

[5]:export the result into a html file

Fig. 8. The Test Executor GUI.

HTML file, which shows test results more clearly and can be saved as important test document. Push the ‘export’ button in the test executor panel, a test report will be generated. Figure 9 shows the report.

The mutation testing report shows the generated mutants list, the number of each operator type, the number of live mutants, the number of killed mutants, and the mutation score. Lower left part shows the number of mutants generated by different operator type.



This is the Mutation Testing Report

table1:Mutants List

Number	OperatorType	ClassName	Line	Description
0	AIO	D:\test\calc.java	5	return a-b;
1	AIO	D:\test\calcTest.java	8	return 4+add(2, 2);
2	AIO	D:\test\Calculator.java	4	return a / b;
3	AIO	D:\test\Example.java	5	return a / b;
4	AIO	D:\test\NewChange.java	29	int index=str.indexOf("-");
5	RIO	D:\test\PlusChangeToMinus.java	15	for(int i=1;i<=a.length;i++){
6	IIO	D:\test\ReadLine.java	17	linenumber--;
7	RIO	D:\test\test.java	15	for(int i=0;i<=n;i++){
8	RIO	D:\test\TestNarcissus.java	5	for(int i=100;i<=1000;i++) {
9	RIO	D:\test\TestPrime.java	6	for(int i=101;i<=200;i++) {
10	AIO	D:\test\TestRabbit.java	11	return function(n+1)+function(n-2);

table2:OP Number

OperatorType	Number
AIO	6
RIO	4
IIO	1

table3:Mutants Results

	Number
LiveMutants#	38
KilledMutants#	99
TotalMutants#	137
MutantScore	72%

Fig. 9. The testing report in a HTML file.

5 Experiment for Evaluation

In order to assess the performance of our method, 6 benchmark programs were selected as the tested programs, all of which were written in JAVA language. Feasibility and effectiveness were assessed using empirical and comparative studies.

5.1 Research Questions

The experiment described here mainly focuses on the following Research Questions (RQs):

> RQ 1. Can our algorithm proposed in this paper effectively reduce the number of mutants? (by calculating the reduction rate of mutants)

The effectiveness of this method in reducing mutants was evaluated by comparing the number of mutants in the proposed algorithm of the selected programs with the number of mutants in traditional method and selective mutation technique using the same mutation operators. We proposed a mutant reduction rate to assess the ability of the mutant reduction. The mutant reduction rate is:

$$MRR = \frac{M_t - M_p}{M_t} \tag{1}$$

M_t: mutant's number generated by traditional method, M_p: mutant's number generated by other methods. The MRR shows that the higher the reduction rate is, the better the effectiveness of this method in reducing mutants becomes.

RQ 2. Can a test case set that kills mutants generated by the proposed algorithm be able to kill mutants of traditional methods?

The set of traditional mutants was tested with the test cases that killed the mutants generated by this algorithm, and the results were expressed as mutation score:

$$MS = \frac{M_K}{M_a - M_e} * 100 \tag{2}$$

M_k: killed mutant's number, M_a: all generated mutant's number, M_e: all equivalent mutant's number.

The test case set used in this paper's experimental evaluation is constructed using traditional test case generation algorithms, such as boundary value analysis, statement coverage, and branch coverage.

5.2 Experiment Subjects

Table 1 shows the detailed description of each experiment subjects. We choose those programs which are popular used in other mutation testing researches. The 'ID' and 'Test Subject' show the names of each experiment subjects and the 'line of code' shows the line number of each code and 'function program' record the function of each program.

Table 1. The description of each experiment subject.

ID	Test subjects	Line of code	Program function description
J1	Trityp	36	Judge the triangle type
J2	Mid	26	Calculate the median of three integers
J3	Quadratic	25	Finding the root of a quadratic equation
J4	Bubble sort	19	Bubble Sorting
J5	Cal	46	Compute days between two days
J6	MyCalendar	50	Ask for a calendar of a certain month

5.3 Experiment Result Analysis

As shown in Table 2 below, by comparing the number of mutants generated by this algorithm and number of mutants generated by selective mutation (SM) with the number of mutants generated by traditional non-optimized methods, we clearly see that the both the number of mutants generated by our proposed method and the number of mutants generated by selective mutation were significantly reduced. For example, the J2 program is reduced from 63 to 35 and 18, and the J6 program is reduced from 66 to 31 and 19.

Table 2. The reduction rate of selective mutation and our algorithm.

ID	Number of mutants in traditional method	Number of mutants in SM	Reduction rate of SM	Number of mutants in our method	Reduction rate of our method
J1	39	23	43.6%	9	76.9%
J2	63	35	44.4%	18	71%
J3	67	34	49.3%	17	74.6%
J4	42	26	38.1%	9	78.6%
J5	40	25	37.5%	9	77.5%
J6	66	31	53.3%	19	71.2%
Average			44.4%		74.96%

In the selective mutation, J6 has the largest reduction rate of 53.3% in 6 programs, and the average reduction rate of 6 programs is 44.4%. In our algorithm, J4 has the largest reduction rate of 78.6% in 6 programs, and the average reduction rate of 6 programs is 74.96%. We can see that all the procedures using our proposed algorithm achieve higher reduction rates, and the reduction rates are higher than the reduction rate of selective mutation, which indicates that our proposed method is better than selective mutation in reducing the number of mutants.

The sufficient test case set for the algorithm proposed in this paper is applied to all the mutants generated in traditional mutation method and the mutation score is calculated. The calculated mutation score finally indicates the effectiveness of the proposed method. The following Table 3 shows the detection results of the test case set that can detect the mutants generated by the proposed method on the traditional mutants set. The mutation score is used as the evaluation index.

From Table 3, we can see that the mutation score of the test case set that is 100% sufficient for the proposed algorithm can reach 90.6% on average, and the lowest one is 87.5%. The J2 program has the highest mutation score of 93.7%, and the J5 program has the lowest mutation score of 87.5%. The experimental results show that for all test subjects, the average mutation score exceeds 90.6%, and we only use a small number of mutants (74.96%).

We also analyzed the number of mutants per mutation operator. From the result we can see that the number of MAO replacement mutants is relatively

Table 3. The reduction rate of each experimental subject.

ID	Number of mutants in traditional method	Number of mutants in proposed method	Reduction rate
J1	39	35	89.7%
J2	63	59	93.7%
J3	67	62	92.5%
J4	42	37	89%
J5	40	35	87.5%
J6	66	60	91%
Average			90.6%

large, while others are relatively small. For the program J1, the number of BOC, NCO is very large and number of the RVO, VMC is zero. Although some operators is used less but we can not say it is useful because those operators are designed to find specific faults which are not shown in our experiment subjects.

The proposed algorithm optimizes traditional mutation method by selecting the appropriate mutation point. Through the analysis of the above results, it can be seen that our algorithm can generate a less number of mutants with a high mutation score, which can significantly reduce the cost of mutation testing while maintaining the effectiveness.

5.4 Result Validity Analysis

As with other case studies, when discussing the validity of the experimental results in this paper, some restrictive factors in the experimental process must be considered, including the following three cases:

- In this experiment, only 6 tested instances were selected, and it is not certain that all the tested instances will have the same experimental results;
- The number of test cases for elections is less than 100. It is uncertain whether the reduction efficiency of test cases will increase for a larger number of test cases;
- All the mutation operators were not selected in the experiment, and there was no manual analysis of whether there were equivalent mutants in the unkilled mutant, so the calculated MS value may be too small.

6 Related Work

In recent years, in order to apply mutation testing in industry and improve the efficiency of mutation testing, many research have been carried out. Here, we briefly show the research progress of test optimization techniques about reducing mutant's number.

Acree [10] proposed a method called mutant sampling, which select a certain proportion of mutants randomly from all mutants generated for mutation testing. This way can effectively reduce the mutant but with lower mutation score.

Hussian [13] proposed a method called the method of Mutant Clustering which classify mutants with similar characteristics, and then randomly select a part of variants from each class for mutation testing. Experiment shows that the clustering method can achieve a good reduction of mutants without affecting the validity of the mutation testing.

Mathur [11] proposed a method which select partial mutation operators applied for mutation testing. This method of generating fewer mutants using a small number of mutation operators is called constraint mutation. Offutt et al. further proposed a method of "selective mutation".

Jia and Harman [12] introduced a Higher Order Mutation method. That is, a high-order mutation consists of multiple single-order mutations, and the use of higher-order mutants can effectively speed up mutation testing.

Delamaro et al. [16] studied the validity of a mutation testing using only one mutation operator. The experimental results show that the SDL operator is probably the most useful mutation operator among all the mutation operators.

Yao et al.'s [15] research shows that some mutation operators are very possible to create equivalent mutants, but the resulting stubborn mutants (which are difficult to detect, but not equivalent mutants) are rare; others are susceptible to stubborn mutants and the generated equivalent mutants are few, so when generating mutants, different selection priorities can be set for the mutation operators.

Although the above mutant reduction technology can reduce the number of mutants, it can not guarantee the path coverage of the program, which affects the sufficiency evaluation ability of the test case set. In our research, a mutant generation algorithm based on basic path coverage and control flow analysis is used to select the appropriate sentence segment to be inserted into error of the basic path, which reduce the number of mutants and realize the basic path coverage. It not only assess whether the test case set can kill the mutants, but also assess whether the test case set can achieve basic path coverage.

7 Conclusion

Mutation testing is widely used to evaluate test case sufficiency and evaluate the effectiveness of software testing techniques. However, a large number of mutants affect the efficiency of mutation testing and limit the application of mutation testing in software testing practice.

In this paper, a mutant generation method based on basic path coverage is proposed for reducing generated mutant's number in mutation testing. Different from the previous methods, by analyzing the control flow structure and basic path of the source program, an identification of path segments suitable for fault insertion and a priority criteria for generating mutants are proposed. By using these criteria to select some appropriate statements for mutation operation, the mutants needed to kill is reduced and the coverage of the basic path is achieved, which improves the effectiveness of the mutation testing. In order to evaluate the efficiency (mutant's number) and effectiveness (accuracy of mutation score) of our proposed method, we implement an automated mutation testing tool to support our algorithm. Our method was applied to 6 tested programs, and the results showed that using the method of this paper, the high mutation score can be maintained while reducing the number of mutants.

This work has opened up a research direction of mutation test optimization technology. The next steps include using more efficient mutation operators to generate mutants, using a larger industrial application sample program to evaluate the effectiveness of the method, and exploring other more efficient mutant reduction techniques.

References

1. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**(5), 649–678 (2011)
2. Offutt, A.J., Untch, R.H.: Mutation 2000: Uniting the Orthogonal. Mutation Testing for the New Century. Kluwer Academic Publishers, Berlin (2001)
3. Just, R., Ernst, M.D., Fraser, G.: Efficient mutation analysis by propagating and partitioning infected execution states (2014)
4. Namin, A.S., Andrews, J.H., Murdoch, D.J.: Sufficient mutation operators for measuring test effectiveness. In: ACM Press the 13th International Conference on Software Engineering - ICSE 2008 - Leipzig, Germany, 10–18 May 2008, p. 351 (2008)
5. Allen, F.E.: Control flow analysis. ACM Sigplan Not. **5**(7), 1–19 (1970)
6. Zapata, F., Akundi, A., Pineda, R., Smith, E.: Basis path analysis for testing complex system of systems. Procedia Comput. Sci. **20**(Complete), 256–261 (2013)
7. Papadakis, M., Malevris, N.: Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. Softw. Qual. J. **19**(4), 691–723 (2011)
8. Eason, G., Noble, B., Sneddon, I.N.: On certain integrals of Lipschitz-Hankel type involving products of Bessel functions. Phil. Trans. Roy. Soc. London **A247**, 529–551 (1955)
9. Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and Oracles. IEEE Trans. Softw. Eng. **38**(2), 278–292 (2012)
10. Acree, A.T.: On mutation. Ph.D. Dissertation, Georgia Institute of Technology (1980)
11. Mathur, A.P.: Performance, effectiveness, and reliability issues in software testing. In: International Computer Software & Applications Conference. IEEE (1991)
12. Jia, Y., Harman, M.: Constructing Subtle Faults Using Higher Order Mutation Testing. In: 2008 Eighth IEEE International Working Conference on Source Code Analysis and Manipulation. IEEE (2008)
13. Hussain, S.: Mutation clustering. Ph.D. Dissertation, King's College, London, UK (2008)
14. Zhang, J.: Scalability studies on selective mutation testing. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), Florence, Italy, 5–24 May 2015, pp. 851–854 (2015)

15. Yao, X., Harman, M., Jia, Y.: 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, Hyderabad, India, 31 May–07 June 2014, pp. 919–930 (2014)
16. Delamaro, M.E., Li, N., Offutt, J., et al.: Experimental evaluation of SDL and one-op mutation for C. In: IEEE Seventh International Conference on Software Testing. IEEE (2014)
17. Hutchins, M., Foster, H., Goradia, T., et al.: Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In: International Conference on Software Engineering. IEEE (1994)
18. Harman, M., Jia, Y., Mateo, R.P., et al.: 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. ACM (2014)
19. Girgis, M.R., Woodward, M.R.: An integrated system for program testing using weak mutation and data flow analysis. In: International Conference on Software Engineering. IEEE Computer Society Press (1985)
20. Krauser, E.W., Mathur, A.P., Rego, V.J.: High performance software testing on SIMD machines. IEEE Trans. Softw. Eng. **17**(5), 403–423 (2002)
21. Allen, F.E., Cocke, J.: A program data flow analysis procedure. Commun. ACM **19**(3), 137 (1976)