

Automatic Test Cases Generation for Unit Testing Using Genetic Algorithm

Song Wang

Department of Mechanical Engineering, Columbia University
116th St & Broadway, New York, NY 10027
US
sw3130@columbia.edu

ABSTRACT

A method of automating the generation of testing cases for unit testing using genetic algorithm combined with mutation testing is proposed. While the method proposed may be well suited to component testing as well. Detailed working process of mutation testing is shown in this paper, different representation and variation method have been tried, two toy problems have been tried to compare with the list bit string representation. Reason why this problem is not that suitable for Genetic Algorithm is discussed.

CCS CONCEPTS

• **Software and its engineering** → **Software creation and management** → **Software verification and validation** → **Formal software verification.**

KEYWORDS

Unit Testing, Mutation Testing, Genetic Algorithm, Test Case Generation

ACM Reference format:

G. Gubbiotti, P. Malagò, S. Fin, S. Tacchi, L. Giovannini, D. Bisero, M. Madami, and G. Carlotti. 1997. SIG Proceedings Paper in word Format. In *Proceedings of ACM Woodstock conference, El Paso, Texas USA, July 1997 (WOODSTOCK'97)*, 4 pages.DOI: 10.1145/123 4

1 INTRODUCTION

During software development, testing is vital to ensure the quality of the code. In testing, we try to meet two goals[6], the first is to show that the developer and customer that the software product meets the requirement in the requirement document. we execute the source codes using test data we selected, then check the results of the testing run of code whether there're errors by comparing outputs with expected ones. Secondly, we try to find inputs where the behavior of the software is incorrect or does not conform to the specification. When testing software for defects, we want to find out unwanted behavior such as crash.

There are three levels of testing[6] under different development stages which are Unit testing: test individual

programs, Component Testing: testing for the integration of individual programs and System Testing where a part of or the overall system is tested. In this project, we only deal with unit testing. One of the most used testing method nowadays is mutation testing, where it is very common that test suites execute all its branches while only partially tested code[5].mutation testing can be able to check whether each statements has been tested.

In the context of Java, individual programs are referred to object classes or class methods. One of the key issue is to provide all the coverage of the object under testing. We should test all operations related to it, these operations include the values of its attributions and any methods that could change the state of the object.

Software bugs cost the U.S. economy \$59.5 billion annually[7]. More than a third of this cost could be avoided if better software testing was performed. Also, Testing consumes significant amount of time and resources in the whole life cycle of software development. More than half of the cost of development is for testing. This is because usually the input space of units under test is very large, the generation of test cases is a NP-hard problem.

In mutation testing, we manually inject errors into the codes under test. The errors injected are called mutants and the programs with mutations are called mutated programs. Regarding a certain input, If the programs runs as there are no mutants it is said the mutant survived under this input which indicates the input can not explore the full coverage of the code under test. Otherwise the mutants are said to be killed. For all the mutated programs, the concept of mutation score is defined for each test input case as the portion of mutants killed over all the mutants generated. A higher mutation score suggests the test case can show more coverage of the tested code.

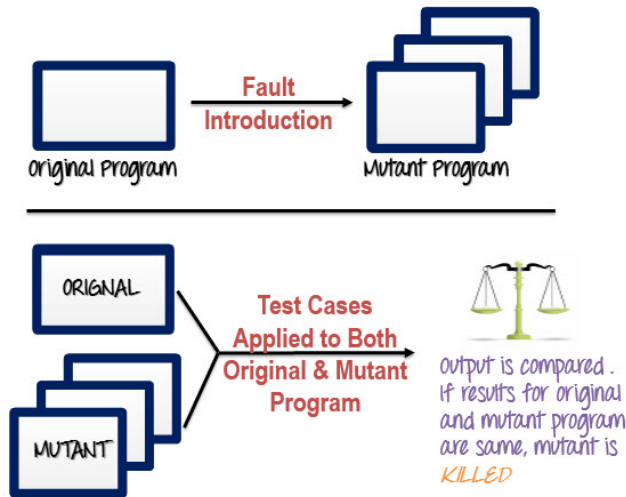


Figure 1: Process of mutation testing.

For a NP-hard problem like the generation of testing cases, an optimization method could be used is Genetic Algorithm. Genetic Algorithm evolves generations of solution population using the mutation score as a fitness measuring to find better and better solution. Khan[4] has done made some primitive experiment on this method where his source code is a 10 lines simple program with two inputs x and y to calculate the x^y , and his manually put four mutants and represent the input into bit string and at each evolution, the individuals with a mutation score less than 0.5 is mutated and crossed individuals with a mutation score more than 0.5.

The results are compared with randomly generated integers which shows GA outperforms the randomly generated ones.

However, since the mutants are generated manually, the nature of the mutants themselves has not been explored, his sample code is too short to represent general program to be tested. And his program is clearly a procedural program, his results may not work for object oriented languages like java. So the results of his work can not be thought general enough. And his work requires manually inserting mutants which is of little use in practical situations where the mutants generated for a single class may be as much as 1000.

Here I proposed to use Pitest with Genetic Algorithm to automated the generation of testing cases for software testing.

Pitest is a robust mutation testing tool developed by Henry Coles, compared to other mutation systems PIT uses coverage analysis and bytecode manipulation to make it run quite quickly 1. Reports generated by Pitest outputs every

generated single mutant itself and whether it is killed or not, then the overall mutation score and line coverage.2. Pitest is very fast and would accelerate the testing in a order of two compared with its counterparts while mutation testing is time expensive. 3. And most of them are useful for a specific scientific research, which would be not practical to use in development teams.4. pit is the only one supported to works with Maven which is the platform for this project.5. Pit is the only one mutation testing that is still under development and the only one to get help from user groups .6. Pit let you have your choice to configure many parameters based on your own machine. For example if you want to speed up the mutation testing, you can change the threads based on the number of cores of your machine, you can also set the max number of mutations allowed per class. Also you can set the target class of interest to be tested. The tool is opensource which means we can change it as we want. Mutation testing is very time consuming even though we use Pitest. and the time depends on the lines of codes (LOC) and the quality of your test.

Sample.java

| Mutations | |
|-----------|--|
| 1. | changed conditional boundary → SURVIVED |
| 2. | changed conditional boundary → SURVIVED |
| 3. | Replaced bitwise OR with AND → SURVIVED |
| 4. | negated conditional → SURVIVED |
| 5. | negated conditional → SURVIVED |
| 6. | negated conditional → SURVIVED |
| 1. | replaced return of integer sized value with (x == 0 ? 1 : 0) → NO_COVERAGE |
| 1. | changed conditional boundary → SURVIVED |
| 2. | negated conditional → SURVIVED |
| 1. | changed conditional boundary → NO_COVERAGE |
| 2. | negated conditional → NO_COVERAGE |
| 1. | Replaced integer division with multiplication → NO_COVERAGE |
| 1. | Changed increment from 1 to -1 → NO_COVERAGE |
| 1. | changed conditional boundary → NO_COVERAGE |
| 2. | negated conditional → NO_COVERAGE |
| 1. | Changed increment from -1 to 1 → NO_COVERAGE |
| 1. | Replaced integer division with multiplication → NO_COVERAGE |
| 1. | Replaced integer multiplication with division → NO_COVERAGE |
| 3. | Replaced integer subtraction with addition → NO_COVERAGE |
| 1. | changed conditional boundary → NO_COVERAGE |
| 2. | Replaced bitwise AND with OR → NO_COVERAGE |
| 3. | negated conditional → NO_COVERAGE |
| 4. | negated conditional → NO_COVERAGE |
| 5. | negated conditional → NO_COVERAGE |
| 1. | Changed increment from -1 to 1 → NO_COVERAGE |
| 1. | changed conditional boundary → NO_COVERAGE |

Figure 2: Pit Report generated by PIT 1.2.5.

2 IMPLEMENTATION DETAILS

2.1 Mutation Testing

Generally, the process of mutation testing consists of as least the following four parts:

2.1.1 Mutation generation

Analysis the class to be tested and generated mutations according rules:

2.1.1.1 Conditionals Boundary Mutator:

The conditionals boundary mutator replaces the relational operators <, <=, >, >= with their boundary counterpart as <=, <, >=, <.

2.1.1.2 *Negate Conditionals Mutator:*

negate <, into >= etc.

2.1.1.3 *Remove Conditionals Mutator:*

replace conditional statement into true always.

2.1.1.4 *Math Mutator:*

randomly change one math operator into another math operator.

2.1.1.5 *Increments Mutator:*

within any loop with a counter, increase the counter by one.

2.1.1.6 *Invert Negatives Mutator:*

change any variable into its negative counterpart.

2.1.1.7 *Inline Constant Mutator:*

randomly change a constant.

2.1.1.8 *Return Values Mutator:*

change the return value while preserve the type of the return type of the method.

2.1.1.9 *Void Method Call Mutator:*

if the program calls a void method, then ignore the call.

2.1.1.10 *Non Void Method Call Mutator:*

if a variable is assigned his value from a method call, replace the method call by a constant.

2.1.1.11 *Constructor Call Mutator:*

if a method return a project, then mutated to return NULL.

Usually there are two kinds of mutants generation schema: source mutators focus on the make changes to the source file and rebuild them, however it's very slow and sometimes can be really hard to integrate in a build. Another choice is Byte code mutators who generate mutants by dealing with bytecode already compiled and which would be rather easy to integrate. In this project, byte code mutators is chosen.

2.1.2 *Test selection*

Select the test based on the classes specified in the pom.xml file of the Maven project. If not specified will scan the whole project and select tests that meets the Junit naming convention, for example, SampleTest.java for Sample.java.

2.1.3 *Mutants insertion:*

mutants are inserted into jvm using Java's instrumentation API, also mutation insertion is the most time consuming parts, in order to accelerate this process as much as we can, a Mutant schemata is used in which a single class contains all the mutants and when running tests each mutants is enable one at a time.

```
public static boolean mutateID1;
public static boolean mutateID2;
```

```
public boolean mutateSchema(int i) {
    if (mutateID1) {
        return i < 12;
    } else if (mutateID2) {
        return i > 12;
    } else {
        // original code
        return i == 10;
    }
}
```

2.1.4 *Mutants detection:*

Run test with mutated programs and check the test results with original programs and output the results. Pitest has one main controlling process. As it analyses mutations it creates a number of child processes (or minions) that do the work of running tests against each mutant. in this project, the output will detailly show which mutants are inserted, survive or not, which line is covered. Dealing with timeouts:

It's not rare for mutations to cause an infinite loop:

Considering the following code:

```
int x = 10;
while (x < 5) {
    x = x - 2;
}
Mutate it into:
int x = 10;
while (x < 5) {
    x = x + 2;
}
```

Where the loop will not terminate until overflow.

To deal with timeouts pitest will first measure the normal execution time (denote as t) of the original tests, pit checks whether timeouts happens by the running time of no longer than t * coefficient (usually as large as 1000, if you want to speed up the mutation you can also set it to be smaller), however this does not always happen. For example, the first tests may execute much longer than its follower because of loading the required classes to run the tests. In this case the pit would classify the mutation as infinite loop by mistake. We can fix this by increase the timeoutConstant in maven while that may make the code running slower. The result is outputted as a html file and parsed through another java tool Jsoup[3].

2.2 Genetic Algorithm

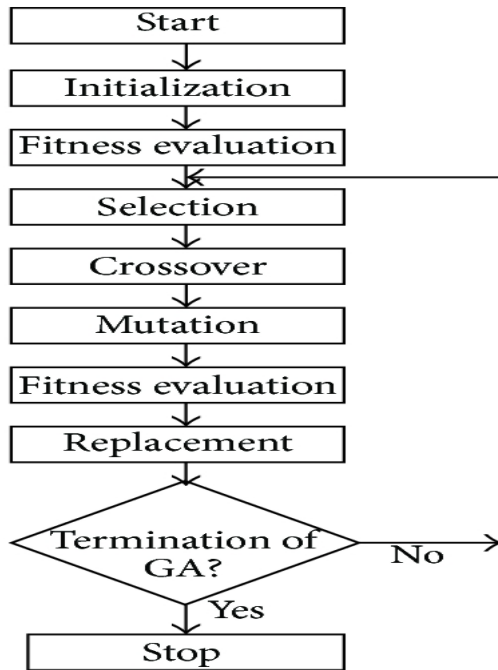


Figure 3: the process of genetic algorithm[2]

After the random initializing of a population, a fitness function to indicate the chance of each individual to survive is defined in our case is the mutation score of each test case. Then the selection process randomly selects individuals of the current generation for the development of the next generation where the individual with a higher fitness should have a higher chance to be selected. Then the selected individuals are mutated and crossed over to create new individuals. Then crossover process takes two individuals at a time and combine their chromosome under one or various points. While mutation usually introduce small changes to each individual in our case is to flip one bit. Provided the newly generated individuals through variation operators we again evaluate them and individuals with higher rank or higher fitness usually have a higher chance to be selected into the next generation. Given the next generation of individuals we loop the processes until a terminate criterion is met (usually the fitness of best individual or the number of evaluations)

Some comments on how genetic algorithm together with mutation testing could help automate the generation of test cases.

2.3 Algorithm Implementation

This projects using genetic algorithm to search good test cases base on their mutation score.

2.1.5 Preparation

This is a maven[1] project who is configured by Project Object Model(POM) and the details is stored in the POM.xml file. At the start of this project, all the dependencies and plugins (Jsoup, Pitest, Junit, Python/Jython) as well the parameters setup (number of threads, timeOutConstant etc)

Then target class to be tested and target test are specified in the POM.xml file. In this project, the target class Sample.java is a program with a constructor and a method named reverse that take a input then output its reverse number (i.e. Sample.reverse(123), it should return 321, Sample.reverse(-492), it should return (-294), if the input is larger than a 8 bits signed integer, this method should return 0. The Sample program is written by myself on an online algorithm coding platform LeetCode[8].

SampleTest.java is generated by the JUnit tool of Eclipse which has a static variable that takes the input from the GA system as well as a test of the reverse method.

2.1.6 Representation

In this project, various representation and variation combination have been tried.

Integer List Representation

Each element of the list(individual) is represent by an Integer generated randomly within the range of 2,147,483,648 (larger than the upper limit of a 32 bits signed integer) to -2147483649 (less than the lower limit of a 32 bits signed integer).

Bit string representation

An 31 bits string is constructed for debugging.

List of Bit String representation

To represent a test cases suite, a list of 120 32 bits signed integer is constructed where each of the tries to detect one of the mutants generated instead of all the mutants.

Evaluation

For a single input the test system communicate with the GA system by the static variable of the test class, every time a new individual is generated in the GA system, we call the static method to set the value of the individual to the static variable in the test, since we can not use java to change current disk by command line, we then call the python script to run the command line to change the current directory into the path roots where this project locates and run the Maven command to launch the Pitest mutation testing system, it scan all the project as well as the information specified in the POM.xml file and choose which test to run, then generated mutants, insert the mutants into the mutate schema, then run against the test to check mutants, lastly pitest will generate a report of html format, after using Jsoup tool to parse the content of the report and get back the mutation score of this individual.

2.1.7 Selection

Selection in this project is based on fitness-proportional selection

2.1.8 Mutation and Crossover

To different schemes have been tried: first is to select the best parents and do crossover and mutation into next generation. Second approach is elitist selection, preserve individuals with good fitness (top 50 %) then do crossover and mutation over the lower 50 %), which together is the next generation.

For crossover of integer list representation, a single point crossover has been tried.

For a single bit string, both single point crossover and two points crossover have been tried.

For 120 elements list bit string, two kinds of multipoints crossover have been tried, the first method is to randomly choose multiple crossover points, another method is to do crossover for every element pair instead of the individual pair, this can be illustrated by figure 7

| | | | | | | | |
|----|----|----|----|-----|-----|-----|------|
| E1 | E2 | E3 | E4 | ... | ... | ... | E120 |
| e1 | e2 | e3 | e4 | | | | e120 |

| | | | | | | | | | |
|---|---|---|---|-----|---|---|---|---|---|
| 1 | 0 | 0 | 1 | ... | 1 | 0 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | ... | 0 | 0 | 1 | 1 | 0 |

Figure 4. Pairwise elements crossover of E4 and e4

Mutation for a integer list representation is to randomly change the value of the integer.

Mutation for bit string representation is to randomly flip bits.

2.1.9 Parameters setup

Parameters can be chosen manually are mutation rate, crossover rate, population size, number of generations, various parameters combinations have been tried.

3 RESULTS AND DISCUSSION

3.1 results for toy problem

In the debugging phase two toy problems have been used in testing, instead of using pitest, just manually create three mutant programs for the source file. Then create initial generation of input with 120 32 bits string into the GA, set the population size to be 100, within two or three generations the bit string list will find the three mutants successfully.

Another toy problem: evolve a single bit string to find the only mutants, while it takes many generations to detect the only mutants.

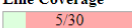
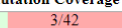
Then use an Integer representation, while it even doesn't work for the toy problems, so Integer Representation is not taken into consideration.

3.2 final result for bit string list

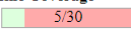
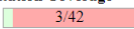
Finally, we use a 120 elements list where each element is a 32 long bit string as our candidate solution, use pairwised elements crossover, vary the mutation rates, crossover rates, replace the worst candidates, but after days of debugging, until today the best solution of each generation seems does not evolve at all. There are 42 mutants generated in the sample program while only three of them are discovered.

Pit Test Coverage Report

Project Summary

| Number of Classes | Line Coverage | Mutation Coverage |
|-------------------|--|---|
| 1 | 17%  5/30 | 7%  3/42 |

Breakdown by Package

| Name | Number of Classes | Line Coverage | Mutation Coverage |
|---------|-------------------|--|---|
| default | 1 | 17%  5/30 | 7%  3/42 |

Report generated by PIT 1.2.5

Figure 5: Pit Report generated by PIT 1.2.5.

4 CONCLUSIONS AND FUTURE WORK

In order to expose problem of my GA system easily, a detailed description of mutation testing is presented. Thinking about the second toy problem, where it takes much time to find a single mutants, which means it is of quite low probability for an individual to outperforms other individuals in the initial population, which means our selection from the population and doing variation over the selected individuals is equivalent to random search at the very early of the evolution, then when the newly generated individuals go back to the population, their parents may be the individuals to be replaced in the next generation. So, for a single string, after many generations there could be a randomly generated individual that could detects the mutants just like random search, however a single input is meaningless for a test since in most cases it is not possible in theory for a single input to explore all the branches of code under test. And to pursue a fully coverage is our goal in this project. we have to design a suit of test inputs, in our case the number of inputs is 120 considering the number of mutants generated. Which is equivalent to evolve 120 dimensions individuals, and clearly the possibility for the GA to go out of the initial random search is very low even after intensive computation.

For the tool I use, the pitest is open sourced, even it is the best mutation testing system I could have currently, its user group is still very small, in addition, I have witnessed some

errors while using the tool and cast doubt over the robustness of the program.
Even though I have spent lots of time reading the source codes, while there are almost no documentation and little comments for the code, I still can not draw a conclusion over the confidence of Pitest.

ACKNOWLEDGMENTS

REFERENCES

- [1] *Apache Maven.*
- [2] *Flight Control Laws Verification Using Continuous Genetic Algorithms.*
- [3] JSOUP, *Jsoup.*
- [4] R. KHAN and M. AMJAD, *Automatic test case generation for unit software testing using genetic algorithm and mutation analysis, Electrical Computer and Electronics (UPCON), 2015 IEEE UP Section Conference on, IEEE, 2015, pp. 1-5.*
- [5] C. RIMMER, *Getting Mutants to Test your Tests.*
- [6] I. SOMMERVILLE, *Software engineering*, Addison-wesley, 2007.
- [7] N. I. O. S. A. TECHNOLOGY, *Software Errors Cost U.S. Economy \$59.5 Billion Annually*, 2002.
- [8] S. WANG, *Reverse Integer*, (2017).