Project Final Report

Course: CS569
Name: Yifan Shen
Content: Final submission

1. Introduction

To explain the work in the project, we need to know some basic information about the background of random test and mutation test algorithms which will be used in the project.

Random testing is a black-box software testing technique where programs are tested by generating random, independent inputs. Results of the output are compared against software specifications to verify that the test output is pass or fail. In case of absence of specifications the exceptions of the language are used which means if an exception arises during test execution then it means there is a fault in the program. Most of time, we use the random test to do some automated software testing, because the random test has many advantages, such as cheap to use, has no bias, quick to find bugs. However, the random test has its own disadvantages for its random characteristic. Sometimes it can only finds the basic bugs. It is only as precise as the specification and specifications are typically imprecise and when it is compared with the other techniques such as static program analysis, it is very poor. [1]

Mutation testing is a testing method used to design new software tests and evaluate the quality of existing software tests. Usually we modify a program in small ways to implement the mutation testing. Each mutated version is called a mutant and tests detect and reject mutants by causing the behavior of the original version to differ from the mutant. This is called *killing* the mutant. Mutants are based on well-defined *mutation operators* that either mimic typical programming errors (such as using the wrong operator or variable name) or force the creation of valuable tests (such as dividing each expression by zero). In this way the testers could develop effective tests or locate weakness in the test data used for the program or in sections of the code that are seldom or never accessed during execution. Most of this article is about "program mutation", in which the program is modified. A more general definition of *mutation analysis* is using well-defined rules defined on syntactic structures to make systematic changes to software artifacts. Mutation analysis has been applied to other problems, but is usually applied to testing. So *mutation testing* is defined as using mutation analysis to design new software tests or to evaluate existing software tests. Thus, mutation analysis and testing can be applied to design models, specifications, databases, tests, XML, and other types of software artifacts, although program mutation is the most common. There are some advantages that the mutation testing has. Mutation testing is a powerful tool to determine the coverage of testing programs. The testing can automatically set the steps in the software, like the creating of mutant software and the

white box method for unit testing. And the mutation testing is capable of comprehensive testing of correctly chosen mutant programs. However, there are also many disadvantages in mutation testing. Testing of mutants is costly and consuming. It requires many test cases to distinguish the mutant from the original software. It also requires a lot of testing before a dependable data is obtained. It needs an automated tool to reduce testing time. It is very complicated to use without an automated tool.[2]

2. Algorithm

In my code, I use separated phases to run different methods. From the contents in the class, we know that the random testing method is still a very efficient method to find the bugs in the software. But the problem in my code is that if we just use the random method, some branches will not be found by the random method. Because the random testing method just chooses the operation randomly, it may not cover all the branches. In this condition, many bugs that hide in the uncovered branches will not be found by the test. So one way to cover more branches is to use the mutation testing method which is very powerful to cover more branches.

In my code, I firstly separate the total time into three parts. The first part is half of the total time. I use this time to do the random testing. By setting the breadth and depth, we can use the BFS to do the random test firstly. In my code I use the random function to generate the random action sequence by using the rgen.shuffle( ). To keep the information of the actions, I use the S to keep the state and test content. we will use the enabled function to get all the actions that we can operate and use the rgen.shuffle( ) to make the action sequence random and we use the sut.safely( ) to check the correctness. The state will change when we use the safely function and we record this new state in the S “pool” to make the algorithm do the next cycle. When I do the BFS random test, I still use another global variable population to record all the list test and all the current branches we have tested. The global variable population will be used in the next phase.

After running the random test, I use about 1/4 of total time to run the mutation test. The mutation phase will use the population that we record in the previous phase. It will sort the population based on the number of branches that the random test covered before. And then choose the top bestNum cases. We pass the chosen cases to mutate function which will randomly choose part of the cases to replay. When we use the replay function, we need to add another parameter catchUncaught = True to make the program continue to run when the test find one bug. Otherwise the program will stop when it find a new bug. After replaying parts of these chosen cases, I randomly choose one action that can be operated and use the sut.safely( ) to check the correctness the case. After checking the correctness, if the test could find new branches, I will add the new current branches to the original global population and then do this operation again and

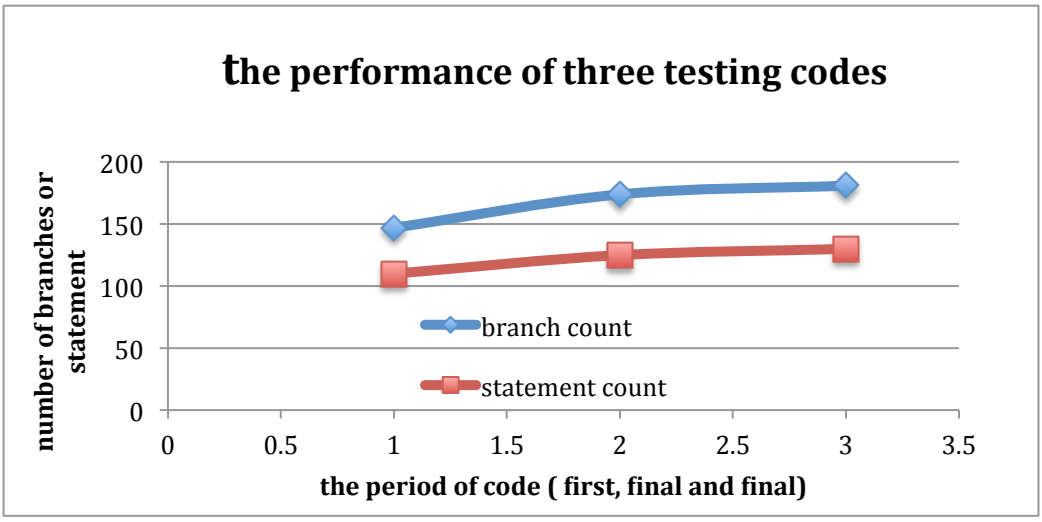
again until the time is used out.
In the final phase, I use the crossover function to crossover two sets of operations and try to find the new branches and check the correctness of the codes.

3. Results

I use the same parameters (30 1 100 5 0 1 1)to run my codes in three periods. The results are shown as below:

period	branch count	statement count
1	147	110
2	174	125
3	181	130

Table 1. the performance of three codes



Graph 1. The performance of three codes

From the graph we can find that the performance has a little improvement especially from the first period to the second period. The reason for this is that I only use the dfs in my first version of codes. In this situation, the test will only run certain number of actions(given by depth). Once the code runs all these actions, it will stop. So the result does not look good. In my second version. I use the randomly bfs method which will continue running actions in the format of BFS until the time is used out. So the performance improves sharply form the first version to the second version. In my final version. I add mutation and crossover function in my codes. By this way, we mainly find bugs and cover most of branches in the first phase of BFS random testing and record all the test actions and branches. In this way I can use the covered branches to mutate and crossover in the second phase and find more new branches and statements. Because the mutation and crossover function in my code is just to find some

ignored branches, the performance improvement is not very obvious. But some new branches are found indeed. To some extends, the mutation can crossover function can assist the random test to find some ignored branches.

[1] Richard Hamlet (1994). "Random Testing". In John J. Marciniak. Encyclopedia of Software Engineering (PDF) (1 ed.). John Wiley and Sons. ISBN 0471540021. Retrieved 16 June 2013.

[2] Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. Hints on test data selection: Help for the practicing programmer. IEEE Computer, 11(4):34-41. April 1978.