# TesterRank

1st Vighnesh Iyer
*dept. of Computer Science*
*of University of Illinois at Urbana Champaign*
Urbana, United States
vniyer2@illinois.edu

2nd Zihao Zhang
*dept. of Electrical and Computer Engineering*
*of University of Illinois at Urbana Champaign*
Urbana, United States
zihao5@illinois.edu

*Abstract*—**With modern day autograders and their speeeds, newer approaches for teaching introductory programming courses has emerged. One of the more popular approaches is the use of Many Small Programs (MSP) where instead of having One Large Project (OLP), instructors are asked to hand out small programming problems with a higher frequency, preferably one problem per week. Previous studies have shown that this has yielded happier students and better grades [1]. With instructors being asked to roll out programming assignments at a faster rate it is a possibility that the testing suite might take a back seat. In order to address this problem, we have come up with this idea of a tool called TesterRank. TesterRank leverages mutation testing to generate mutants of the instructor provided solution set and runs the test cases against them thereby assigning a score to each test case and also generates a rank for each test case. Additionally, TesterRank also generates two combinations of test cases, both of which tend to be subsets of the original testing suite thereby optimizing the test-suite by reducing the number of testcases based on various criteria discussed further in the paper.**

*Index Terms*—**MAJOR Mutation Framework,**

## I. Introduction

Mutation analysis is a well-known technique to assess the quality of test suites [2]. With the induction of MSPs in teaching introductory programming courses, it is becoming increasingly important to keep the quality of the testing suites in check. In order to measure the quality of testing suites which is essential for grading purposes in an academic setting, we've decided to leverage the concept of mutation testing. In the subsequent sections we begin with describing the open source tool (Major Mutation Framework) that we've chosen to use and the rationale behind this decision. Next, we'll discuss about the output files of the Major Mutation framework and their formats. Further, we will examine the various techniques we employed to wrangle the data and massage it to be used as input for our ranking algorithm. In the next section, we'll talk in depth about our ranking algorithm and the various metrics chosen to determine a rank. Later, we'll discuss about the optimized combinations of the testing suite and how the TesterRank tool generates these combinations. In the next section, we'll discuss the observations and results of using TesterRank on two java projects. We'll conclude this paper by giving an insight into our future work to evaluate TesterRank and make this tool an asset for instructors teaching introductory programming courses.

Identify applicable funding agency here. If none, delete this.

## II. Related Work

In our project, we give each test cases a rank and then perform test case reduction based our ranking.

Previous test case ranking techniques usually focused on code coverage or other indirect factors that may have affected the test cases' ability to find faults. However, our basic idea is to find mutants that individual test case can kill. Use this as main criterion to evaluate test cases.

As for test suite reduction, previous research has focused on code coverage as a measure of redundancy. The identification of such redundancy led to intense research in reducing the test-suite size by eliminating tests that are redundant with respect to the testing requirements. This led to techniques that remove tests without affecting the overall coverage achieved by the test suite. While meeting testing requirements is important, this does not guarantee the absence of bugs. Moreover, removing tests can negatively impact the fault-detection capability of the test suite. The effectiveness of the reduced test suite is generally measured in terms of the reduced test suites capability to detect faults compared to the original test suites capability to detect faults. Previously, re- searchers have used either mutation score or manually seeded faults to measure capability of test suite to detect faults.

Previous research on test-suite reduction, used two metrics to evaluate the quality of the reduction techniques: reduction in test-suite size and fault-detection capability. The fault-detection capability is most commonly measured as the ratio of the number of faults detected by the reduced test suite to the number of faults detected by the original test suite. The results reported by prior techniques show relatively high reduction in test-suite size and varying loss of fault-detection capability. Rothermel et al. [6]even found that the fault-detection capability of the reduced test suite is severely lowered. However, we found that the fault detection capability does not substantially decrease.

## III. Tool Selection

Among all the open source tools available on the internet, we selected several tools to compare them and to consider them as our tools to generate mutants automatically and analyze them. Below are the tools that we selected to compare for our project:

- Bacterio: A proprietary tool developed at University of Castilla-La Mancha (Spain), for which an evaluation version can be requested online.
- Javalanche: An open-source tool developed at Saarland University (Saarbrucken, Germany)
- Jester: An open-source tool developed by Ivan Moore.
- Judy: A proprietary tool developed at Worcaw University of Technology (Poland), freely available.
- Jumble: An open-source tool, originally developed by a software engineering company named ReelTwo (New Zealand).
- MAJOR: A proprietary tool developed as a collaboration of Ulm University (Germany) and Allegheny College (USA), available on request.
- MuJava: A proprietary tool developed as a collaboration between Korea Advanced Institute of Science and Technology (KAIST) and George Mason University (USA), and its third-party plugin for Eclipse, named MuClipse.
- PIT: An open-source tool developed by Henry.

Among these tools, MAJOR can be applied to most of the computer science education condition and can perform pretty efficiently. And MAJOR offers a particularly interesting feature: it allows to select the mutation operators in a tool-specific language, and it comes with reduced sets of mutation operators that are designed to have the same power of mutations of all possible operators without redundant mutations [**?**].

## IV. MAJOR Mutation Framework

MAJOR Mutation Framework is a compiler-integrated and noninvasive tool that provides fast fault seeding for arbitrary purposes. [8] It can generate mutants for small programs and then analyze the how much mutants the testsuite for this program can kill. When comparing the MAJOR with existing tools such as Jumble [3], MuJava [4], or Javalanche [5], we find that MAJOR is more suitable for our project. This is because MAJOR is integrated into the Java compiler and does not require a specific mutation analysis framework. This feature is very crucial since the small problem in computer science education can vary a lot. Compatibility of the tool is highly valued. Additionally, since it is compiler-integrated it only generates mutants which are compilable thereby reducing the load of sieving through the generated mutants.

### *A. Output of Major Mutation Framework*

- mutants.log-List all the Mutants generated by Major-javac.
- summary.csv-Export summary of mutation analysis including the mutants killed and the runtime details.
- covMap.csv-Export mutation coverage map.
- killMap.csv-Export mutation kill map.
- testMap.csv-Export mapping of test id to test name.

## V. Implementation Details

For our project, we take the result of the MAJOR Mutation framework as the input of our tool. Based on the results, we generate two building blocks, which synthesized all the information we needed for our algorithm which is discussed later in the paper. Then we designed our special metrics for determining the rank of each test case. Next, we implemented our algorithm to rank all the test cases based on fixed weights for each metric. In the end, we established the method to generate two kinds of optimized combinations.

### *A. Our building blocks*

As mentioned in the previous section the Major Mutation Framework generates a list of csv files as output which in their original form cannot be leveraged as input to our algorithm. As a result, we had to write a program to read through the various csv files. First of them was a method to generate the mapping of each test case with a list of mutants the test case covered; we named this map - CountCovMap. In order to populate the CountCovMap we had to read through the covMap.csv file. Next, it was also important to generate a mapping of each test case with a list of mutants the test case killed, and for this purpose we wrote a method to read and analyze the killMap.csv; we named this map - CountKillMap. Thus the fundamental building blocks of our input to the ranking algorithm were CountCovMap and CountKillMap. As an example, please take a look at the two tables to get a feel of how both the maps look.

TABLE I
CountCovMap

| **TestCaseNumber** | ***List of Mutants Covered*** |
|---|---|
| 1 | [1,3,5,6,7] |
| 2 | [2,4,9,11] |

TABLE II
CountKillMap

| **TestCaseNumber** | ***List of Mutants Killed*** |
|---|---|
| 1 | [1,3,7] |
| 2 | [2,9,11] |

As it is evident from both the tables, the list of mutants killed is a subset of list of mutants covered for each test case. Please note that, these are samples and not actual results.

### *B. Metrics for each test cases*

To evaluate the performance of a test case, we designed three parameters to show the test cases' ability to evaluate their rank in a test suite. Our Metrics include three major parts, which are, Mutants Killed, Efficiency, and Distinctness. All these three parameter are weighted differently in the ranking algorithm which will be discussed later. In the following part in this section, the parameters will be introduced in detail.

- Mutants Killed-The number of mutants killed be individual test cases. Which also plays the most important role in evaluating the test case's ability to find faults in program.
- Efficiency-Sometimes even if test cases can find mutants, they fail to kill them. So we design the efficiency to

evaluate the test case ability to kill mutants separately. Its basic idea is to see the portion of killed mutants among all the mutants covered by the test case.

$$Efficiency = \frac{MutantsKilledbyTestCase}{MutantsCoveredbyTestCase} \quad (1)$$

- Distinctness-Mutants killed by test cases can have huge overlap. For instance, say test case A kills mutant No.1,No.2,and No.3, and test case B kills mutant No.1 and No.2. In this case, test case B is redundant. So we designed a parameter to check this kind of redundancy, which is Distinctness. It measures the portion of mutants killed uniquely by the test case in the mutants killed by this test case.

$$Distinctness = \frac{MutantsKilledbyTestCaseUniquely}{MutantsKilledbyTestCase} \quad (2)$$

Using the three metrics above, we can evaluate the quality of individual test cases in detail. After various factors being considering in the algorithm, our ranking will be more reasonable and more representative.

### C. TestRank Algorithm

The pseudo code for our TestRank algorithm is as follows

```
TestRank (MutantsKilledMap,
     EfficiencyMap, DistinctNessMap)

Map results;
For Each TestCase in TestCases:
efficiency = EfficiencyMap[TestCase];
mutantsKilled = MutantsKilledMap[TestCase];
distinctNess = DistinctNessMap[TestCase];

double score = (0.6 * mutantsKilled +
                0.2 * efficiency +
                0.2 * distinctNess);
results[TestCase] = score;

return results;
```

As mentioned in our previous section, the fundamental building blocks of our project were the CountCovMap and CountKillMap. Our TestRank algorithm wouldn't directly take CountCovMap and CountKillMap as inputs, therefore both the maps had to be fine-tuned further. As discussed in another previous section, we had determined the metrics that we would use to rank the ability of the test cases viz. MutantsKilled, Efficiency and Distictness. We decided to use these three metrics as a motivation to develop our input to the TestRank algorithm. We first began by populating the MutantsKilledMap, which had the test case number as a key and the number of mutants that particular test case killed as value. Populating this map was relatively simple because it was directly derived from the CountKillMap. Next, it was the turn to populate the EfficiencyMap which had the test case number as the key and efficiency of that particular test case as its value. Populating this map involved leveraging both our building blocks viz. CountKillMap and CountCovMap. The last one to be handled was the DistinctNessMap, this again had the test case number as the key and the distinctness value of that particular test case as its value. Thus we had the inputs to our TestRank algorithm which were three maps - MutantsKilledMap, EfficiencyMap and DistinctNessMap.

TABLE III
COMBINATION EXAMPLE

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| A | X | X | X | | | | |
| B | | | X | X | | | |
| C | | X | X | X | X | X | X |
| D | | | | X | X | X | |
| E | | | | | | | X |

Having introduced the input parameters of our TestRank algorithm, lets dive straight into its implementation details. The first design choice that we as researchers had to make was the weightage distribution of our metrics, as in how much weight does he metric hold while finalizing the score of each test case. Our main heuristic behind assigning the weight of mutantsKilled metric as 0.6 was that, we wanted this particular metric to dominate the result of the score equation, as we wanted the number of mutants killed to be the dominating factor which which would mask the sum of the weights of both efficiency and distinctness. The rationale behind this was, even in the industry when it comes to mutation testing, a test cases' ability to execute mutants is supposed to be driving factor for determining its worth. As you can see from the pseudo code provided above, for each test case, we first fetched the metrics(mutantsKilled, efficiency and distinctness) from the input maps and multiplied the metrics with their respective weights (0.6, 0.2 and 0.2 in this case). We took a summation of these three values and stored that as a score for each test case in a map called results which had the test case number as the key and its score as value. Sorting this map based on its values gave us the desired ranking. When we presented this project to our class, we received valuable feedback saying that the weights assigned to each metric should not be fixed. On the other hand it should be inputted by the user who will be using the tool. Hence we have decided to tweak this algorithm to allow for more flexibility as suggested by our instructor. But that is part of our future work.

### D. Optimized Combinations

In addition to ranking the test cases, we not only generate the score to determine whether the test case is good or not, but we also leverage this information to optimize the test suites by deleting unnecessary test cases and redundant test cases.

We have written the algorithm to give out the combination with least number test cases and the combination with least redundant test cases. First, we should explain what are these combinations.

In TABLE III - A, B, C, D, and E are test cases where as 1,2,3,4,5,6,7 are mutants. We can see that the Combination with Least Number Of Test Cases is [A,C] as test case A kills mutants 1,2,3 and test case C kills 2,3,4,5,6,7. The combination [A,C] can kill all the mutants that can be killed by this test suite and has the least number of test cases. So we can call it Combination with Least Number Of Test Cases.

As for Combination with Least Redundancy Of Mutants Killed. In this example, it should be [A,D,E]. As we can see, test case A kills mutant 1,2,3. Test case D kills 4,5,6. Test case E kills mutant 7. This combination kills all the mutants and there is no overlap between the mutants killed by each test case. There will be cases where achieving zero overlap will be impossible, in such cases our program will generate a combination which will have minimal overlap.

## VI. Result

We tested the TesterRank tool on two java source files. One of them was a Triangle.java class which was provided by the developers of the Major mutation framework as part of their tutorial. Since we wanted this to be a fair study which doesn't necessarily over-fit, we decided to write a java program which would mirror a MSP (Many Small Problem) type problem. So we wrote a SampleStackImplementation. Our motive behind choosing this problem was that we wanted to come up with a problem such that students will be asked to write their own implementation for a stack of integers. In addition to the source file we also wrote test cases that we would use for grading this assignment. For this section, we would like to focus on the results of testing the TesterRank tool on our SampleStackImplementation as it more appropriately mirrors the class of problems we intend to use our TesterRank tool on.

After feeding the java source file along with the test cases the Major Mutation Framework developed log files which on proper examination helped in generating this map in this format -: TestCaseNumber - [List of Mutants Killed] 1 - [1,2,3,4,13,14,15,16,17,18,19,20,30,31,32,34,35,36,37,38,39] 2 - [1,2,3,4,13,14,15,,16,17,18,19,20,22,23,24,25,26,27,28,29] 3 - [1,2,3,4,13,14,15,16,17,18,19,20] 4 - [1,2,3,4,5,6,7,9,10,11,12] 5 - [1,2,13,14,15,21] 6 - [1,2,3,4] 7 - [1,2,22,23,24] 8 - [1,2,5,6,7,8] 9 - [1,2,3,4,22,23,24]

TesterRank assigned the following ranks, and scores to each test case - Rank 1 was TestCaseNumber 2 with score of 6.2125000059604645; Rank 2 was TestCaseNumber 1 with score of 4.126488097012043; Rank 3 was TestCaseNumber 4 with score of 3.80194804923875; Rank 4 was TestCaseNumber 3 with score of 3.6458333283662796; Rank 5 was TestCaseNumber 9 with score of 1.6071428582072258; Rank 6 was TestCaseNumber 8 with score of 1.3333333358168602; Rank 7 was TestCaseNumber 6 with score of 1.125; Rank 8 was TestCaseNumber 7 with score of 0.8000000007450581; Rank 9 was TestCaseNumber 5 with score of 0.7916666679084301;

TesterRank produced the following two combinations. Combination with least number of TestCases is 1,2,4,5,7,8. This combination as you can see from map mentioned above kills all the mutants by eliminating Test cases 3, 6 and 9. The tool chose to remove these test cases because there are no mutants these testcases (3,6,9) kill which are already not killed by the other test cases. When it comes to the combination with the least redundancy, TesterRank produces the same combination - 1,2,4,5,7,8 which encompasses all those test cases which kill all the mutants with minimal overlap. Please note that although both the combinations i.e. the combination with the least number of test cases and the combination with the least redundant test case is the same in this particular SampleStackImplementation problem, it won't necessarily be the same for all problems.

As for Triangle project, which has 33 test cases and 150 MAJOR generated mutants for this project. The results of two projects are similar, so we do not discuss the result for Triangle project in detail anymore.

## VII. Conclusion and Discussion

In this project, we implemented MAJOR Mutation framework as the base of our study. After analyzing the output of MAJOR Mutation framework, we use the output of MAJOR Mutation framework as the input to our tool. Based on the output of MAJOR Mutation framework, we created two building blocks, which are the mapping of test cases to the mutants they killed and the mapping of test cases to the mutants they covered. Then we designed our special metrics for determining the rank of each test case. Next, we implemented our algorithm to rank all the test cases based on fixed weights for each metric. We studied the practical need for test cases and gave each parameter reasonable weight. In the end, we established the method to generate two kinds of optimized combinations for test suite reduction.

However, our project is not perfect yet, it still has a few drawbacks. Currently, it is not flexible enough to handle various kinds of computer science courses. Moreover,the project that we conduct research on is limited, which will make the result not representative enough. Also, the algorithm itself isn't the most optimal when it comes to time and space efficiency. We used Brute Force idea so the algorithm is both space complex and time complex. Last but not least, our result lacks a empirical data set for computer science education to validate the result of our project.

## VIII. Future work

For our project, we think there are several ways to conduct further research to make it more practical and useful.

- Customize: In our rank algorithm, based on a general education requirements we set the weight of the parameters. However, they can be customized to meet various need for computer science education.
- Representative: More projects should be tested and analyzed to make the result more representative.
- Improve on Complexity: To make the tool fast enough. More advanced method should be used to generate final output.
- ValidationTrue validation and results of this tool will be tested when we get access to student submissions. Once

we've access to the student submissions, we can will feed the instructor generated solution and test cases to the major mutation framework. We'll use the results of the Major Mutation Framework as input to our TesterRank tool and check if there is any deviation in final student scores after reducing the testing suite.

## IX. Documentation of the TesterRank Tool

One of the key challenges that we encountered when we began working on this tool was using the Major Mutation Framework. The documentation provided by the makers of the Major Mutation Framework was too high level for inexperienced researchers like us and their documentation significantly lacked details. In order to begin using our tool the first thing you'll have to do is to download the official documentation from this site - (http://mutation-testing.org/doc/major.pdf). Next step is to download and install Java7 on your machine since the Major Mutation Framework is integrated with the Java7 compiler. If you've multiple Java versions installed on your Mac follow the instructions on this link to choose Java7 - (https://stackoverflow.com/questions/21964709/how-to-set-or-change-the-default-java-jdk-version-on-os-x).

Now that Java7 has been installed you can clone our project in a folder location of your choice. It should have the following folders - bin, config, doc, example, lib and mml. Please note that inorder to kick start the Major Mutation Framework you'll have to update your environment and prepend Majors bin directory to your execution path (PATH variable). For a Mac OS, the terminal command will be export PATH="/Users/vighneshiyer/Downloads/major/bin:$PATH$" Please note that, this path will be different for you based on where you have cloned your repository. I had cloned it in a folder called major in my Downloads folder and hence that terminal-command worked for me. Once this step is done, run this command in your terminal "javac -version" without the quotes. The output should be "javac 1.7.0-Major-v1.3.4". If this is not your output then it means that the major framework is not ready to work yet. If you're not getting this ouput please take a look at the official documentation and follow their instructions and do some googling to get that output. We struggled a lot to get this working and hence have detailed the steps we followed, but if they don't work for you please contact us.

I am assuming that you've reached the point where javac -version returns "javac 1.7.0-Major-v1.3.4". Now the next step is to first check if everything is in place and is working. cd into example/ant and run the command "ant clean".

Now in the ant folder, cd into src/triangle and when you do "ls" you should be able to see SampleStack.java. Note that if you want to upload your own source file you will have to delete SampleStack.java and paste your file in this exact same folder. (We know it is annoying but thats how the major mutation framework has been configured)

Please go back to the ant folder and cd into test/triangle/test when you do "ls" you should be able to see TestSuite.java. Note that if you want to upload your own test suite file, you will have to name it TestSuite.java and copy that file in the exact same directory and the first line of your TestSuite.java should be "package triangle.test;". (Please note that, these instructions were provided no where in the documentation and we had to figure this all by ourselves, hence it will be beneficial for you to follow our instructions)

Go back to the ant folder again, and run the command ant -DmutOp="=all.mml.bin" compile. If everything is working properly the terminal should say " [javac] Generated Mutants: 53 (61 ms)". This command basically compiles your src file and generates mutants.

The next step is to run the command "ant compile.tests" without the quotes. This command essentially compiles your TestSuite.java file

The next step is to run the command "ant mutation.test" without the quotes. This command does all the magic and the Major Mutation Framework will generate a list of log files which our program is going to consume as inputs.

After running these three commands you must still be in the ant folder.

Now it is time to unleash TesterRank. If you do "ls" you'll be able to see a TesterRank.java file in the ant folder. And the real sauce lies in this file.

The next step is to run this program for that run the following command in your terminal - "javac -Xlint TesterRank.java" without the quotes. It should not give you any errors if it does, go and fix them

The next step is to run the command "java TesterRank" and this will generate a ranking of all the test cases of your testing suite along with the two combinations.

## X. Work Done And Lines Of Code

For this project we (Vighnesh and Zihao) wrote close to 700 lines of code which includes the TesterRank.java file ( 500 lines of code), SampleStack.java file ( 100 lines of code) and TestSuite.java file( 100 lines of code). TesterRank.java takes in as inputs the various csv files and wrangles the data in them to maps which are represented by the various methods in the TesterRank.java file. Additionally all the code that determines the rank of each test case along with the combinations is all written TesterRank.java file in various methods. SampleStack.java is our sample solution which we mutated to generate mutants. TestSuite.java is our testing suite which has 9 test cases.

## References

[1] J. M. Allen, F. Vahid, A. Edgcomb, K. Downey and K. Miller, "An Analysis of Using Many Small Programs in CS1" SIGCSE '19, Minneapolis, MN, USA, pp. 585–591, Feb - March 2019.

[2] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. IEEE Computer, 11(4):3441, 1978

[3] Irvine, Sean A., et al. "Jumble java byte code to measure the effectiveness of unit tests." Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION (TAICPART-MUTATION 2007). IEEE, 2007.

[4] Ma, Yu-Seung, Jeff Offutt, and Yong-Rae Kwon. "MuJava: a mutation system for Java." Proceedings of the 28th international conference on Software engineering. ACM, 2006.

[5] Schuler, David, and Andreas Zeller. ”(Un-) covering equivalent mutants.” 2010 Third International Conference on Software Testing, Verification and Validation. IEEE, 2010.

[6] G. Rothermel, M. J. Harrold, J. von Ronne, and C. Hong, Empirical studies of test-suite reduction, STVR, vol. 12, 2002.

[7] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, “Electron spectroscopy studies on magneto-optical media and plastic substrate interface,” IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].

[8] Just, Rene, Franz Schweiggert, and Gregory M. Kapfhammer. ”MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler.” 2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011). IEEE, 2011.