ADVANCED SOFTWARE TESTING AND ANALYSIS
PROJECT REPORT

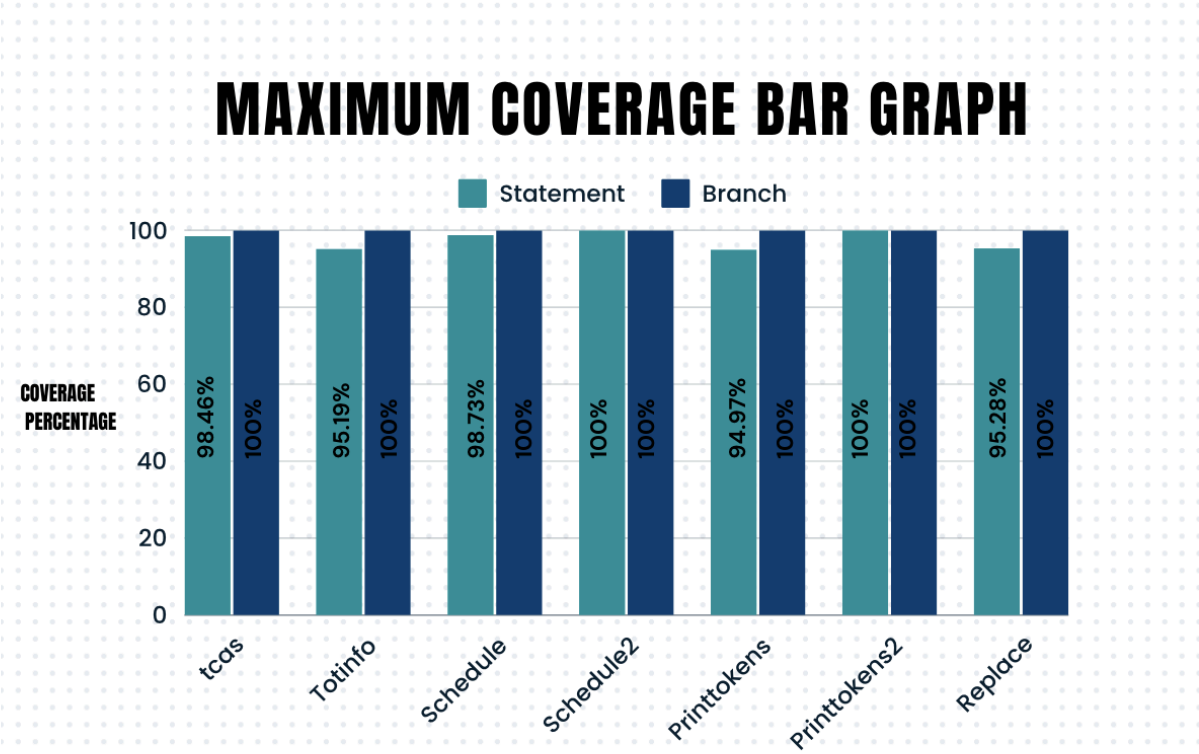
Team Mutants

Deepakindresh Narayana Gandhi (862464481)
Ramsundar Konety Govindarajan (862467731)

INTRODUCTION:

For each of the seven benchmark programs (tcas, totinfo, schedule, schedule2, printtokens, printtokens2, replace), the goal is to create six test suites. These test suites will be created by combining two coverage criteria (statements and branches) with three prioritization methods (random, total, and additional).

After creating the test suites, each benchmark program and its faulty versions will be compiled. For each test suite created for each benchmark program, the goal is to identify the set of faults that are exposed by that particular test suite.



Upon running the benchmark algorithms for all the programs we got the above maximum statement and branch coverage for each of those programs, which was the goal coverage for all the 6 types of test suites.

LANGUAGES USED:

PYTHON
C

CONSTRUCTION OF COVERAGE PRIORITIZATION:

STATEMENT COVERAGE:

The idea here is to extract the statement coverage details from gcov, which are stored in a text file called “coverage.txt”. We then extract the percentage of lines executed from this file. Next, we delete the gcda files to extract the individual statement coverage information of each test case in the “universe.txt” test suite.

BRANCH COVERAGE:

For branch Coverage we did the same as above but for gcov we included an option “-b” to include branch coverage details in the “coverage.txt” and we extracted the line that contained branches executed and took the % value from it.

RANDOM Prioritization:

For this, we applied a simple random swapping algorithm in the “randomP.txt” file by creating a copy of the “universe.txt” and running those test cases without deleting `.gcda` files, so it stores cumulative statements and branch coverage until it reaches the maximum coverage of the benchmark program which ran on the “universe.txt”. Thus, after this process, the random test suite size was reduced to an average of 6-8 test cases.

TOTAL Prioritization:

For Total Prioritization, we initially ran all the test cases in the universe.txt test suite and stored individual coverage information for both statement and branch. We stored this information in an array and sorted them and wrote it to 2 new files which are “totalPrioritization.txt” and “branchTotalPrioritization.txt”.

We then ran the same benchmark program in these 2 files and created “totalP.txt” and “branchP.txt” which stored the test cases that were required and did not constitute of the same line or same branch coverage by having a cumulative % that kept checking if there is an increase in the coverage without deleting gcda files and only if there was we added

them to the “totalP.txt” and “branchP.txt” until they reached the maximum coverage for the particular programs (tcas, totinfo, etc…).

ADDITIONAL Prioritization:

This was by far our best work and the most challenging. Since we were not very comfortable with c and it took us a lot of time to just do the basics, we shifted to Python and started from there. Here we took the “totalPrioritization.txt” and “branchTotalPrioritization.txt” as the starting test suite instead of the universe.txt since they were already sorted in terms of which had the maximum coverage.

Then this was the costliest step as it had a Worst Case of $O(n^2)$ but it did not actually take that much since we were able to identify all the test cases that took for full coverage within 5-6 test cases max.

How the algorithm works is that the first line from Prioritization.txt is extracted and stored in **cummulativeCoverage** array. All the lines of the test suite are stored in an array called **statementLines**. Now only the first line is run and removed from the statementLines array and let us assume the gcda file gives us 86% as statement coverage. Now running the second line even if the second line had 86% or 84% as its individual coverage, the gcda file will update its coverage only if there is an increase in new statements covered. So what we did was after running the second line we stored a **maxCoverage** value that got updated only if there was an increase in coverage and we ran the next line from statementLines after deleting the current line and also erase the gcda file since it now has the coverage of first two lines so we remove it and then rerun the test cases that are stored in the **cummulativeCoverage** array. Thus the maximum cumulative coverage is populated again and then we ran all the lines like this until we get the maxCoverage after running the first line. Now we got two lines and let us assume the statement coverage increased to 94% in two test cases, **now we appended the cummulativeCoverage** with the maxCoverage testcase and deleted all the gcda files and repopulated the entire statementLines Array but removed the testcases that were in cummulativeCoverage array.

Now we rerun the testCases that were in cumulative coverage and thus the currentCoverage becomes 94% and now we repeat the process until we get the maximum statement coverage reached the benchmark’s maximum coverage.

VERSION TESTING:

After getting all the test suites it was now time to run all the test suites and compare the outputs with the benchmark program. We stored the results of the benchmark program in an array and ran all the versions with each of the test suites and if we found that the outputs did not match we stored the results in a “**summary.txt”** file.

RESULTS:

PROGRAM: TCAS

COVERAGE CRITERIA	PRIORITIZATION	TEST SUITE SIZE	FAULTS IDENTIFIED
Statement	Random	6	9/41
Statement	Total	5	10/41
Statement	Additional	4	10/41
Branch	Random	5	9/41
Branch	Total	4	11/41
Branch	Additional	2	7/41
Universe.txt	-	1590	41/41

PROGRAM: TOTINFO

COVERAGE CRITERIA	PRIORITIZATION	TEST SUITE SIZE	FAULTS IDENTIFIED
Statement	Random	6	23/23
Statement	Total	6	23/23
Statement	Additional	5	23/23
Branch	Random	2	23/23
Branch	Total	1	23/23
Branch	Additional	1	23/23
Universe.txt	-	1026	23/23

PROGRAM: PRINTTOKENS

COVERAGE CRITERIA	PRIORITIZATION	TEST SUITE SIZE	FAULTS IDENTIFIED
Statement	Random	18	2/7

Statement	Total	7	6/7
Statement	Additional	5	5/7
Branch	Random	4	0/7
Branch	Total	1	1/7
Branch	Additional	1	1/7
Universe.txt	-	4072	7/7

PROGRAM: SCHEDULE

COVERAGE CRITERIA	PRIORITIZATION	TEST SUITE SIZE	FAULTS IDENTIFIED
Statement	Random	8	5/9
Statement	Total	4	2/9
Statement	Additional	3	2/9
Branch	Random	3	2/9
Branch	Total	1	0/9
Branch	Additional	1	0/9
Universe.txt	-	2634	9/9

PROGRAM: SCHEDULE2

COVERAGE CRITERIA	PRIORITIZATION	TEST SUITE SIZE	FAULTS IDENTIFIED
Statement	Random	5	1/9
Statement	Total	1	3/9
Statement	Additional	1	3/9
Branch	Random	3	1/9
Branch	Total	1	3/9
Branch	Additional	1	3/9

Universe.txt	-	2679	9/9

PROGRAM: REPLACE

COVERAGE CRITERIA	PRIORITIZATION	TEST SUITE SIZE	FAULTS IDENTIFIED
Statement	Random	16	13/31
Statement	Total	13	10/31
Statement	Additional	9	6/31
Branch	Random	5	6/31
Branch	Total	1	2/31
Branch	Additional	1	2/31
Universe.txt	-	5542	31/31

PROGRAM: PRINTTOKENS2

COVERAGE CRITERIA	PRIORITIZATION	TEST SUITE SIZE	FAULTS IDENTIFIED
Statement	Random	14	8/9
Statement	Total	5	7/9
Statement	Additional	4	7/9
Branch	Random	8	5/9
Branch	Total	1	6/9
Branch	Additional	1	6/9
Universe.txt	-	4057	9/9

INTERESTING EXPERIMENTS:

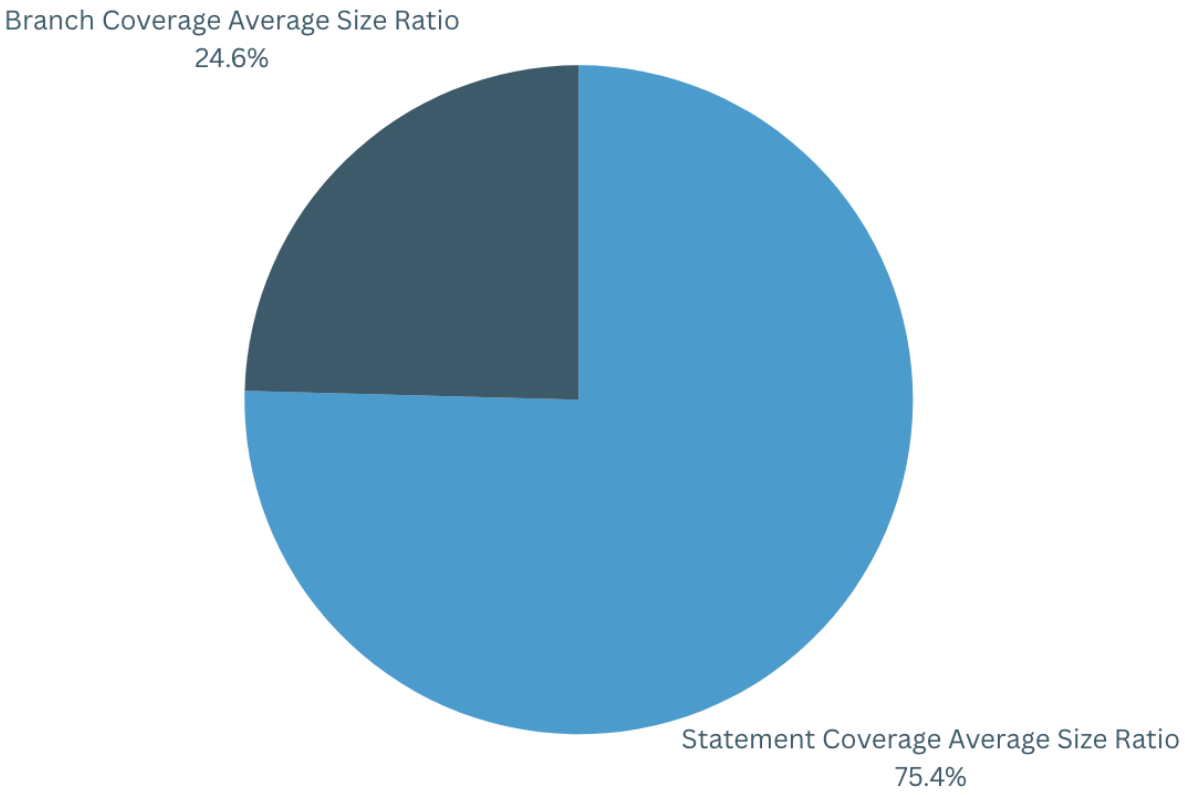
How small are your test suites as compared to the original number of available test cases?

Significantly smaller. The average size of the new test suites is 4.59 while compared to the average size of the original test suites is around 3805, which is almost a reduction of 99.87% in the size, but this did have its tole in the time that took to create these test suites and also even though the statement coverage and branch coverage reached the benchmark they were not able to identify the faults that the universe.txt test suite could identify.

How do the suite sizes change according to different coverage criteria?

Statement Average Test Suite size: 6.9

Branch Average Test Suite size: 2.28



How many faults are exposed by your test suites as compared to the total number of available faults, and as compared to the original test suite?

This can be viewed in the above table where we have clearly mentioned how the original test suite outperformed the new test suites but it would take a lot more time to run the entire suite as compared to the new test suites.

Which coverage criteria seem to be the least and most effective at being able to expose faults?

The most effective at being able to expose faults in this case is Total Coverage, while the least was Additional Coverage. Since, total coverage was a mid-point among both random and additional, it had the right test suite size and coverage, thus making it perform better than the rest. While additional although covered the same coverage with lesser testcases, the fault identification properties of the algorithm reduced with the reduction in test suite size and it was not worth the long wait that took to build it.

What other conclusions can you draw from your observations?

We were surprised at many things, since we thought that if we achieved the benchmark coverage, we would be able to identify at least 80-90% of the faults but that was rarely the case. All these efforts but we were not able to identify even 50% of the faults with the new test suites although we were able to significantly reduce the test suite size and achieve the same statement and branch coverage as the original test suite. The conclusion that we can draw from this is that 100% statement coverage or 100% branch coverage does not mean that the code is bug-free and we could still encounter many faults.

HOW DID WE UTILISE CHATGPT:

We utilized ChatGPT for simple functions like reading a file in c and Python, for things like swapping test cases in text files, truncating coverage files created by gcov, solving some errors that we encountered such as unicode errors where we then included encoding = “latin-1” into our read method in our Python file, and creating the project's docker file.

CONTRIBUTION OF TEAM:

- Initially, we both spent time individually reading the project description, understanding the question, and reviewing prioritization papers to gain insight into the concepts. We then discussed coverage criteria, cloning the repository, and had a basic overview of prioritization methods.
- For the implementation of coverage criteria, we both individually analyzed statement coverage and branch coverage using gcov. Deepak’s approach was efficient, and then we discussed it and decided to use that algorithm.

- For Random and Total Prioritization, we first brainstormed together and encountered some initial errors. After that, Ram's idea for solving the Random and Total problem was efficient, so we decided to follow that idea for Tcas.
- For the implementation of Additional Prioritization for branch and statement coverage for Tcas, Deepak proposed a good approach, so we decided to proceed with it.
- Afterwards, we discussed automating the version testing algorithm, and Deepak took charge of implementing the version testing and Docker components. We both implemented and tested all the programs for Tcas. Ram implemented all the programs for Printtokens, Schedule, and Schedule2, and we verified the outputs. For Printtokens2, Replace, and Totinfo, Deepak implemented all the programs and recorded the outputs.For the project report, we both took charge and worked on it together.

CONCLUSION:

In conclusion, the comprehensive approach of combining different coverage criteria (statements and branches) with prioritization methods (random, total, and additional) has enabled us to systematically create diverse and shortened test suites for each of the seven benchmark programs (tcas, totinfo, schedule, schedule2, printtokens, printtokens2, replace).

By compiling each benchmark program and its faulty versions, we were able to effectively assess the fault-exposing capabilities of each test suite. This process has provided valuable insights into the effectiveness of different testing strategies in uncovering faults within the software systems.

Even after testing with different criteria having full statement and branch coverage does not give guarantees that the code is bug-free and additional tests are always required for identification.

Overall, this methodology facilitates a systematic approach to testing and enables us to tailor test suites to specific criteria and prioritize testing efforts based on various factors. This comprehensive testing approach enhances the reliability and robustness of software systems by identifying and addressing potential faults early in the development lifecycle.

Steps to run the codebase and test:

You could also have a look at the readme.md file in github
Github Link

docker pull deepakintherace/benchmark:2.0

docker run -it deepakintherace/benchmark:2.0

Warning the container may run for around **2 hours** so please wait patiently!!!
This will run all the files but the **outputs will be stored in summary.txt** file, inorder to access it you might have to get into the docker CLI and use touch commands

The summary file locations are
/benchmark/tcas/summary.txt
/benchmark/totinfo/summary.txt

and so...

For Evaluation purposes and to check the summary in an instance we have also uploaded our program that has fully run in the branch **“data-branch”** instead of “master” where the master branch does not have any of the executables or the output text files.

To check for the generated test cases:
Please go to the branch **"data-branch"** in Github
In every program folder (like inside tcas, totinfo) you can find the generated testSuites
The names of the test case are given as follows:-
random-statement-suite.txt - **randomP.txt**
random-branch-suite.txt - **branchRandomP.txt**
total-statement-suite.txt - **totalP.txt**
total-branch-suite.txt - **branchTotalP.txt**
additional-statement-suite.txt - **additionalP.txt**
additional-branch-suite.txt - **branchAdditional.txt**