Effectiveness of Different Test Case Prioritization Methods Based on Coverage Criteria

CS 206 Project, Winter 2024

Handed Out: **February 5, 2024**
Due Date: **March 4, 2024, 11:59pm**
Grade Weight: **20% of total course grade**
Note: **This project can be completed in groups of two.**

Overview

In this project, you will compare the effectiveness of different kinds of test case prioritization methods in exposing faults via multiple coverage criteria. You are provided with a set of benchmark programs, a set of test cases for each benchmark, and a set of faults for each benchmark.

- You will collect the coverage information for each test case.
- For each benchmark program, you will use test case prioritization methods to create test suites according to the different coverage criteria.
- For each created test suite, you will evaluate its fault-exposing potential by identifying the set of available faults that are exposed by the test suite.
- Finally, you will report on your experimental results and observations.

Teams

You may decide whether you want to work individually or in a team of two. In either case, please choose a **team name** (whether you are working individually or in a group of two). If you are working in a group of two, **you must add your team information to the following spreadsheet**:

https://docs.google.com/spreadsheets/d/18Pxas1WQGtWBhSEUClE5SLnrQZXcLH0T8V0I_YzdrMA/edit?usp=sharing

IMPORTANT: you must enter your team information by **Feb. 13, 2024 at 5pm** in order to be allowed to work in a team. After this deadline, you will **not** be able to join a team and will be required to complete the project individually.

Get Required Files

We will be using **GitHub Classroom** to distribute the initial files and to collect assignments. To get started, create a GitHub account if you have not already. Then, while logged in, visit this invitation link:

https://classroom.github.com/a/A0n3YMjX

If you're working in a pair and your partner has already created your team, you should see it listed, and go ahead and join it. Otherwise, type in your team name to create your team. Once completed, GitHub will create a repository for your team containing the benchmark codes. You can then clone this repository to your machine and push your changes as you make them.

You should also receive by email an invitation to join the ucr-cs206 GitHub organization, which you should accept.

Benchmark Programs

You are provided with a set of 7 relatively small benchmark programs written in `C`. Each program is associated with a set of faults and a set of test cases:

Program Name	# of Available Faults	# of Available Test Cases
tcas	41	1590
totinfo	23	1026
schedule	9	2634
schedule2	9	2679
printtokens	7	4072
printtokens2	9	4057
replace	31	5542

General Directory Structure for Benchmarks
All benchmark programs are contained within a folder named "`benchmarks`". Within this folder, there is a subdirectory for each of the seven benchmark programs.
Within the subdirectory for a benchmark program, you will find the following folders and files:

- **The benchmark program**
  The benchmark program is specified as a `.c` file, and possibly one or more associated `.h` files.
  Example: Program `tcas` is located in the file "`benchmarks/tcas/tcas.c`" (there are no associated `.h` files for this program).
- **The set of faults**
  The faults are specified as a set of "faulty versions" that are associated with each benchmark program (**one fault per faulty version**). Each faulty version is contained within a folder that is named with the letter "`v`", followed by the number of the faulty version. The program code for each faulty version is identical to that of the (original) benchmark program, except for a slight modification to the code that represents the fault.

Example: Program `tcas` is associated with 41 different faulty versions, contained in the folders named "`benchmarks/tcas/v1`" through "`benchmarks/tcas/v41`".

- **The set of test cases**
  The test cases are specified in the file "`universe.txt`" (**one test case per line**).
  Example: The test cases for program `tcas` are contained in the file "`benchmarks/tcas/universe.txt`".
- **Test case input file directories**
  Any other folders that may be present contain input files that are used by the associated test cases. You don't need to do anything with these folders except to make sure they are present in the current working directory when running the benchmark program.

Compiling and Running Each Benchmark Program
All benchmark programs take some input values (either command-line parameters, the name of an input file, or both), and produce output that is written to the standard output stream (by default, the screen). The following table shows how to compile each program, how to execute the compiled program using the first test case specified in the associated "`universe.txt`" test case file, and the names of any input file directories that are required by the associated test cases. Note that the test cases assume that any necessary input file directories are contained in the current working directory when running the benchmark program.

Program Name	How to Compile (ignore compiler warnings)	Example Command to Execute Program	Input File Directories
tcas	`gcc -Wno-return-type -g -o tcas tcas.c`	`./tcas 1258 1 0 897 174 7253 1 629 500 0 0 1`	(none)
totinfo	`gcc -Wno-return-type -g -o totinfo totinfo.c -lm`	`./totinfo < universe/jkADl.mat`	`universe`
schedule	`gcc -Wno-return-type -g -o schedule schedule.c`	`./schedule 5 1 1 < input/dat027`	`input`
schedule2	`gcc -Wno-return-type -g -o schedule2 schedule2.c`	`./schedule2 5 1 1 < input/dat027`	`input`
printtokens	`gcc -Wno-return-type -g -o printtokens printtokens.c`	`./printtokens < inputs/newtst122.tst`	`inputs`
printtokens2	`gcc -Wno-return-type -g -o printtokens2 printtokens2.c`	`./printtokens2 < inputs/newtst122.tst`	`inputs`
replace	`gcc -Wno-return-type -g -o replace replace.c -lm`	`./replace '@|' 'E)m' < input/ruin.1373`	`input`, `moni`, `temp-test`

The `-Wno-return-type` flag is required to prevent compile errors in some older-style C code. With the above commands, you should be able to compile all the original and faulty versions of the programs successfully (ignore all compiler warnings). If you run into an unexpected error, let Prof. Sridharan know.

We strongly recommend using GCC version 9.1 or higher to compile the benchmark code. You can run `gcc --version` to see which version of gcc is installed on your system by default; you may need to update it.

Important notes for Mac OS: the version of the `gcc` command installed by default actually runs the clang compiler; you will need to install gcc separately (e.g., via homebrew, by running `brew install gcc`). As of Feb. 5, 2024, Homebrew installs gcc version 13.x. To actually run this version of gcc, you should use the command `gcc-13`, **not** `gcc`. Similarly, to run the right version of gcov, run the command `gcov-13`, **not** `gcov`.

Important notes for Windows: On Windows, we *strongly* recommend doing the project using the Windows Subsystem for Linux. All our instructions assume a Unix-like environment, and the default Windows shell may cause issues we have not anticipated.

Coverage Criteria

We will consider 2 different kinds of coverage criteria in this project.

- **Statement coverage**. Statement coverage is the selection of tests so that every statement has been executed at least once.
- **Branch coverage**. Branch coverage is a requirement according to which, for each conditional branch in the program (such as due to if statements, loops etc.), the branch condition must have been true at least once and false at least once during testing.

Obtaining Coverage Information

For each test case associated with each benchmark program, you have to collect the coverage information for each of the two coverage criteria listed above. The coverage information can be collected by using a UNIX tool called **gcov**. It is your responsibility to read the gcov documentation to figure out how to run it, how the coverage information is stored in files, and how to parse and use that information. Here are some useful but older links for the usage of gcov:

- Monitoring Statement Coverage with gcov
- Monitoring Branch Coverage with gcov

You may want to look into the json format that can be output by gcov, as it may be easier to parse; it is documented here. JSON support requires version 9.1 or higher of gcov, which should be installed if you are using a recommended gcc version (see above). We **strongly** suggest writing some small tests (i.e., your own small C programs with corresponding inputs to exercise different paths) to ensure that your generation and parsing of coverage information is working as expected.

Note in particular the following from the documentation:

> Each time you run the program, the execution counts are summed into any existing .gcda files, so be sure to remove any old files if you do not want their contents to be included.

So, if you are computing coverage for individual test cases (not overall coverage), be sure to delete .gcda files before each run of the program. Otherwise, you will get cumulative coverage.

Test Case Prioritization

Test case prioritization techniques schedule test cases in an execution order according to some criterion. Here are the prioritization methods you will use:

- **Random prioritization**. Randomly order the test cases in a test suite.
- **Total Coverage prioritization**. Prioritize test cases according to the total number of statements/branches they cover simply by sorting them in order of their total statement/branch coverage.
- **Additional Coverage prioritization**. Iteratively perform the following two steps until all statements/branches are covered by at least one test case: (i) select a test case that yields the greatest additional statement/branch coverage; and (ii) then adjust the coverage information on subsequent test cases to indicate their coverage of statements/branches not yet covered by a test already chosen for the suite.

If you want to read more about the above methods you may refer to the following papers for more details (all available on Canvas under Modules > Research Papers for Course Project).

References:

- S. Elbaum, A. G. Malishevsky, and G. Rothermel, Test Case Prioritization: A Family of Empirical Studies. IEEE Transactions on Software Engineering, 2002.
- B. Jiang, Z. Zhang, W. K. Chan, and T. H. Tse, Adaptive Random Test Case Prioritization. IEEE/ACM International Conference on Automated Software Engineering, 2009 (ASE'09).
- G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, Test Case Prioritization: An Empirical Study. IEEE International Conference on Software Maintenance, 1999 (ICSM'99).

Detailed Requirements

The following tasks will need to be performed to complete this project.

1. Create the following test suites
   - For each benchmark program, create six test suites using the combinations of two coverage criteria (statements and branches) and three prioritization methods (random, total, and additional).
   - Each test suite should be "adequate" with respect to the given coverage criterion. That is, the generated test suite should cover all of the entities that are covered by the entire set of available test cases. The final test suite you generate for each case should be significantly smaller than the set of all available test cases. For example, even though there may be over 2000 test cases available for a benchmark program, a test suite may only need 10 or 15 test cases in order to become adequate for some particular coverage criterion. **NOTE:** do **not** include redundant tests in your generated test suites. E.g., if you are doing the random prioritization technique, your suite thus far covers statements {1,2,3,4,5}, and the next randomly-chosen test covers {1,2}, you should **not** add that test to the suite. This will ensure that the test suites are not unnecessarily large.
2. Evaluate the fault-exposing potential of each test suite
   - Compile each benchmark program, and compile each of the faulty versions associated with each benchmark program.
   - For each created test suite for each benchmark program, identify the set of faults that are exposed by the test suite. A fault is considered to be "exposed" by a test suite if there exists at least one test case in the test suite that causes the program to produce *different* output when run on the faulty version, as compared to when run on (the original version of) the program. In other words, the output from the original version of the program on any input is **always** assumed to be correct, and any deviation from that output is incorrect. Note that even though there may be over 20 total faults available for a benchmark program, it may turn out that a particular test suite is only able to expose 2 or 3 of these faults.
   - Also report the set of faults exposed by the original test suite (i.e., all the originally-provided tests).
3. Report your experimental results and observations
   - Create a data table listing your experimental results. The table should show, for each created test suite associated with each benchmark program, the size of the test suite (number of test cases), and the number of faults exposed by the test suite. Give the same data for the original test suite (number of test cases and number of faults exposed).
   - Write about any interesting observations you can make from your experimental results. How small are your test suites as compared to the original number of available test cases? How do the suite sizes change according to different coverage criteria? 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? Which coverage criteria seem to be the least and most effective at being able to expose faults? What other conclusions can you draw from your observations?

Implementation and Submission

Programming Languages to Use

You may use any language you choose to implement the project. You must commit into your repository a description file (README.md) for how to compile and run your code. The instructors *must* be able to successfully run your code on their own machine (with a reasonable amount of effort) given your instructions in order for you to receive full credit. We recommend (but do not require) that you create and share a Docker container with all required dependencies to make your code run, to make this process easier, particularly if you choose a language where installation of the language runtime / libraries will be non-trivial. Full documentation for Docker can be found here: https://docs.docker.com.

Third-Party Libraries

Generally speaking, the only libraries you can use for your project are the standard library for your programming language and a JSON-parsing library if it is not included. Use of any other third-party library requires **prior approval** of the instructor.

Use of the gcovr Python library, or similar libraries for interacting with gcov, is **not** allowed. The code you need to write to parse the output from the standard `gcov` command should be straightforward, and part of the point of this project is to get you familiar with using this standard coverage tool and parsing its output.

ChatGPT / LLMs

Use of ChatGPT and other LLMs is *allowed* for the project. You should be able to get free access to GitHub Copilot as a student; see here for how to verify your student status. If you choose to use any LLM to assist in coding your project, you **must** do the following:

1. Add a section to your project report documenting how you used the LLM, how it was helpful, and what problems you encountered.
2. For any significant chunks of code in your implementation taken from LLM output (more than 1-2 lines), add a comment in the code indicating it was written with the help of an LLM.

Whether you use an LLM or not, you must still abide by the Academic Integrity rules listed at the end of this document.

Items to Submit

Please submit the following items for this project in your GitHub repository:

- A file `name.txt` in the root of your Git repository, which contains the following information for each team member on a separate line: `Name, Email, SID`

- The report you have prepared regarding your experimental results and observations, according to the above detailed requirements. Place the report in a `report` folder. A PDF is preferred.
- The test suites you have created for each benchmark program. The test suites should be stored in .txt files named appropriately (e.g., `random-statement-suite.txt`), and they should be in the same format as the `universe.txt` files, containing one program input on each line.
- The programs you wrote in order to get your experimental results, along with a README file to describe how to compile and run each program. Please also document the location(s) of your test suites.
- If you are turning in a Docker container, provide a link to the container in the README file. It is better to host the container on Google Drive rather than checking it into your Git repository.

Please do not commit executables or other generated files into the repository. You can add a `.gitignore` file to ensure these files do not get added.

Academic Integrity

Each team is expected to complete the assignment **on their own**. You can discuss the assignment with others in the class, but code **cannot** be shared between teams or be copy-pasted from the Internet. If you are in doubt as to whether some code excerpt can be reused, please ask the instructors. See also the prohibition of third-party library usage and the discussion of LLM usage above under "Implementation and Submission".

Any sharing of your project code, either with other teams or on a public web site / repository, is **prohibited** and will be treated as an academic integrity violation. We recommend you only push the project code to the private repository created for you as part of the GitHub Classroom. See also the Academic Integrity document distributed at the beginning of the quarter, which you were required to read.