Jakin Chan
862181945
jchan419@ucr.edu

Ryan Lu
862136685
rlu033@ucr.edu

Faults Found - file locations

Notable files:

benchmarks/<prog_name>/results/uni_results.txt

A list of all the faults found from the original test suite (universe.txt).

- tcas: https://github.com/ucr-cs206/cs206-project-on-rounds/blob/master/benchmarks/tcas/results/uni_results.txt
- totinfo: https://github.com/ucr-cs206/cs206-project-on-rounds/blob/master/benchmarks/totinfo/results/uni_results.txt
- schedule: https://github.com/ucr-cs206/cs206-project-on-rounds/blob/master/benchmarks/schedule/results/uni_results.txt
- schedule2: https://github.com/ucr-cs206/cs206-project-on-rounds/blob/master/benchmarks/schedule2/results/uni_results.txt
- printtokens: https://github.com/ucr-cs206/cs206-project-on-rounds/blob/master/benchmarks/printtokens/results/uni_results.txt
- printtokens2: https://github.com/ucr-cs206/cs206-project-on-rounds/blob/master/benchmarks/printtokens2/results/uni_results.txt
- replace: https://github.com/ucr-cs206/cs206-project-on-rounds/blob/master/benchmarks/replace/results/uni_results.txt

/benchmarks/<prog_name>/results/by_case/universe_bycase_results.txt

A list of faults exposed by each individual test case.

The number on the left is the line number of the test case and the numbers on the right are the list of faults discovered by each test case.

benchmarks/<prog_name>/results

A list of the faults discovered by each test suite.

benchmarks/<prog_name>/results/by_case

A list of the faults exposed by each test case in each test suite.
Note that the number on the left is not the line number of the test case but more of a list numbering.

Results

Types of Coverage:

- Statement Coverage (S)
- Branch Coverage (B)

Prioritization Techniques:

- Random prioritization (Rand)
- Total Coverage prioritization (Total)
- Additional Coverage prioritization (Add)

Program	Coverage+Priority	Size (Number of Test Cases)	Faults Found
tcas	**Original**	1590	41/41
	S-Rand	6	6/41
	S-Total	5	9/41
	S-Add	5	7/41
	B-Rand	13	11/41
	B-Total	13	13/41
	B-Add	11	9/41
totinfo	**Original**	1026	21/23
	S-Rand	8	15/23
	S-Total	5	13/23
	S-Add	5	13/23
	B-Rand	7	13/23
	B-Total	5	13/23
	B-Add	5	13/23
schedule	**Original**	2634	9/9
	S-Rand	6	1/9
	S-Total	3	2/9
	S-Add	3	2/9

	B-Rand	12	7/9
	B-Total	10	3/9
	B-Add	7	5/9
schedule2	**Original**	2679	9/9
	S-Rand	8	2/9
	S-Total	4	4/9
	S-Add	4	4/9
	B-Rand	12	5/9
	B-Total	8	6/9
	B-Add	5	2/9
printtokens*	**Original**	4072	1/7
	S-Rand	17	2/7
	S-Total	5	5/7
	S-Add	4	5/7
	B-Rand	16	3/7
	B-Total	6	5/7
	B-Add	5	4/7
printtokens2*	**Original**	4057	1/9
	S-Rand	8	7/9
	S-Total	2	6/9
	S-Add	2	6/9
	B-Rand	15	5/9
	B-Total	5	8/9
	B-Add	3	7/9
replace	**Original**	5542	12/31
	S-Rand	23	8/31
	S-Total	17	6/31
	S-Add	10	5/31
	B-Rand	29	6/31

	B-Total	21	6/31
	B-Add	12	6/31

Interesting Observations on…

The number of test cases:

Overall in general, the number of test cases in each test suite is significantly lower than the total number of test cases to choose from. This makes sense since a large test suite is going to have duplicate coverage, which is not the goal of this project.

Among the different criteria, the test suites where the test cases are chosen by random results in the highest number of test cases. This makes sense as it is essentially shooting in the dark and hoping to hit the target. Total coverage and additional coverage test suites generally have a similar number of test cases, with total coverage sometimes pulling ahead. This also makes sense because just having the additional coverage requirement makes it so that there is not significant overlap in the path/coverage.

The best test suite criteria:

The best test suite differs for the criteria, but total coverage for branches consistently is either at the top or near the top in terms of faults found. This makes sense because in order to find more faults, one needs more unique paths to be covered. In this case, branch coverage is superior because it forces more unique paths compared to statement coverage, increasing the chances for more faults found.

Within branches, randomization does help to a certain degree since it results in more total paths, and subsequently more chances of finding unique paths and faults (hence randomization sometimes finds the most faults above). However, it does come at the cost of having more tests. Additional coverage, on the other hand, is more efficient in terms of using less test cases, but it optimizes for paths that are completely different from each other with no overlap. This pushes away cases that involve specific branch choices of one test and choices of another test.

The worst test suite criteria:

In terms of the worst criteria for finding a test suite, it differs, but random statement coverage tends to rank at the bottom fairly frequently. This is interesting because in some cases, randomization produces the best test suite. We think that this is just a product of randomization in general, one either gets a good test suite or a bad one.

The number of faults exposed:

The total number of faults exposed by the original test suite is not 100%. A good chunk of the time, it does find all the faults, but not all the time. We think that there is most likely an issue with our code.

For tcas, our test suites find roughly a quarter of the total faults. For totinfo, our test suites are finding a little over half of the available faults and faults from the original test suite. In the schedule code, a quarter of the faults were found by our test suites, and in schedule2, roughly half of the faults are discovered. In printtokens and printtokens2, the original test suite only found 1 of the total faults, which should be an error on our end. However, the test suites we created found on average more than half of the faults. Lastly, for replace, the original test suite only found 12/31 of the faults, but the test suites we made found on average more than half of the faults of the original test suite.

Initially, we thought that branch coverage would involve a lot more tests compared to statement coverage because one needs to cover the cases where the branch is taken and when it is not. Interestingly, the branch coverage test suites were larger, but not by a significant amount compared to statement coverage most of the time. This makes sense since to achieve statement coverage, one usually needs to take both sides of a branch, so the jump to branch coverage is smaller than we thought.

Our work steps:

1. Build something that can compile and run the test cases in universe.txt
2. Run all of them on the original file and track coverage from the json files. Save and sort.
3. Build 3 tests suites for - random prioritization, total coverage prioritization, and additional coverage prioritization for tcas.c
4. Run the final test suites on all of the faulty files. If output doesn't match, tick fault for failing test case. (NO GCOV). Store into .txt file in folders.
    a. bash to get outputs on all files
5. In .cpp file, collect all .txt files' data and compile it into a single text file as a solution.
    a. cpp
6. Create a set/table where we list the files/fault case each test case finds a fault.
7. Each program has its own compile command, so we are just going to input them all manually.

Issues we ran into/ things that threw us off:

- '<': Some programs use a file for input. However, since we were using shell script, < was not recognized as part of the console command. To fix this, we had to parse through each line, looking for where that arrow was, and run the commands with the newly formatted arguments.
- Some faulty cases threw segmentation fault errors, which could affect the fault-finding. To fix this, we made sure to write both stdout and stderr outputs to our comparison files.
- Jakin's machine runs the below code concurrently, resulting in race conditions. However, this was not the case for Ryan's machine. This caused a lot of headaches.

```
gzip -dk $1.gcov.json.gz && rm $1.gcov.json.gz
```

- There are multiple branches on one line
    - This threw off the original assumption and data structure used to store the changes

- The priority queue has a slightly different order every time for nodes with the same coverage/additional coverage
    - We are assuming this is caused by iterating through the map to push into the priority queue
- There were also a lot of errors and setbacks trying to figure out the semantics of creating and running a bash script
    - Many many many issues with getting bash scripts to work as this is new to both of us.
- Issues with testing the code on a smaller dataset first
    - Does not always translate to the actual test cases
- The original test suite themselves do not reach 100% coverage
    - We tested this by not deleting the .gcda files and let it stack up
- Time stamps for commands not lining up (these are warnings rather than errors I think)
    - We just deleted the time stamp checkers

Sources:

- https://ioflood.com/blog/bash-parse-json/
- https://www.cyberciti.biz/faq/unix-howto-read-line-by-line-from-file/
- https://www.geeksforgeeks.org/bash-scripting-for-loop/
- https://gcc.gnu.org/onlinedocs/gcc/Invoking-Gcov.html
- https://askubuntu.com/questions/420981/how-do-i-save-terminal-output-to-a-file
-

Libraries

- Decided to use C++ for parsing json files.
    - https://github.com/nlohmann/json

Things we used Chatgpt for:

- Figuring out how to even start (how to automatically compile and run files)
- This was helpful just because we had no idea how to start.
- It was helpful for idea generation

You

is there a way to automate compiling and running c files

ChatGPT

Yes, you can automate the process of compiling and running C files using various tools and scripting languages. Here's a simple way to do it using a shell script:

bash

```bash
#!/bin/bash

# Compile the C file
gcc -o program program.c

# Check if compilation was successful
if [ $? -eq 0 ]; then
    echo "Compilation successful. Running program..."
    # Run the compiled program
    ./program
else
    echo "Compilation failed."
fi
```

Save this script in a file, let's say `compile_and_run.sh`, and make it executable using the command `chmod +x compile_and_run.sh`. Then you can simply run the script using `./compile_and_run.sh`, assuming your C file is named `program.c`.

This script will compile the C file using `gcc` and then run the compiled program. If the compilation fails, it will display an error message. You can customize it further based on your specific requirements. Additionally, you can integrate this script into your development workflow or use build automation tools like Makefile or CMake for more complex projects.