SFU CMPT 473 SPR2019 Assignment 3

Selecting a Project

Software Under Test

- jgrapht (specifically jgrapht-core) https://www.openhub.net/p/jgrapht/enlistments

Requirements:

1. Openhub Project (check)
2. > 10,000 LOC (check)
    - jgrapht-core: **src = 32623 LOC** and **test = 13105 LOC**. Obtained from the 'Statistic' plugin of IntelliJ IDE.
3. Project uses JUnit (check)

Project Metrics:

- Identification of the open-source project
    - jgrapht-core
- Supporting Organization
    - https://jgrapht.org/
- Size of code base
    - Total LOC = 45728: **src = 32623 LOC** and **test = 13105 LOC**. Obtained from the Statistic plugin of IntelliJ IDE.
- Proposed evaluation platform (OS, Language)
    - MacOSX 10.14 and GNU/Linux (Ubuntu 16), Java
- Build time to compile and link an executable from the source

    - 12s

- Number of tests, lines of code, and execution time for the test suite.
    - 358 tests, runs in 16s

Group Analysis and Results

```
Number of mutants generated:
    MutantsGenerated = 7454

Number of mutants covered by the test suite:
    MutantsCovered  = MutantsKilled + MutantsLive
                    = 6037
Number of mutants killed by the test suite:
    MutantsKilled   = 4276

Number of live mutants:
    MutantsLive         = 1761

Overall mutation score & adequacy of test suite:
    MutationScore   = MutantsKilled / MutantsGenerated
                        = 57.37%
    TestSuiteAdequacy = MutantsKilled / MutantsCovered
                        = 70.8%
```

Discussion and Explaination of Observed Results

- Does killed + live = covered or … = generated? Why or why not? What do the results tell you about your test suite?

Killed + Live = Covered. This is because the generated mutants can include mutants that are not detectable (covered) by the testsuite. We found that there were functions in the src code for which mutants were generated, but these functions were not called by any test code…or used anywhere else in the codebase for that matter.

- Does the test suite exhibit weaknesses? How can it be improved?

The result show that the testsuite doesn't detect almost half of the generated mutants (mutation score of 57.37%). This means there could be potential bugs that the testsuite will not catch. This can be improved by adding more testcases that target the uncovered mutants.

- Does the test suite exhibit strengths? How do you recognize them?

One strength of the testsuite is that for the mutants that it covers, it can effectively detect 70% of them, meaning there is only a 30% chance of a bug slipping through. Another strenght is the fast speed of test execution, this allows mutants to be analyzed quickly, hence more kinds of mutations can be introduced and analyzed.

- Contrast the cost & benefit of mutation testing with other testing methods.

One benefit of mutation analysis is that it enables us to measure the adequacy of the testsuite rather than the program under test. Secondly, it does so in a purely automated fashion. Other methods like input-space partitioning enable us determine our test coverage level, but require a great deal of planning and manual design. The downside to mutation testing though is that it can be slow for a large codebase because many mutants are generated for a single function and this can grow exponentially.

- What was easy? What was difficult? What obstacles did you face in applying mutation analysis to a real world project, and how did you overcome them?

Running the mutation tests with Major was easy. We just had to wait while it churned through thousands of mutants and then generated a summary report. However, setting up the Java project to work with Major was very painful! Mostly because of limited documenation of version incompatibilities.

From examining our build error logs, we hypothesized that we needed an older version of JDK compatible with Major's tools (this was not documented by Major). Consequently we would have to choose an opensource project compatible with that version of Java or alternatively use an older version of the project. We chose the later. The rest of the time was spent reading the Major documenation and examining the examples to understand what each component does and how to properly integrate it to our project.

- Do you have any other interesting insights or opinions on the experience?

We are of the opinion that the above experience could be made a lot better if Major's documentation clearly states the version of its dependencies. Also, Major could be upgrated to support the later versions of Java, so it can be used for newer projects…this will enable baking mutation testing right in from the start, hence resulting in better quality software.