REDEFINING PRIORITISATION

Liang, J. | Elbaum, S. | Rothermel, G. (2018)

Aim

In this work, a lightweight and highly-efficient cost-saving algorithm (CCBP), based on test suite failure and execution history, is used to prioritise commits, contrary to traditional Test Case Prioritization (TCP). Results were obtained by cross-checking this approach against 3 illustrative CI datasets.

Google Dataset

Google splits CI testing into two phases: *pre-commit and post-commit,* were first only smaller tests are applied. The dataset used is the Google Shared Dataset of Test Suite Results, that contains over 3.5 Million test suite executions, over 30 days. Features include test suite ID's, change requests, outcome status and duration of test executions, etc.

Rails Dataset

Prominent open-source project, written in Ruby, that contains over 35,000 builds on Travis CI Platform, collected over 5 months. The dataset includes information such as test suite ID's,outcome status and duration of test executions, job and build ID's, start and finish times, etc.

CCBP Algorithm

Continuous, Commit-Based Prioritization takes into consideration two events associated with commits: the moment a commit arrives for testing and the conclusion of the execution of the test suites associated with a commit. When a commit arrives, it is queued and if computer resources are available, the queue is prioritized and the highest ranked commit begins execution. Continuous Prioritization is applied by considering failing and execution history data. CCBD favours commits that contain a large percentage of test suites that have failed recently, and in the absence of failures, it favours commits with test suites that have not been executed recently.

Pros

- Lightweight, cost-effective and operates quickly.
- Scalable approach (as teams grow, the algorithm progressively saves more time and resources).
- Can be combined with TCP and Machine Learning techniques.
- Already proven to obtain promising results, when applied to Google and Rails datasets.
- Easily tunable to adapt to different CI systems.

Cons

- Assumes commits are independent.
- Produces better results with concurrent changes.
- Findings may not be extendable to slower-paced CI systems.
- Commit "starvation" - arrival of new commits may overshadow the execution of an earlier one.

Weighted APFD

$$APFD_C = \frac{\sum_{i=1}^{m}(\sum_{j=TF_i}^{n} t_j - \frac{1}{2}t_{TF_i})}{\sum_{j=1}^{n} t_j \times m}$$

Originally, the APFD metric, considers all tests to have the same cost, in terms of running time. So it has become important to improve this metric, by also taking into account test case size. In the formula above, *t* represents the test case cost, *TFi* the first test case that reveals the *i_th* fault, in a set of *m* failures.

Machine Learning (ML)

ML techniques can be used to prioritise commitQ, based on past information (not only execution and failing history). The point is to rank the commits from more to less likely to fail. (*inter-commit prioritization)*

This approach can be conjugated with Test Case Prioritization, where ML can pinpoint which combination of test cases should be applied first, in order to maximize fault detection (*intra-commit prioritization)*

Detailed View - CCBD

Algorithm 1: CCBP: Prioritizing Commits

```
parameter failWindowSize
parameter exeWindowSize
resources
queue commitQ

Procedure onCommitArrival(commit)
    commitQ.add(commit)
    if resources.available() then
        commitQ.prioritize()
    end

Procedure onCommitTestEnding()
    resources.release()
    if commitQ.notEmpty() then
        commitQ.prioritize()
    end

Procedure commitQ.prioritize()
    for all commit_i in commitQ do
        commit_i.updateCommitInformation()
    end
    commitQ.sortBy(failRatio, exeRatio)
    commit = commitQ.remove()
    resources.allocate(commit)

Procedure commit.updateCommitInformation(commit)
    failCounter = exeCounter = numTests = 0
    for all test_i in commit do
        numTests.increment();
        if commitsSinceLastFailure(test_i) ≤ failWindowSize then
            failCounter.increment()
        end
        if commitsSinceLastExecution(test_i) > exeWindowSize then
            exeCounter.increment()
        end
    end
    commit.failRatio = failCounter / numTests
    commit.exeRatio = exeCounter / numTests
```

Above we can see a detailed description of what the algorithm does. Commit arrival and completion invoke the respective procedures, *onCommitArrival()* and *OnCommitTestEnding()*. Procedure *prioritize()* updates the commits stacked into commitQ and then sorts them by looking at the failRatio and exeRatio parameters.

Finally, *updateCommitInformation()* stores information about commit history, that can be bounded by failWindowSize and exeWindowSize parameters. If a test suite has failed within failWindowSize, the failure counter is incremented and analogously for test suits that have not been executed within exeWindowSize.