Test Minimization, Selection, and Prioritization

Arnav Gupta

April 21, 2024

Contents

1 Motivation

Software testing is expensive and can have too many tests to run. Most tests don't fail.

2 Minimization

Given a test suite T, a set of requirements $\{r_1, \dots, r_n\}$ that must be satisfied to provide the desired adequate testing of the program, and subsets of T $\{T_1, \dots, T_n\}$ associated with each r_i such that any test case t_j belonging to T_i can be used to achieve requirement r_i.

The problem is to find a representative set of test cases t_j that will satisfy all r_i.

Finding the minimal set of test cases is a minimal hitting set problem. The set cover problem is to identify the smallest sub-collection whose union is the universe.

If r_1 can be satisfied by only 1 test case, the test case is **essential**. If a test case satisfies only a subset of the test requirements satisfied by another test case, it is redundant.

GE Heuristic:

1. select all essential test cases in the test suite
2. for the remaining test requirements, use the additional greedy algorithm
    (a) select the test case that satisfies the max number of unsatisfied test requirements

GRE Heuristic:

1. remove all redundant test cases in the test suite, which may make some test cases essential
2. perform the GE heuristic on the reduced test suite

Hitting Set Heuristic:

1. all test cases that occur in single element T_i are first included in the hitting set and all T_i containing any of these elements are marked
2. all unmarked T_i with 2 elements are considered, the test case that occurs in the maximum number of T_i of two elements is chosen and included in the hitting set
3. all unmarked T_i with 3 elements are considered, only test cases involved in a tie from earlier are considered here
4. back to step 2

2.1 Data Flow Graph

To minimize tests with data flow graph, get du pairs and associated testing sets:

- select essential tests
- select remaining tests that might be needed

3 Test Selection

While test case selection reduces size of a test suite, most selection techniques are **modification-aware**, meaning selection is not only temporary but also focused on modified parts of the program.

Test cases are selected because they are relevant to changed parts of the system under test, which involves a white-box static analysis of the program code.

3.1 Regression Testing

Given a program P, its modified version P', and the test set T used to test P, find a way using T to gain sufficient confidence in the correctness of P'.

Steps for regression testing:

1. identify the modifications made to P
2. select $T' \subseteq T$, the set of tests to re-execute on P', which may use the identified modifications
3. retest P' with T', establishing the correctness of P' with respect to T'
4. if necessary, create new tests for P'
5. create a new complete test set for P', including tests from steps 2 and 4, and old tests that are not selected, provided that they remain valid

To reduce the time involved in re-testing, choose only tests from T that produce different output when run on P', called modification-revealing tests.

A test T_i is **modification-revealing** if it produces different outputs in P and P'.

Cannot decide if a test is modification-revealing, but can identify the necessary condition. For a test to produce different output in P and P', it must execute some code modified from P to P'.

A test T_i is modification-traversing if it executes a new or modified statement in P', or misses a statement in P' that is executed in P.

Modification-revealing tests are necessarily modification-traversing, but the other way is not necessary.

3.2 Evaluation Criteria

Criteria are:

- inclusiveness
  - if T contains n modification-revealing tests and S selects m of those tests, the inclusiveness of S is m/n
  - if a method S always selects all modification-revealing tests, S is safe
- precision
  - measure the extent to which a selective strategy omits tests that are non-modification-revealing
  - if T contains n non-modification-revealing tests and S omits m of those tests, precision of S is m/n

To identify fault-revealing test cases for P', find the modification-revealing test cases for P and P' using:

- **P-Correct-For-T assumption**: assume that for each test $t \in T$, when P was tested with t, P halted and produced the correct output
- **Obsolete-Test-Identification assumption**: assume that there exists an effective procedure for determining, for each test $t \in T'$, whether t is obsolete for P'

A modification-revealing test must also be fault-revealing.

Controlled-Regression-Testing assumption: when P' is tested with t, all factors that might influence the output of P', except the code in P', are kept constant with respect to their states when P was tested with t

Given that the Controlled-Regression-Testing assumption holds, a non-obsolete test case t can thereby be modification-revealing only if it is also modification-traversing for P and P'.

The all 3 assumptions hold $T_{fr} = T_{mr} \subseteq T_{mt} \subseteq T$.

3.3 Slicing

Not every statement that is executed under a test case has an effect on the program output of the test case (RIP).

Slicing: a program slice consisting of all statements, including conditions in the program that might affect the value of variable V and point P

Backward slices S(v,n): refer to statement fragments that contribute to the value of v and statement n, where statement n is a use node of variable v

Forward slices S(v,n): refer to all program statements that are affected by the value of v and statement n, referring to the predicate uses and computation uses of the variable v

Slices constructed can be very large.

Dynamic slice is constructed with respect to the traditional static slicing criterion together with dynamic nfo (input sequence to program during specific execution).

Dynamic slicing criteria are:

1. variables to be slices (same as static slicing)
2. point of interest in the program (same as static slicing)
3. sequence of input values for which the program was executed

2 main properties desirable in slicing algorithms are **soundness** and **completeness**.

To be **sound** an algorithm must never delete a statement from the original program which could have an effect upon the slicing criterion. This allows analyzing a slice in total confidence that is contains all statements relevant to the criteria.

To be **complete** an algorithm must remove all statements which cannot affect the slicing criterion. Completeness is unachievable due to undecidability.

The goal of good slicing algorithms is to delete as many statements as possible without giving up soundness. The closer an algorithm approximates completeness, the most precise the slices it constructs will be.

4 Prioritization

Given a test suite T, the set of permutations of T PT, and a function from PT to real numbers, $f : PT \to R$.

The problem is to find $T' \in PT$ such that for all T'' such that $T'' \in PT$ and $T'' \neq T'$, $f(T') \geq f(T'')$.

Prioritization concerns ordering test cases for early maximization of some desirable properties, such as rate of fault detection. Seeks to find the optimal permutation of the sequence of test cases, and does not involve selection, assuming all test cases may be executed in the order of the permutation it produces, but that testing may be terminated at some arbitrary point during the testing process.

Coverage is a metric often used as the prioritization criteria since early maximization of structural coverage will also increase the change of early maximization of fault detection. Prioritization seeks to achieve higher fault detection rate, but actual aim is to maximize early coverage. Can be done with random, hill climbing, genetic, greedy, etc.

4.1 Test history

Can have historical failure rates for tests, after each test run, there is more info, such as co-failure rate (given one test failed, get failure probability of queued tests).

Reorder tests based on cofailure rates.

In reality, too many test requests to globally reorder. Conditional probability to reorder tests for a single change, and use a scoring function to globally order tests. Scoring function is

$$new_s c = prev_s c + (P(t_2 = \text{fail} \mid t_1 = \text{fail}) - 0.5)$$

where $prev_s c$ is adjusting the existing score, the probability is given that the previous test failed, the probability that the next one will fail, and reduce the score if the relationship is < 0.5.

Some tests may starve during prioritization. The dispatch queue is used for prioritization. If it is empty, tests from waiting queue go to dispatch queue, in original order, otherwise if it is smaller than a threshold, tests in the head of the waiting queue will be picked and put in the dispatch queue in a random order.

5 Flaky Tests

A flaky tests passes and fails on the same build. This could be fine if tests have inherent non-determinism (hardware/environmental, asynchronous, AI based). Could also be broken and expensive to fix.

At Facebook, they ran failed tests up to 10 times to find if flaky. At Google, ran tests 3 times and only report fault if it fails 3 times in a row. These can be expensive. At Microsoft, run test 1000 times and check if below a flaky threshold.

To quantify flaky tests:

1. measure of the degree of test flakiness, where flake rate is flakes/runs
2. establish the flask rate baseline on stable build or release
3. how flaky are tests?
    (a) number of runs to have statically confident stable flake rate

Flaky test outcomes include true positive and true negative (fail means fault, pass means no fault), but also false positive (fail but no fault) and false negative (pass but fault slips through).

Accuracy of a test is $(TP + TN)$/runs. Flake rate of a test is $(FP + FN)/\text{runs} = 1 - \text{accuracy}$. Can measure flake rate over time.

A stable build is one that has been successfully running in production. A stable build is needed so that any test failure is a flaky failure, and any test pass is a good pass.

Stable accuracy is TN/runs or passes over runs. Stable flake rate is FP/runs or fails over runs.

For 99% confidence in flake rate, have 666 runs. For 99.9% confidence in flake rate, have 1083 runs. Run a test over 1000 times on a stable build. Use

$$n = \frac{Z^2 p(1-p)}{e^2}$$

with n being the number of reruns, p being the pass rate, and Z being

To use flaky test failure rate to prioritize tests:

1. likelihood of change in stable state (binomial distribution)
2. prioritizing re-runs to find instabilities (probability of a set of runs)

To get the likelihood of getting the set of test results from re-runs, use \$$\mathrm{P}_\mathrm{t}$(f, r, flakerate) $= \mathrm{r}\left(_{f*\mathrm{flakerate}^{\mathrm{f}}*(1-\mathrm{flakerate})^{\mathrm{r\text{-}f}} withtheprobablityofgettingf}\right)$ failures with r test runs. Flake rate is calculated from a stable build and estimated by test failure rate. Determine if the rate has changed at statistically significant levels.

Tests no longer have a binary outcome, but become unstable relative to the baseline build flake rate (test failure rate).

Signal is when the test starts failing more than expected, which means someone likely changed code and introduced a fault.

Noise is when the test fails at expected levels, so just normal interference (async/environmental).

To prioritize re-runs, use P_t. Re-run tests in order by P_t ascending. Stop when the result is statistically significant or no more re-runs are available. Will quickly find the tests that have become unstable.

An algorithm to find changes in stable test failure rate:

1. run each test once
2. calculate P_t
3. order tests by P_t
4. run the test with the lowest P_t
5. investigate tests that show highly unlikely fail rates (are unstable)
6. go to step 5 until no more test runs budgeted or high confidence reached (fail rate stable and behaviour stable)

Without flaky tests:

- indicator of something bad (bug) is test fail
- confidence of test results is irrelevant, just run once
- prioritization is which test is more likely to fail

With flaky tests:

- indicator of something bad (bug) is ratio between pass/fail changing
- confidence of test results is achieved from running multiple times, either to reach a budget or achieve statistical confidence

- prioritization is which tests has the most unlikely results

After finding all new pass/fail distributions for each test, compare with history. If it has significant deviance (using stats binomial test), this is an indicator of something bad, otherwise don't do anything.

Tests are not binary (anomaly detection) so check change in stable flake rate. As long as behaviour of system is stable, don't fix flaky test (still get signal from the test by re-running and examining P_t). Making a test less flaky means less noise, so failure becomes more unlikely and requires fewer re-runs. Tradeoff cost to fix test and cost to re-run and calculate P_t.