Università degli Studi di Salerno
Dipartimento di Informatica

Tesi di Laurea di I livello in
Informatica

Template tesi ISISLab

Relatore
Nome Cognome
Correlatore
Dott. Nome Cognome

Candidato
Nome Cognome

Academic Year 2021-2022

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Contents

Chapter 1

Introduction

Things to add:

- where different types of coverage are useful
- Add formulas for coverage criteria

Coverage testing

Coverage is one of the metrics employed during testing to asses what portion of the source code is "covered" by the test suite i.e., what portion of the code is executed when the tests run. Coverage is essential to extract information about the general quality of a test suite and helps determine how comprehensively the software is being verified. As a result, coverage can be classified as a white-box testing technique.

Source code coverage can be expressed according to different sub-metrics:

- Statement coverage aims executing every single statement in the code.
- Branch coverage, also known as decision coverage, measures how many decision structure have been fully explored by the test cases.
- Mutation coverage, also known as fault-based testing, aims at purposefully introducing faults in the program in order to check whether the test suite is able to identify them. If the fault is correctly detected, the mutant is "killed". One issue of mutation is scalability, since generating and compiling the mutants, before running the test cases, can be time-consuming end quickly exhaust testing resources. Additionally, the introduced mutations can be classified as weak or strong: with strong

mutation, the artificial fault is propagated to an observable behavior, while weak mutants are confined to more specific environments.

- Function coverage measures how many functions have been called by the test cases.
- Condition coverage determines the number of boolean conditions/expressions executed in the conditional statements.

To reach statement coverage, it is sufficient to execute a branch in which the statement is control dependent.

A high coverage can sometimes be deceiving, however: in the case of Machine Learning Systems (MLS), where typically the source code is made up of a sequence of library functions and API invocations [1], thus resulting in very high statement and branch coverage with relatively modest test suites. Additionally, the effectiveness of such systems is highly determined by the dataset employed for model training and validation, which cannot be covered by tradition test cases.

Coverage can also be measured at any testing levels; while at the unit test-level we focus mostly on the coverage of statements and branches, at the system-testing level, the coverage targets shift towards more complex elements, such as menu items, business transactions or other operations that require multiple components of the system to work properly.

A coverage criterion is A coverage goal is a particular target that we want to cover in respect to the chosen criterion, i.e. a particular branch of an if statement.

Automatic test case generation

Test case generation can be seen as a multi-objective problem, given that the goal is to cfover multiple test targets.

Search-based approaches for test case generation use optimization algorithms to attempt to find the best candidate test case with the objective to maximize fault detection. Genetic Algorithms (GAs) are an example of an evolutionary search approach for test case generation; starting from an initial, often randomlly genrated, population of test cases, the algorithm keeps evolving the individuals according to simulated natural evolution theory principles. In this context, a typical fitness function of a GA would measure the distance between the execution trace of the generated test cases and the coverage targets.

Testing in the Internet of Things

Chapter 2

Literature

Problem formulation

The simplest way of approaching the evolutionary search problem for test case generation is by iteratively determining a coverage goal in the source code (i.e. a particular branch), and executing the GA to find a test case that achieves this coverage. A strategy for iterative search typically includes:

- Enumerating the targets to cover, i.e. the individual branches.
- Performing the single-objective search for each target, until all the targets are covered, or the budget has expired.
- Building the final test suite by combining all the generated test cases.

Formal problem formulation

Focusing on one coverage goal at a time is ultimately a poor strategy, however. Foremost, a search algorithms could get "trapped" while attempting to cover an expensive branch and waste a large portion of the testing budget [2]. Secondly, the order by which coverage targets are selected may end up largely impacting the final performance. Additionally, this approach assumes that all coverage goal are equally important and independent of each other; this is often not the case as, for example, covering the true branch of an if statement may be easier than covering the true branch of another if statement that requires a complex chain of operations to be properly satisfied. Finally, covering a particular branch may have collateral coverage over other branches in the test case's path.

These issues suggest that multi-target evolutionary approaches may reveal more effective and reliable. These approaches are known as whole test suites approaches and their goal is to evolve the entire test suite simultaneously, rather than iteratively covering the single branches/statements. [3]

Specifically, the fitness function is mainly based on two measures: approach level and branch distance. The approach level represents how far is the execution path of a given test case from covering the target branch, while the branch distance represents how far is the input data from changing the boolean value of the condition of the decision node nearest to the target branch. As the branch distance value could be arbitrarily greater than the approach level, it is common to normalize the value of the branch distance.

State of the art

EvoSuite is an example of an evolutionary algorithm that optimizes the whole test suite towards just one coverage criterion, rather than generating test cases directed towards multiple coverage criteria. With EvoSuite, any collateral coverage isn't a concern since all coverage is intentional, given that the ultimate goal is to generate the whole test suite. The algorithm starts with a randomly generated population of test suites. The fitness function rewards better coverage of the source code; if two suites have the same coverage, the one with fewer statements is chosen. For each test suite, its fitness is measured by executing all of its test cases and keeping track of the executed methods and of the minimal branch distance for each branch.

Expand on bloat in EvoSuite

Another popular algorithm for multi-target search problems is the Non-dominated Sorting Genetic Algorithm II (NSGA-II). This algorithm is based on three principles:

- It uses elitism when evolving the population: the most fit individuals are carried over along the offsprings.
- It uses an explicit diversity-preserving mechanism, the Crowding distance.
- It emphasizes the non-dominated solutions, as its name suggests.

First of all, in the context of test cases, domination can be expressed by the following relation:

The NSGA-II algorithm works as follows:

Definition 1: *A test case x dominates another test case y (also written $x \prec y$) if and only if the values of the objective function vector satisfy the following conditions:*

$$\forall i \in \{1, \dots, k\} \quad f_i(x) \le f_i(y)$$
$$\text{and}$$
$$\exists j \in \{1, \dots, k\} \text{ such that } f_j(x) < f_j(y)$$

Figure 2.1: Test case domination

- Starting from an initial population of individuals Pt, generate an offspring population Qt of equal size and merge the two together, obtaining the population Rt.
- Perform non-dominated sorting of the individuals in Rt based on target indicators and classify them by fronts, i.e. they are sorted according to an ascending level of non-domination. This ensures that the top Pareto-optimal individuals will survive to the next generation.
- If one of the fronts in the sorted sequence doesn't fit in terms of population size, crowding distance sorting is performed.
- Create the new population based on crowded tournament selection, then perform crossover and mutation.

Figure 2.2 summarizes the main loop of the algorithm:

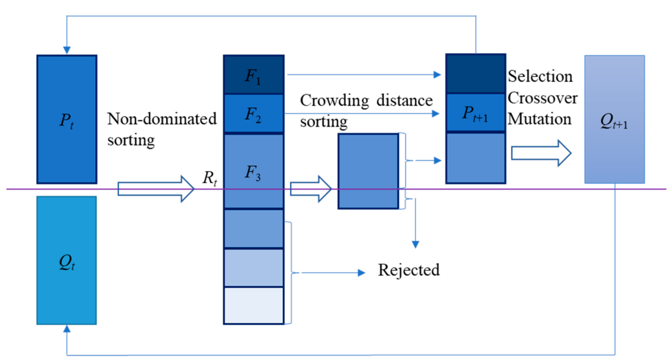

Figure 2.2: NSGA-II algorithm main loop

In the context of software engineering, NSGA-II has been applied to problems such as software refactoring and test case prioritization, with two or three objectives. If the number of objectives begins to grow, however, the performance of the algorithm doesn't scale up well [4]. To overcome these limitation,

MOSA...

DynaMOSA, Dynamic Many-Objective Sorting Algorithm [5] is an approach that focuses on ..., and has been developed as an evolution of MOSA. This latter solution implements a many-objective GA to tackle test case generation and has three main features:

- instead of ranking candidates for selection based on their Pareto optimality, it uses a preference criterion. This criterion selects the test case with the lowest objective score for each uncovered target; these selected individuals are given a higher chance of survival, while other test cases are ranked with the traditional NSGA-II approach.

- The search is focused only on the uncovered coverage targets.

- All tests that satisfy one or more of the uncovered targets will be archived and used as the final test suite once the search ends.

In many-objective optimization problems, candidate solutions are typically evaluated in terms of Pareto dominance and Pareto optimality.

DynaMOSA has been employed with Java classes.

Optimal Coverage sEarch-based tooL for sOftware Testing, OCELOT [6] is a test case generation tool for C programs that implements both a multi-target approach based on MOSA, and new iterative single-target appproach named LIPS, Linear Independent Path-based Search.

Tested with MOSA but not with DynaMOSA.

Chapter 3

Conclusions

Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Bibliography

[1] Vincenzo Riccio et al. "Testing machine learning based systems: a systematic mapping". In: vol. 25. 6. 2020, pp. 5193–5254. DOI: 10.1007/s10664-020-09881-0. URL: https://doi.org/10.1007/s10664-020-09881-0.

[2] Gordon Fraser and Andrea Arcuri. "Whole Test Suite Generation". In: vol. 39. 2. 2013, pp. 276–291. DOI: 10.1109/TSE.2012.14. URL: https://doi.org/10.1109/TSE.2012.14.

[3] Gordon Fraser and Andrea Arcuri. "Evolutionary Generation of Whole Test Suites". In: *International Conference On Quality Software (QSIC)*. Los Alamitos, CA, USA: IEEE Computer Society, 2011, pp. 31–40. DOI: http://doi.ieeecomputersociety.org/10.1109/QSIC.2011.19.

[4] Bingdong Li et al. "Many-Objective Evolutionary Algorithms: A Survey". In: vol. 48. 1. 2015, 13:1–13:35. DOI: 10.1145/2792984. URL: https://doi.org/10.1145/2792984.

[5] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. "Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets". In: vol. 44. 2. 2018, pp. 122–158. DOI: 10.1109/TSE.2017.2663435. URL: https://doi.org/10.1109/TSE.2017.2663435.

[6] Simone Scalabrino et al. "Search-Based Testing of Procedural Programs: Iterative Single-Target or Multi-target Approach?" In: *Search Based Software Engineering - 8th International Symposium, SSBSE 2016, Raleigh, NC, USA, October 8-10, 2016, Proceedings*. Ed. by Federica Sarro and Kalyanmoy Deb. Vol. 9962. Lecture Notes in Computer Science. 2016, pp. 64–79. DOI: 10.1007/978-3-319-47106-8_5. URL: https://doi.org/10.1007/978-3-319-47106-8%5C_5.

[7] A. Author and A. Author. *Book reference example*. Publisher, 2099.

[8] A. Author. "Article title". In: *Journal name* (2099).

[9] *Example*. URL: `https://www.isislab.it`.

[10] A. Author. “Tesi di esempio ISISLab”. 2099.