Unit test generation for common and uncommon behaviors

Master's Thesis

Björn Evers

Unit test generation for common and uncommon behaviors

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Björn Evers
born in Vlaardingen, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Unit test generation for common and uncommon behaviors

Author: Björn Evers
Student id: 4205340
Email: b.evers@student.tudelft.nl

Abstract

Various search-based test generation techniques have been proposed to automate the process of test generation to fulfill different criteria (e.g., line coverage, branch coverage, mutation score, etc.). Despite these techniques' undeniable accomplishments, they still suffer from a lack of guidance coming from the data gathered from the production phase, which makes the generation of complex test cases harder for the search process. Hence, previous studies introduced many strategies (such as dynamic symbolic execution or seeding) to address this issue. However, the test cases created by these techniques cannot assure the full coverage of the execution paths in software under test. Therefore, this thesis introduces common and uncommon behavior test generation (CUBTG) for search-based unit test generation. CUBTG uses the concept of commonality score, which is a measure of how close an execution path of a generated test case is from reproducing the same common and uncommon execution patterns observed during the real-world usage of the software.

To evaluate the performance of CUBTG, we implemented it in EvoSuite and evaluated it on 150 classes from JabRef, an open-source application for managing bibliography references. We found that CUBTG managed to cover more common behaviors than plain the many-objective sorting algorithm (MOSA) in 75% of the cases, and more uncommon behaviors in 60% of the cases. In up to 10% of the cases CUBTG managed to find more mutants seeded by PIT by using method sequences that plain MOSA did not find.

Thesis Committee:

Chair:	Prof. Dr. A. Zaidman, Faculty EEMCS, TU Delft
University supervisor:	Prof. Dr. A. Zaidman, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. Panichella, Faculty EEMCS, TU Delft
	Dr. C. Lofi, Faculty EEMCS, TU Delft
	Dr. X.D.M. Devroey, Faculty EEMCS, TU Delft
	P. Derakhshanfar, MSc, Faculty EEMCS, TU Delft

Preface

This thesis represents the final part of my education at TU Delft. I have learned a lot here throughout the years. Looking back, the version of myself that started his Bachelor's programme here years ago would be quite content with the knowledge and skills I have gained, and I am thankful to all people who have played a part in that.

In relation to this final project, for their help and for their valuable advice during the meetings we had, I would like to thank Andy Zaidman, Xavier Devroey and Pouria Derakhshanfar.

For always being there for me, and for helping me practically always when I ask for it, also during this final project, I would like to thank my father Eric, my mother Anneke, my brother Niels, and my grandmother Annie.

Even though one always keeps learning, I am now closing the education chapter of my life. I am glad this chapter was part of it.

Björn Evers
Vlaardingen, the Netherlands
May 31, 2020

Contents

List of Figures

Chapter 1

Introduction

Software testing has been an important part of developing software for almost as long as software has been around. At first, this was mostly a manual effort, but automated software testing soon became a topic of research, for a large part because of the time investment that is needed to write tests [9]. Several methods for automatic testing have been devised, including methods for automatically generating a set of test cases based on the existing code base.

Most of these test generation methods concern themselves with obtaining high coverage values in certain coverage methods based solely on the source code, like number of source code lines, or number of code branches covered. Not much research has been done on creating test cases based on actual use of the system in practice, which may result in a more realistic test suite, and might uncover errors that are not found by test suites generated based on traditional coverage metrics.

To contribute to this relatively unexplored field, in this thesis we present a method for guiding unit test generation towards generating tests for common behaviors in a software unit (e.g., a class), or uncommon behaviors in the unit, which we call common and uncommon behavior test generation (CUBTG). CUBTG is designed primarily to work with the many-objective sorting algorithm (MOSA) [20], but may be adapted in the future to work with other algorithms. MOSA is a genetic algorithm for generating unit tests, and searches for solutions to multiple coverage goals (e.g., branches, lines, exceptions) at once, guided by a fitness function (FF) for each coverage goal, and a secondary objective (SO) to decide which test case to keep when there are multiple test cases covering the same goal.

CUBTG uses the concept of *commonality score*, which we define in this thesis. In short, the commonality score describes to what extent a test exercises code branches that are executed a lot during normal usage of the software under test (SUT), using log data from production runs of the SUT. Based on this commonality score, we define FFs and SOs to be used with MOSA, which favor either common execution paths or uncommon execution paths during test generation, producing test cases and test suites that exercise more common, or more uncommon behaviors in the SUT.

These newly devised FFs and SOs influence the search for solutions to standard coverage goals in MOSA. The FFs attempt to influence the commonality score of test cases while they are being generated to cover a standard coverage goal throughout the iterations

of MOSA. The aim of the SOs is to keep the test with the highest or lowest commonality score when multiple test cases are found for the same coverage goal. We hypothesize that using these new FFs and SOs can lead to a better, or more specific guidance of the search process, which can lead to test cases covering existing coverage goals in different ways, increasing coverage.

1.1 Research questions and contributions

We evaluated the performance of CUBTG on 150 classes of the open-source application JabRef[1], using an implementation in EvoSuite [11], an application for generating unit tests using genetic algorithms, and answered the following research questions:

RQ1 Do tests generated by CUBTG achieve a better commonality score compared to standard MOSA, and how do the different CUBTG methods compare?

RQ2 How do software faults revealed by CUBTG differ from standard MOSA, and how do the different CUBTG methods compare?

RQ3 How do tests generated by CUBTG compare to standard MOSA in terms of standard code coverage metrics, and how do the different CUBTG methods compare?

RQ4 Does the usage of the added FFs and SOs affect the efficiency of the EvoSuite test generation?

RQ5 How much time of EvoSuite test generation does it take for commonality score to converge?

We found that CUBTG managed to cover more common behaviors than standard MOSA in 75% of the cases, and more uncommon behaviors in 60% of the cases. We also performed mutation testing on the generated test suites, and found that CUBTG performed the same or worse than standard MOSA in most cases. There were a few exceptions in which CUBTG managed to find some mutants by using method sequences that standard MOSA did not find.

CUBTG generally performs the same or a little bit worse in terms of standard coverage metrics, causes EvoSuite to go through less generations, causes test suite sizes to be smaller, and causes longer test cases to be generated. There does not appear to be a significant effect on the efficiency of EvoSuite test generation, but at least 100 seconds were needed for the commonality score to converge for the most part during our evaluation.

In summary, this thesis makes the following contributions to the field:

- The novel CUBTG method for test generation, which uses newly devised FFs and SOs and the concept of commonality score to influence the unit test generation process in MOSA towards generating tests that exercise more common or uncommon behaviors.

[1] https://www.jabref.org/

- An evaluation of this CUBTG method on the JabRef application, evaluating its effect on the commonality score, fault revealing capability, and standard coverage metrics obtained by generated test cases, along with an evaluation of the effect of CUBTG on the efficiency of EvoSuite and the efficiency of CUBTG itself.

1.2 Thesis structure

The remainder of this thesis is structured as follows. Chapter 2 will describe background and related work relevant for the rest of this thesis. In Chapter 3, the concept of commonality of test cases and test suites is defined, and based on that CUBTG is introduced to generate tests based on execution counts of code branches from real world usage of a system. Chapter 4 describes how we evaluated the CUBTG methods on the JabRef open-source project, including among other things our research questions, how we obtained execution data, and what data we collected for analysis. The results of this evaluation, along with a discussion of those results and conclusions based on them, are presented in Chapter 5. Finally, we present our overall conclusion and note our suggestions for future work in Chapter 6.

Chapter 2

Background and related work

The quality of a piece of software is important for ensuring that it works as intended. Software testing is an important aspect of determining the quality of a piece of software. Its goal generally is to run a piece of software in an effort to simulate a real execution, and verify if the system behaves according to specification. In the early days of software testing, this was always done by hand, manually specifying the pieces of code to run, and the expected outcome. Writing these tests can be a tedious and time consuming task, and can even consume as much, or even more, time than developing the software that is being tested.

It comes to no surprise that eventually, a movement towards more automatic methods of software testing took place, and is still ongoing. In 1976, Jessop et al. published one of the earlier papers on automatic software testing, describing a system that verifies if a software system conforms to a formal model of how it should work [15]. Ramamoorthy et al. discuss software testing tools available at the time, including the less formal ones [24]. Moving to this century, we see the main rise of non-formal automatic testing tools. Several methods for automatic testing have been devised over the years, including fuzzing [35, 13], concolic testing [17, 30], and search-based testing [19, 25].

As a further background for this thesis, Section 2.1 discusses genetic algorithms, Section 2.2 discusses parts of MOSA relevant for this thesis, and finally Section 2.3 discusses usage-based test generation.

2.1 Genetic algorithms

As a subset of search-based software testing, genetic algorithms have been used widely for software testing [31]. Genetic algorithms use the basic principles of evolution of species in the real world to search for solutions (individuals) to a wide variety of problems that perform well according to some set of fitness metrics. In the case of test generation, the solutions are test cases or test suites, and the fitness metrics are some kind of measure of how well the tests test the software under test.

Like evolution of species in the real world, genetic algorithms use the processes of selection, crossover, and mutation to find incrementally better solution. The general process is shown in Figure 2.1. An initial population of some defined size is first generated, more

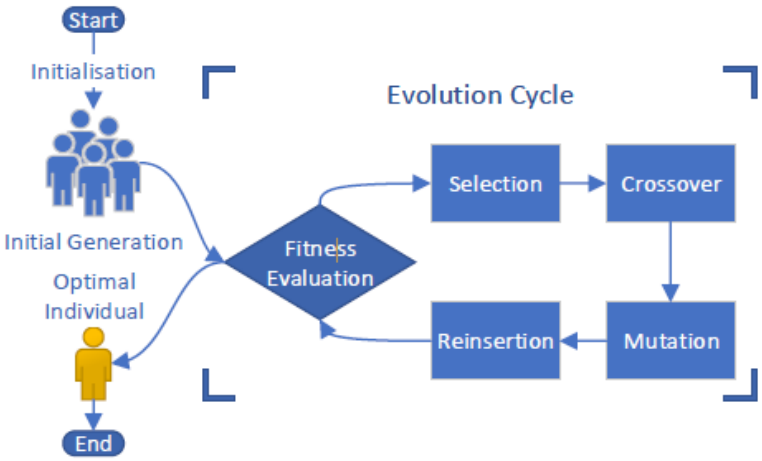

Figure 2.4: Overview of a Genetic Algorithm

Figure 2.1: Overview of the general steps in a genetic algorithm. Taken from [36].

or less randomly in most cases. Then an iterative process of evolution is started. For all individuals in the population, their fitness is calculated using one or more fitness functions. Using some selection procedure, a subset of the individuals is chosen to be kept, using their fitness values. Those individuals are then used in a crossover procedure to produce children. In this procedure, pairs of individuals are mixed to produce new individuals (analogous to producing children in the real world). In the case of test cases, the crossover operation might consist of combining the first half of the statements of one test case with the last half of the statements of the other, for example. Mutation is then randomly applied to some of the individuals, changing/adding/removing a random part of an individual in a random way. One might remove a statement from a test case, for example. The individuals are then reinserted into the population for the next iteration of evolution. This process continues until some constraint on the run time is met, for example an absolute amount of time, or until a satisfactory individual is found. One or more individuals are then outputted from the algorithm.

Genetic algorithms have been used in quite a few ways in software testing. Some recent examples are Qi et al., where a parallel genetic algorithm is implemented to perform pairwise testing [23], Rawat et al., where fuzzing is implemented using a genetic algorithm [26], and Arcuri, where a genetic algorithm is used to generate tests on the system level for web services [2, 3].

A fairly well-known application for software testing using genetic algorithms is EvoSuite [11]. It applies the methods as stated above. Originally, it used an algorithm which considers a whole test suite as an individual, instead of a separate test case. It uses coverage of the whole class under test (CUT) as a goal. This way the outcome does not depend on the order or differences in difficulty of satisfying separate goals (e.g., covering branches). However, it meant that the algorithm became less focused on specific goals that had to be

covered.

Recently developed search-based genetic algorithms for software testing are the many-objective sorting algorithm (MOSA) [20] and DynaMOSA [22], which are also incorporated in the EvoSuite application. These algorithms consider test cases as individuals, and incorporate separate fitness functions for separate coverage goals. Using a method similar to the non-dominated sorting genetic algorithm II (NSGA-II) [8], using non-dominated fronts, the algorithm tries to generate test cases in the direction of multiple coverage goals in parallel, within one evolution population. This made it possible to generate tests aiming to cover specific goals, while not letting the test generation be stuck on covering a single goal for a long time.

2.2 Test case selection in MOSA

To explain how we have incorporated the common and uncommon behavior test generation (CUBTG) methods in MOSA in Section 3.2.1, we will first touch upon the parts of MOSA that are most relevant for the discussion. As stated, MOSA is an evolutionary algorithm. It uses the steps typical for such an algorithm. It starts with a random population of individuals, selects individuals for reproduction based on their fitness values according to some fitness function (FF), performs crossover and mutation on those individuals, and creates the population for the next generation. This cycle continues until the search budget has run out. In the case of MOSA, the population consists of test cases, a subset of which is returned at the end of the generation as a test suite. This is in contrast with whole suite algorithms (as mentioned above), which consider a population of test suites, choosing the whole suite that performs best at the end.

For our purpose here, it is important to understand the selection step of MOSA. Suppose we have a population of test cases, and a set of coverage goals (e.g., branches, lines, exceptions). MOSA then categorizes test cases in sets that go from good to bad test cases, according to the coverage goals. It does this by categorizing them into so-called *non-dominated fronts*, in a way very similar to NSGA-II [8], but preceded by an extra categorization that makes the selection work much more effectively using the property that the population consist of test cases.

This first, extra step is to take for each goal that has not been covered yet the test case that covers it best, and putting them into a set. In case of a tie it will take the shortest test case. Then from the remaining test cases in the population, it creates non-dominated fronts of test cases. Each consecutive set of test cases only contains test cases that are not dominated by any of the other test cases, and each time the population that remains is considered until the maximum population size for the next iteration is reached. A test case dominates another test case if it is better in terms of all coverage goals. If the last front to be added to the next population cannot be fully added because of the maximum population size, the *crowding distance* is used to determine which test cases to add from that last front.

Example This system of categorizing the population of test cases can be more easily understood by looking at an example used in the paper introducing MOSA [20], which is

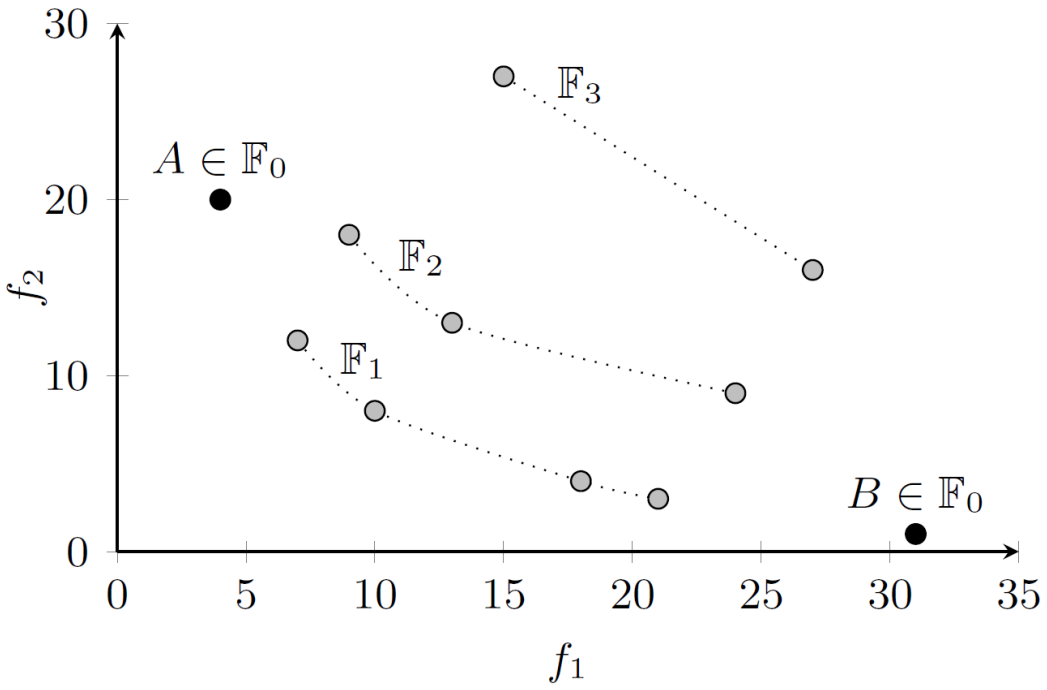

Figure 2.2: Categorization of test cases in non-dominated fronts for MOSA. Figure taken from the paper which introduced MOSA [20].

shown in Figure 2.2. In this example we have two coverage goals with corresponding FFs f_1 and f_2, where a goal has higher coverage if the fitness value is lower. Each of the dots represents a test case in the population. If we follow the selection procedure of MOSA, the test cases A and B will be put in the first front, $\mathbb{F}_0$, because they perform best in terms of f_1 and f_2, respectively. Then the test cases marked with $\mathbb{F}_1$ in the figure are added to the next front, because they are the remaining non-dominated test cases in the population. From the remaining test cases, $\mathbb{F}_2$ and $\mathbb{F}_3$ can be constructed. This procedure works the same for a typical scenario with tens or hundreds of code branches, but with a lot more dimensions.

The aspect of this selection procedure that is important for our purpose here, and that we will use to incorporate the CUBTG methods, is that each of the test generation goals (through their corresponding FFs) has an effect on the direction of the search of the algorithm. For each of the goals, a test case will be selected that performs best in terms of that goal, and possibly more test cases that perform well for that goal will be kept in the remaining non-dominated fronts. Those test cases will be used to generate new test cases in following iterations, which means that they have an effect on what the new test cases look like, also on test cases that are not directly related to covering that specific goal.

2.3 Usage-based test generation

In most automatic test generation approaches, the aim is to achieve high values for several coverage metrics. Often used metrics are the well-known line coverage, branch coverage, or more recently mutation coverage, which simulates bugs in software to evaluate how good a test suite is at finding real faults in a piece of software. These metrics do not take into account how the execution patterns of the generated tests compare with the way software is being executed in production use. Wang et al. looked into this and found that developer written tests as well as automatically generated tests do not represent typical execution patterns during execution that well [34].

One could argue that creating tests that reflect (or on the opposite, not at all reflect) the behavior of actual users reveal faults in software that would otherwise have been left uncovered, because of the use of certain method call sequences, for example. Additionally, code that is not often used in practice may be left relatively untested because it is rarely exercised in production. Recently, a method based on symbolic execution has been devised to recreate behavior of users using log data from a system run in production [33], which allowed finding the same faults in a system that are encountered by a user.

Generating tests based on actual usage of a system is what this thesis aims to expand upon. As opposed to Wang et al. [33], where the aim is to more or less exactly replicate a full behavior executed by a user using symbolic execution, the method in this thesis aims to only guide the search of a genetic algorithm towards executing or not executing certain branches. Like [33], log data is used to determine the execution counts of code branches. The method in this thesis can also be used to guide the search for test cases away from user behaviors, as opposed to guiding the search towards it.

Chapter 3

Defining commonality and using it in an evolutionary algorithm

This chapter defines the concept of *commonality* for test cases and test suites, and describes how to use it in an evolutionary test generation algorithm. The former is discussed in Section 3.1, and the latter in Section 3.2.

3.1 Defining commonality

Intuitively, commonality describes to what extent a test exercises code branches that are executed a lot during normal usage of the software under test (SUT). If a test executes a lot of branches that are used a lot in practice, compared to branches that are executed less in practice, it would score high in terms of commonality. If, on the other hand, it executes a lot of branches that are executed very sparsely in practice, and not many branches that are executed commonly, it would score low in terms of commonality.

More specifically, for one test case, commonality is defined as the average of how commonly a branch is executed over all branches covered by the test case. And in case of a test suite, commonality is defined as the average commonality over all its test cases. Note that defining the commonality in this way means that multiple executions within the test suite are taken into account for the score.

In the remaining of this section, the following will be discussed. First, in Section 3.1.1, the structure of the SUT as relevant for the discussion here will be described, along with an example which will be used throughout the chapter. Then test cases and test suites along with a definition of their commonality will be defined in Section 3.1.2, along with an example of computing the commonality score.

3.1.1 Structure of the SUT

Since the methods presented in this thesis generate unit test cases, which exercise a single class under test (CUT), we are only looking at the commonly and uncommonly executed code branches within one target CUT at the same time. Let us define what a CUT consists of. Depending on the programming language, classes could contain several constructs like

fields, methods, constructors, etc. For the discussion here, it is assumed that a class is just a set of concrete (implemented) methods that can be executed separately by a test case. For each of these methods, we can construct a control flow graph (CFG). CFGs were first introduced by Allen [1]. We will describe the relevant parts here in short.

A CFG is a directed graph in which the nodes represent *basic blocks*, and in which the edges represent control flow paths. Basic blocks are pieces of code that are always executed together. They will be called branches within this thesis most of the time. Suppose a branch B is a list of statements, and that $E(s)$ means that statement s is executed in some run of the code represented by the CFG. Then the following holds, which intuitively says that if one statement s in the branch B is executed ($E(s)$), they are all executed, and if one is not executed, none of them are executed.

$$\forall B\colon (\forall s \in B\colon (E(s) \iff \forall t \in B \setminus \{s\}\colon E(t))) \tag{3.1}$$

Note that this does only hold strictly in theory, because in practice there could be circumstances that cause the program to stop running mid-branch, like system crashes, program exceptions, etc.

Branches can have multiple incoming edges and multiple outgoing edges. There is always exactly one entry point in the CFG without any incoming edges, and one exit point without any outgoing edges. It is possible for a branch to contain zero statements.

Example CFG To illustrate what a CFG looks like and how it is used with respect to the commonality score defined in this chapter, let us look at some example pseudo code along with its corresponding CFG. The (very abstract) code for an example method is shown in Algorithm 1. Each `statement<x>` represents a single statement, and each branch in the code has been numbered using a comment at the start of the branch. The corresponding CFG is shown in Figure 3.1. In this figure, each node B_x corresponds to branch x in the example method. So for example, `Branch 3` in the pseudocode contains statements 3, 4 and 5, and corresponds to node B_3 in the CFG. The branch can only be reached from branch 1 (the root), if `condition1` is not met. Control flow can continue to either branch 4 or 5, depending on whether `condition2` is met or not. The other nodes and edges connected to them in the CFG can be explained in a similar manner.

3.1.2 Formal definition of commonality score

For the purpose of computing the commonality score of a test case or test suite, it is necessary to quantify how common the execution of a branch is during an execution of the CUT. The method described in this thesis uses the *execution weight* of a branch to do that, which is derived from the *execution count*. They are defined as follows.

Definition 1 *The* execution count *of a branch is the number of times a branch has been executed during some usage session.*

Definition 2 *The* execution weight *of a branch is an integer number specifying the number of times a branch has been executed relatively according to some set of execution data,*

Algorithm 1: Example method

```
/* Branch 1                                    */
if condition1 then
    /* Branch 2                                */
    statement1;
    statement2;
else
    /* Branch 3                                */
    statement3;
    statement4;
    statement5;
    if condition2 then
        /* Branch 4                            */
        statement6;
    /* Branch 5                                */
    statement7;
/* Branch 6                                    */
statement8;
```

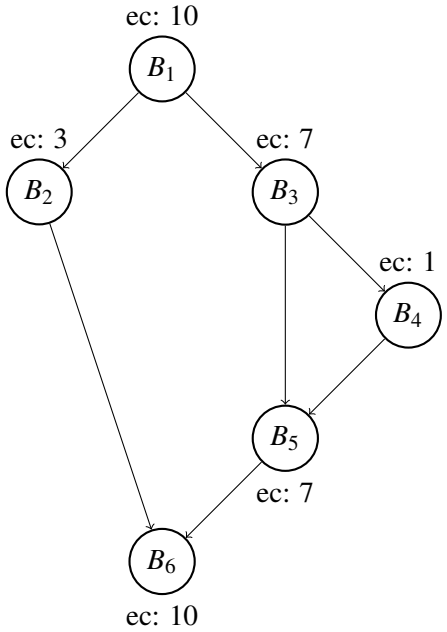

Figure 3.1: Example CFG. Each node indicates its execution count x in the form of a label $ec : x$

compared to other branches in the same CUT. The execution weight can be the same as, or can be derived from, the execution count defined in Definition 1.

For example, if one branch has an execution weight of 2, and another branch has an execution weight of 1, it means that the first branch is executed twice as often in a typical execution scenario. These execution weights can represents actual execution counts originating from one or more specific usage sessions, but this is not required. For example, one might want to scale down the execution counts that have been collected during an execution to keep them within reasonable size, which makes them easier to understand and possibly take up less space. One might replace counts for two branches 1005 and 493 with 2 and 1, for example. This leads to the same result in terms of commonality of test cases and suites, besides some minor loss of precision. Also, one might want to estimate weights, based on personal knowledge about the system, if obtaining execution counts is not feasible. Alternatively, one could choose to only use execution counts from one specific group of users. A situation may also occur where only part of the branches have known execution weights. In that case, it may be possible to infer weights for those branches using other weights in the CFG.

Now let us define the *commonality score* of a test case and test suite. For a test case, its commonality score depends only on the branches it covers (i.e., on the branches it executes), and on the execution weights of the branches in the whole CUT that have the highest and lowest execution weights. This leads to a value between 0 and 1, with a higher value indicating a test case executing a higher amount of common behaviours in the CUT. Branches without execution weights (actual or inferred) are ignored when computing the commonality score. Suppose we have the following:

- a test case t
- n branches in the CUT, covered by t, and having an execution weight: B_x with $0 \leq x \leq n-1$
- execution weight of branch B_x: $w(B_x)$
- execution weight of the branch with the highest execution weight in the CUT: H
- execution weight of the branch with the lowest execution weight in the CUT: L

Then the commonality score for test case t can be denoted as in Equation (3.2) below.

$$c(t) = \frac{\sum_{x=0}^{n-1} (w(B_x) - L)}{n \cdot (H - L)} \tag{3.2}$$

The commonality score for a test suite is defined as the average of the commonality scores of all test cases in the suite.

It may be important to emphasize that the commonality score is computed over the whole CUT. In the example here only one method with its CFG is shown, but in practice a class will have multiple methods, each with its own CFG. The highest execution weight H and the lowest execution weight L are defined for the whole CUT, not just for the method

	*condition*1	*condition*2
t_1	`false`	`true`
t_2	`true`	*N/A*
t_3	`false`	`false`

Table 3.1: Truth values for branching conditions in algorithm 1, for test cases t_1, t_2, and t_3

that the branch happens to be part of. This allows the commonality score to reflect how likely a part of the CUT is to be executed in practice compared to other parts of the class, as opposed to only within a single method.

Commonality score example

Some example execution weights are shown in the example CFG in fig. 3.1. Let us assume that in this case the execution weights represent actual execution counts from some usage session of the example method, and that no errors occurred during execution. The weights of the root and exit nodes show us that the method has been executed 10 times in total. B_2 has been executed 3 times, while B_3 has been executed 7 times. Note that it makes sense that the counts for theses branches add up to 10, because that is the total number of times the method has been executed, and in the corresponding `if-else` statement (see algorithm 1), either the `if` or `else` branch has to be taken. Additionally, we see that `condition2` was satisfied in 1 of the 10 runs of the method.

Let us now illustrate the definition of the commonality score using three example test cases t_1, t_2, and t_3. Pseudo-code will not be given for them. It will only be stated whether `condition1` and `condition2` are `true` for them, as those expressions determine the control flow. We will assume here that the example method in algorithm 1 is the only method in the CUT, and that the two condition expressions just mentioned can somehow be influenced by the test cases to make them either `true` or `false`.

The values for `condition1` and `condition2` for the test cases are as shown in table 3.1. Let us compute the commonality score for the simplest test case (which covers the least branches), t_2. By having a value of `true` for `condition1`, it branches left at B_1 (see fig. 3.1). It then reaches B_2, and goes on to the last branch it covers, B_6. Using eq. (3.2) to compute the commonality score $c(t_2)$, we get the following computation and result.

$$\begin{aligned} c(t_2) &= \frac{\sum_{x\in\{1,2,6\}} (w(B_x) - L)}{n(H-L)} \\ &= \frac{\sum_{x\in\{1,2,6\}} (w(B_x) - 1)}{3\cdot(10-1)} \\ &= \frac{(10-1)+(3-1)+(10-1)}{27} \\ &= \frac{20}{27} \approx 0.741 \end{aligned}$$

Using an analogous computation for $c(t_1)$ and $c(t_3)$, we get the following commonality scores.

$$c(t_1) = \frac{2}{3} \approx 0.667$$
$$c(t_3) = \frac{5}{6} \approx 0.833$$

Going by these commonality scores, t_3 exercises the most common behaviors of the CUT, followed by t_2, with t_1 exercising the most uncommon behaviors. This seems intuitive, because t_1 takes a path through the method that is executed only 1 out of 10 times, t_2 takes a path that is executed 3 out of 10 times, and t_3 takes a path that is executed 7 out of 10 times. Note that t_3 and t_1 have respectively the highest and lowest commonality score that can be obtained by test cases with a single call to this method. Note that this shows that the highest and lowest obtainable scores are not necessarily 1 and 0. In fact, most of the time they are not, because it would require covering only branches with the highest and lowest execution weight, which is often not possible to do.

Now suppose we have two test suites s_1 and s_2. s_1 consists of only t_1 and s_2 consists of t_2 and t_3. We can compute their commonality scores, as defined, by taking the average of all commonality scores of the test cases they contain. In this simple example, s_1 only contains t_1, and hence its commonality score is that of t_1, which is $\frac{2}{3} \approx 0.667$. For s_2, we take the average of the commonality scores of t_2 and t_3, which is $\left(\frac{20}{27} + \frac{5}{6}\right)/2 = \frac{85}{108} \approx 0.787$. Going by these numbers, s_2 covers more common parts of the code than s_1.

3.2 Incorporating commonality in an evolutionary algorithm

This section describes how the commonality score defined in Section 3.1 can be used to influence the search direction of an evolutionary unit test generation algorithm. The search can be steered in the direction of covering more common behaviors, more uncommon behaviors, or both, in the CUT. We call the resulting method of test generation common and uncommon behavior test generation (CUBTG).

During regular test generation, when not using CUBTG, the goal is solely to obtain the highest possible coverage values for the metrics that are being used (e.g., branch coverage). It generally does not matter what the test cases used to obtain coverage look like, as long as they contribute towards raising the coverage value. When using CUBTG methods, on the other hand, the goal changes somewhat. In addition to obtaining high coverage values, which still remains an important objective, we also value the commonality score of the test cases and test suites that are generated. The aim is to satisfy coverage goals for test generation using test cases that exercise the commonly or uncommonly executed parts of the code.

There are several algorithms available for evolutionary test generation. The current state of the art algorithm is the many-objective sorting algorithm (MOSA) [20]. Another, older relevant algorithm is the algorithm for whole test suite optimization introduced with EvoSuite [11], and variants thereof. Also relevant is the more novel algortihm is DynaMOSA

[22]. This is a refined version of MOSA that works more efficiently and somewhat more effectively, especially when using smaller search budgets. According to two recent studies [21, 6], both MOSA and DynaMOSA outperform whole suite approaches and other many-objective algorithms. For DynaMOSA it is necessary to order the search objectives, while MOSA considers them all at the same time. The lattter fits best with how we want to incororate CUBTG. For these reasons, we chose to build upon MOSA for this study.

In the remaining of this section, relevant parts of the test case selection procedure in MOSA will be explained in Section 2.2, and our additions to MOSA for CUBTG will be discussed in Section 3.2.1.

3.2.1 Additions to MOSA

We made two modifications to MOSA to incorporate CUBTG in the algorithm. Note the discussion of test case selection in MOSA in Section 2.2 for a discussion of the parts of MOSA relevant for the discussion here.

Fitness functions

First, we created two fitness functions (FFs) from the commonality score defined in section 3.1: one of them to steer the search process to test cases with high commonality score (covering more common behaviors), and the other to steer the search process to test cases with low commonality score (covering more uncommon behaviors). For the former, we want the FF value to get lower once the commonality score for the test case gets higher (because a lower FF value means performing better in terms of the corresponding coverage goal). And for the latter, we want this value to get lower once the commonality score gets lower. We define these two FFs as in Definitions 3 and 4. Note that the values for these two FFs will always be between 0 and 1 (both inclusive), because that is the case for $c(t)$ too.

Definition 3 (Common behavior fitness function) *For a test case t and its commonality score $c(t)$, the fitness function value $f_{common}(t)$ is defined as follows.*

$$f_{common}(t) = 1 - c(t)$$

Definition 4 (Uncommon behavior fitness function) *For a test case t and its commonality score $c(t)$, the fitness function value $f_{uncommon}(t)$ is defined as follows.*

$$f_{uncommon}(t) = c(t)$$

One can use one, both, or none of the above FFs depending on the goal of the test generation. For both of the FFs, the intention is for them to steer the direction of the search in a specific direction, not to cover a specific goal. By adding an additional dimension to the search space, the MOSA algorithm will look for test cases performing well in that direction,

in this case test cases with high or low commonality scores. The goal is to influence the test cases that will cover other goals (e.g., branches) to have higher or lower commonality scores.

Secondary objectives

Second, we modified the criterion that is used to choose which test case to keep in case of a tie, for the situation where two or more test cases have the same fitness value for a coverage goal. This is called the secondary objective (SO). By default, as mentioned above, MOSA uses the length of the test case (number of statements) as a SO, and keeps the shorter one. We added two additional SOs to this based on the commonality score of the test case. One that keeps the test case with the higher commonality score, and another that keeps the test case with the lower commonality score.

We still kept the test case length as an aspect for deciding which test case to keep, because we would still like the test case length to be limited. To combine the two types of SO, we used the following method. For two tied test cases, we used their relative difference in terms of length and commonality in two new secondary objectives. A value < 1 means that the first test case is better and a value > 1 means that the second test case is better. A value equal to 1 means that both test cases perform the same in terms of the secondary objective. In that case, we arbitrarily choose to keep the first test case. So for example, if the first test case is twice as good as the second, the SO value would be 0.5. If the situation would be reversed, the value would be 2. Let us state this more formally.

Definition 5 (Common behavior secondary objective) *For two test cases t_1, t_2 with lengths l_1, l_2, and configurable weights α and β, the comparison between the two test cases is done using the following formula:*

$$common(t_1,t_2) = \frac{\alpha\left(\frac{l_1}{l_2}\right) + \beta\left(\frac{1-c(t_1)}{1-c(t_2)}\right)}{\alpha+\beta}$$

If $common(t_1,t_2) \leq 1$, then t_1 is kept, otherwise t_2 is kept.

Definition 6 (Uncommon behavior secondary objective) *For two test cases t_1, t_2 with lengths l_1, l_2, and configurable weights α and β, the comparison between the two test cases is done using the following formula:*

$$uncommon(t_1,t_2) = \frac{\alpha\left(\frac{l_1}{l_2}\right) + \beta\left(\frac{c(t_1)}{c(t_2)}\right)}{\alpha+\beta}$$

If $uncommon(t_1,t_2) \leq 1$, then t_1 is kept, otherwise t_2 is kept.

Example To illustrate this method of combining secondary objectives, let us revisit the example from Section 3.1.2. Remember that the commonality score for t_1 was $2/3$ and that the commonality score for t_3 was $5/6$. Suppose for these test cases that $l_1 = 2$ and $l_3 = 3$, and that we are using the common behavior secondary objective defined in Definition 5

above to decide which test case to keep, because they cover the same goal (e.g., they both cover $B3$, see table 3.1). Let us use weights $\alpha = 1$ and $\beta = 2$. Then we can calculate the secondary objective value as follows:

$$\begin{aligned} common(t_1, t_3) &= \frac{\alpha\left(\frac{l_1}{l_3}\right) + \beta\left(\frac{1-c(t_1)}{1-c(t_3)}\right)}{\alpha+\beta} \\ &= \frac{1 \cdot \frac{2}{3} + 2 \cdot \frac{1-2/3}{1-5/6}}{1+2} \\ &= \frac{\frac{2}{3}+4}{3} = \frac{\frac{14}{3}}{3} = \frac{14}{9} \end{aligned}$$

As $14/9 > 1$, we will choose to keep t_3 in favor of t_1 in this case.

Summary

This section defined the FFs and SOs for incorporating CUBTG in MOSA. Using them allows unit tests generation aiming to cover traditional coverage goals, while steering the generation towards common or uncommon behaviors in the CUT. The FFs aim to influence the commonality score of a test while it is still being generated throughout the iterations of MOSA. The aim of the SOs is to keep the test with the highest or lowest commonality score when multiple test cases are found for the same coverage goal.

Chapter 4

Empirical evaluation - methodology

This chapter describes the methods used to evaluate the performance of common and uncommon behavior test generation (CUBTG) described in Chapter 3. The main goal of the evaluation is to compare the performance of CUBTG with the performance of the many-objective sorting algorithm (MOSA) [20] as implemented in EvoSuite. Because we are not sure of any additional effects of adding the CUBTG fitness functions (FFs) and secondary objectives (SOs) to the many-objective MOSA algorithm, we also compare against the non-dominated sorting genetic algorithm II (NSGA-II) algorithm (which is a multi-objective algorithm) supplemented with the CUBTG FFs and SOs.

For the purpose of this evaluation, we implemented the CUBTG methods described in Chapter 3 in EvoSuite. The implementation is available at `https://github.com/STAMP-project/evosuite-ramp/tree/cub-test-gen`. A replication package for this evaluation is available at `https://github.com/Bjorn48/cubtg-es-evaluation`, and a docker version of this replication package can be found at `https://github.com/Bjorn48/cubtg-es-evaluation-docker`.

We executed the implementation on a server with two Intel Xeon E5-2660 v3 CPUs (running at 2.60 GHz). Each CPU has 10 cores and can run 20 threads at once. The server has about 378 GiB of RAM.

We will start this chapter with stating and clarifying our research questions in Section 4.1. Second, we will discuss the subject for this evaluation in Section 4.2. Third, in Section 4.3 we will discuss the method we used to obtain execution data. Fourth, we will discuss the EvoSuite configurations we used in Section 4.4. And last in Section 4.5, we will describe what result data we collected and how we analyzed it.

4.1 Research questions

In this evaluation, we aim to address the overall question of how effective and efficient test generation using CUBTG methods is compared to standard MOSA, and compared amongst themselves. The following research questions are addressed for that purpose.

RQ1 Do tests generated by CUBTG achieve a better commonality score compared to standard MOSA, and how do the different CUBTG methods compare?

RQ2 How do software faults revealed by CUBTG differ from standard MOSA, and how do the different CUBTG methods compare?

RQ3 How do tests generated by CUBTG compare to standard MOSA in terms of standard code coverage metrics, and how do the different CUBTG methods compare?

RQ4 Does the usage of the added FFs and SOs affect the efficiency of the EvoSuite test generation?

RQ5 How much time of EvoSuite test generation does it take for commonality score to converge?

The purpose of **RQ1** is to verify the effect of the FFs and SOs added in CUBTG. If they work as intended, the maximum commonality score FF and SO cause generated tests to have a higher commonality score, while the minimum variants cause them to have lower commonality score.

With **RQ2** we try to find out whether using CUBTG makes generated tests find different kind of faults than when using standard MOSA. This may equate to a better fault revealing potential in terms of number of covered bugs, for example, but this does not need to be the case.

RQ3 is meant to check whether the additional search criteria that are added in CUBTG affect the performance of standard coverage metrics, like branch coverage. One could imagine standard coverage metrics to be negatively impacted, simply because more criteria are added, but also because the search through the solution space is steered in a certain direction. It is possible that this causes a global optimum for standard goals to be more difficult to find.

RQ4 focuses mainly on comparing the time EvoSuite needs for converging to stable coverage values when using the various CUBTG methods and standard MOSA. Because extra, and maybe relatively expensive, calculations are done to compute commonality score, convergence of coverage metrics to a stable value might take longer. For this RQ we look at the default EvoSuite coverage criteria as specified in Section 4.4.

Finally **RQ5** looks at the convergence pattern of standard MOSA and the CUBTG methods in terms of the newly implemented commonality score. As for **RQ4**, it looks at the convergence time for the fitness value.

4.2 Subjects

Software that could potentially used as a subject for this evaluation has to conform to the following requirements:

R1 It has to be written in Java, because EvoSuite and the infrastructure described in section 4.3.2 work with Java classes.

R2 It has to be runnable on a Windows laptop (corresponding to the system available for executing the gathering of execution data), possibly by running a virtual machine.

R3 It has to be executable on a personal computer to allow us to collect execution data, without needing server-like hardware.

R4 It has to allow user interactions, by using either a command-line interface or a GUI to allow enough variability between the different executions. Concretely, that means libraries or frameworks are ruled out.

It turns out that the first, second and last requirements together rule out a lot of options. Almost all open-source Java software are either libraries or frameworks, or applications to be run on servers. Taking the various requirements into account, we decided to focus on JabRef[1], "*an open source bibliography reference manager*". It can work with BibTeX and biblatex files, and features a GUI created using JavaFX[2].

We sampled a subset of 150 classes, including classes of varying nature to prevent biasing results (for example, by choosing classes that are too easy to cover with EvoSuite). The following three sets of classes have been selected. The cyclomatic complexity (i.e., McCabe's complexity) [18] and lines of code metrics have been computed using *CK*[3].

- The 75 classes with the highest cyclomatic complexity.
- The 38 classes with the largest number of lines of code, excluding all classes from the category above.
- The 37 classes that were executed the most according to the execution data, excluding all classes from the categories above. How often a class has been executed was determined by taking the execution weight of the branch with the highest execution weight.

Note that some classes were excluded because they did not work well with EvoSuite after initial tests. The excluded classes were those in the `org.jabref.gui` and `org.jabref.logic.importer.fileformat` package. The reason was that EvoSuite does not work well with JavaFX, used by JabRef, and with classes performing I/O operations.

Also, note that only a subset of these 150 classes are used when comparing EvoSuite configurations (which are described below). This has two reasons. First, some combinations of EvoSuite options and specific classes led to runtime errors in EvoSuite. To compare EvoSuite configurations reliably, enough successful runs are needed. And second, for comparing commonality scores, sometimes the branches executed by a generated test case or test suite did only exercise a part of the class under test (CUT) for which not enough execution data was available. The results corresponding to those test cases and test suites have not been considered.

[1]https://www.jabref.org/
[2]https://openjfx.io/
[3]https://github.com/mauricioaniche/ck

4.3 Obtaining execution count data

For computing the commonality score of a test case or test suite as defined in Section 3.1, and consequently for the application of the CUBTG methods as described in Section 3.2, execution count data is needed for the CUT. More specifically, the execution counts (see definition 1) of the basic blocks, or branches, in the control flow graph (CFG) of the CUT (see section 3.1.1) are needed. To obtain the most accurate results, these execution counts will have to be obtained from one or more usage sessions of the system that the CUT is part of.

In this section will first describe three methods to obtain execution count data we have considered in Section 4.3.1, and will describe in a bit more detail how we implemented our chosen method in Section 4.3.2.

4.3.1 Approaches for gathering execution data

This subsection will briefly describe the methods we considered for obtaining execution count data. First, one can use regular, existing log statements to obtain execution counts for the branches in the CUT. This approach consists of matching log messages to the log statement that outputted the message. One way to do this is using static analysis [37, 10], which has been shown to work well in practice [29]. In short, this method consists of creating templates from the log statements in the source code. It uses the observation that most log statements are built from a static part and a dynamic part. Log messages from a log file that has been outputted are then matched against those templates, using their static part, to determine which log statement they most likely originate from. For our purpose here, this can be used to determine how often the branch containing each log statement has been executed during a usage session. We did not use this method for our evaluation, because there is no implementation readily available (there is an implementation by Schipper et al. [29], but it has not been published).

The second approach one can take is to adjust the configuration of the logging framework that is being used to include the location in the source code of the log statement with each log message that is outputted. This is a straightforward approach requiring no additional software, but it causes significant performance loss at runtime. For example, the Log4j [4] framework throws an exception when the log statement is executed and retrieves location information from the stack trace. According to the developers, this is 1.3 to 5 times slower for synchronous loggers, and 30 to 100 times slower for asynchronous loggers, compared to not outputting location information [5]. This makes it infeasible to use in most systems, especially in production. We tried out this approach, but the application we used for our evaluation (JabRef) became too slow to use. Hence, we decided not to use this approach.

Third, one can add new log statements at specific locations in the code (source code, compiled code, or anything in between) specifically for tracing the execution during a usage session. Each tracing log statement can be configured to print an identifier unique to that

[4] https://logging.apache.org/log4j/2.x/

[5] https://logging.apache.org/log4j/2.x/manual/layouts.html#LocationInformation

statement, so that it can be easily inferred from the resulting log file which log statement has been executed, including its location. In essence, this is a simplified version of the first method requiring additional log statements. We used this method for the evaluation because it was relatively easy to implement and it did not impact the responsibility of JabRef too much. One of the main downsides of this method is that it requires modification of the code.

4.3.2 Implementation of our approach

This subsection describes the way we implemented the third method from the last subsection for obtaining execution data for our evaluation. The approach we took consists of three steps which we will describe here in order.

Step 1: inserting tracing log statements

The first step is the addition of log statements to the CUT. JabRef uses Log4j itself, so we decided to use it for the tracing log statements too. We decided to insert the new log statements directly into the source code, as opposed to the bytecode, because we had full control over this experiment, and inserting the new log statements into source code makes it easier for us to comprehend the resulting source code including the new log statements.

To insert the new log statements, we used the *Spoon*[6] library. Quoting its website's front page, "Spoon is an open-source library to analyze, rewrite, transform, transpile Java source code". Because we wanted the execution data to be as accurate as possible, we decided to take the somewhat rigorous approach of adding a log statement in almost every code branch. To be explicit, these are the locations in which log statements were added:

1. At the start of the *then* part of every if-statement.
2. At the start of the *else* part of every if-statement.
3. At the start of the body of every catch-block in a try-catch-statement.
4. At the start of the body of every method.
5. At the start of the body of every constructor.

There were some parts of the code in which log statements were not added for technical or practical reasons, even if they qualified as one of the above locations:

1. If the statement would be inserted in a default method[7]. Code in a default method cannot reference class fields, which we used to implement the trace logging in JabRef.
2. If a method does not have an existing body. This is the case for unimplemented interface methods and abstract methods in abstract classes. Adding a log statement here would mean breaking binary compatibility with the original version, and we want the execution data to represent an execution of the original software without the added log statements.

[6] http://spoon.gforge.inria.fr/
[7] https://docs.oracle.com/javase/tutorial/java/IandI/defaultmethods.html

Each added log statement simply prints an identifier (we used a `long` value) unique for that log statement. This led to a modified JabRef version with the only difference in behavior being that an extra log file was outputted containing one of these unique identifiers on each line.

Note that the tracing log statements can be enabled and disabled using the Log4j configuration files. Suppose that one would want to use this method in production, one could enable and disable these tracing log statements at runtime whenever one wants to turn tracing on and off.

Our goal was to obtain execution data that was representative for a typical execution of JabRef. To reasonably closely approximate this, we decided to let four people plus the author of the thesis use JabRef for some predefined random task, as described below, for approximately 5 minutes. Before the start of the short usage session, we made sure that it was clear for the person how to use JabRef, by giving a short explanation (obviously this was not needed for the author). We collected the trace logs resulting from these usage sessions, and used them as an input for step 2.

JabRef task This is the task we asked the participants to perform for collecting execution data. Note that we allowed the participants to perform any secondary actions which they felt were natural during the task.

1. Start the JabRef application.
2. Create a new bibliography database.
3. Add approximately 3 entries (e.g., articles) to the database corresponding to something you are currently working with, or are interested in.
4. Modify one or more of the added entries.
5. Save the bibliography database to disk.
6. Close the JabRef application.

Step 2: converting log statement identifiers to source code locations

In this step we convert the file with an identifier on each line, outputted from Step 1, to a file with on each line the code location of the log statement corresponding to the identifier. We again used Spoon for that purpose. We let Spoon output the location in the software under test (SUT) in the following format: `<class-name>|<method-name>|<line-number>`, where:

- `<class-name>` is the fully qualified name of the class in which the log statement resides. This needs to be included, because we may be collecting execution data for multiple classes at once (which we were in this case).
- `<method-name>` is the name of the method in which the log statement resides, without a parameter list and without parentheses after the name.

```
org.jabref.gui.util.ThemeLoader|addAndWatchForChanges|94
org.jabref.gui.util.ThemeLoader|addAndWatchForChanges|99
org.jabref.gui.JabRefDialogService|notify|274
org.jabref.gui.DefaultInjector|injectMembers|73
org.jabref.logic.citationstyle.CitationStyle|createCitationStyleFromFile|98
org.jabref.gui.JabRefDialogService|notify|274
org.jabref.gui.JabRefDialogService|notify|274
org.jabref.gui.JabRefDialogService|notify|274
org.jabref.gui.DefaultInjector|injectMembers|73
org.jabref.gui.DefaultInjector|injectMembers|73
org.jabref.gui.JabRefDialogService|notify|274
```

Figure 4.1: Part of an example location log file, containing the location of an executed log statement on each line.

- `<line-number>` is the line number of the log statement.

A part of an example of such a file is shown in Figure 4.1. One may wonder why we did not write the location of the log statement directly to the log file in step 1, instead of an identifier, by inserting it directly in the message to be logged. The reason for this was that writing only a identifier to the file costed much less time and disk space at runtime compared to writing the whole location at runtime. JabRef became a lot less responsive when writing the full location to the log file at runtime.

Step 3: creating an aggregated JSON count file

We inputted the file ouputted from Step 2 into a Kotlin script. This script combines the location data into a hierarchical JSON file, going from class, to method, to line, and then stating the execution count for each separate line. This file can be used by EvoSuite to extract execution count data to apply CUBTG methods. An example file is shown in Figure 4.2.

4.4 EvoSuite configurations

The configurations used to run EvoSuite are listed in Table 4.1 with an identifier, and the search algorithm, the coverage criteria, and the SOs used. Note that the default set of coverage criteria and default test case length SO are referred to in table 4.1 using the abbreviation “def.”. They are described below. Similarly, the maximum and minimum commonality score variants of the FFs and SOs are referred to using the abbreviations “max.” and “min.”. For a more detailed explanation of these commonality related FFs and SOs, see Section 3.2.1. A search budget of 180 seconds was used for all configurations. According to related literature [12, 21], three minutes is a good compromise between run time and coverage.

In the following, the terms *maximum configuration variants* and *minimum configuration variants* will sometimes be used. The former refers to the configurations `fit_max_sec_max` and `fit_def_sec_max`, and the latter refers to the configurations `fit_min_sec_min` and

```
{
    "className": "org.jabref.gui.util.ThemeLoader",
    "methods": [
        {
            "methodName": "addAndWatchForChanges",
            "executionCounts": [
                {
                    "lineNumber": 94,
                    "count": 1
                },
                {
                    "lineNumber": 99,
                    "count": 1
                }
(...)
```

Figure 4.2: Part of an example JSON execution count file.

Configuration identifier	*Search algorithm*	*Coverage criteria*	*SOs*
`fit_def_sec_def`	MOSA	def.	def.
`fit_max_sec_max`	MOSA	def. + max.	def. + max.
`fit_min_sec_min`	MOSA	def. + min.	def. + min.
`fit_def_sec_max`	MOSA	def.	def. + max.
`fit_def_sec_min`	MOSA	def.	def. + min.
`fit_max_min_sec_def`	MOSA	def. + max. + min.	def.
`nsgaii_max`	NSGA-II	def. + max.	def. + max.
`nsgaii_min`	NSGA-II	def. + min.	def. + min.

Table 4.1: EvoSuite configurations

`fit_def_sec_min`. This corresponds to the type of FFs and SOs are used in those configurations.

EvoSuite default coverage criteria We decided to use the following set of standard coverage criteria for EvoSuite: line, branch, exception, weak mutation, input, output, method, method (without exceptions), context branch.

Default SO Note that, as described in Section 2.2, the default SO used by EvoSuite favors test cases and suites that are shorter in terms of number of statements. If a default SO is referred to, it is this one.

The configurations in table 4.1 were included in the evaluation for the following reasons. The `fit_def_sec_def` configuration was included as a baseline. It does not use any of the functionality added to EvoSuite, and hence represents EvoSuite as it is used in a state-of-the art fashion. The `fit_max_sec_max` and `fit_min_sec_min` configurations are meant to look at the effect of the newly implemented FFs and SOs while steering the search in one specific direction. The `fit_def_sec_max` and `fit_def_sec_min` configurations have the same purpose, but only look at the effect of the added SOs, while using the default set of coverage criteria. The `fit_max_min_sec_def` configuration is meant to look at the effect of using both the minimum and maximum variants of the added FFs. Using this configuration, the added FFs steer the search towards common behaviors and uncommon behaviors simultaneously. Finally, the `nsgaii_max` and `nsgaii_min` configurations are added to compare the results to runs using a classical multi-objective search algorithm. They are meant to detect any patterns in the results that might be caused by using MOSA, with the newly implemented functionality, which might not occur using a multi-objective search algorithm.

Note finally that for the MOSA configurations, EvoSuite has been configured to not discard search goals once they are covered by some test case. The purpose of disabling this (default) behavior is to allow the search to find test cases that are better in terms of the minimum and maximum commonality score SOs, even if they have already been covered before. This might slow down the search, but it allows the result to improve in terms of commonality score.

4.5 Data collection and analysis

Let us now look at what data has been collected to answer the research questions, and how it has been collected. This will be stated for each research question separately. Each research question will first be repeated for ease of reading, in the format "Q: ⟨ RQ ⟩", where "⟨ RQ ⟩" is the actual research question.

RQ1 Q: "Do tests generated by CUBTG achieve a better commonality score compared to standard MOSA, and how do the different CUBTG methods compare?" To compute the commonality score of an individual test case, we need to know which branches it covers. We configured EvoSuite to output this information for each generated test case.

RQ2 Q: "How do software faults revealed by CUBTG differ from standard MOSA, and how do the different CUBTG methods compare?" To answer this question, we would need known faults (i.e., bugs) in JabRef to check whether CUBTG methods uncovers them better than standard MOSA. There is no dataset of real bugs available for JabRef (in a format that is easily usable). As has been researched before, strong mutation testing can be a good simulation of real faults in software [16]. We used PIT[8] for mutation testing, and ran it separately on each generated test suite, for each configuration, resulting in a mutation score for each generated test suite.

[8]https://pitest.org/

RQ3 Q: "How do tests generated by CUBTG compare to standard MOSA in terms of standard code coverage metrics, and how do the different CUBTG methods compare?" Information about standard coverage metrics on the test suite level can be outputted by EvoSuite in its statistics file. We configured EvoSuite to output the data we needed for standard coverage metrics to that file. In addition, we added code to EvoSuite to output to a file the length of each generated test case.

RQ4 Q: "Does the usage of the added FFs and SOs affect the efficiency of the EvoSuite test generation?" We need FF values for different points in time during the runtime of EvoSuite to observe convergence patterns of those FF values. To obtain this data, we modified EvoSuite to output intermediate FF coverage values to its log file.

RQ5 Q: "How much time of EvoSuite test generation does it take for commonality score to converge?" Like for the previous research question, intermediate values for commonality score are needed to answer this question. It would take very large amounts of space to store these intermediate values for every test case, so we only stored them per test suite, in the same way as for the other FFs.

For determining the statistical significance and magnitude of the obtained results, we have compared them using the Mann–Whitney–Wilcoxon (MWW) test [14] and the Cliff's delta measure [7], respectively. For both of these, we have grouped the results by class in order to obtain a fair comparison between test cases, as suggested by [16]. For significance, we have taken a p value of 0.05 resulting from the MWW test to be significant result. The Cliff's delta measure is a value between -1 and 1, where a value > 0 means that the first configuration is higher, while a value < 0 means that the configuration it is compared against gives a higher result. The absolute value indicates the magnitude of the result. When talking about effect sizes below, we mean the absolute value, unless stated otherwise. Romano [27] suggested the following qualitative assessments for the effect size e, which are used in this chapter:

- $e < 0.147$: negligible
- $0.147 <= e < 0.33$: small
- $0.33 <= e < 0.474$: medium
- $e >= 0.474$: large

To obtain statistically relevant data, we repeated the evaluation 30 times for each of the selected classes. Taking into account that we used 8 different EvoSuite configurations, the total amount of EvoSuite executions (and hence generated test suites) is 36,000.

Chapter 5

Empirical evaluation - results

This chapter shows the results of evaluating the common and uncommon behavior test generation (CUBTG) implementation on JabRef, using the setup described in the previous chapter. The results relevant for each research question will be shown separately, each in their own section. At the end of this chapter, possible threats to the validity of this evaluation will be discussed. In this chapter, the standard the many-objective sorting algorithm (MOSA) configuration, `fit_def_sec_def`, will often be referred to as the "default configuration" or "standard MOSA", because it is the main existing configuration the newly implemented configurations are compared to.

Note that results for the EvoSuite configurations using the the non-dominated sorting genetic algorithm II (NSGA-II) algorithm are mostly not discussed in this section, because they were added to detect any (odd) patterns caused by using them in MOSA versus a multi-objective algorithms, as discussed in the previous section, not for comparing effectiveness or efficiency as is done here. The results are still shown for completeness.

5.1 RQ1: Commonality score

This section presents the results, discussion and conclusion for **RQ1**, which was "Do tests generated by CUBTG achieve a better commonality score compared to standard MOSA, and how do the different CUBTG methods compare?"

5.1.1 Results

For answering this question, let us take a look at the commonality score per test case. Overall coverage per configuration is shown in Figure 5.1. Additionally, all configurations have been compared pairwise. For all classes for which there were statistically significant differences ($p < 0.05$) in commonality score between configurations, effect sizes are shown in Figure 5.2. The total number of classes that were compared pairwise, per pair of configurations, is shown in Table 5.1. Note that results have not been grouped by class. This means that test suites with higher number of test cases in the suite have a higher weight than those with lower number of test cases in the suite. This is intended, because this figure is only meant to compare results on the test case level. It gives us more information than the suite

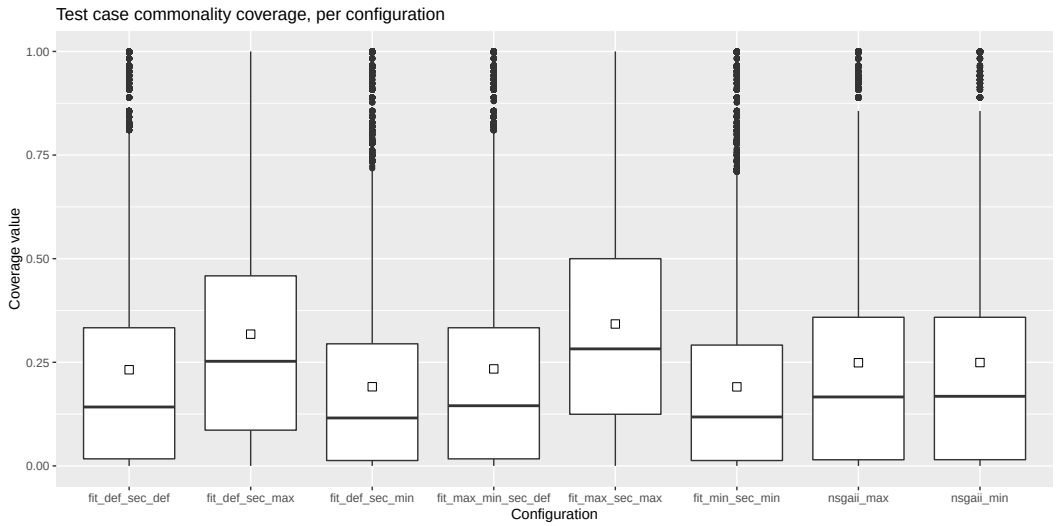

Figure 5.1: Commonality score per configuration. Each data point is a test case.

level, because instances of common or uncommon behaviors being covered multiple times by different test cases are hidden by looking at the results for test suites as a whole.

	Conf. name	1	2	3	4	5	6	7	8
1	fit_def_sec_def		81	81	80	81	80	81	81
2	fit_def_sec_max	81		82	82	82	82	82	82
3	fit_def_sec_min	81	82		82	82	81	82	81
4	fit_max_min_sec_def	80	82	82		82	81	83	83
5	fit_max_sec_max	81	82	82	82		82	83	83
6	fit_min_sec_min	80	82	81	81	82		82	81
7	nsgaii_max	81	82	82	83	83	82		83
8	nsgaii_min	81	82	81	83	83	81	83	

Table 5.1: Number of classes for which data is available for comparing two configurations for comparing commonality coverage.

These are the most important things to observe from this data:

- The configurations using the maximum commonality secondary objectives (SOs) and fitness functions (FFs) achieve a higher commonality score than the default configuration for a large part of the tested classes (58/81 for `fit_def_sec_max` and 65/81 for `fit_max_sec_max`). For both of these configurations there is a small to medium effect size for a lot of classes. For `fit_max_sec_max` in particular one can notice (from fig. 5.2) that a large part of the classes have a large effect size, a pattern that is much less present for `fit_def_sec_max` versus standard MOSA.

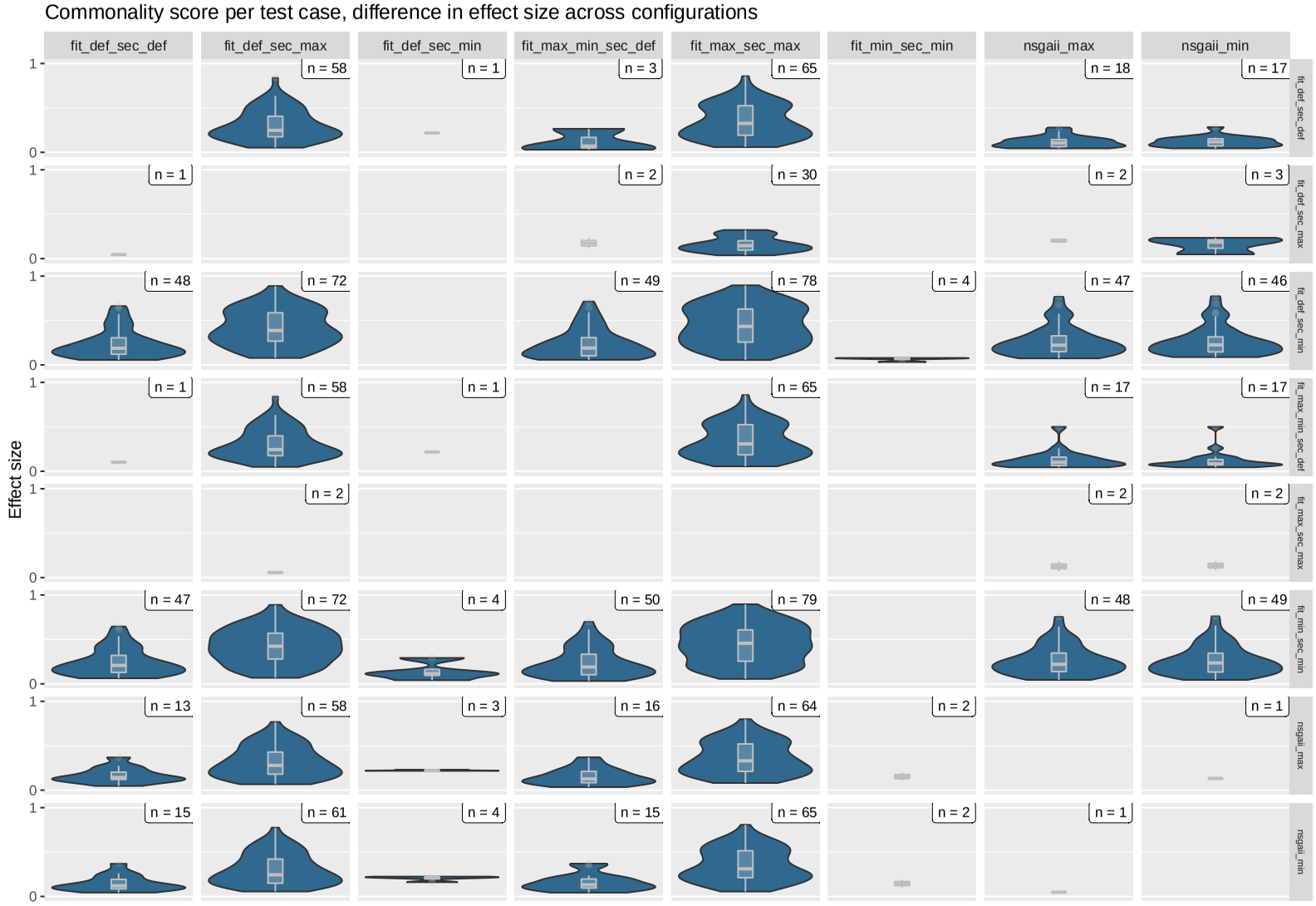

Figure 5.2: Effect size of commonality score difference per class, per configuration. Only significant results are shown ($p < 0.05$). In each cell, data is shown for which the column configuration resulted in a higher value than the row configuration. The number of observations is shown in the upper-right corner of each cell.

- For 30/82 classes, the `fit_max_sec_max` configuration gives a somewhat higher result than the `fit_def_sec_max` configuration. For most of these classes the difference is small (effect size around 0.25, see fig. 5.2), and for a few the difference is somewhat larger. Also, `fit_max_sec_max` never gives a lower coverage result than `fit_def_sec_max`.

- The configurations using the minimum commonality SOs and FFs achieve a little lower commonality score than the default configuration for a bit more than half of the classes in the test set (48/81 for `fit_def_sec_min` and 47/81 for `fit_min_sec_min`). The effect size is mostly small, with some classes having a medium to large effect size.

- For 4/81 classes `fit_min_sec_min` has a lower commonality score than `fit_def_sec_min`, and for another 4/81 classes this is the other way around. In both cases the effect sizes are small, although they are a bit larger when `fit_min_sec_min` has lower coverage.

- For 79/82 classes, `fit_max_sec_max` has a higher result than `fit_min_sec_min`, and the former never has a lower result than the latter (as was intended). Similar patterns can be seen in the pairwise comparisons when comparing the other minimum and maximum configuration variants. The effect sizes of the differences are mostly large or medium, with median values close to 0.5, as can be seen in fig. 5.2.

- For the `fit_max_min_sec_def` configuration, there are only 4/80 classes for which there is a significant difference in coverage compared to standard MOSA. If there is a difference, it is negligible except for 1 class.

- The commonality score is not particularly high on average. As can be seen in fig. 5.1, the median is about 0.3 for `fit_max_sec_max`, the configuration with the highest median.

It is worth noting that the results for most classes are much more significant than the $p < 0.05$ threshold as used in this section. For example, if we take a $p < 0.001$ instead, `fit_max_sec_max` is still significantly higher for 58/81 classes compared to standard MOSA, and `fit_min_sec_min` is still significantly lower for 39/80 classes. In other words, the results are very significant for most classes, and they are closer to the $p < 0.05$ threshold only for a smaller number of classes. For the exact significance values and effect sizes per class for the two configurations just mentioned, see Tables A.1 and A.2 in the appendix.

We have also checked the results for a possible correlation of the effect size and class complexity, lines of code, or how often a class has been executed. There does not seem to be such a correlation.

5.1.2 Discussion

Overall, the results for this research question show that the maximum CUBTG configuration variants achieve a higher commonality score for about 75% of the classes, compared to the

standard MOSA configuration, and that the minimum configuration variants achieve a lower commonality score for about 60% of the classes. This allows us to conclude that the addition of the FFs and SOs achieves the goal of increasing or decreasing the commonality score in most of the cases. Also the `fit_max_min_sec_def` configuration does not cause a significant difference in commonality score overall, as was intended. For only 4/80 classes there is a significant difference.

Looking a little bit closer at the results, the first thing to notice is that the maximum configuration variants often have a larger and more significant effect on the commonality score than the minimum configuration variants. In other words the commonality score that the minimum variants achieve is relatively closer to the coverage the standard MOSA configuration achieves, compared to the difference in achieved coverage between the maximum variants and standard MOSA. For example, `fit_max_sec_max` achieves a higher commonality score than standard MOSA for 65/81 (80%) of the classes, while `fit_min_sec_min` achieves a lower commonality score for 47/80 (59%) of the classes. Also, one can see in Figure 5.2 that the effect of the maximum configuration variants is generally greater than the minimum configuration variants.

A reason for this could be that there are relatively few branches with a high execution weight. This means that if you do not pay attention to commonality score (like the default configuration), you would not reach them as often as the branches with a lower execution weight. The maximum configuration variants give extra weight to reaching those branches, possibly causing longer test cases, for example, but would be able to make a relatively large difference in commonality score in this way. The minimum configuration variants would still be able focus on taking branches that have a lower execution weight, but because standard MOSA already chooses mostly branches with a relatively low execution weight, there would not be that much of an improvement.

Another possible explanation for this observation is that branches with a really low execution weight are very difficult to reach by the test generation algorithm. If we assume that, then it makes sense that the minimum configuration variants manage to cover some more of them than the default configuration, but not nearly all of them, causing only a small difference in commonality score between those configurations. The maximum configurations would then be looking to cover more easily accessible branches with higher execution weight, causing a larger difference in coverage. If the execution weights come from real executions of the software under test (SUT) (which is the case for this evaluation), then this explanation assumes that difficult to reach branches in the code are not executed often in those real executions. Intuitively, this sounds reasonable.

Something else that could have an effect on the minimum configuration variants being less pronounced is the way the commonality score is calculated (see Section 3.1). It takes into account the whole set of branches that are covered by the test case (i.e., the path it takes through the code). If we assume, like in the previous explanation, that branches with less execution weight are more difficult to reach, reaching them might require also covering branches that are higher in execution weight. Using the way of calculating commonality score outlined in section 3.1, the effect of covering the difficult to reach branch with low execution weight on the coverage value would be partly undone by covering the branches required to reach it. Looking at it this way, the effect of the minimum commonality score

FF and SO could actually be greater than is shown in the coverage value.

A second observation that needs discussion is that the addition of the maximum and minimum CUBTG SOs have a greater effect on commonality score than the corresponding FFs. For a possible explanation for this we have to think of how MOSA uses the FFs and SOs. This is discussed in more detail in Section 2.2. In short, search goals are created for all FFs (e.g., for all branches, lines, exceptions). A single search goal is also created for the minimum and maximum FFs. During each evolution generation, MOSA keeps the generated test cases that are not worse in terms of all search goals compared to any of the other test cases (i.e., if it is non-dominated, see section 2.2). Using the minimum and maximum FFs causes MOSA to look for test cases in the direction of covering more or less execution weight. Secondary objectives, on the other hand, are used when two test cases are found that cover the same goal (e.g., branch). The test case that has either the highest or the lowest commonality score is then kept. Test cases are found that cover the same goal, and even more often because we disabled removal of satisfied goals for the evaluation. Each time this occurs, the SO is used to decide which test case to keep, increasing or decreasing the commonality score of the test case satisfying that goal. The corresponding FFs, on the other hand, only help steer the search process in the direction of covering more commonly or uncommonly executed branches. They are only one search goal among the many other search goals. This probably causes the effect to be much less pronounced than for the SOs.

Third, using the minimum commonality score FF in addition to the corresponding SO has a lesser effect than for their maximum counterparts. This can be attributed to the same reason as outlined above for the performance of the minimum variants being lesser than for the maximum variants. Because it is already difficult to achieve less commonality score, it will also be difficult to realize the extra bit of effectiveness of the minimum FF.

Fourth, the commonality score is not particularly high or low overall, for none of the tested configurations. One possible reason for this is that, like discussed above, in most cases it is not possible to cover only a specific branch (with for example particularly high or low execution weight). Most of the time a test has to execute other code to reach a branch, which somewhat averages out the commonality score. Another possible reason is that achieving a high or low commonality score is not the main objective of the test generation. The main objective stays covering the main goals (covering branches, lines, exceptions, etc.). The goal is to do that in a way that covers more or less execution weight.

Fifth and last, the `fit_max_min_sec_def` configuration makes commonality score a little bit higher than standard MOSA. This can be most clearly seen from the effect size as shown in Figure 5.2, although there are only a small number of significant results. A possible reason for the small shift towards more commonality score is because the maximum commonality FF has more effect than the minimum one, as discussed above. In this configuration they are used both, but if the maximum FF causes more tests to be generated with higher commonality score than the minimum FF for low commonality score, there will be a small shift towards higher coverage.

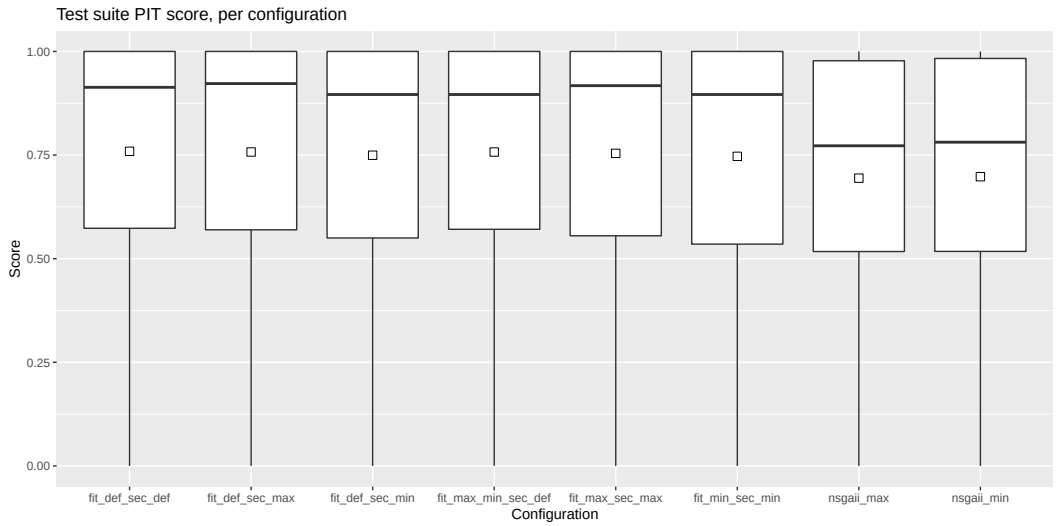

Figure 5.3: PIT score per configuration. Each data point is a test suite.

5.1.3 Conclusion

Tests generated using CUBTG FFs and SOs achieve better commonality score most of the time, and almost never achieve worse commonality score, compared to a standard MOSA configuration. For the maximum configuration variants, coverage is significantly higher in about 75% of the cases, and for minimum variants it is lower in about 60% of the cases. The effect of the maximum variants is higher. The CUBTG configuration including the maximum commonality score FF often performs significantly better than the configuration using only the SO. For the minimum CUBTG variants this difference cannot be observed.

5.2 RQ2: Fault revealing capability

This section presents the results, discussion and conclusion for **RQ2**, which was "How do software faults revealed by CUBTG differ from standard MOSA, and how do the different CUBTG methods compare?"

5.2.1 Results

Like for the previous research question, the PIT score distribution per configuration is shown in Figure 5.3, pairwise comparison of the effect size across configurations is shown in Figure 5.2, and the total number of classes used for these pairwise comparisons is shown in Table 5.2. Note that unlike for commonality score in the previous research question, these results are per test suite, not per test case.

These are the most important things to observe from this data:

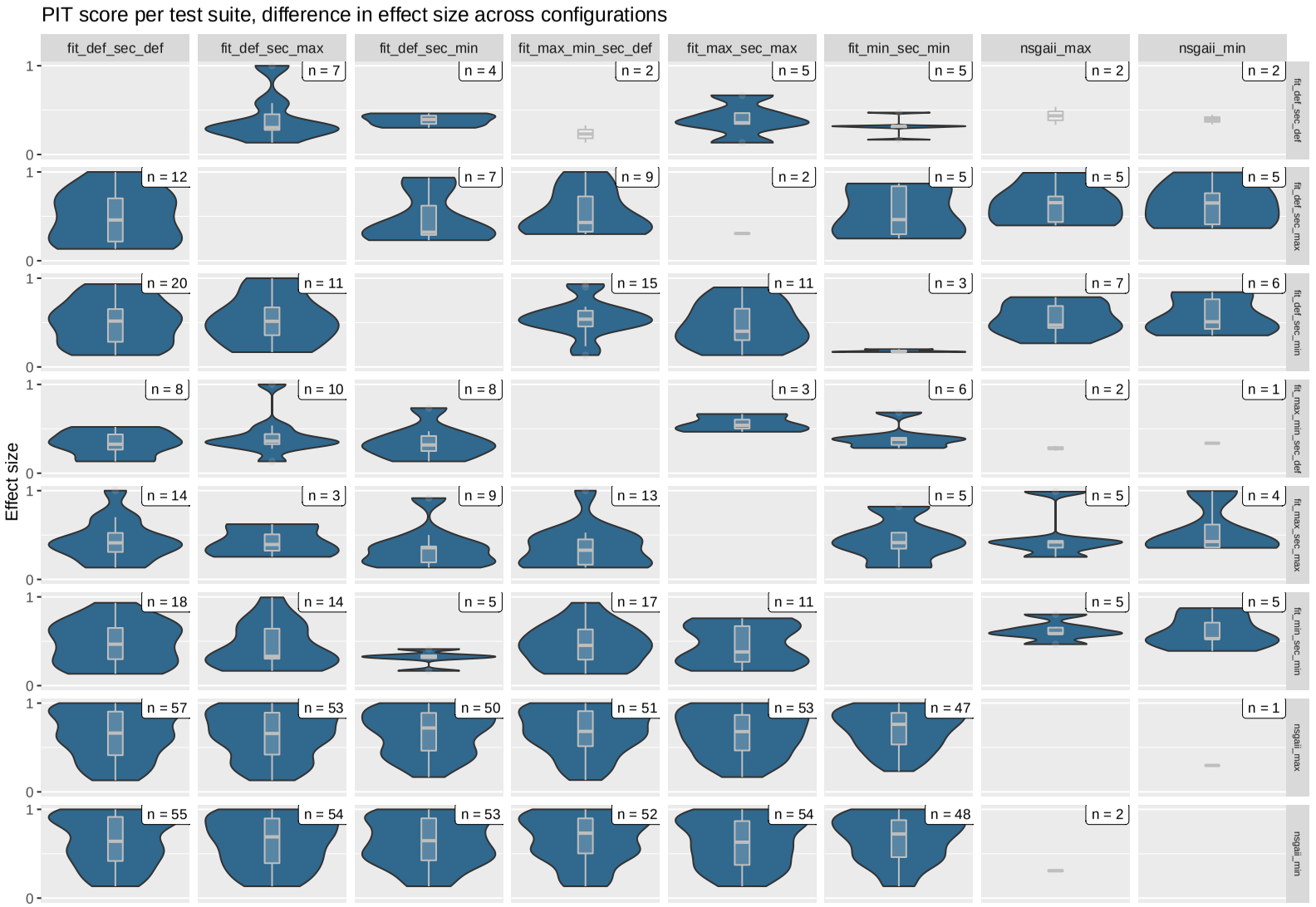

Figure 5.4: Effect size of PIT score difference per class, per configuration. Only significant results are shown ($p < 0.05$). In each cell, data is shown for which the column configuration resulted in a higher value than the row configuration. The number of observations is shown in the upper-right corner of each cell.

	Conf. name	1	2	3	4	5	6	7	8
1	fit_def_sec_def		70	74	60	71	74	79	79
2	fit_def_sec_max	70		75	71	72	76	83	82
3	fit_def_sec_min	74	75		75	76	76	84	83
4	fit_max_min_sec_def	60	71	75		71	75	80	80
5	fit_max_sec_max	71	72	76	71		76	83	83
6	fit_min_sec_min	74	76	76	75	76		84	83
7	nsgaii_max	79	83	84	80	83	84		82
8	nsgaii_min	79	82	83	80	83	83	82	

Table 5.2: Number of classes for which data is available for comparing two configurations for comparing PIT score.

- The differences in PIT score between configurations are not large in general (that is, between the MOSA configurations). The largest number of classes for which the PIT score is significantly better is for `fit_def_sec_def` compared to `fit_def_sec_min` (for 20/74 classes).
- The maximum configuration variants give slightly higher PIT scores compared to the default configuration, in terms of the median (0.922 and 0.917 versus 0.913), but the differences are very small. On the other hand, the mean for the max configurations is slightly lower compared to the default configuration.
- When comparing to standard MOSA, `fit_def_sec_max` gives a higher PIT score for 7/70 classes, and a lower score for 12/7 classes. For the classes for which it performs better, the effect size is mostly small to medium, and large for two of the classes. For the classes where the PIT score is better for standard MOSA, the effect sizes are fairly evenly distributed from 0 to 1. We have taken a closer look at the classes for which the PIT score turned out to be significantly better for `fit_def_sec_max`. 6 of those 7 classes have been executed relatively often according to the execution counts generated during the collection of execution data (see Chapter 4). The classes have a medium or low complexity compared to other classes in the test set. One of the classes, `org.jabref.logic.util.StandardFileType` (an `enum` class), achieves a result that is a lot better, with an effect size of 1 and a p-value of 1.18e-13 (i.e., there can be almost no doubt that the result is valid in a statistical sense). The average PIT score for this class when using `fit_def_sec_def` is 0.556, while it is 0.963 when using `fit_def_sec_max`. There is one other class for which there is a large effect size in terms of PIT score. For four of the classes there is a small effect size, and for one there is a negligible effect size. For more details about the results per class in terms of average PIT score, significance of the result and effect size, see Tables A.3 and A.4 in the appendix, comparing `fit_def_sec_max` and `fit_max_sec_max` to the default configuration, respectively.
- `fit_def_sec_max` gives a higher PIT score for some of the classes compared to the other CUBTG configurations, but it also performs worse than them for a not much

smaller number of classes. For the latter case, the effect sizes can also be reasonably high. See for example the comparisons with `fit_def_sec_min`, `fit_max_min_sec_def`, and `fit_min_sec_min` in fig. 5.4, where the effect size is 0.5 or higher (i.e., large), for a large part of the classes.

- `fit_min_sec_min` performs better than `fit_def_sec_def` for 5/74 classes. The effect sizes here are small for one class, medium for 2, and large for one. It performs worse for 18/74 classes, with a fairly evenly distributed effect size. Also, it performs pretty bad compared to other CUBTG configurations, in terms of number of classes and effect size.

5.2.2 Discussion

Let us start the discussion of this research question with a discussion about the consequences of using mutation testing, using PIT, to answer it. As noted in Chapter 4, we can use mutation testing only to simulate real faults in software. Unlike databases of real bugs, the locations where mutations are introduced in the code do not depend on how often that piece of code has been executed during normal usage. That is, one can imagine bug databases from real world scenarios to contain relatively more bugs in pieces of the software that end-users use a lot, while this is not the case for mutation testing. Of course it would be possible theoretically, but that is not how mutation testing is implemented by mutation testing software (like PIT) that is used today. Because the CUBTG methods of test generation rely on execution data from real world usage, results in finding bugs might be different for faults seeded by PIT, real world bug databases, or even yet to be uncovered bugs.

With that in mind, let us discuss the overall results with respect to the obtained PIT scores. As was shown in the results, there is no significant difference for by far the most classes (at most in 32% of the cases), and if there is a significant difference, it is almost always in favor of standard MOSA. Apparently, the addition of CUBTG does not help that much in improving PIT scores in general, besides in case of a small number of classes. However, as was shown in Figure 5.4, the `fit_def_sec_max` configuration does relatively well, compared to the other configurations (better for 10% of classes).

In general, one can imagine why using CUBTG methods would decrease the PIT score in some cases. Using the FFs for commonality score slightly steers the search process away from the other (main) goals of covering branches, lines, exceptions, etc. If less of those goals are eventually covered, one can imagine the amount of mutants being detected getting lower too.

The results show that adding the maximum commonality score FF makes the performance in terms of PIT score worse, compared to using only the corresponding SO. One possible reason for this could be that it would steer the test generation away too much from less executed branches, leaving mutants in those parts of the code alive.

As shown in the results, the classes on which `fit_def_sec_max` performs relatively well are classes that are executed often. This supports the point above that mutations might often be placed in pieces of the code that are executed often. The class on which the perfor-

mance was especially good compared to standard MOSA was a really small enum class (`org.jabref.logic.util.StandardFileType`).

After taking a closer look at the class, it appears that in a large majority of the cases (25/30), the tests generated using the `fit_def_sec_max` configuration contain a method sequence that is not present in the tests generated by `fit_def_sec_def`. From inspection of the execution counts that have been gathered, it turns out that there is only one branch, the single branch of a method, that has been executed by the data collection participants. This method is consistently involved in the test cases that kill the mutants which the tests generated by standard MOSA fail to kill most of the time. The fact that the method that is executed a lot is also involved in the method call sequence needed to kill the mutants seems to be coincidental. However, this observation is evidence for the theory that using CUBTG can generate method sequences in tests to satisfy the traditional coverage goals in different ways than standard MOSA, possibly finding faults in the class under test (CUT) of a different kind (like in this case). In either case, the behavior of generating a different method sequence is unlikely to be coincidental, evidenced by the p-value of 1.18e-13.

5.2.3 Conclusion

Tests generated using CUBTG FFs and SOs are not significantly better or worse in terms of achieved PIT score compared to standard MOSA, for most classes. If there is a significant difference, it is most of the time in favor of standard MOSA, and only in a few cases in favor of the CUBTG configuration. The `fit_def_sec_max` configuration stands out, performing better than standard MOSA for 10% of the classes, which are mostly executed often by the people who provided the execution data. It appears that using CUBTG can result in satisfying existing coverage goals (like) branches in different ways, which may result in finding faults in the CUT of a different nature than standard MOSA is able to. The minimum configuration variants perform relatively bad compared to other CUBTG configurations and standard MOSA.

5.3 RQ3: Standard coverage criteria and EvoSuite runtime metrics

This section presents the results, discussion and conclusion for **RQ3**, which was "How do tests generated by CUBTG compare to standard MOSA in terms of standard code coverage metrics, and how do the different CUBTG methods compare?"

5.3.1 Results

For all standard coverage metrics the newly added configurations performed mostly the same or somewhat worse than the default configuration. The differences between CUBTG configurations were not large, in general. We will not go into the details of each standard coverage metric. To illustrate the results, we will look at branch coverage as an example. As for the EvoSuite runtime metrics, the results for number of EvoSuite generations, test suite size, and test case length will be stated.

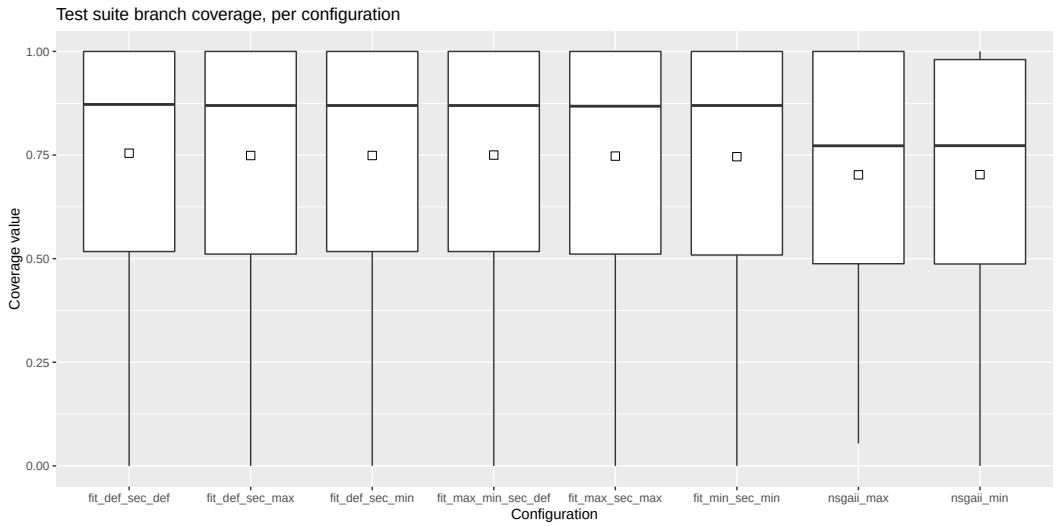

Figure 5.5: Branch coverage per configuration. Each data point is a test suite.

Branch coverage

Like for the previous research question, the branch coverage distribution per configuration is shown in Figure 5.5, pairwise comparison of the effect size across configurations is shown in Figure 5.6, and the total number of classes used for these pairwise comparisons is shown in Table 5.3. Indeed, almost no difference can be noticed between the configurations, save for a small number of classes for which some configurations perform better or worse. Two small observations that can be made are that `fit_min_sec_min` performs worse than the other MOSA configurations for a few classes (with mostly a medium effect size), and that `fit_max_min_sec_max` performs similar to `fit_def_sec_def` when compared to other MOSA configurations. When the latter two configurations perform better, it is mostly with a medium effect size, and with a large effect size for some classes. Apparently, the maximum and minimum commonality score SOs have the largest negative effect on branch coverage, when compared to the corresponding FFs.

#Generations

Let us now discuss the number of EvoSuite evolution generations. Again, the distribution of number of generations per configuration is shown in Figure 5.7, pairwise comparison of the effect size across configurations is shown in Figure 5.8, and the total number of classes used for these pairwise comparisons is shown in Table 5.4. Note that outliers have been omitted from fig. 5.7 for readability.

For all CUBTG configurations, the mean and median lie lower than for the default configuration, although this is less pronounced for `fit_max_min_sec_def`. If we look at the number of classes with significant differences, and the effect size of those differences, compared to standard MOSA, we see a lot more pronounced results. Compared to each of the

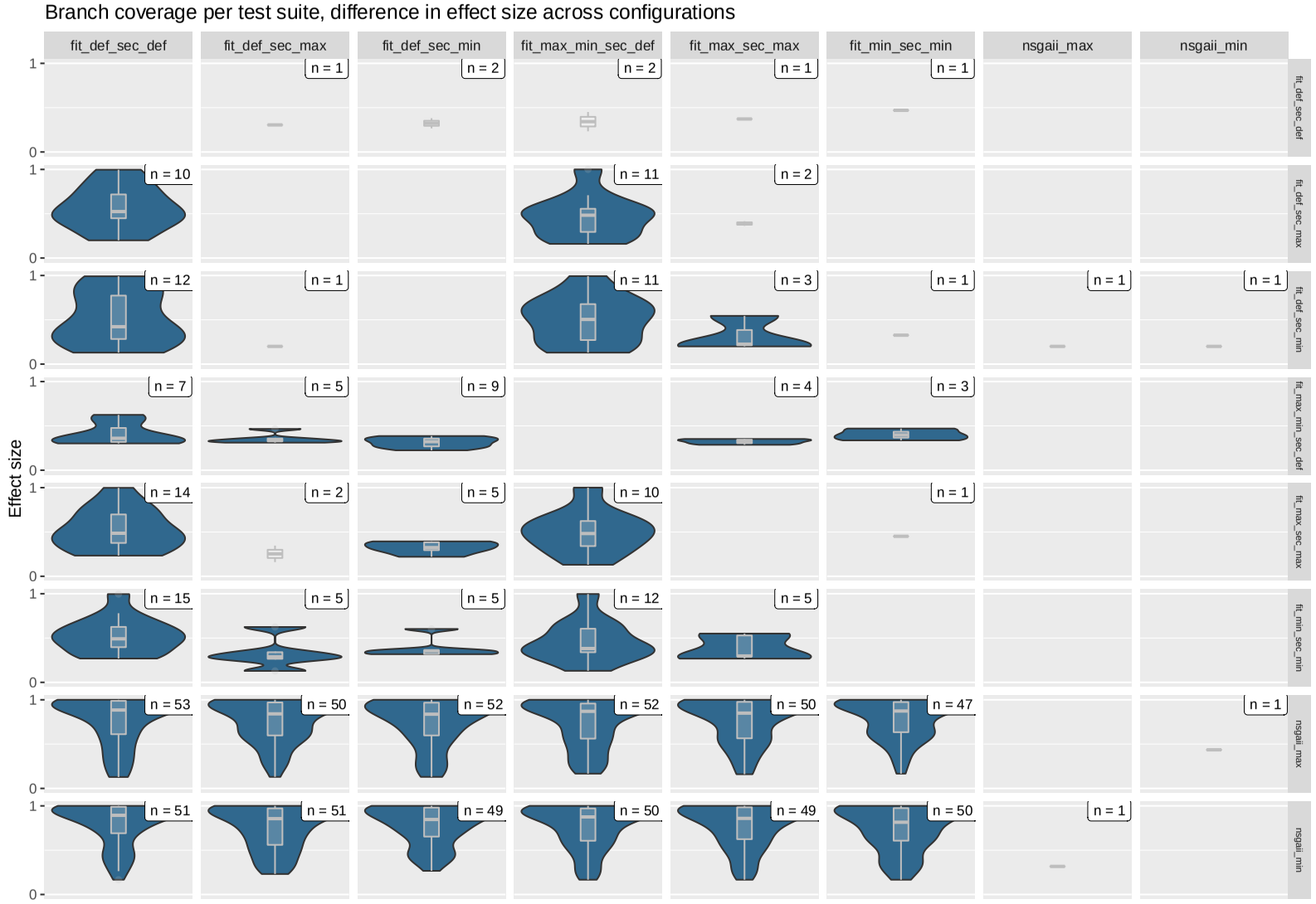

Figure 5.6: Effect size of branch coverage difference per class, per configuration. Only significant results are shown ($p < 0.05$). In each cell, data is shown for which the column configuration resulted in a higher value than the row configuration. The number of observations is shown in the upper-right corner of each cell.

	Conf. name	1	2	3	4	5	6	7	8
1	fit_def_sec_def		60	61	57	60	59	61	65
2	fit_def_sec_max	60		62	61	62	63	62	67
3	fit_def_sec_min	61	62		60	63	61	62	66
4	fit_max_min_sec_def	57	61	60		59	58	60	64
5	fit_max_sec_max	60	62	63	59		62	61	66
6	fit_min_sec_min	59	63	61	58	62		62	65
7	nsgaii_max	61	62	62	60	61	62		64
8	nsgaii_min	65	67	66	64	66	65	64	

Table 5.3: Number of classes for which data is available for comparing two configurations for comparing branch coverage.

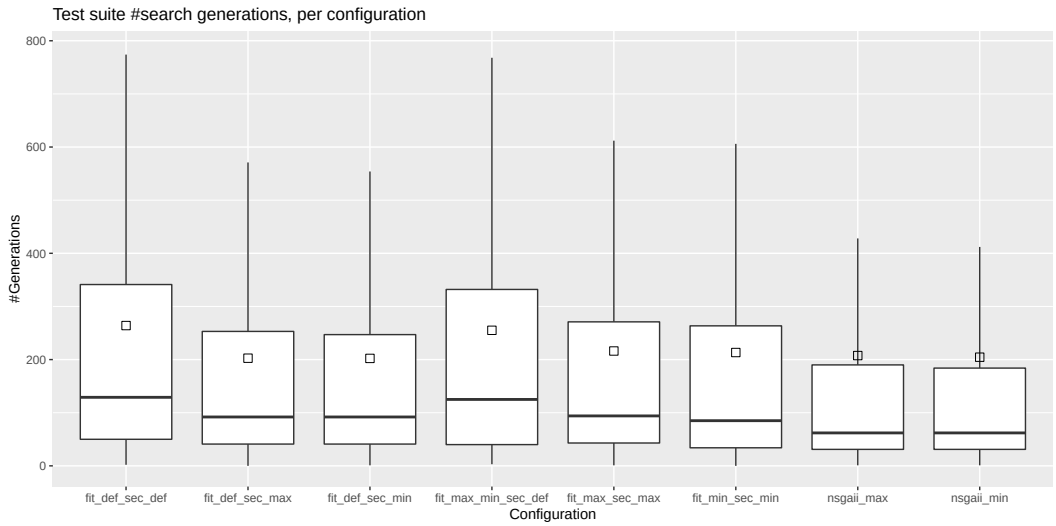

Figure 5.7: #Generations per configuration. Each data point is a test suite.

CUBTG configurations, standard MOSA goes through more generations for more than half of the classes in the test set. This is the most extreme when comparing to `fit_min_sec_min`, where `fit_def_sec_def` goes through more generations for 72/94 classes. When looking at the effect sizes for the classes where `fit_def_sec_def` goes through more generations than the CUBTG configurations, we see that they dominantly lie above 0.5 (i.e., large effect size), and that the effect size is close to or equal to 1 for a significant number of classes.

For the configurations including the maximum and minimum FFs, there are also quite some classes for which the CUBTG configuration goes through more generations. This ranges between 9 and 14 classes, 14 being for `fit_max_min_sec_def`. We have taken a closer look at the classes for which the number of generations is higher for `fit_max_min_sec_def`. It turns out that the classes for which is the case are classes that are very low in complexity, and they generally have been executed a lot according to the execution data. Also, the effect size compared to standard MOSA is pretty high (large in most cases). For the

Figure 5.8: Effect size of number of EvoSuite generations difference per class, per configuration. Only significant results are shown ($p < 0.05$). In each cell, data is shown for which the column configuration resulted in a higher value than the row configuration. The number of observations is shown in the upper-right corner of each cell.

	Conf. name	1	2	3	4	5	6	7	8
1	fit_def_sec_def		94	94	94	94	94	90	90
2	fit_def_sec_max	94		94	95	94	95	91	91
3	fit_def_sec_min	94	94		95	94	95	91	91
4	fit_max_min_sec_def	94	95	95		96	95	92	92
5	fit_max_sec_max	94	94	94	96		95	92	92
6	fit_min_sec_min	94	95	95	95	95		91	91
7	nsgaii_max	90	91	91	92	92	91		92
8	nsgaii_min	90	91	91	92	92	91	92	

Table 5.4: Number of classes for which data is available for comparing two configurations for comparing the number of EvoSuite generations.

CUBTG configurations including the added SOs, the effect sizes are also almost all large.

When comparing the CUBTG configurations pairwise, for about 25% (with small differences between configurations) of the classes the evolution goes through more generations than for the other configuration, in both directions. Between `fit_def_sec_max` and `fit_def_sec_min`, the number of classes for which there is a difference is the least (7/94 and 9/94), while it is the greatest where `fit_max_min_sec_def` has a higher number of generations (52/95, at most). For all of the pairwise comparisons among CUBTG configurations effect sizes are, again, predominantly large. When `fit_max_min_sec_def` has a higher number of generations, there are also some classes for which the effect size is close to 1. The results suggest that the maximum and minimum commonality score SOs have a larger effect on the number of generations than the corresponding FFs.

Suite size

Now let us take a look at the suite size metric. The suite size distribution per configuration is shown in Figure 5.9, pairwise comparison of the effect size across configurations is shown in Figure 5.10, and the total number of classes used for these pairwise comparisons is shown in Table 5.5. Outliers have been omitted from fig. 5.9 for readability.

These are the most important things to notice:

- `fit_max_min_sec_def` results in a higher suite size for about 75% of the classes, compared to the other CUBTG configurations (70/86 at most, compared to `fit_max_sec_max`). For most classes the effect size is large, and the distribution graph widens towards an effect size of 1. Compared to standard MOSA there are only a handful of classes with a significant difference.

- A similar statement can be made for `fit_def_sec_def`. It results in higher suite sizes for about (75%) of the classes compared to the CUBTG configurations, except for `fit_max_min_sec_def`, with a similar effect size pattern.

- All CUBTG configurations besides `fit_max_min_sec_def` result in lower suite sizes for a lot of classes compared to other MOSA configurations, particularly compared to

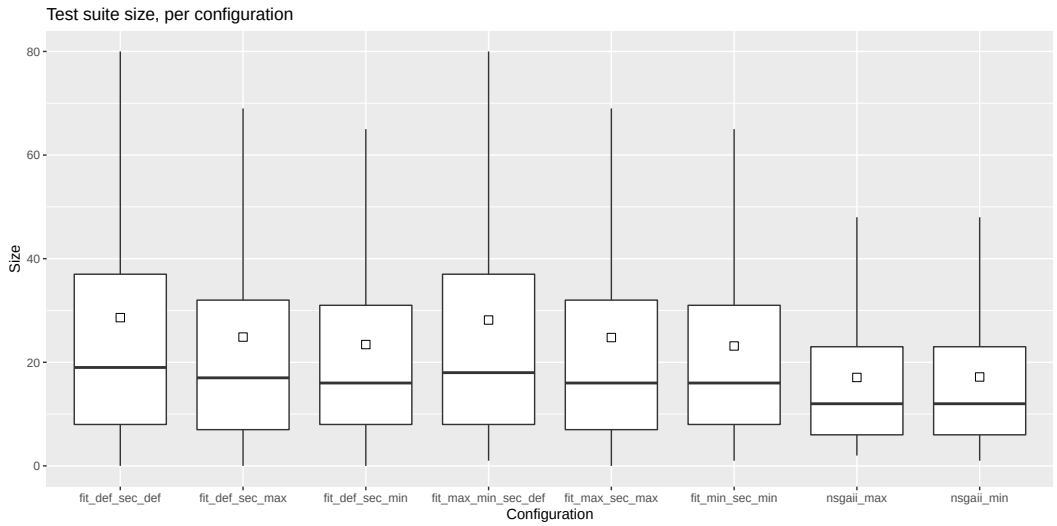

Figure 5.9: Suite size per configuration. Each data point is a test suite.

`fit_max_min_sec_def` and `fit_def_sec_def` (the effect size pattern was mentioned above). From this and the above points, it appears that the maximum and minimum commonality score SOs have a larger effect on the suite size than the corresponding FFs, like for the number of generations. For comparisons among these configurations, the effect size is around 0.5 for a lot of the classes, and the distribution gets smaller when going towards a larger effect size. Still, the effect size does not get lower than 0.25, which means it is still non-negligible in all cases.

	Conf. name	1	2	3	4	5	6	7	8
1	fit_def_sec_def		86	86	81	84	88	86	86
2	fit_def_sec_max	86		88	88	86	90	89	89
3	fit_def_sec_min	86	88		89	87	90	90	90
4	fit_max_min_sec_def	81	88	89		86	88	88	88
5	fit_max_sec_max	84	86	87	86		90	90	90
6	fit_min_sec_min	88	90	90	88	90		90	90
7	nsgaii_max	86	89	90	88	90	90		82
8	nsgaii_min	86	89	90	88	90	90	82	

Table 5.5: Number of classes for which data is available for comparing two configurations for comparing suite size.

Test case length

Finally, we will show the results for the test case length metric. The test case length distribution per configuration is shown in Figure 5.11, pairwise comparison of the effect size

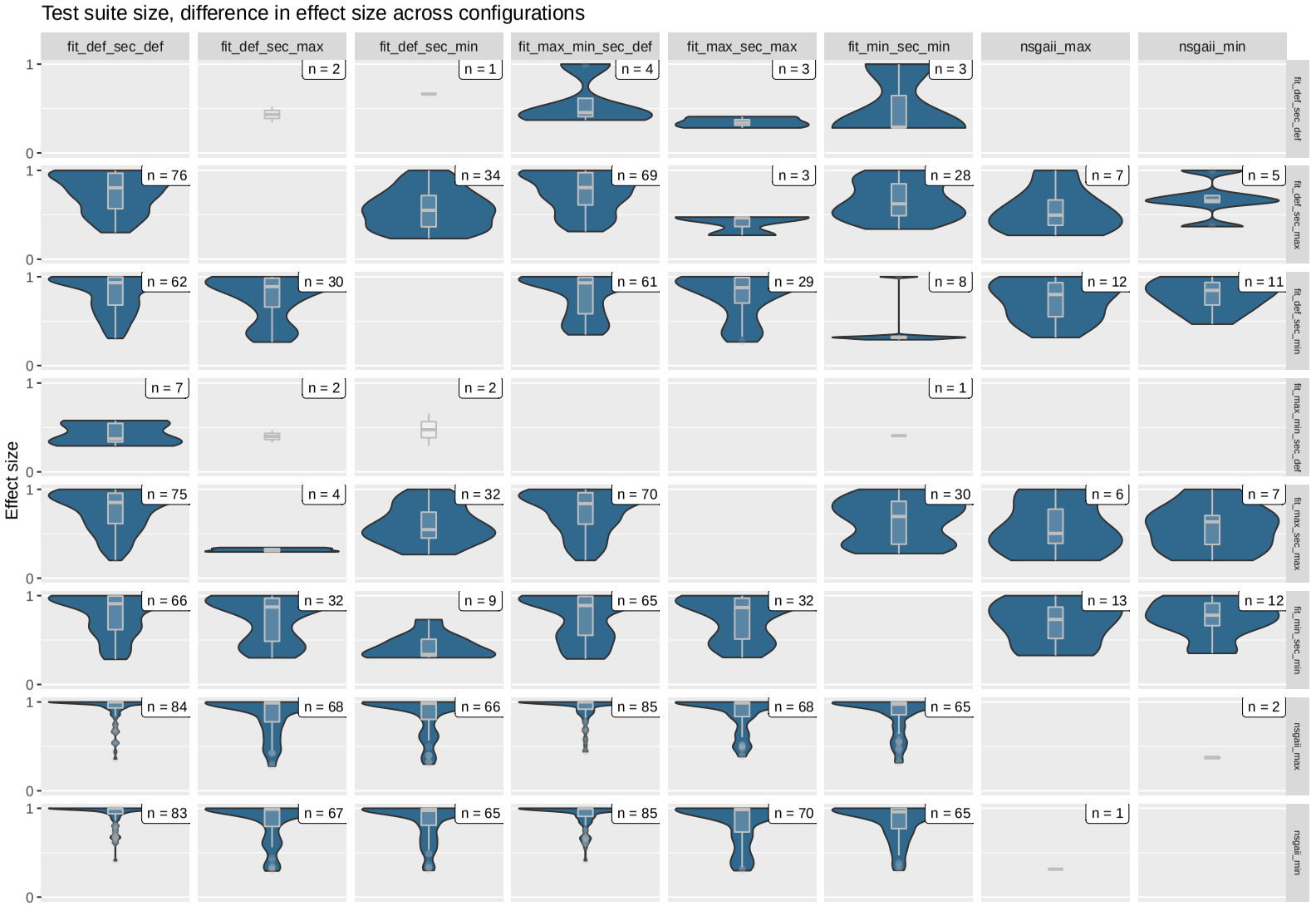

Figure 5.10: Effect size of suite size difference per class, per configuration. Only significant results are shown ($p < 0.05$). In each cell, data is shown for which the column configuration resulted in a higher value than the row configuration. The number of observations is shown in the upper-right corner of each cell.

across configurations is shown in Figure 5.12, and the total number of classes used for these pairwise comparisons is shown in Table 5.6. Outliers have been omitted from fig. 5.11 for readability.

The following are the most important observations that can be made from these:

- For almost all classes in the test set, standard MOSA (`fit_def_sec_def`) and `fit_max_min_sec_def` produce shorter test cases than all other MOSA configurations. The most extreme case in this sense is `fit_max_min_sec_def` compared to `fit_def_sec_max`, with 81/85 classes. The effect size of the difference is mostly medium, with some classes having a small, negligible, and large effect size. Note also that there are almost no classes for which these two configurations produce longer test cases compared to other MOSA configurations (at most 2/85 when comparing `fit_def_sec_def` to `fit_max_sec_max`). When comparing these configurations to each other, we see that `fit_max_min_sec_def` produces shorter test cases for 9/80 classes, and `fit_def_sec_def` for 4/80 classes, and the effect size is negligible for almost all of those.

- When comparing the remaining CUBTG configurations (`fit_def_sec_max`, `fit_def_sec_min`, `fit_max_sec_max`, `fit_min_sec_min`), there are the most and the largest differences in test case length between the maximum configuration variants and the minimum configuration variants. In the largest cases, there are 40/88 classes for which `fit_def_sec_min` produces shorter test cases than `fit_def_sec_max`, and also 40/88 classes for which it produces shorter test cases than `fit_max_sec_max`. Effect sizes are mostly small and negligible, but there are some classes with larger effect sizes (as can be seen from fig. 5.10). From this it appears that the minimum configuration variants tend to produce shorter test cases than the maximum configuration variants, in a large part of the cases. When comparing the maximum or minimum configurations amongst themselves, the number of classes for which there is a significant difference is a lot smaller, with the largest case being 18/86 classes where `fit_def_sec_max` produces shorter test case than `fit_max_sec_max`. The effect sizes are small or negligible, save for a few exceptions.

	Conf. name	1	2	3	4	5	6	7	8
1	fit_def_sec_def		84	87	80	85	87	81	82
2	fit_def_sec_max	84		88	85	86	88	85	86
3	fit_def_sec_min	87	88		88	88	89	87	87
4	fit_max_min_sec_def	80	85	88		86	88	83	84
5	fit_max_sec_max	85	86	88	86		88	87	87
6	fit_min_sec_min	87	88	89	88	88		88	88
7	nsgaii_max	81	85	87	83	87	88		81
8	nsgaii_min	82	86	87	84	87	88	81	

Table 5.6: Number of classes for which data is available for comparing two configurations for comparing test case length.

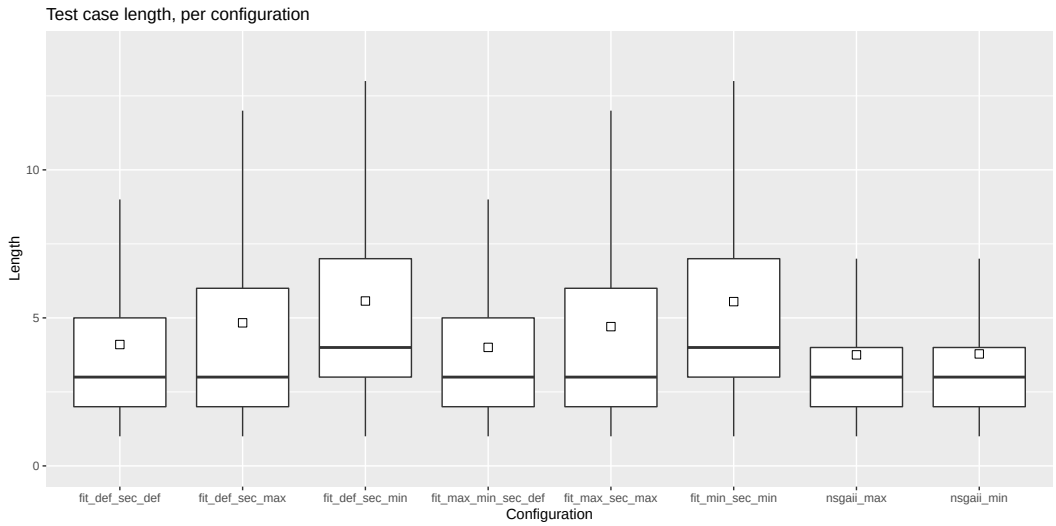

Figure 5.11: Test case length per configuration. Each data point is a test case.

5.3.2 Discussion

We saw that the CUBTG configurations generally performed the same or somewhat worse than the standard MOSA configuration for all of the standard metrics that were evaluated. A possible explanation for this is that covering the standard goals (branches, lines, etc.) becomes a bit less important when adding the CUBTG FFs and SOs. Using those distracts the search from covering some of the goals, leading to a lower coverage. One can imagine this could have an impact on the coverage of branches that require several steps of evolution to reach, for example.

This observation can also be related to the execution speed of the test generation. When using the CUBTG FFs and SOs, the time spent per evolution generation will be larger, because fitness values have to be computed for the CUBTG FFs and SOs. The fitness values are not easy to compute, because computing them requires looping through all branches in the class and retrieving the execution weight for those branches. This leads to the algorithm getting not as far through the search process when the search budget runs out (after 180s in this experiment).

The largest differences were observed in not the standard coverage metrics, but in properties of the EvoSuite search process: number of processed generations, the resulting size of the test suite, and generated test case size. For the number of generations, it was observed that it is generally less for the CUBTG generations compared to standard MOSA, although this difference was smaller for the `fit_max_min_sec_def` configuration. This observation is probably related to the efficiency argument above. If the search process takes longer to complete an evolution generation, less generations will be completed in total by the end of the search process.

The effect on number of generations seems to be larger for CUBTG SOs than for FFs. This could be caused by the fact that SOs are invoked every time a test case is found that

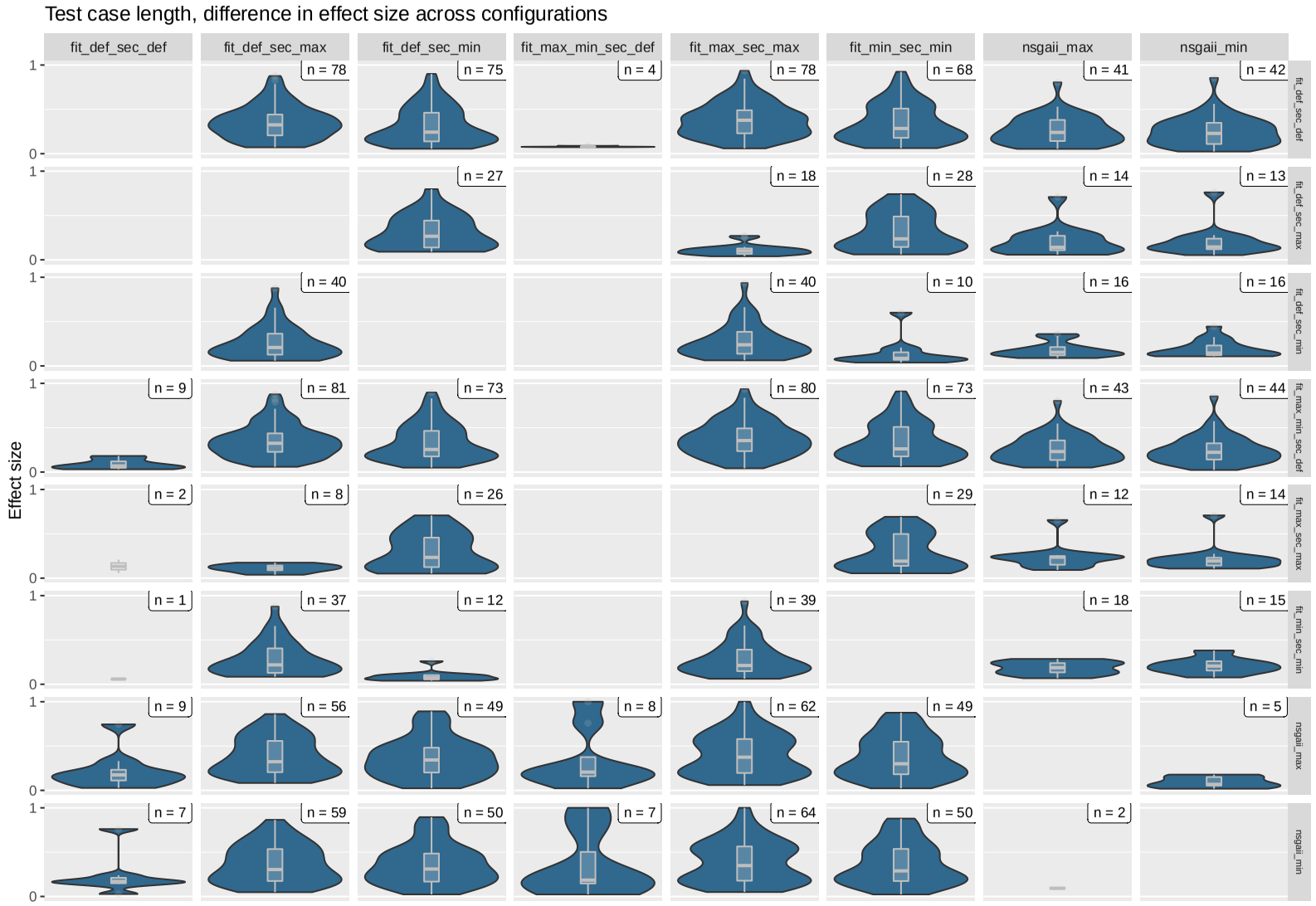

Figure 5.12: Effect size of test case length difference per class, per configuration. Only significant results are shown ($p < 0.05$). In each cell, data is shown for which the column configuration resulted in a higher value than the row configuration. The number of observations is shown in the upper-right corner of each cell.

covers the same goal (e.g., branch) as another test case already covers. Because we have disabled the EvoSuite option to remove a goal from the set of goals to be covered once it has been covered by one test case, the number of times two test cases have to be compared on the basis of the SO increases. On top of that, the SO value has to be computed for both test cases that are compared, so the total time needed to do the computation can add up fast. The FF values also have to be computed for each test case, but this computation is not being repeated, as opposed to the current implementation for SOs.

An observation that needs discussion is that when using the CUBTG FFs, there are some classes for which the number of generations processed gets a lot higher. We have looked at the class for which this was the most extreme, `org.jabref.model.strings.LatexToUnicodeAdapter`. For this class, the Cliff's delta effect size of `fit_max_min_sec_def` compared to standard MOSA was 0.982 (213 on average versus 385). It is a class with some static fields and 1 static method with a single branch (i.e., no branching statements). From the EvoSuite log files it is unclear where the large difference in number of generations processed comes from. There is no apparent difference during the evolution process, and the total time taken for the search is not different (as one would expect when setting a search budget limit). It could be that the generation process for these classes occurred when there was less load on the server on which the generation was done, although this seems unlikely because we tried to make sure to not use more resources than available. We are not aware of any other processes running on the server at the time of test generation. Another explanation could be that test cases that cause exceptions are generated during a lot of generations. One could imagine a generation ending quicker if this would be the case. But again, we have no direct reason to believe this is the case.

For suite size, one observation was that standard MOSA for a large majority of classes resulted in larger suite sizes than configurations using the CUBTG SOs. One reason for this could be the decreased efficiency when using those SOs, causing less goals to be covered, and in turn causing less test cases to be needed for covering them.

For test case length, the main observation is that the test case length is almost always higher for configurations where the CUBTG SOs were used, and that the effect is the largest for the minimum commonality score variants. The explanation for the general pattern is that the length SO, which is the only SO used by default in EvoSuite and prefers shorter test cases, is given a much smaller weight than the commonality score SOs during the experiment, as described in Section 3.2.1. This means a test case that is at least as long as the shortest one of the two is chosen, and hence the average will always be at least as high as when using the default length SO, and higher in almost all cases.

The reason for the effect being larger for the minimum configuration variants means that covering the branches with least execution weight requires taking longer paths through the tested classes on average, compared with paths for covering branches with higher execution weight. In other words, there are probably less cases in which the test case taking the path with least commonality score is also the test case that has the fewest statements.

5.3.3 Conclusion

The CUBTG configurations generally performed the same or somewhat worse than the standard MOSA configuration for all of the standard coverage metrics that were evaluated. As for the EvoSuite runtime metrics, using the CUBTG configurations generally causes EvoSuite to go through less evolution generations (for up to 72/94 classes), causes test suite size to be smaller, and causes longer test cases to be generated (for up to 81/85 classes).

5.4 RQ4: Efficiency for standard metrics

This section presents the results, discussion and conclusion for **RQ4**, which was "Does the usage of the added FFs and SOs affect the efficiency of the EvoSuite test generation?"

5.4.1 Results

To answer this question we looked at the evolution of the fitness value for the several standard fitness metrics over time. For only one of the metrics there seems to be a significant difference in the time it takes to converge to a stable value when comparing standard MOSA to the CUBTG configurations. The one metric for which a slight difference can be seen is output coverage. Of course, in several cases there is a difference in the final obtained value, as discussed for RQ3, but the way in which the value converges to that final value is not different, apart from very slight differences.

The evolution of output coverage during test generation is shown in Figure 5.13. Note that for all plots showing the evolution of the fitness values, there is a data point for every 5 seconds from the start. The value shown is the median of the best coverage value available during the generation over all generated test suites (i.e., for all classes and all experiment repetitions). The vertical black line indicates the point of 180 seconds from the start of the test generation, which is the configured search budget. For output coverage, all configurations evolve the same way, until the point at about 150 seconds into the generation. At that point, the `fit_def_sec_def` configuration still manages to increase the coverage somewhat until the end of the configuration. At approximately the end of the configuration, the `fit_def_sec_max` and `fit_def_sec_min` configurations manage to achieve a little bit of extra coverage. Each of those three configurations might not have converged to a final value at the point where the search budget runs out, while the other configurations seem to be converged.

For all other standard metrics, no significant differences between the configurations seem to be present. To illustrate this, let us look at the evolution of the branch coverage metric, as an example, in Figure 5.14. Indeed there seem to be no apparent differences between configurations for this metric in terms of time until convergence. Other differences in the evolution pattern are also not present. It seems that for all configurations, they are not finished converging after the search budget has run out. The evolution of the other standard metrics will not be discussed further here. Plots analogous to the one shown here for those other metrics can be found in Appendix B.

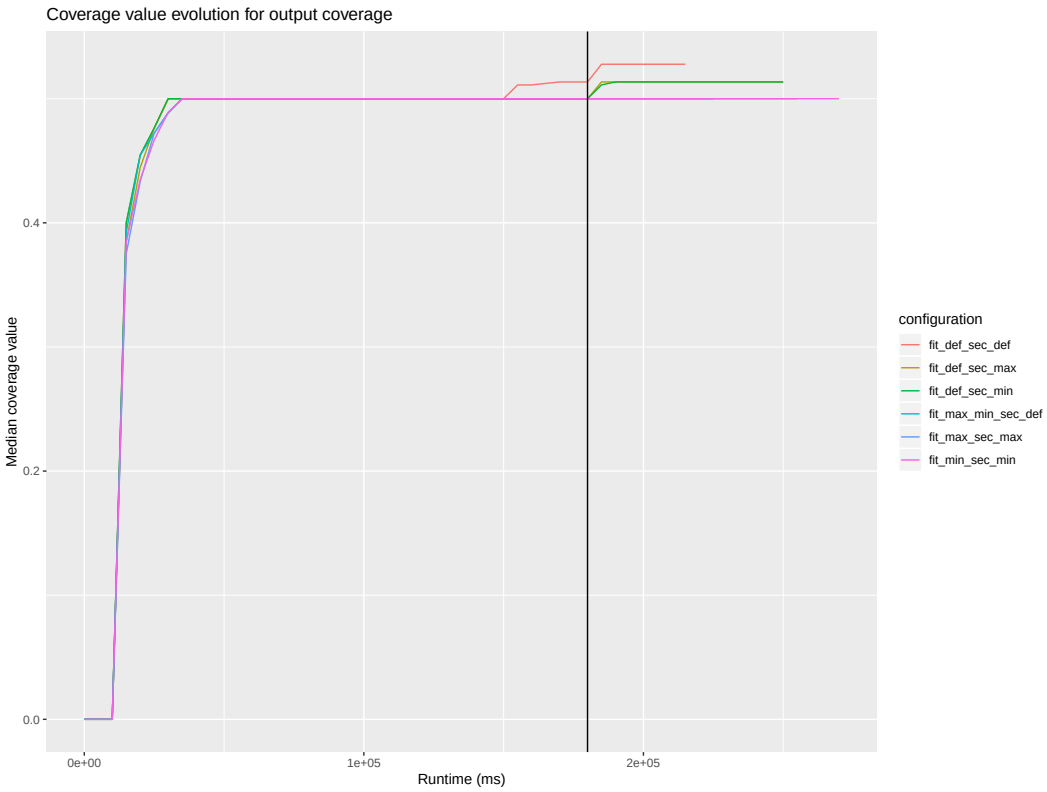

Figure 5.13: Evolution of the median test suite output coverage value for different configurations.

5.4.2 Discussion

It was observed that for the output coverage, the configurations that do not include the CUBTG FFs manage to improve coverage a little bit towards the end of the search process. This can be caused by the maximum and minimum commonality FFs steering the generation in those two specific directions, which makes it less likely for test generation to cover situations that require more specific sequences of method calls or method argument values in the generated test cases. As for why there only seems to be a somewhat significant difference in the evolution for output coverage: one can imagine that by steering the search in the specific direction covering often or not often executed branches, the code will be executed in similar ways multiple times, causing the same kinds of output for the code. On the other hand, if this would be the case, one would also expect to see a difference with respect to input coverage, which does not seem to be there (see Figure B.3).

5.4.3 Conclusion

There is no apparent difference in the evolution of fitness values during test generation between standard MOSA and configurations using the CUBTG methods. Only a small

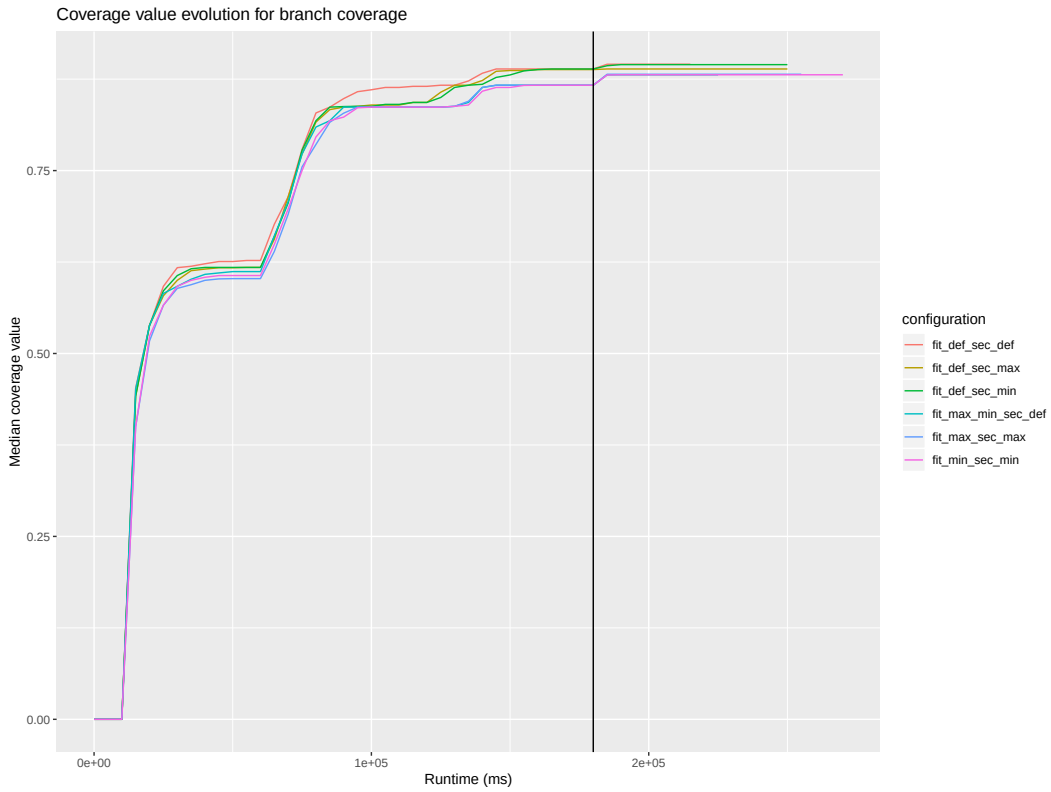

Figure 5.14: Evolution of the median test suite branch coverage value for different configurations.

difference for output coverage can be observed. Hence, there is no apparent significant performance difference from that perspective.

5.5 RQ5: Efficiency for CUBTG

This section presents the results, discussion and conclusion for **RQ5**, which was "How much time of EvoSuite test generation does it take for commonality score to converge?"

5.5.1 Results

As stated before, this question is very similar in nature to the previous one, but now we only look at the new commonality score metric. The evolution of the commonality score for the different configurations is shown in Figure 5.15. A few things can be noted from this figure.

- For the maximum configuration variants it seems like they have not necessarily converged to a final value after the search budget has been used up. This does not seem to be the case for the minimum configuration variants, for which the commonality score seems to have converged after about 140 seconds. For the default configuration

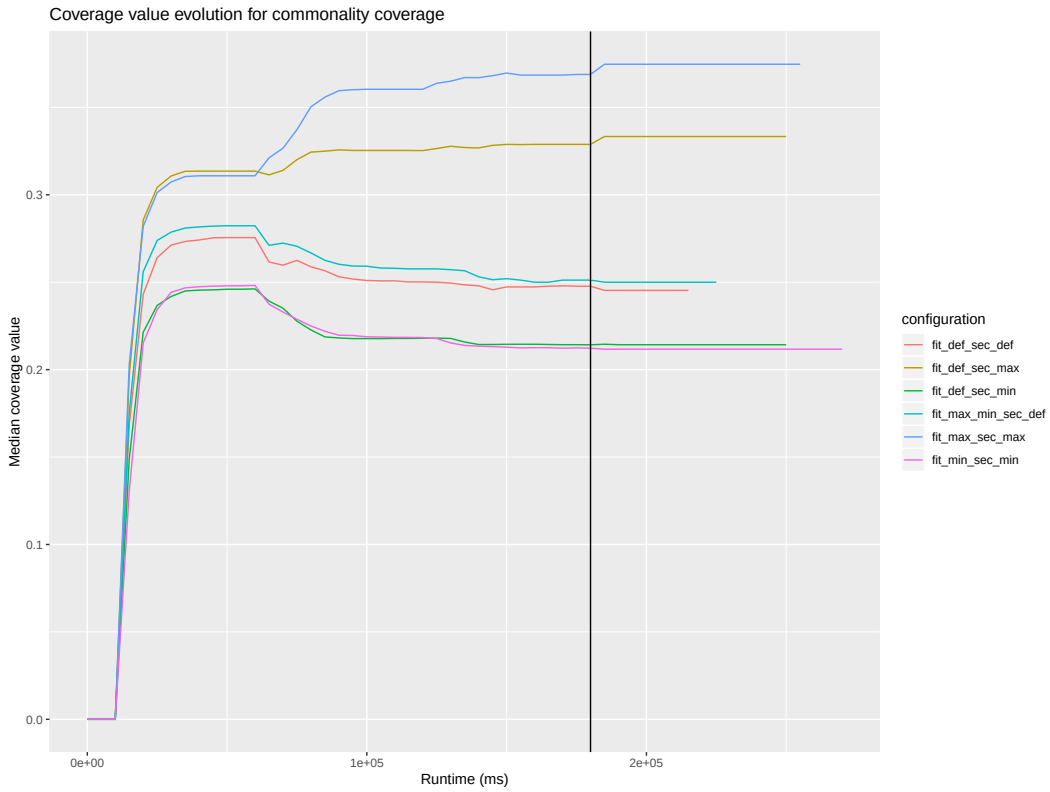

Figure 5.15: Evolution of the median test suite commonality score value for different configurations.

and `fit_max_min_sec_def`, it is unclear from the data whether they were finished converging.

- After about 100 seconds, the largest part of converging seems to have been done. The curve still rises a bit for maximum configuration variants and lowers a bit for the minimum variants, mixed, and default variants.

- After about 60 seconds, all configuration variants have risen to a point that is not too far apart for the different configurations. Only after that the maximum and minimum variants really start to diverge.

5.5.2 Discussion

Let us discuss two observations done in the results section. First, the maximum configuration variants require more time to converge to a stable commonality score value. As noted in the discussion for RQ3, it looks like shorter test cases often lie closer to covering less execution weight than more, and longer ones seem to be required to cover more execution weight. One can imagine that it takes more time for EvoSuite to generate a suitable longer

test case to cover goal branches than to do the same for goals that require fewer statements in the test case. Then it makes sense that converging to a stable commonality score value takes more time for the maximum variants.

And second, there is the pattern that can be seen from Figure 5.15 where the coverage values for the 6 configurations stay relatively close together until about 1 minute into the search process, after which they diverge more. This pattern makes sense if we follow how MOSA works and what has been discussed above for RQ3 and RQ4. In the first part of the test generation by MOSA, a lot of new test cases will be found to cover the set of goals, because no test cases are yet present that cover those goals. It does not matter as much for MOSA at that point how well the test case performs in terms of the SOs, because there are no or very few alternatives at that point of the search process. There are still differences between the configurations. If we look at the maximum configuration variants, for example, they end up with a slightly higher commonality score at about 1 minute into the search process. This is explainable by noticing that if multiple test cases covering the same goal are still found in this short period, the one with the highest commonality score will still be kept, and the maximum commonality FF will also exercise some influence on the direction the search goes. An analogous argument can be made for the minimum coverage variants. We see that the `fit_max_min_sec_def` configuration ends up a bit higher in coverage compared to standard MOSA (and it also stays that way throughout the search process). This has been discussed in the discussion for RQ1.

Then if we continue the search process, the focus shifts more to finding better solutions for the goals that are already covered. Of course, new solutions are still found, but the largest part of goals that will eventually be covered is already covered. If we take as an example again the maximum configuration variants, test cases already found will continuously be replaced by test cases covering the same goal, but having a higher execution count. The rate at which test cases are replaced will slowly dampen, because it becomes increasingly difficult to find test cases with a higher commonality score. Again, an analogous explanation can be given for the minimum configuration variants. We see that the coverage values for `fit_def_sec_def` and `fit_max_min_sec_def` also drop a bit going forward from the 1 minute mark. This can again be attributed to the theory that test cases with lower commonality score generally contain less statements. Because these two configurations use the default SO, which favors test cases with fewer statements, commonality score will drop a bit as shorter test cases are found.

5.5.3 Conclusion

The maximum commonality score configuration variants take longest to converge, as commonality score is still converging for them when the search budget of 180 seconds has run out. The minimum configuration variants are done converging after about 140 seconds. This is also the case for the configurations using the default SO, but the pattern is less clear there. After about 100 seconds, most of the convergence has been done for all configurations, except for smaller improvements. 60 seconds is a too short time to use the CUBTG FFs and SOs, because they comparatively do not affect the search process that much until after that point.

5.6 Threats to validity

Internal validity EvoSuite test generation is inherently based on randomness (as are all techniques based on evolution). To make the effect of this randomness on the results reasonably small, we repeated the test suite generation 30 times, per configuration, per class.

Characteristics of the server on which the experiments have been performed have influenced the results of the experiment. For example, a faster or slower CPU would have probably caused higher and lower resulting coverage values, respectively. Also, because the server was not being used exclusively for this experiment, other processes may have been going on during the test generation or PIT score computation, leading to unfair comparisons between results.

We tried to use parameters for EvoSuite that were as close as possible to the defaults used in practice and in other research. Previous research suggests that this is a good compromise between optimality of the parameters and time spent looking for optimal parameters, although it may lead to poor results in some cases [5, 28]. For the added parameters related to CUBTG test generation, we tried to use values that would be reasonable to use in practice. However, note that as this was the first experiment using the newly implemented additions to EvoSuite, there was no baseline configuration to go from, and the parameters chosen were based on the limited experience of running the tool on small examples.

One other aspect to consider is the correctness of the implementation of the CUBTG methods as described in Chapter 3. As goes for all software, the implementation is likely not bug-free. To reduce the chance of bugs influencing the evaluation results, we have tested the implemented FFs and SOs to confirm they work as intended. Results that were obviously invalid (e.g., a negative FF value) were removed from the data set.

External validity The execution data needed for the experiment was gathered from a small number of people in a relatively structured manner, as opposed to from actual real-life executions of JabRef. As described in Chapter 4, the test participants were briefly told how to use JabRef, and were given approximately 5 minutes to use it on some task they might perform with it. The given explanation and the relatively structured and time limited way in which this was executed may have significantly influenced the execution data on which test generation was based. Additionally, only a small number of people (5) participated in the collection of execution data, and one of them was the author. That said, we think that given the setup used to gather execution data, using a larger group would not have made a significant difference because of the structured and time limited nature of the method of gathering the data.

Perhaps more importantly, the experiment was executed only on JabRef, which means the results are probably biased towards JabRef. To offset this, we used a large number of classes in the test set, and tried to include a reasonable variety of classes (see Chapter 4). Though of course, the classes that we were able to test on were only the ones for which execution data was available, which is still a more limited number of classes than all classes available in JabRef. To confirm the results presented here, a repetition of the evaluation on other projects have to be done.

Construct validity For measuring the effect of using the CUBTG test generation methods on standard coverage metrics, we used metrics that are used as defaults in other research and in practice, as recommended in [4]. The goal of this is for results to be representative for comparing to other research and executions in practice.

For commonality score, the same metric was used to compute coverage during the evaluation as for computing coverage within the FFs and SOs during EvoSuite test generation. The metric itself was invented during this study, so there are no other sources stating its usefulness that we can compare against.

As for measuring ability of finding faults in software, PIT was used as described in Chapter 4. The main concern here is that errors seeded by PIT (or any other mutation tool that could have been used) are not real world errors, although they should be able to quite closely represent them [16].

Conclusion validity To ensure conclusion were only drawn based on statistically significant results, the Mann–Whitney–Wilcoxon (MWW) test was used to determine significance of the results. We decided to view a result as significant if $p < 0.05$, a value that is deemed appropriate in other research [16]. To determine the magnitude of results, we used the Cliff's delta measure. This measure can be computed directly from the Vargha and Delaney A (VD.A) measure [32], which is another often used metric [16].

For answering research questions RQ4 and RQ5 we did not make use of statistical tests, because the answers were based on subjective observation of graphs showing the evolution of fitness values. A more detailed observation would be necessary for more quantitative answers.

Chapter 6

Conclusion and future work

This chapter will conclude the thesis. A summary of the methods introduced and of our findings is given in Section 6.1. Section 6.2 described the implications of the findings from the evaluation. Finally, Section 6.3 suggests possible future work that may be interesting to perform.

6.1 Summary

In this thesis we have described common and uncommon behavior test generation (CUBTG) methods for generating tests for common and uncommon behaviors in the class under test (CUT), based on the execution counts of code branches extracted from real world log data. CUBTG as described here is to be used in a genetic algorithm, and has primarily been designed for use with the many-objective sorting algorithm (MOSA) in this work.

The aim of CUBTG is to guide the search for tests in the direction of more commonly or uncommonly executed code branches, while still trying to satisfy traditional coverage goals, like line or branch coverage. In other words, the aim is not to satisfy a new goal, or to exactly replicate user behavior, but instead to influence the method call sequences that are used in the tests generated for existing goals.

For this purpose, we defined the concept of *commonality score* for test cases and test suites, which intuitively describes how commonly used in practice (according to log data) the branches are that are being executed by the test case or test suite, relative to other branches in the CUT.

We created fitness functions (FFs) for use in MOSA that use this commonality score to quantify how commonly or uncommonly executed the code branches that generated tests execute are. These aim to influence the search direction of MOSA, using the way that algorithm uses non-dominated fronts to determine the search direction for new test cases.

Additionally, we defined two new secondary objectives (SOs), which are used in MOSA to decide which test case to keep if multiple test cases satisfy the same goal, or otherwise have the same sub-optimal FF value for the goal. They use the commonality score to keep the test case or suite that executes either the most common or most uncommon behaviors. To be able to use these new SOs along with the existing SO (which prefers shorter test

cases), we devised a method to combine them by allowing weights to be configured when multiple SOs are used at once.

We implemented these FFs and SOs, and the method to combine SOs, in EvoSuite, and used this implementation to evaluate the CUBTG methods. We obtained execution counts from a small group of people using the open-source Java application JabRef for about 5 minutes, and executed EvoSuite with 8 different configurations, each using a different combination of FFs and SOs.

We found that CUBTG managed to cover more common behaviors than plain MOSA in 75% of the cases, and more uncommon behaviors in 60% of the cases (**RQ1**). We also performed mutation testing on the generated test suites, and found that CUBTG performed the same or worse than plain MOSA in most cases. Though, there were a few exceptions in which CUBTG managed to find some mutants by using method sequences that plain MOSA did not find (**RQ2**).

CUBTG generally performs the same or a little bit worse in terms of standard coverage metrics, causes EvoSuite to go through less generations, causes test suite sizes to be smaller, and causes longer test cases to be generated (**RQ3**). There does not appear to be a significant effect on the efficiency of EvoSuite test generation (**RQ4**), but at least 100 seconds were needed for the commonality score to converge for the most part during our evaluation (**RQ5**).

In summary, this thesis makes the following contributions to the field:

- The novel CUBTG method for test generation, which uses newly devised FFs and SOs and the concept of commonality score to influence the unit test generation process in MOSA towards generating tests that exercise more common or uncommon behaviors.
- An evaluation of this CUBTG method on the JabRef application, evaluating its effect on the commonality score, fault revealing capability, and standard coverage metrics obtained by generated test cases, along with an evaluation of the effect of CUBTG on the efficiency of EvoSuite and the efficiency of CUBTG itself.

6.2 Implications

Our results show that in a large majority of the cases, using CUBTG does not have a large negative (nor positive) effect on standard coverage metrics, like branch coverage, and using it does not impact efficiency. The degree to which common or uncommon behaviors (depending on the configuration) are used increases by a large amount in most cases. This means that using CUBTG is a good idea in most cases if one wants to generate test cases that exercise parts of the CUT that are commonly or uncommonly executed in practice, without having to worry about performance declining in terms of traditional metrics. Note that CUBTG is not designed to replicate exact user behavior. Other methods are better suited for that, like the one introduced by Wang et al. [33]. CUBTG can only be used to steer test generation towards common behaviors, not to replicate them.

It was also shown that CUBTG does not work particularly well (nor particularly bad) for generating tests that perform well during mutation testing. There are cases in which CUBTG

generates method sequences in test cases that are not generated when using standard MOSA. Those test cases can be valuable for covering code in ways that are not found during standard test generation, and our results show that it can increase mutation coverage. There are cases, though, in which CUBTG performs quite bad in terms of finding mutants. In those cases it would not be advisable to use CUBTG for test generation, or at least not solely.

6.3 Future work

These are some suggestions we have for future work:

- The fault finding capability of CUBTG has only been tested here in terms of mutation testing using PIT. It may be interesting to see how CUBTG performs for finding real world faults. This could be evaluated, for example, by using a database like Defects4J [1].
- Related to the previous point, one could also expect more bugs to be present in uncommonly used parts of a system. It would be interesting to see if test generated using CUBTG would be able to uncover those more effectively than other test generation approaches.
- An evaluation on more, and a wider variety of, applications would be beneficial. The evaluation in this thesis used JabRef as the only subject. Even though we tried to choose a wide variety of classes, stronger conclusions could be drawn from a larger evaluation.
- Similarly to the previous point, an evaluation using more, and more realistic log data would improve the external validity of the results. For the evaluation in this thesis, log data was obtained from a fairly restricted test and from a small group of people. Log data obtained from a larger group of people and from real world executions over a longer period of time would make the results stronger.
- We saw in our results that there are some cases in which CUBTG produces method sequences in test case which are not generated using standard MOSA (see section 5.2.2). It may be interesting to further research the differences in method sequences, for example if any patterns can be found in it when looking at a larger number of cases in which this happens, and if there is a general difference in the generated method sequences at all.
- The current computation of commonality score treats all branches as separate units and does not consider the connections between branches and their position in the control flow graph (CFG). Basing the computation of commonality score based on paths through the CFG instead of on branches might improve the accuracy of the commonality score. It might make the computation more complex and less efficient, however.

[1] https://github.com/rjust/defects4j

- The results show a small, though non-negligible amount of cases in which CUBTG performs worse than standard MOSA in terms of standard metrics, like line or branch coverage, or mutation score. It might be interesting to look at the characteristics of the classes for which this is the case, and at methods that may be used to mitigate this. For example, one may look at ways to dynamically disable CUBTG if it appears that it significantly, negatively affects generated tests.

Bibliography

[1] Frances E. Allen. Control flow analysis. *ACM SIGPLAN Notices*, 5(7):1–19, jul 1970. doi: 10.1145/390013.808479.

[2] Andrea Arcuri. EvoMaster: Evolutionary multi-context automated system test generation. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, apr 2018. doi: 10.1109/icst.2018.00046.

[3] Andrea Arcuri. RESTful API automated test case generation with EvoMaster. *ACM Transactions on Software Engineering and Methodology*, 28(1):1–37, feb 2019. doi: 10.1145/3293455.

[4] Andrea Arcuri and Lionel Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, nov 2012. doi: 10.1002/stvr.1486.

[5] Andrea Arcuri and Gordon Fraser. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, feb 2013. doi: 10.1007/s10664-013-9249-9.

[6] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology*, 104:207–235, dec 2018. doi: 10.1016/j.infsof.2018.08.010.

[7] Norman Cliff. *Ordinal Methods for Behavioral Data Analysis*. Taylor & Francis Ltd., 2014. ISBN 9781317781431. URL `https://www.ebook.de/de/product/22385281/norman_cliff_ordinal_methods_for_behavioral_data_analysis.html`.

[8] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[9] Chartchai Doungsa-ard, Keshav P Dahal, M Alamgir Hossain, and Taratip Suwannasart. An automatic test data generation from uml state diagram using genetic algorithm. *Proceedings of the Second International Conference on Software Engineering Advances (ICSEA 2007)*, 2007.

[10] Jason Flinn. lprof: A non-intrusive request flow profiler for distributed systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, pages 629–644, Berkeley, CA , USA, 2014. USENIX Association. ISBN 9781931971164.

[11] Gordon Fraser and Andrea Arcuri. Evolutionary generation of whole test suites. In *2011 11th International Conference on Quality Software*. IEEE, jul 2011. doi: 10.1109/qsic.2011.19.

[12] Gordon Fraser and Andrea Arcuri. A large-scale evaluation of automated unit test generation using evosuite. *ACM Trans. Softw. Eng. Methodol.*, 24(2), December 2014. ISSN 1049-331X. doi: 10.1145/2685612. URL https://doi.org/10.1145/2685612.

[13] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation - PLDI '08*. ACM Press, 2008. doi: 10.1145/1375581.1375607.

[14] Myles Hollander, Douglas A. Wolfe, and Eric Chicken. *Nonparametric Statistical Methods*. John Wiley & Sons, 2013. URL https://www.ebook.de/de/product/21853168/myles_hollander_douglas_a_wolfe_eric_chicken_nonparametric_statistical_methods.html.

[15] WH Jessop, J Richard Kane, S Roy, and JM Scanlon. Atlas-an automated software testing system. In *Proceedings of the 2nd international conference on Software engineering*, pages 629–635. IEEE Computer Society Press, 1976.

[16] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. ACM Press, 2014. doi: 10.1145/2635868.2635929.

[17] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, may 2007. doi: 10.1109/icse.2007.41.

[18] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, dec 1976. doi: 10.1109/tse.1976.233837.

[19] Phil McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, mar 2011. doi: 10.1109/icstw.2011.100.

[20] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Reformulating branch coverage as a many-objective optimization problem. In *2015 IEEE 8th international conference on software testing, verification and validation (ICST)*, pages 1–10. IEEE, 2015.

[21] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information and Software Technology*, 104:236–256, dec 2018. doi: 10.1016/j.infsof.2018.08.009.

[22] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. *IEEE Transactions on Software Engineering*, 44(2):122–158, feb 2018. doi: 10.1109/tse.2017.2663435.

[23] Rong-Zhi Qi, Zhi-Jian Wang, and Shui-Yan Li. A parallel genetic algorithm based on spark for pairwise test suite generation. *Journal of Computer Science and Technology*, 31(2):417–427, mar 2016. doi: 10.1007/s11390-016-1635-5.

[24] C. V. Ramamoorthy and S. F. Ho. Testing large software with automated software evaluation systems. In *Proceedings of the international conference on Reliable software -*. ACM Press, 1975. doi: 10.1145/800027.808461.

[25] Aurora Ramírez, José Raúl Romero, and Sebastián Ventura. A survey of many-objective optimisation in search-based software engineering. *Journal of Systems and Software*, 149:382–395, mar 2019. doi: 10.1016/j.jss.2018.12.015.

[26] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. VUzzer: Application-aware evolutionary fuzzing. In *Proceedings 2017 Network and Distributed System Security Symposium*. Internet Society, 2017. doi: 10.14722/ndss.2017.23404.

[27] Romano, Jeffrey D. Kromrey, Jesse Coraggio, and Jeff Skowronek. Appropriate statistics for ordinal level data: Should we really be using t-test and cohen'sd for evaluating group differences on the nsse and other surveys. In *annual meeting of the Florida Association of Institutional Research*, pages 1–33, 2006.

[28] Abdel Salam Sayyad, Katerina Goseva-Popstojanova, Tim Menzies, and Hany Ammar. On parameter tuning in search based software engineering: A replicated empirical study. In *2013 3rd International Workshop on Replication in Empirical Software Engineering Research*. IEEE, oct 2013. doi: 10.1109/reser.2013.6.

[29] Daan Schipper, Maurício Aniche, and Arie van Deursen. Tracing back log data to its log statement: from research to practice. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 545–549. IEEE, 2019.

[30] Koushik Sen and Gul Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, pages 419–423. Springer Berlin Heidelberg, 2006. doi: 10.1007/11817963_38.

[31] Chayanika Sharma, Sangeeta Sabharwal, and Ritu Sibal. A survey on software testing techniques using genetic algorithm. *International Journal of Computer Science*, 2014.

[32] András Vargha and Harold D. Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, jun 2000. doi: 10.3102/10769986025002101.

[33] Qianqian Wang and Alessandro Orso. Mimicking user behavior to improve in-house test suites. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, may 2019. doi: 10.1109/icse-companion.2019.00133.

[34] Qianqian Wang, Yuriy Brun, and Alessandro Orso. Behavioral execution comparison: Are tests representative of field behavior? In *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, mar 2017. doi: 10.1109/icst.2017.36.

[35] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. In *2010 IEEE Symposium on Security and Privacy*. IEEE, 2010. doi: 10.1109/sp.2010.37.

[36] Shang Xiang. Fit2crash: Specialising fitness functions for crash reproduction. Master's thesis, TU Delft, 2020. URL `http://resolver.tudelft.nl/uuid:26da088e-25e1-4de4-bfc2-6935e32646ab`.

[37] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*. ACM Press, 2009. doi: 10.1145/1629575.1629587.

Appendix A

Results per class

This appendix contains detailed results per class for a few cases corresponding to results shown in Chapter 5.

	class	complexity	loc	max. exec. count	fit_def_sec_def	fit_max_sec_max	significance	effect size	magnitude
1	org.jabref.cli.JabRefCLI	36	146	56	0.00798	0.129	3.55e-177	0.806	large
2	org.jabref.Globals	14	77	2614	0.0786	0.112	0.048	0.559	negligible
3	org.jabref.JabRefGUI	43	183	574	0.239	0.374	0.000541	0.606	small
4	org.jabref.logic.autosaveandbackup.BackupManager	21	117	153	0.208	0.221	0.289	0.523	negligible
5	org.jabref.logic.bibtex.BibEntryWriter	28	124	8691	0.0952	0.133	8.07e-08	0.596	small
6	org.jabref.logic.bibtex.comparator.BibDatabaseDiff	23	94	23	0.0381	0.0759	0.000621	0.565	negligible
7	org.jabref.logic.bibtex.comparator.EntryComparator	28	103	60	0.516	0.529	0.0248	0.544	negligible
8	org.jabref.logic.bibtex.comparator.FieldComparator	29	119	808	0.247	0.307	1.58e-20	0.684	medium
9	org.jabref.logic.bibtex.DuplicateCheck	55	245	7599	0.201	0.296	2.76e-14	0.637	small
10	org.jabref.logic.bibtex.FieldContentParser	4	31	3201	0.427	0.472	1.61e-05	0.612	small
11	org.jabref.logic.bibtex.LatexFieldFormatter	67	206	50661	0.154	0.186	8.32e-11	0.618	small
12	org.jabref.logic.bibtex.LatexFieldFormatterPreferences	6	27	2183	0.152	0.372	1.01e-20	0.756	large
13	org.jabref.logic.bibtexkeypattern.BibtexKeyGenerator	31	140	341	0.142	0.19	1.04e-30	0.666	medium
14	org.jabref.logic.bibtexkeypattern.BracketedPattern	245	1016	1605	0.0518	0.122	1.06e-77	0.665	small
15	org.jabref.logic.citationstyle.CitationStyle	32	148	39564	0.0509	0.101	4.15e-53	0.778	large
16	org.jabref.logic.exporter.BibDatabaseWriter	42	175	145	0.0619	0.087	8.87e-07	0.555	negligible
17	org.jabref.logic.exporter.BibtexDatabaseWriter	27	127	145	0.274	0.302	0.0919	0.525	negligible
18	org.jabref.logic.exporter.TemplateExporter	36	235	105	0.919	0.919	0.956	0.499	negligible
19	org.jabref.logic.exporter.VerifyingWriter	8	34	1762	0.322	0.457	2.35e-05	0.615	small
20	org.jabref.logic.importer.fetcher.ArXiv	33	264	87	0.163	0.391	1e-51	0.762	large
21	org.jabref.logic.importer.fetcher.AstrophysicsDataSystem	15	99	132	0.128	0.284	2.74e-66	0.788	large
22	org.jabref.logic.importer.fetcher.CrossRef	21	110	129	0.5	0.833	5.98e-08	0.666	medium
23	org.jabref.logic.importer.fetcher.GoogleScholar	26	126	117	0.13	0.263	7.62e-21	0.729	medium
24	org.jabref.logic.importer.fetcher.IacrEprintFetcher	29	145	20	0.124	0.237	0.000617	0.613	small
25	org.jabref.logic.importer.Importer	21	122	497	0.191	0.311	4.3e-13	0.598	small
26	org.jabref.logic.importer.ParserResult	33	138	130	0.223	0.306	4.89e-55	0.668	medium
27	org.jabref.logic.importer.util.MetaDataParser	29	128	193	0.311	0.318	0.00279	0.547	negligible
28	org.jabref.logic.importer.WebFetchers	14	96	13	0.562	0.679	3.76e-05	0.585	small
29	org.jabref.logic.integrity.AbbreviationChecker	4	16	2295	0.445	0.483	0.167	0.535	negligible
30	org.jabref.logic.integrity.BracketChecker	7	26	1205	0.353	0.458	0.0747	0.55	negligible
31	org.jabref.logic.integrity.PersonNamesChecker	9	25	4590	0.432	0.432	1	0.5	negligible
32	org.jabref.logic.journals.Abbreviation	19	69	556488	0.376	0.495	1.25e-103	0.775	large
33	org.jabref.logic.journals.AbbreviationParser	17	72	92754	0.198	0.221	0.297	0.52	negligible
34	org.jabref.logic.journals.JournalAbbreviationRepository	15	58	139122	0.265	0.336	2.16e-47	0.693	medium
35	org.jabref.logic.layout.AbstractParamLayoutFormatter	12	54	4502	0.384	0.513	0.0094	0.582	small
36	org.jabref.logic.layout.format.Authors	63	280	1792	0.252	0.28	6.6e-17	0.605	small
37	org.jabref.logic.layout.format.HTMLChars	44	176	87993	0.417	0.551	1.86e-05	0.623	small
38	org.jabref.logic.layout.Layout	24	93	4283	0.198	0.273	2.05e-13	0.632	small
39	org.jabref.logic.layout.LayoutEntry	169	497	35976	0.155	0.239	7.61e-17	0.618	small
40	org.jabref.logic.layout.LayoutHelper	82	266	74500	0.118	0.131	0.0979	0.548	negligible
41	org.jabref.logic.net.ProxyPreferences	29	96	16	0.266	0.362	2.13e-37	0.663	small
42	org.jabref.logic.net.URLDownload	40	206	82	0.925	0.94	0.965	0.5	negligible
43	org.jabref.logic.openoffice.OOBibStyle	142	696	672	0.0449	0.0464	0.353	0.508	negligible
44	org.jabref.logic.protectedterms.ProtectedTermsLoader	31	124	21	0.419	0.431	7.83e-09	0.576	small
45	org.jabref.logic.protectedterms.ProtectedTermsParser	14	72	6615	0.101	0.118	0.139	0.524	negligible
46	org.jabref.logic.TypedBibEntry	7	44	1047	0.618	0.679	0.0288	0.571	negligible
47	org.jabref.logic.util.io.FileBasedLock	17	97	73	0.576	0.576	1	0.5	negligible
48	org.jabref.logic.util.io.FileUtil	42	239	591	0.294	0.424	0.00186	0.553	negligible
49	org.jabref.logic.util.strings.StringLengthComparator	3	12	102578	0.736	0.741	0.846	0.506	negligible
50	org.jabref.logic.util.UpdateField	20	107	93	0.471	0.674	6.86e-17	0.606	small
51	org.jabref.logic.util.Version	47	215	622	0.254	0.336	1.87e-33	0.642	small
52	org.jabref.model.bibtexkeypattern.AbstractBibtexKeyPattern	21	97	167	0.743	0.78	0.00084	0.54	negligible
53	org.jabref.model.ChainNode	15	111	274	0.0398	0.268	5.35e-49	0.729	medium
54	org.jabref.model.database.BibDatabase	83	443	11410	0.0676	0.28	4.05e-306	0.789	large
55	org.jabref.model.database.BibDatabaseContext	45	193	7652	0.034	0.119	1.91e-128	0.765	large
56	org.jabref.model.entry.Author	60	270	1191	0.227	0.346	2.94e-138	0.821	large
57	org.jabref.model.entry.AuthorList	77	371	4717	0.459	0.535	4.44e-22	0.583	small
58	org.jabref.model.entry.AuthorListParser	84	328	496	0.351	0.373	0.0103	0.544	negligible
59	org.jabref.model.entry.BibEntry	128	618	103656	0.185	0.189	4.23e-07	0.529	negligible
60	org.jabref.model.entry.BiblatexEntryTypes	2	492	510	0.298	0.358	5.6e-05	0.583	small
61	org.jabref.model.entry.BibtexEntryTypes	2	193	133	0.361	0.37	0.0959	0.522	negligible
62	org.jabref.model.entry.BibtexSingleField	21	98	94328	0.0368	0.317	2.7e-178	0.928	large
63	org.jabref.model.entry.Date	26	101	187	0.0451	0.22	3.59e-113	0.775	large
64	org.jabref.model.entry.event.EntryEvent	4	28	2837	0.418	0.538	6.47e-08	0.778	large
65	org.jabref.model.entry.event.FieldChangedEvent	14	78	2601	0.489	0.564	1.32e-28	0.74	large
66	org.jabref.model.entry.FieldName	8	171	423	0.466	0.64	1.01e-18	0.708	medium
67	org.jabref.model.entry.InternalBibtexFields	52	321	29207	0.484	0.59	1.65e-21	0.621	small
68	org.jabref.model.entry.KeywordList	37	137	301	0.32	0.434	1.4e-54	0.658	small
69	org.jabref.model.entry.Month	23	120	216	0.0752	0.438	5.38e-56	0.769	large
70	org.jabref.model.entry.specialfields.SpecialField	15	57	1112	0.206	0.706	6.04e-59	0.858	large
71	org.jabref.model.EntryTypes	36	182	2368	0.267	0.583	2.36e-68	0.754	large
72	org.jabref.model.FieldChange	22	75	1932	0.446	0.632	2.26e-37	0.676	medium
73	org.jabref.model.groups.AbstractGroup	31	129	1476	0.0417	0.288	1.14e-216	0.821	large
74	org.jabref.model.groups.AllEntriesGroup	6	24	1476	0.112	0.339	3.37e-27	0.767	large
75	org.jabref.model.groups.GroupTreeNode	54	229	1476	0.0714	0.22	4.78e-179	0.77	large
76	org.jabref.model.metadata.MetaData	50	232	7679	0.0256	0.145	1.48e-148	0.726	medium
77	org.jabref.model.metadata.SaveOrderConfig	19	142	16	0.567	0.683	4.51e-09	0.59	small
78	org.jabref.model.search.rules.SentenceAnalyzer	12	43	6256	0.359	0.388	0.124	0.546	negligible
79	org.jabref.model.strings.StringUtil	154	547	2183	0.284	0.426	2.6e-113	0.667	medium
80	org.jabref.preferences.JabRefPreferences	168	1368	16694	0.119	0.119	0.716	0.496	negligible
81	org.jabref.preferences.PreviewPreferences	8	81	2110	0.122	0.278	8.1e-99	0.832	large

Table A.1: Commonality coverage for `fit_def_sec_def` and `fit_max_sec_max` compared per class

	class	complexity	loc	max. exec. count	fit_def_sec_def	fit_min_sec_min	significance	effect size	magnitude
1	org.jabref.cli.JabRefCLI	36	146	56	0.00798	0.00621	0.4	0.505	negligible
2	org.jabref.Globals	14	77	2614	0.0786	0.0855	0.884	0.496	negligible
3	org.jabref.JabRefGUI	43	183	574	0.239	0.183	0.877	0.496	negligible
4	org.jabref.logic.autosaveandbackup.BackupManager	21	117	153	0.208	0.18	0.00629	0.442	negligible
5	org.jabref.logic.bibtex.BibEntryWriter	28	124	8691	0.0952	0.0793	0.000186	0.431	negligible
6	org.jabref.logic.bibtex.comparator.BibDatabaseDiff	23	94	23	0.0381	0.0337	0.702	0.494	negligible
7	org.jabref.logic.bibtex.comparator.EntryComparator	28	103	60	0.516	0.504	0.0555	0.461	negligible
8	org.jabref.logic.bibtex.comparator.FieldComparator	29	119	808	0.247	0.247	0.922	0.498	negligible
9	org.jabref.logic.bibtex.DuplicateCheck	55	245	7599	0.201	0.161	0.222	0.479	negligible
10	org.jabref.logic.bibtex.FieldContentParser	4	31	3201	0.427	0.416	0.229	0.469	negligible
11	org.jabref.logic.bibtex.LatexFieldFormatter	67	206	50661	0.154	0.129	5.79e-09	0.392	small
12	org.jabref.logic.bibtex.LatexFieldFormatterPreferences	6	27	2183	0.152	0.121	0.00868	0.427	negligible
13	org.jabref.logic.bibtexkeypattern.BibtexKeyGenerator	31	140	341	0.142	0.121	3.86e-05	0.442	negligible
14	org.jabref.logic.bibtexkeypattern.BracketedPattern	245	1016	1605	0.0518	0.0466	0.00303	0.469	negligible
15	org.jabref.logic.citationstyle.CitationStyle	32	148	39564	0.0509	0.0486	6.6e-30	0.296	medium
16	org.jabref.logic.exporter.BibDatabaseWriter	42	175	145	0.0619	0.0478	0.137	0.485	negligible
17	org.jabref.logic.exporter.BibtexDatabaseWriter	27	127	145	0.274	0.271	0.631	0.493	negligible
18	org.jabref.logic.exporter.TemplateExporter	36	235	105	0.919	0.749	3.45e-23	0.33	medium
19	org.jabref.logic.exporter.VerifyingWriter	8	34	1762	0.322	0.318	0.0745	0.446	negligible
20	org.jabref.logic.importer.fetcher.ArXiv	33	264	87	0.163	0.15	1.37e-06	0.43	negligible
21	org.jabref.logic.importer.fetcher.AstrophysicsDataSystem	15	99	132	0.128	0.099	2.65e-08	0.4	small
22	org.jabref.logic.importer.fetcher.CrossRef	21	110	129	0.5	0.5	1	0.5	negligible
23	org.jabref.logic.importer.fetcher.GoogleScholar	26	126	117	0.13	0.12	0.64	0.489	negligible
24	org.jabref.logic.importer.fetcher.IacrEprintFetcher	29	145	20	0.124	0.125	0.971	0.501	negligible
25	org.jabref.logic.importer.Importer	21	122	497	0.191	0.163	0.448	0.489	negligible
26	org.jabref.logic.importer.ParserResult	33	138	130	0.223	0.186	6.4e-69	0.299	medium
27	org.jabref.logic.importer.util.MetaDataParser	29	128	193	0.311	0.306	0.0109	0.461	negligible
28	org.jabref.logic.importer.WebFetchers	14	96	13	0.562	0.56	0.941	0.498	negligible
29	org.jabref.logic.integrity.AbbreviationChecker	4	16	2295	0.445	0.427	0.0412	0.445	negligible
30	org.jabref.logic.integrity.BracketChecker	7	26	1205	0.353	0.32	0.000714	0.394	small
31	org.jabref.logic.integrity.PersonNamesChecker	9	25	4590	0.432	0.374	8.33e-21	0.268	medium
32	org.jabref.logic.journals.Abbreviation	19	69	556488	0.376	0.353	6.76e-05	0.451	negligible
33	org.jabref.logic.journals.AbbreviationParser	17	72	92754	0.198	0.21	0.534	0.513	negligible
34	org.jabref.logic.journals.JournalAbbreviationRepository	15	58	139122	0.265	0.213	3.89e-16	0.39	small
35	org.jabref.logic.layout.AbstractParamLayoutFormatter	12	54	4502	0.384	0.302	0.000146	0.377	small
36	org.jabref.logic.layout.format.Authors	63	280	1792	0.252	0.238	1.6e-06	0.44	negligible
37	org.jabref.logic.layout.format.HTMLChars	44	176	87993	0.417	0.392	0.0564	0.446	negligible
38	org.jabref.logic.layout.Layout	24	93	4283	0.198	0.162	1.64e-06	0.411	small
39	org.jabref.logic.layout.LayoutEntry	169	497	35976	0.155	0.116	0.000866	0.451	negligible
40	org.jabref.logic.layout.LayoutHelper	82	266	74500	0.118	0.109	0.306	0.467	negligible
41	org.jabref.logic.net.ProxyPreferences	29	96	16	0.266	0.148	1.14e-51	0.285	medium
42	org.jabref.logic.net.URLDownload	40	206	82	0.925	0.894	7.83e-19	0.396	small
43	org.jabref.logic.openoffice.OOBibStyle	142	696	672	0.0449	0.0363	1.22e-29	0.387	small
44	org.jabref.logic.protectedterms.ProtectedTermsLoader	31	124	21	0.419	0.356	4.18e-09	0.421	small
45	org.jabref.logic.protectedterms.ProtectedTermsParser	14	72	6615	0.101	0.0822	0.313	0.484	negligible
46	org.jabref.logic.TypedBibEntry	7	44	1047	0.618	0.598	0.436	0.475	negligible
47	org.jabref.logic.util.io.FileBasedLock	17	97	73	0.576	0.481	1.01e-08	0.36	small
48	org.jabref.logic.util.io.FileUtil	42	239	591	0.294	0.281	0.79	0.494	negligible
49	org.jabref.logic.util.strings.StringLengthComparator	3	12	102578	0.736	0.605	5.4e-08	0.298	medium
50	org.jabref.logic.util.UpdateField	20	107	93	0.471	0.407	5.42e-05	0.447	negligible
51	org.jabref.logic.util.Version	47	215	622	0.254	0.208	8.76e-15	0.407	small
52	org.jabref.model.bibtexkeypattern.AbstractBibtexKeyPattern	21	97	167	0.743	0.472	1.35e-110	0.194	large
53	org.jabref.model.ChainNode	15	111	274	0.0398	0.0385	0.881	0.499	negligible
54	org.jabref.model.database.BibDatabase	83	443	11410	0.0676	0.0506	1.68e-79	0.349	small
55	org.jabref.model.database.BibDatabaseContext	45	193	7652	0.034	0.0299	0.0857	0.481	negligible
56	org.jabref.model.entry.Author	60	270	1191	0.227	0.215	0.0084	0.467	negligible
57	org.jabref.model.entry.AuthorList	77	371	4717	0.459	0.257	1.62e-48	0.367	small
58	org.jabref.model.entry.AuthorListParser	84	328	496	0.351	0.294	2.79e-48	0.231	large
59	org.jabref.model.entry.BibEntry	128	618	103656	0.185	0.13	0	0.177	large
60	org.jabref.model.entry.BibtexEntryTypes	2	193	133	0.361	0.359	0.331	0.494	negligible
61	org.jabref.model.entry.BibtexSingleField	21	98	94328	0.0368	0.0316	0.00127	0.446	negligible
62	org.jabref.model.entry.Date	26	101	187	0.0451	0.0374	0.384	0.491	negligible
63	org.jabref.model.entry.event.EntryEvent	4	28	2837	0.418	0.472	0.0807	0.594	small
64	org.jabref.model.entry.event.FieldChangedEvent	14	78	2601	0.489	0.42	5.1e-18	0.292	medium
65	org.jabref.model.entry.FieldName	8	171	423	0.466	0.389	1.1e-05	0.393	small
66	org.jabref.model.entry.InternalBibtexFields	52	321	29207	0.484	0.241	6.4e-45	0.287	medium
67	org.jabref.model.entry.KeywordList	37	137	301	0.32	0.231	9.34e-40	0.352	small
68	org.jabref.model.entry.Month	23	120	216	0.0752	0.0268	0.302	0.487	negligible
69	org.jabref.model.entry.specialfields.SpecialField	15	57	1112	0.206	0.159	0.741	0.51	negligible
70	org.jabref.model.EntryTypes	36	182	2368	0.267	0.223	0.178	0.477	negligible
71	org.jabref.model.FieldChange	22	75	1932	0.446	0.323	1.67e-13	0.408	small
72	org.jabref.model.groups.AbstractGroup	31	129	1476	0.0417	0.0347	0.35	0.493	negligible
73	org.jabref.model.groups.AllEntriesGroup	6	24	1476	0.112	0.116	0.00222	0.428	negligible
74	org.jabref.model.groups.GroupTreeNode	54	229	1476	0.0714	0.0691	0.196	0.488	negligible
75	org.jabref.model.metadata.MetaData	50	232	7679	0.0256	0.0172	1.4e-172	0.234	large
76	org.jabref.model.metadata.SaveOrderConfig	19	142	16	0.567	0.391	3.56e-17	0.366	small
77	org.jabref.model.search.rules.SentenceAnalyzer	12	43	6256	0.359	0.381	0.41	0.526	negligible
78	org.jabref.model.strings.StringUtil	154	547	2183	0.284	0.221	1.6e-05	0.463	negligible
79	org.jabref.preferences.JabRefPreferences	168	1368	16694	0.119	0.108	4.98e-13	0.422	small
80	org.jabref.preferences.PreviewPreferences	8	81	2110	0.122	0.0761	2.88e-07	0.41	small

Table A.2: Commonality coverage for `fit_def_sec_def` and `fit_min_sec_min` compared per class

	class	complexity	loc	max. exec. count	fit_def_sec_def	fit_def_sec_max	significance	effect size	magnitude
1	org.jabref.cli.JabRefCLI	36	146	56	0.487	0.526	0.118	0.618	small
2	org.jabref.Globals	14	77	2614	0.092	0.139	0.81	0.517	negligible
3	org.jabref.JabRefGUI	43	183	574	0.0982	0.0944	0.154	0.466	negligible
4	org.jabref.logic.autosaveandbackup.BackupManager	21	117	153	0.418	0.32	0.0107	0.309	medium
5	org.jabref.logic.bibtex.BibEntryWriter	28	124	8691	0.92	0.923	0.793	0.48	negligible
6	org.jabref.logic.bibtex.comparator.BibDatabaseDiff	23	94	23	0.489	0.491	0.116	0.616	small
7	org.jabref.logic.bibtex.comparator.EntryComparator	28	103	60	0.772	0.8	0.0286	0.664	small
8	org.jabref.logic.bibtex.comparator.FieldComparator	29	119	808	0.496	0.544	0.722	0.527	negligible
9	org.jabref.logic.bibtex.DuplicateCheck	55	245	7599	0.639	0.632	0.958	0.504	negligible
10	org.jabref.logic.bibtex.FieldContentParser	4	31	3201	0.948	0.969	0.452	0.534	negligible
11	org.jabref.logic.bibtex.LatexFieldFormatter	67	206	50661	0.811	0.789	0.592	0.459	negligible
12	org.jabref.logic.bibtexkeypattern.BracketedPattern	245	1016	1605	0.629	0.573	3.83e-06	0.152	large
13	org.jabref.logic.citationstyle.CitationStyle	32	148	39564	0.663	0.663	0.168	0.467	negligible
14	org.jabref.logic.exporter.BibDatabaseWriter	42	175	145	0.76	0.742	0.552	0.456	negligible
15	org.jabref.logic.exporter.BibtexDatabaseWriter	27	127	145	0.791	0.807	0.643	0.468	negligible
16	org.jabref.logic.exporter.TemplateExporter	36	235	105	0.121	0.12	0.57	0.483	negligible
17	org.jabref.logic.exporter.VerifyingWriter	8	34	1762	0.692	0.675	0.799	0.483	negligible
18	org.jabref.logic.importer.fetcher.ArXiv	33	264	87	0.189	0.189	1	0.5	negligible
19	org.jabref.logic.importer.fetcher.CrossRef	21	110	129	0.208	0.178	1.06e-09	0.0776	large
20	org.jabref.logic.importer.fetcher.GoogleScholar	26	126	117	0.108	0.0917	2.07e-08	0.138	large
21	org.jabref.logic.importer.fetcher.IacrEprintFetcher	29	145	20	0.0825	0.0796	0.161	0.467	negligible
22	org.jabref.logic.importer.Importer	21	122	497	1	0.948	0.00031	0.317	medium
23	org.jabref.logic.importer.ParserResult	33	138	130	0.971	0.937	0.577	0.464	negligible
24	org.jabref.logic.importer.util.MetaDataParser	29	128	193	0.953	0.962	0.112	0.563	negligible
25	org.jabref.logic.importer.WebFetchers	14	96	13	1	1	1	0.501	negligible
26	org.jabref.logic.integrity.BracketChecker	7	26	1205	1	0.96	0.0419	0.433	negligible
27	org.jabref.logic.journals.Abbreviation	19	69	556488	0.923	0.981	2.5e-05	0.789	large
28	org.jabref.logic.journals.AbbreviationParser	17	72	92754	0.517	0.508	0.334	0.483	negligible
29	org.jabref.logic.layout.AbstractParamLayoutFormatter	12	54	4502	0.692	0.72	0.627	0.534	negligible
30	org.jabref.logic.layout.format.Authors	63	280	1792	0.985	0.896	2.42e-06	0.168	large
31	org.jabref.logic.layout.format.HTMLChars	44	176	87993	0.481	0.449	0.384	0.434	negligible
32	org.jabref.logic.layout.format.NameFormatter	22	95	752	0.835	0.852	0.0834	0.628	small
33	org.jabref.logic.layout.Layout	24	93	4283	0.773	0.838	0.189	0.58	small
34	org.jabref.logic.layout.LayoutEntry	169	497	35976	0.262	0.228	0.000366	0.232	large
35	org.jabref.logic.layout.LayoutHelper	82	266	74500	0.228	0.258	0.0933	0.579	small
36	org.jabref.logic.net.ProxyPreferences	29	96	16	0.999	0.999	0.597	0.484	negligible
37	org.jabref.logic.net.URLDownload	40	206	82	0.517	0.581	0.21	0.598	small
38	org.jabref.logic.openoffice.OOBibStyle	142	696	672	0.353	0.311	0.259	0.414	small
39	org.jabref.logic.protectedterms.ProtectedTermsLoader	31	124	21	0.986	0.985	0.334	0.483	negligible
40	org.jabref.logic.protectedterms.ProtectedTermsParser	14	72	6615	0.773	0.773	1	0.5	negligible
41	org.jabref.logic.TypedBibEntry	7	44	1047	0.947	0.891	0.286	0.433	negligible
42	org.jabref.logic.util.io.FileUtil	42	239	591	0.847	0.882	0.0495	0.652	small
43	org.jabref.logic.util.StandardFileType	3	15	2113	0.556	0.963	1.18e-13	1	large
44	org.jabref.logic.util.strings.StringLengthComparator	3	12	102578	0.991	0.991	0.75	0.514	negligible
45	org.jabref.logic.util.Version	47	215	622	0.86	0.854	0.17	0.45	negligible
46	org.jabref.model.bibtexkeypattern.AbstractBibtexKeyPattern	21	97	167	0.999	1	0.161	0.533	negligible
47	org.jabref.model.ChainNode	15	111	274	0.878	0.892	0.0419	0.567	negligible
48	org.jabref.model.database.BibDatabase	83	443	11410	0.847	0.8	0.0698	0.362	small
49	org.jabref.model.database.BibDatabaseContext	45	193	7652	0.959	0.921	0.126	0.384	small
50	org.jabref.model.entry.Author	60	270	1191	0.995	0.999	0.723	0.487	negligible
51	org.jabref.model.entry.AuthorList	77	371	4717	0.805	0.787	0.848	0.485	negligible
52	org.jabref.model.entry.AuthorListParser	84	328	496	0.964	0.966	0.283	0.563	negligible
53	org.jabref.model.entry.BibEntry	128	618	103656	0.803	0.829	0.254	0.406	small
54	org.jabref.model.entry.Date	26	101	187	0.968	0.956	0.703	0.472	negligible
55	org.jabref.model.entry.event.EntryEvent	4	28	2837	0.8	0.821	0.039	0.638	small
56	org.jabref.model.entry.event.FieldChangedEvent	14	78	2601	0.985	0.986	1	0.501	negligible
57	org.jabref.model.entry.FieldName	8	171	423	0.953	0.892	0.806	0.483	negligible
58	org.jabref.model.entry.InternalBibtexFields	52	321	29207	0.988	0.993	0.0966	0.623	small
59	org.jabref.model.entry.KeywordList	37	137	301	0.966	0.979	0.227	0.573	negligible
60	org.jabref.model.entry.Month	23	120	216	0.542	0.551	0.0765	0.368	small
61	org.jabref.model.entry.specialfields.SpecialField	15	57	1112	0.573	0.352	1.18e-12	0	large
62	org.jabref.model.EntryTypes	36	182	2368	0.9	0.945	0.0417	0.639	small
63	org.jabref.model.FieldChange	22	75	1932	0.997	0.986	0.0227	0.384	small
64	org.jabref.model.groups.AbstractGroup	31	129	1476	0.885	0.884	0.925	0.507	negligible
65	org.jabref.model.groups.GroupTreeNode	54	229	1476	0.98	0.97	0.468	0.467	negligible
66	org.jabref.model.metadata.MetaData	50	232	7679	0.984	0.976	0.633	0.482	negligible
67	org.jabref.model.metadata.SaveOrderConfig	19	142	16	0.961	0.983	0.277	0.548	negligible
68	org.jabref.model.search.rules.SentenceAnalyzer	12	43	6256	0.984	0.956	0.0419	0.433	negligible
69	org.jabref.model.strings.StringUtil	154	547	2183	0.943	0.934	0.0552	0.356	small
70	org.jabref.preferences.JabRefPreferences	168	1368	16694	0.0215	0.0178	0.0358	0.414	small

Table A.3: PIT score for `fit_def_sec_def` and `fit_def_sec_max` compared per class

	class	complexity	loc	max. exec. count	fit_def_sec_def	fit_max_sec_max	significance	effect size	magnitude
1	org.jabref.cli.JabRefCLI	36	146	56	0.487	0.528	0.332	0.573	negligible
2	org.jabref.Globals	14	77	2614	0.092	0.122	0.602	0.467	negligible
3	org.jabref.logic.autosaveandbackup.BackupManager	21	117	153	0.418	0.322	0.0338	0.33	medium
4	org.jabref.logic.bibtex.BibEntryWriter	28	124	8691	0.92	0.874	0.272	0.418	small
5	org.jabref.logic.bibtex.comparator.BibDatabaseDiff	23	94	23	0.489	0.577	0.0169	0.672	medium
6	org.jabref.logic.bibtex.comparator.EntryComparator	28	103	60	0.772	0.779	0.234	0.589	small
7	org.jabref.logic.bibtex.comparator.FieldComparator	29	119	808	0.496	0.494	0.7	0.529	negligible
8	org.jabref.logic.bibtex.DuplicateCheck	55	245	7599	0.639	0.581	0.00525	0.293	medium
9	org.jabref.logic.bibtex.FieldContentParser	4	31	3201	0.948	0.98	0.217	0.552	negligible
10	org.jabref.logic.bibtex.LatexFieldFormatter	67	206	50661	0.811	0.809	0.829	0.517	negligible
11	org.jabref.logic.bibtexkeypattern.BracketedPattern	245	1016	1605	0.629	0.574	5.02e-06	0.153	large
12	org.jabref.logic.citationstyle.CitationStyle	32	148	39564	0.663	0.663	0.342	0.483	negligible
13	org.jabref.logic.exporter.BibDatabaseWriter	42	175	145	0.76	0.757	0.772	0.522	negligible
14	org.jabref.logic.exporter.BibtexDatabaseWriter	27	127	145	0.791	0.717	0.676	0.47	negligible
15	org.jabref.logic.exporter.TemplateExporter	36	235	105	0.121	0.121	1	0.5	negligible
16	org.jabref.logic.exporter.VerifyingWriter	8	34	1762	0.692	0.692	1	0.5	negligible
17	org.jabref.logic.importer.fetcher.ArXiv	33	264	87	0.189	0.189	0.334	0.517	negligible
18	org.jabref.logic.importer.fetcher.CrossRef	21	110	129	0.208	0.2	6.9e-05	0.23	large
19	org.jabref.logic.importer.fetcher.GoogleScholar	26	126	117	0.108	0.0978	6.26e-05	0.283	medium
20	org.jabref.logic.importer.fetcher.IacrEprintFetcher	29	145	20	0.0825	0.0778	0.0419	0.433	negligible
21	org.jabref.logic.importer.Importer	21	122	497	1	0.965	0.00557	0.383	small
22	org.jabref.logic.importer.ParserResult	33	138	130	0.971	0.969	0.516	0.54	negligible
23	org.jabref.logic.importer.util.MetaDataParser	29	128	193	0.953	0.926	0.151	0.429	negligible
24	org.jabref.logic.importer.WebFetchers	14	96	13	1	1	1	0.501	negligible
25	org.jabref.logic.integrity.AbbreviationChecker	4	16	2295	1	0.991	0.334	0.483	negligible
26	org.jabref.logic.integrity.BracketChecker	7	26	1205	1	0.988	0.161	0.467	negligible
27	org.jabref.logic.journals.Abbreviation	19	69	556488	0.923	0.95	0.0117	0.679	medium
28	org.jabref.logic.journals.AbbreviationParser	17	72	92754	0.517	0.508	0.0419	0.433	negligible
29	org.jabref.logic.layout.AbstractParamLayoutFormatter	12	54	4502	0.692	0.686	0.722	0.475	negligible
30	org.jabref.logic.layout.format.Authors	63	280	1792	0.985	0.933	6.2e-07	0.149	large
31	org.jabref.logic.layout.format.HTMLChars	44	176	87993	0.481	0.395	0.00598	0.294	medium
32	org.jabref.logic.layout.format.NameFormatter	22	95	752	0.835	0.854	0.497	0.551	negligible
33	org.jabref.logic.layout.Layout	24	93	4283	0.773	0.83	0.284	0.567	negligible
34	org.jabref.logic.layout.LayoutEntry	169	497	35976	0.262	0.234	0.0183	0.322	medium
35	org.jabref.logic.layout.LayoutHelper	82	266	74500	0.228	0.239	0.952	0.504	negligible
36	org.jabref.logic.net.ProxyPreferences	29	96	16	0.999	0.99	0.0881	0.433	negligible
37	org.jabref.logic.net.URLDownload	40	206	82	0.517	0.507	0.609	0.541	negligible
38	org.jabref.logic.openoffice.OOBibStyle	142	696	672	0.353	0.336	0.744	0.475	negligible
39	org.jabref.logic.protectedterms.ProtectedTermsLoader	31	124	21	0.986	0.976	0.0815	0.45	negligible
40	org.jabref.logic.protectedterms.ProtectedTermsParser	14	72	6615	0.773	0.767	0.584	0.484	negligible
41	org.jabref.logic.TypedBibEntry	7	44	1047	0.947	0.929	0.985	0.498	negligible
42	org.jabref.logic.util.io.FileUtil	42	239	591	0.847	0.864	0.176	0.604	small
43	org.jabref.logic.util.StandardFileType	3	15	2113	0.556	0.763	1.05e-07	0.833	large
44	org.jabref.logic.util.strings.StringLengthComparator	3	12	102578	0.991	0.993	0.71	0.516	negligible
45	org.jabref.logic.util.Version	47	215	622	0.86	0.858	0.18	0.451	negligible
46	org.jabref.model.bibtexkeypattern.AbstractBibtexKeyPattern	21	97	167	0.999	0.995	0.604	0.48	negligible
47	org.jabref.model.ChainNode	15	111	274	0.878	0.892	0.0419	0.567	negligible
48	org.jabref.model.database.BibDatabase	83	443	11410	0.847	0.843	0.575	0.457	negligible
49	org.jabref.model.database.BibDatabaseContext	45	193	7652	0.959	0.953	0.738	0.474	negligible
50	org.jabref.model.entry.Author	60	270	1191	0.995	0.997	1	0.5	negligible
51	org.jabref.model.entry.AuthorList	77	371	4717	0.805	0.854	0.191	0.599	small
52	org.jabref.model.entry.AuthorListParser	84	328	496	0.964	0.967	0.216	0.574	small
53	org.jabref.model.entry.BibEntry	128	618	103656	0.803	0.817	0.185	0.393	small
54	org.jabref.model.entry.BibtexSingleField	21	98	94328	1	0.999	0.334	0.483	negligible
55	org.jabref.model.entry.Date	26	101	187	0.968	0.969	0.555	0.542	negligible
56	org.jabref.model.entry.event.EntryEvent	4	28	2837	0.8	0.853	0.000454	0.733	medium
57	org.jabref.model.entry.event.FieldChangedEvent	14	78	2601	0.985	1	0.334	0.517	negligible
58	org.jabref.model.entry.FieldName	8	171	423	0.953	0.903	0.407	0.443	negligible
59	org.jabref.model.entry.InternalBibtexFields	52	321	29207	0.988	0.992	0.168	0.601	small
60	org.jabref.model.entry.KeywordList	37	137	301	0.966	0.975	0.293	0.563	negligible
61	org.jabref.model.entry.Month	23	120	216	0.542	0.547	0.00125	0.262	large
62	org.jabref.model.entry.specialfields.SpecialField	15	57	1112	0.573	0.353	1.2e-12	0	large
63	org.jabref.model.EntryTypes	36	182	2368	0.9	0.857	0.771	0.479	negligible
64	org.jabref.model.FieldChange	22	75	1932	0.997	0.974	0.371	0.464	negligible
65	org.jabref.model.groups.AbstractGroup	31	129	1476	0.885	0.855	0.563	0.458	negligible
66	org.jabref.model.groups.GroupTreeNode	54	229	1476	0.98	0.991	0.655	0.517	negligible
67	org.jabref.model.metadata.MetaData	50	232	7679	0.984	0.951	0.213	0.447	negligible
68	org.jabref.model.metadata.SaveOrderConfig	19	142	16	0.961	0.996	0.221	0.554	negligible
69	org.jabref.model.search.rules.SentenceAnalyzer	12	43	6256	0.984	0.977	0.57	0.517	negligible
70	org.jabref.model.strings.StringUtil	154	547	2183	0.943	0.934	0.0472	0.351	small
71	org.jabref.preferences.JabRefPreferences	168	1368	16694	0.0215	0.0193	0.113	0.448	negligible

Table A.4: PIT score for `fit_def_sec_def` and `fit_max_sec_max` compared per class

Appendix B

Additional coverage evolution figures

This appendix contains several additional figures showing the evolution of coverage values for standard coverage metrics not discussed in Section 5.4.

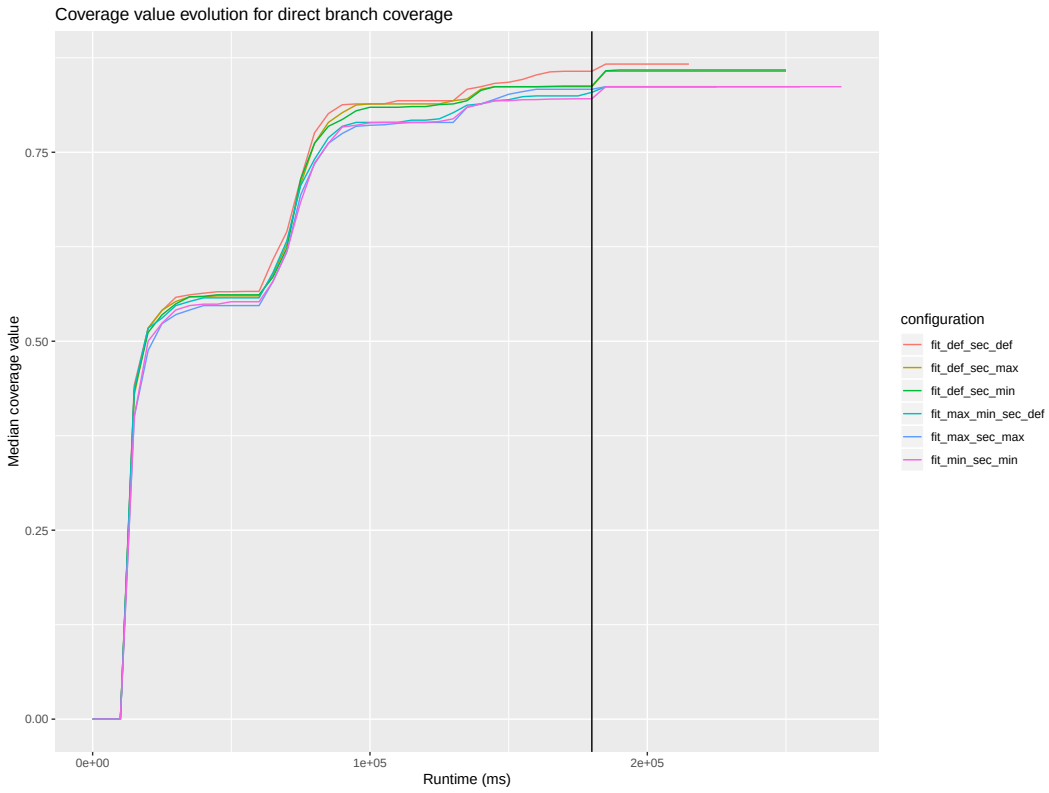

Figure B.1: Evolution of the median test suite direct branch coverage value for different configurations.

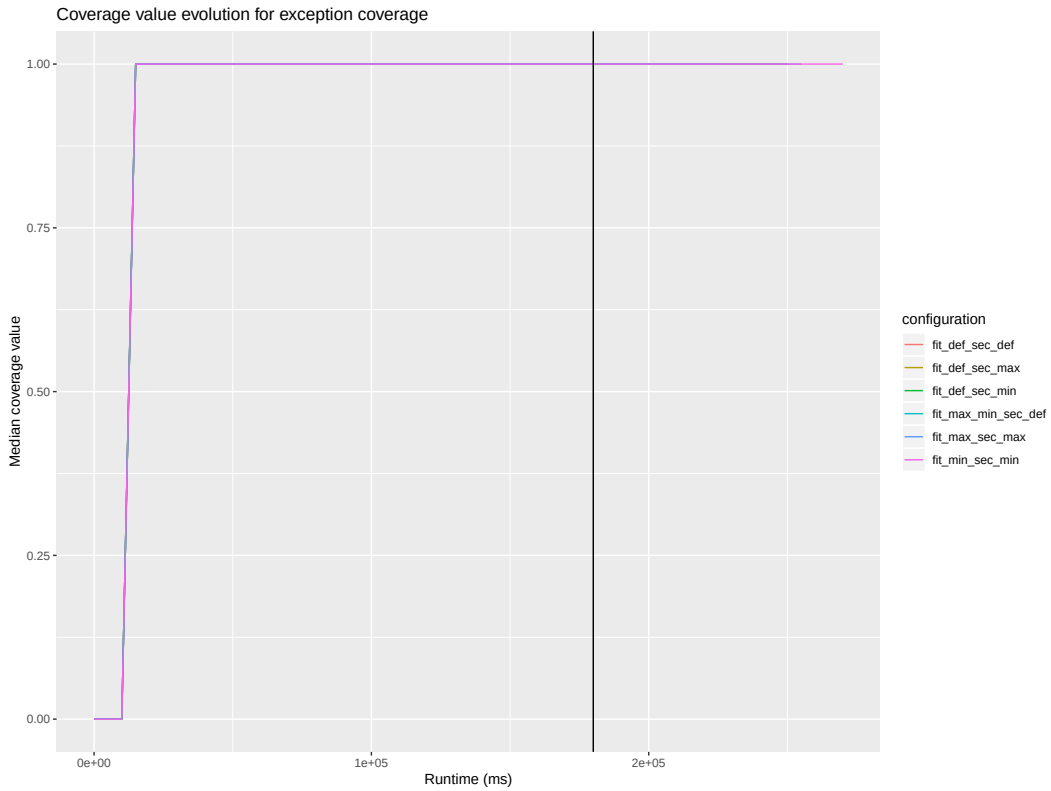

Figure B.2: Evolution of the median test suite exception coverage value for different configurations.

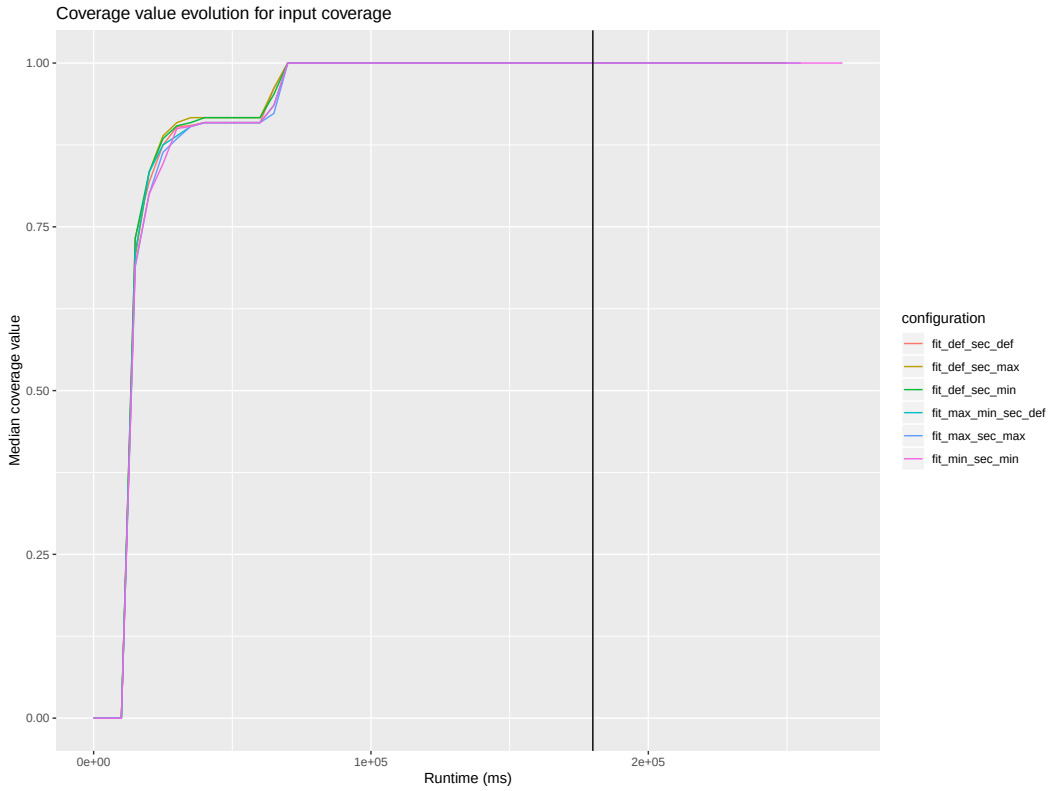

Figure B.3: Evolution of the median test suite input coverage value for different configurations.

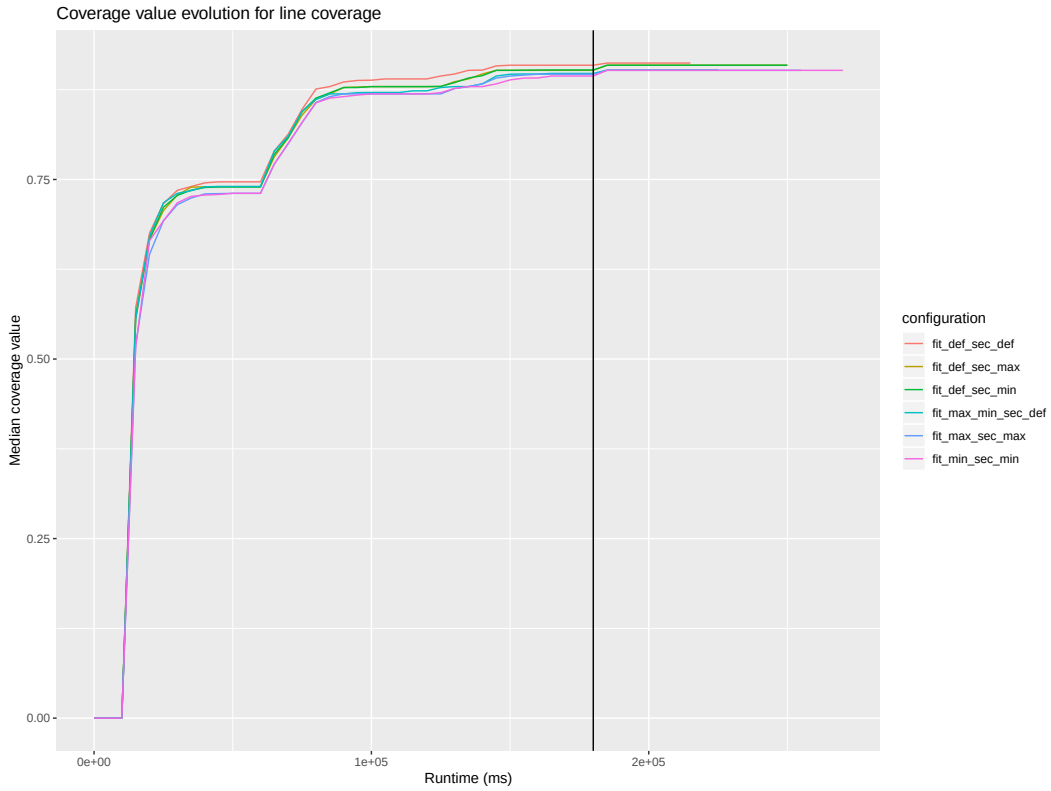

Figure B.4: Evolution of the median test suite line coverage value for different configurations.

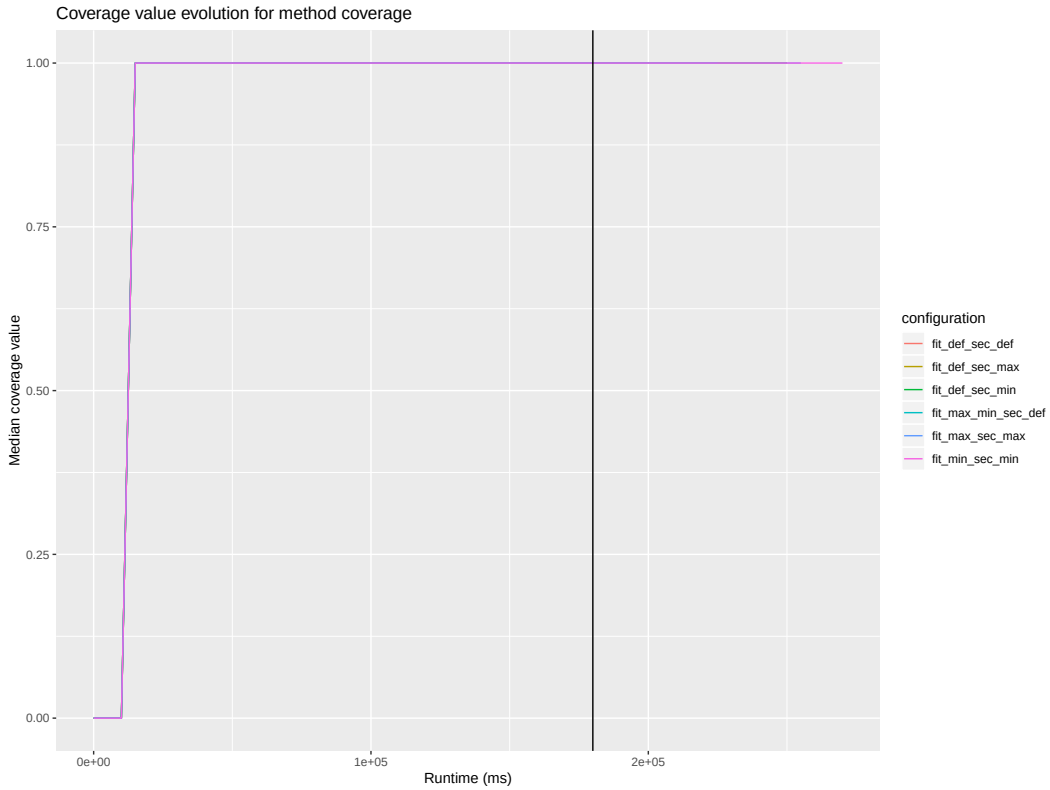

Figure B.5: Evolution of the median test suite method coverage value for different configurations.

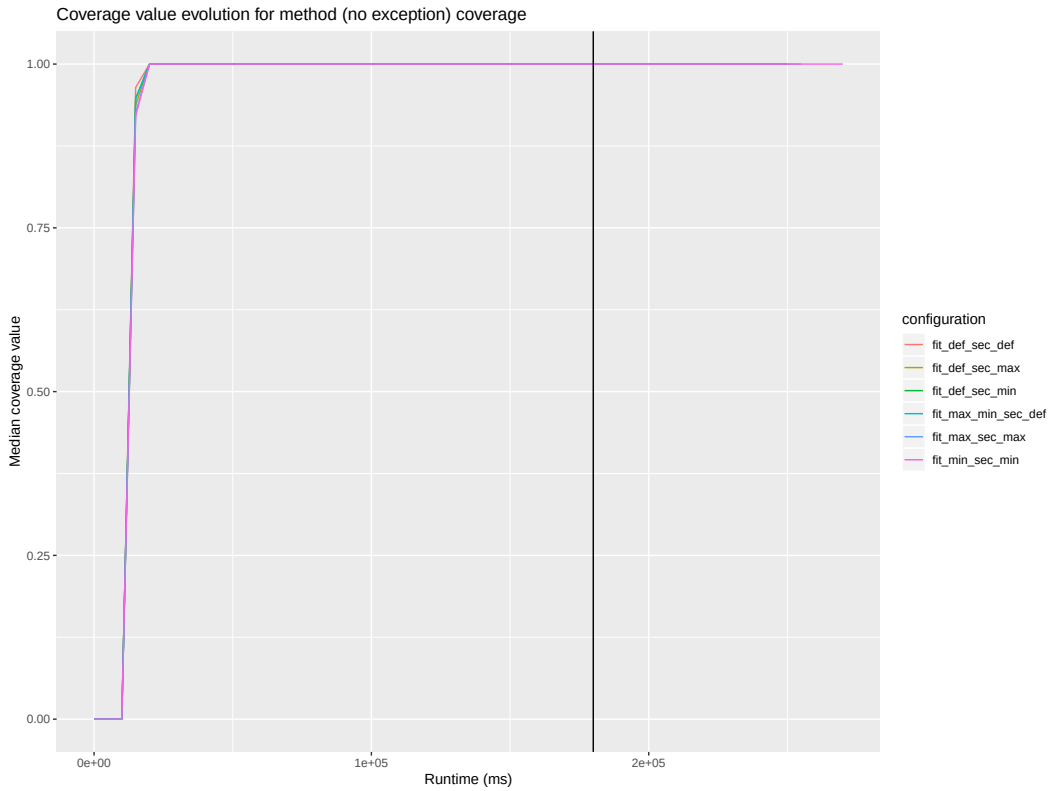

Figure B.6: Evolution of the median test suite method (no exception) coverage value for different configurations.

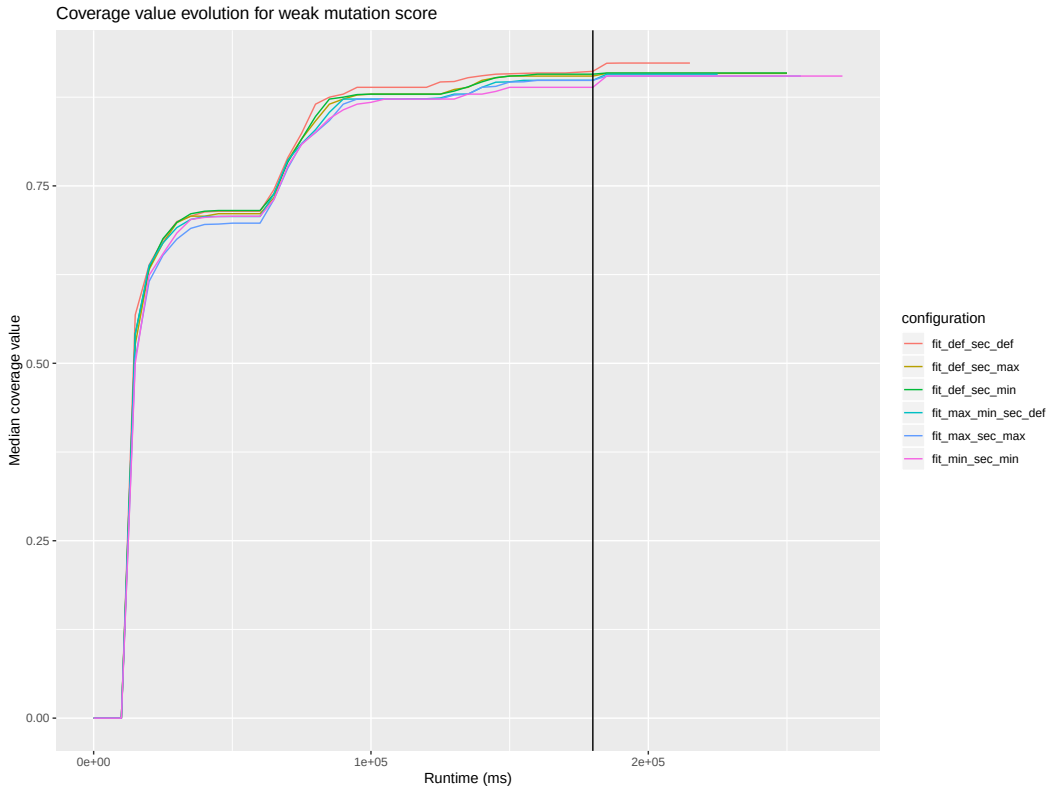

Figure B.7: Evolution of the median test suite weak mutation score for different configurations.

Glossary

CFG control flow graph. vii, 12–15, 24, 63

CUBTG common and uncommon behavior test generation. i, 1–3, 7, 8, 16, 17, 19, 21, 22, 24, 27, 29–31, 34, 36, 37, 39–42, 44, 46, 49, 50, 52–54, 57–59, 61–64

CUT class under test. 6, 11, 12, 14–16, 19, 23–25, 41, 61, 62

FF fitness function. 1, 2, 7, 8, 17, 19, 21, 22, 27–30, 32, 34–37, 40–42, 44, 46, 47, 50, 52–54, 57–59, 61, 62

MOSA the many-objective sorting algorithm. i, v, vii, 1, 2, 5, 7, 8, 16–19, 21, 22, 28–32, 34–37, 39–42, 44, 46, 49, 50, 52–54, 57, 61–64

MWW Mann–Whitney–Wilcoxon. 30, 59

NSGA-II the non-dominated sorting genetic algorithm II. 7, 21, 28, 31

SO secondary objective. 1, 2, 18, 19, 21, 22, 27–30, 32, 34–37, 40–42, 46, 47, 50, 52, 53, 57–59, 61, 62

SUT software under test. 1, 11, 26, 35

VD.A Vargha and Delaney A. 59