



FACULTY OF ENGINEERING AND TECHNOLOGY
MASTER OF SOFTWARE ENGINEERING

**Optimizing Software Test Case Generation
using Light-Weight Variations of
Indicator-Based Evolutionary Algorithm**

Author:

Hadi Awad

Supervisor:

Dr. Abdel Salam Sayyad

*A thesis submitted in fulfillment of the requirements for the
degree of Master of Science in Software Engineering at
Birzeit University, Palestine*

June 7, 2020



FACULTY OF ENGINEERING AND TECHNOLOGY

MASTER OF SOFTWARE ENGINEERING

Master Thesis

Optimizing Software Test Case Generation using Light-Weight Variations of Indicator-Based
Evolutionary Algorithm

تحسين إنشاء اختبارات فحص جودة البرمجيات باستخدام تعديلات على الخوارزمية التطورية القائمة على المؤثر

Author:

Hadi Awad

Supervisor

Dr. Abdel Salam Sayyad

Committee

Dr. Abdel Salam Sayyad

Dr. Ahmad Afaneh

Dr. Samer Zein

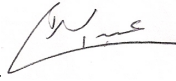
*A thesis submitted in fulfillment of the requirements for the degree of Master of Science
in Software Engineering at Birzeit University, Palestine*

June 7, 2020



Approved by the thesis committee:


Dr. Abdel Salam Sayyad, Birzeit University



Dr. Ahmad Afaneh, Birzeit University



Dr. Samer Zein, Birzeit University



Date approved:

2020/09/07

Declaration of Authorship

I, Hadi Awad, declare that this thesis titled, "Optimizing Software Test Case Generation using Light-Weight Variations of Indicator-Based Evolutionary Algorithm" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a master degree at Birzeit University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

Date:

Abstract

As the development of an application grows larger and becomes more rapid and complex, and since the modern software development paradigm surge extreme development and continuous delivery, Software testing becomes a core activity in software development life cycle to ensure that the software product meets the requirements and has no failures. However, it has been generally observed that traditional manual testing can't meet the development pace in terms of providing high code coverage as well as effectiveness in finding bugs, and thus continuous integration and automated testing has risen recently to cover this gap.

In this research, we intend to use white box testing techniques to analyze a program source code and automatically generate feasible test cases that examine the system under test and check its behavior. The test cases properness and feasibility are measured by certain metrics, such as code coverage, execution time and resource usage. For a test case to be valuable for the developer it should at least achieve code coverage with minimal time to execute.

Many evolutionary algorithms have been successfully adopted in automating test case generation. Nevertheless, Indicator Based Evolutionary Algorithm (IBEA) was never examined since it requires intense computation time to evaluate the quality of Solutions. Thus, we intend to employ light-computation flavors of IBEA as a multi objective algorithm in generating test suites

ملخص

ان تطوير التطبيقات ينمو بشكل كبير وقد أصبح أكثر سرعة وتعقيدًا، ونظراً لأن نماذج تطوير البرامج الحديثة يتطلب ارتفاع في نسق التطوير وآليات التسليم المتابعة، فإن اختبار البرمجيات يصبح نشاطًا أساسيًا في دورة حياة تطوير البرمجيات، لضمان تلبية المنتج للمتطلبات من دون اي خطأ. ومع ذلك، فقد وجد عمومًا أن الاختبار اليدوي التقليدي لا يلي سرعة التطوير من حيث توفير تغطية عالية للغة البرمجة المكون الاساسي للبرمجيات ومن حيث الفعالية في العثور على الأخطاء، وبالتالي، فإن كتابة فحوصات البرمجيات الآلية ودمجها المستمر في آليات البرمجة قد وجدت مؤخرًا لتغطية هذه الفجوة.

في هذا البحث العلمي، نعتمد استخدام تقنيات فحص الجودة القائمة على وجود المصدر لتحليل اللغة المكونة للبرمجيات وإنشاء حالات اختبار ملائمة بشكل تلقائي والتي من واجبها فحص النظام قيد الاختبار والتحقق من سلوكه. يتم قياس ملاءمة حالات الاختبار هذه والحدوى منها على مقاييس معينة، مثل التغطية العالية للغة البرمجة، وقت التنفيذ واستخدام موارد الحاسوب. وعليه، لكي تكون حالة الاختبار ذات قيمة لمطور البرمجيات، يجب أن تحقق على الأقل التغطية العالية للغة البرمجة مع الحد الأدنى من الوقت للتنفيذ.

مؤخرًا، تم اعتماد العديد من الخوارزميات التطورية بنجاح في أتمتة انشاء حالات الاختبار. ومع ذلك، لم يتم فحص الخوارزمية التطورية القائمة على المؤشر لأنها تتطلب وقتًا مكثفًا لتقييم جودة الحلول. ولذلك، فإننا نعتمد استخدام تعديلات على الخوارزمية التطورية القائمة على المؤشر كخوارزمية متعددة الأهداف في انشاء حزمة الاختبار.

Acknowledgements

First all, I would like to express my sincere gratitude to my family; especially those two super heroes for their unlimited support and courage during the past years.

I would like to express my sincere appreciation to my advisor Dr. Abdel Salam Sayyad for his guidance, instructions and advice.

Special thanks to all instructors of the software engineering master at Birzeit university whom I had the chance to cooperate with and learn from.

Contents

1	Introduction	1
1.1	Problem Statement	3
1.2	Research Questions	6
1.3	Research Contribution	10
1.3.1	Enhancing <i>EVOSUITE</i> Tool with IBEA algorithms	10
1.3.2	Validating Contribution on well Known set of Classes	11
1.3.3	Addressing the Problem Through Many objectives	12
1.3.4	Comparison Between IBEA, mIBEA and NSGA-II .	13
1.4	Research Overview	14
1.5	Research Activities	15
2	Background	16
2.1	Evolutionary Algorithm	17
2.1.1	General Design of Genetic Algorithm	18
2.1.2	Genetic Operators	19
2.2	Single-objective Evolutionary Algorithms	21
2.3	Multi-objective Evolutionary Algorithms	22
2.3.1	Indicator based Evolutionary Algorithm	24

2.3.2	A Modified Indicator-based Evolutionary Algorithm (mIBEA)	25
2.3.3	Non-dominated Sorting Genetic Algorithm II	27
2.4	EVOSUITE	30
2.4.1	Unit Testing and Class Under Test	31
2.4.2	Types of Code Coverage	32
2.4.3	Whole Test Suite Generation	36
2.4.4	Problem Representation	37
3	Related Work	39
3.1	Single Structural Target Approach	40
3.2	Multi Structural Target Approach	45
3.3	Adjusting Evolutionary Algorithms	53
3.4	Thesis Distinction from Other Studies	56
3.5	Literature Review Summary	57
4	Research Methodology And Experiment Setup	59
4.1	Experiment Data Sources	60
4.2	Experiment Setup	61
4.2.1	Chromosome Structure	61
4.2.2	Fitness Function	64
4.2.2.1	Branch distance	65
4.2.2.2	Approach Level	66
4.2.2.3	Fitness Function: Branch Coverage	66
4.2.3	Evaluation Metrics	68

4.3	Algorithms and EvoSuite	68
4.3.1	Genetic Search Operators of EVOSUITE	69
4.3.1.1	Chromosome Crossover	70
4.3.1.2	Chromosome Mutation	71
4.3.2	Random Test Case Generation	73
4.4	Experiment Assumptions	74
5	Experiment Results and Analysis	76
5.1	Algorithm Comparison	76
5.1.1	Experiment Results	77
5.1.1.1	Experiment Results as Box Plots	78
5.1.1.2	Descriptive Statistics of Experiment Results	82
5.2	Branch Coverage Analysis	83
5.3	Generation Time Analysis	87
5.4	Testing Metrics Analysis	89
5.5	Statistical Analysis	93
6	Conclusion And Future Work	97
6.1	Threats to Validity	97
6.1.1	Threats to construct validity	97
6.1.2	Threats to Internal validity	98
6.1.3	Threats to conclusion validity	99
6.1.4	Threats to external validity	99
6.2	Conclusion	100
6.3	Difficulties and Obstacles	102

6.3.1	Mastering <i>EVOSUITE</i>	102
6.4	Future Work	103
6.4.1	Analyzing different Variations of IBEA	103
6.4.2	Analyzing CUT with large number of objectives . .	103
6.4.3	IBEA to cover single branches at a time	104
6.4.4	Contribute to EvoSuite	105
	Bibliography	107
	Appendices	117
	A Thesis External Links	118
	B Preliminary Experiment Results	119
B.1	Preliminary Experiment Results	120
B.2	Preliminary Experiment box Plots	121

List of Figures

2.1	IBEA Pseudo Code [52]	26
2.2	NSGA-II Algorithm Procedure [1]	29
4.1	Example class and test case	63
4.2	Chromosome Structure in WTS	64
4.3	Code Snippet to explain Branch distance	65
4.4	A code Snippet to explain the Branch distance and Approach level	67
4.5	Test Suite Crossover	70
4.6	Test Case Mutation example, the three operations delete, insert and change are shown [22]	73
5.1	Generated Test Suite Branch Coverage	79
5.2	Generated Test Suite Length	80
5.3	Generated Test Suite Size	81
5.4	Algorithm Generation Time	81
5.5	Branch Coverage when Number of Branches is between 22 and 70	84

5.6	Branch Coverage when Number of Branches is between 71 and 100	84
5.7	Branch Coverage when Number of Branches is between 101 and 200	85
5.8	Branch Coverage when Number of Branches is between 201 and 300	85
5.9	Branch Coverage when Number of Branches is between 301 and 1215	86
5.10	Evolution Time to Branch Numbers on full Coverage . . .	87
5.11	Test-suite Size to Target Goals	90
5.12	Test-suite Length to Target Goals	91
5.13	Test-suite Execution Time to Target Goals	92
5.14	Test-suite Mutation Score to Target Goals	93
B.1	Total Execution Time	121
B.2	Generated Test Suite Coverage	122
B.3	Generated Test Suite Mutation Score	122
B.4	Generated Test Suite Size	122
B.5	Generated Test Suite Length	123
B.6	Generated Test Suite Covered Goals	123

List of Tables

3.1	Literature Review Summary	58
4.1	Classes to be used in the Empirical Experiment	62
4.2	EvoSuite PARAMETER SETTINGS	69
5.1	Mean Values that summarizes the full Experiment Collected Results	79
5.2	Add caption	82
5.3	Add caption	83
5.4	Add caption	83
5.5	Data Summary for Full Goals Coverage	88
5.6	Friedman test of metrics collected	94
5.7	Evolution Time Pair Wise Comparison	95
5.8	Evolved Test-suite Branch Coverage	95
5.9	Evolved Test-suite Length	95
5.10	Evolved Test-suite Execution Time	96
5.11	Evolved Test-suite Size	96
5.12	Evolved Test-suite Mutation Score	96
B.1	<i>EVOSUITE</i> Experimental Data-set	119

B.2 Preliminary Experiment Result 120

Chapter 1

Introduction

Software testing is one of the most important activities in the software development life cycle; testing activities consume 50% of the total effort and cost of the whole development effort according to Chartchai et al [13]. The very early stage of testing any software is unit testing; unit testing can be defined as a software testing method by which the developers test their individual unit of source code. The unit can be referred to as a single component or small piece of code that is under test; nevertheless, its well known among the developers that the unit is the class under test (CUT) or a method in a class.

Unit testing is an activity that has been highlighted a lot as a core process in Software methodologies like Agile and Extreme Programming; what adds more importance to unit testing nowadays, is the ability to integrate them easily in team's software development process (like DevOps) using a lot of integration tools. Nevertheless, the primary key practice for unit testing is automating it so that it can be executed regularly or on demand [64].

Writing unit tests for a program are considered a tedious task for most of the developers [7, 54]. Moreover, such an activity became an essential process in modern software development methodologies such as extreme programming, Scrum Agile and Kanban Agile in which the delivery of new software functionalities could occur on a daily basis [7, 28]. Thus, easing the writing of unit tests and automatically deriving them is one of the important topics in primary software engineering [45].

Many Software engineering activities can be considered as an optimization problem [27], and unit testing is one of them as we aim to generate tests that execute the software systemically using fewer resources, shortest time possible and achieving higher code coverage (like statement or branch coverage) while detecting the maximum defects existing in the software [19]. Not only Generating the tests, but also create tests' oracle that evaluates the correctness of the system under test (SUT) behavior during the unit tests execution.

A lot of algorithms have been analyzed in the domain of automating test case inputs generation. Both Single and Multi Objective Genetic Algorithms were examined and researchers improved those algorithms by empowering them with hybrid approaches and techniques [43, 62, 40]. Recently, there were limited researches that tried to study new MOEAs. This Study analyzed IBEA [66] and since IBEA is known for its high computation cost [44], I employed a modified enhanced flavor IBEA named Modified IBEA [35] and conducted a comparison with the famous NSGA-II [11].

This study compared IBEA, Modified IBEA and NSGA-II in terms of branch coverage (effectiveness), Evolution time (efficiency) and other non-coverage testing metrics like evolving test-suites that have the minimum execution time and length possible. I found out that NSGA-II and Modified IBEA achieved higher branch coverage over IBEA. NSGA-II was the most efficient algorithm and IBEA evolved test suites that have shortest execution time and smaller in size.

1.1 Problem Statement

With the modern software development methodologies and the expeditious development that it employs, constructing test suites that could effectively detect defects would slow down the development pace. Consequently, having a tool that could generate those unit tests within a reasonable amount of time would be a valuable asset for software developers and would benefit the development process.

Many optimization algorithms have been adopted to generate test inputs for the system under test automatically. Moreover, considerable research has adjusted and modified the algorithms for the sake of achieving better results. Nonetheless, and with the significant contribution of search-based software testing, I could not find to our best knowledge, a research that analyses the use of Indicator-Based Evolutionary Algorithms (IBEA). Add to this, I selected IBEA due to its popularity in the

Search-based software engineering field as one of the effective Indicator-based MOEAs. IBEA tries to incorporate practical decision making and valuable information to decision-makers who are interested in the regions of Pareto Front. This is all occurring in IBEA which accomplishes that by either maximizing or minimizing some performance indicator. And thus, IBEA is considered as one of the powerful MOEAs as a collective approach that tries to employ the burden of selection operation in order to maximize the performance of all individuals in the population [14].

Many evolutionary algorithms (EA) have been used and examined in the test generation problem. However, most of them adopted a single objective as a criterion of completeness, meaning, they aimed to find a single test case that achieves a particular target at a time. The main drawback of this approach is that it assumes that all goals are equally important, have the same difficulty to be achieved, and are independent of each other. In reality, none of these assumptions are valid. For example, most of the algorithm resources could be spent in trying to cover an infeasible branch while there are other more feasible branches which are easier to be covered. To get over those issues, the concept of whole test suite optimization (WTS) has arisen [18]. Briefly, WTS focuses on generating a test suite (which is a group of test cases) that achieves a specific set of goals/objectives. So, instead of generating a test case that covers a branch at a time, WTS produces a set of test cases that covers a group of

branches I specify. It was proven that WTS outperforms previous search-based approaches used in test case generation [20].

WTS tries to generate test-suites that achieve several groups of targets that belong to the same family collectively, code coverage is as an example of such a target; code coverage is of many types, it could be statement, path, branch or even function. its upon the research preferences to choose what type of coverage to use in his research. I intend to introduce another objective to the solutions generated by the WTS approach that is of specific interest to the software engineers working on an industrial project. This study will use the test case execution time as the second objective and hence the MOEA will be implied to generate test-suites that achieve the higher branch coverage with less execution time possible. Thus, I intend to employ the Indicator-Based Evolutionary Algorithm (IBEA) [66] as a multi objective-based algorithm for optimizing generated test suites to achieve high branch coverage without causing an increase in test cases execution time.

Instead of using Pareto dominance to evaluate the quality of solutions, IBEA uses an indicator function (typically hypervolume but other indicator functions can be used). Zitzler and Künzli [66] have shown on different benchmarks that IBEA has performed better than the two popular algorithms NSGA-II and SPEA2, concerning different performance measures. IBEA was also proven to have an advantage over other algorithms in many-objective problems, i.e., problems with four or more objectives [52], [41].

1.2 Research Questions

In this thesis, I intend to use white-box testing techniques to analyze a program source code, and automatically generate feasible test cases with a proper set of assertions that examine the System Under Test (SUT) and checks its behavior. The test cases correctness and feasibility are measured by specific metrics, such as code coverage, number of mutations revealed, execution time and resource usage. For a test-case to be valuable for the developer, it should at least achieve branch coverage with minimal time to execute; developers also favour test cases that are of smaller size, because they would be easier to read, maintain. Moreover, writing test oracles for smaller test cases are highly preferred over larger complex ones.

To improve the performance of IBEA and reduce the computation time, I will be examining a modified flavour of the general IBEA. Our main objective is to examine the basic IBEA and analyze how it would perform in the domain of test case generation. The Basic algorithm will be compared to one of the MOEAs algorithms such as NSGA-II. Then I will try to modify IBEA and analyze its results again. Previous studies that attempted to improve IBEA are not too many, one study proposed a simple and fast hyper-volume indicator-based MOEA (FV-MOEA), which was

suggested to quickly update the exact HV contributions of different solutions by deleting the irrelevant ones [34], hence reducing IBEA's computation time. A second study presented a new algorithm called Modified Indicator-based Evolutionary Algorithm (mIBEA) which improves the general IBEA by embedding an additional Pareto-dominance based component for selection and thus tries to achieve diversity of the evolved solutions set [35].

RQ1: What is the branch coverage of IBEA and Modified IBEA when being compared to other Multi-Objective Algorithms (MOAs) in generating test cases?

The first objective that any developer has in mind when writing a test suite is to construct a set of test cases that achieves the highest coverage possible for the class under test. In order to accomplish that, one needs to provide test cases with different test inputs that cover every feasible condition in his code.

The most commonly used coverage metric in Software-based search testing is branch coverage. A software application consists of conditional statements such as if conditions and loops which are guarded by logical predicates. Branch coverage requires that each predicate should be evaluated to both true and false. A branch that is not reachable or has a predicate that will never evaluate to both true and false is called an infeasible branch. Thus, an optimal test suit would be the one that can cover all feasible branches.

One way to compare the algorithms is to check which one produces test suites that pull off higher branch coverage within a specified search budget. So, our first question aims to measure the effectiveness of the algorithms by figuring out the branch coverage score of IBEA algorithms and compare them with other algorithms in the literature.

RQ2: To what extent is IBEA efficient when being compared to other MOAs in generating test cases?

One crucial factor that should be taken into consideration while generating the test suites is time. Since I have described how software development paradigms enforce rapid developments and delivery, it's important that the algorithms should be capable of generating test suite in a reasonable period for the system under test.

Execution time is defined as the time taken for the software test suite generation process [44]. Execution time will be affected by some factors that are related to the nature of the system under test, like dependency between classes, size of classes (i.e. the number of code lines), number of branches to cover as well as to the machine resources on which the generation will be held.

This study work towards optimizing test cases to attain high coverage test suites with minimal execution time. In other words, answering this question will reveal how efficient are IBEA and modified versions of the IBEA algorithm in such an optimization problem where I would need to

get the highest coverage possible with minimal time. The lower the execution time the more cost-effective would the algorithm be.

RQ3: What is the size of the test suite generated by IBEA when being compared to other MOAs in generating test cases?

Automating the process of generating test Suites is an excellent tool and makes life easier for developers. Yet, the developer has to evaluate the test cases manually and examine if they meet the needs and are logically correct. As a result and for the tools to be usable by practitioners, its generated test cases should be of small size otherwise adding test oracles to the generated test cases would be a tedious task.

Having said that, such an optimization problem to minimize test suite size and preserve branch coverage must be shed light on. The relation of test cases size and its coverage percentage is correlative; since in most cases, larger test cases cover more branches. However, that might not be the situation in test Case generation tools where the code statements are being injected into a test case randomly and only then can be evaluated against the subject goals.

The last question of our test cases aims to evaluate what we call the test oracle cost. Test Oracles are used to assess the correctness of the observed behavior upon executing the test suite [24]. A usual scenario during writing tests for a software application is therefore test inputs are being generated automatically and developers add the test oracles manually. Hence, for a test case to be considered valuable, it should achieve

high branch coverage and be small in size, since more minor test cases are more comfortable with producing assertions for.

1.3 Research Contribution

This thesis would be the first to use basic IBEA in test suite generation that aims to achieve a group of predetermined goals all together at a time. Due to the fact that IBEA has some calculation cost, lots of researchers believed that it wouldn't be useful in such problems where many objectives have to be achieved together [43]. However, I aim to apply a modified versions of IBEA and employ it in a computationally light manner and investigate whether this approach can outperform or at least equally compete with other used algorithms.

1.3.1 Enhancing *EVOSUITE* Tool with IBEA algorithms

I plan to contribute to *EVOSUITE* tool by providing the implementations of evolutionary algorithms that I am going to examine as part of this study. *EVOSUITE* is a tool that automatically generates test suite with their assertions for classes that are written in any language compiled to Java Virtual machine byte-code, it targets code coverage criteria such as branch or statement coverage. Moreover, *EVOSUITE* helps developers in determining whether the generated test suites are valuable and correct enough to be dependent on by providing a small group of assertion statements that validates SUT behavior.

As *EVOSUITE* is designed to be extensible, then the main advantage of using this tool is that it saves us from investing efforts and time in performing byte-code instrumentation, as well as to handle static and dynamic analysis on the source code and develop the needed infrastructure to generate test cases based on the previous activities. *EVOSUITE* lacks any IBEA implementations. Hence, I will be the first among those who contributed to *EVOSUITE* with the primary and adjusted IBEA algorithm.

In addition to developing the algorithms, I found that the tool was not ready to have the test-suite execution time and length as an Objective functions, so I implemented the missing objective functions, integrated them with the tool and used them in our study. Add to this, some metrics of test-cases and suites are not natively measured in *EVOSUITE* so I added the missing code to measure expose those metrics to be reported properly as a other *EVOSUITE* metrics.

1.3.2 Validating Contribution on well Known set of Classes

After implementing the algorithms and integrating them with *EVOSUITE*, they will be examined against a well know set of open source libraries and applications. I used the same data-set used by most of the researches conducted on *EVOSUITE*. Mainly I will be using a subset of classes that were used by Fraser and Arcuri. [16]. In this study, I will select a group of diverse classes from different software applications. Some of which are

open source projects developed by Google and Apache Software Foundation.

To avoid having biased results, I will bear in mind the domain of classes during the result analysis and discussions; meaning, I will differentiate between classes that have many String or text operations from those who are numerical or have a lot of array processing. The nature of the class really affects the test case generation, or to be more precise, the type of branches inside a class under test has a direct impact, because different branch types have various means of condition resolution, and as a result, the branch coverage is affected. For example, String-based conditions are harder to satisfy when being compared to integer-based conditions [55].

1.3.3 Addressing the Problem Through Many objectives

Unlike most research that has been conducted in this area, I intend to use more than one criterion in the evaluation function of the evolutionary algorithms. Most studies took branch coverage as the main criterion and the size of the generated test suite as the secondary criterion, so if two test suites have been evaluated to be of the same value to the system under test, the one with a smaller size will be selected. This study will consider optimizing the branch coverage achieved by a test-suite to its execution time and size.

In this scope we define the optimization objectives as:

- A How many branches the test case covers.
- B Test case Execution time
- C Test case length i.e size

1.3.4 Comparison Between IBEA, mIBEA and NSGA-II

Search based software engineering fields have been used in many software engineering aspects. Moreover, a lot of algorithms have been harnessed in search-based software testing. Whenever a new algorithm was employed or a new approach was examined, a comparison is always held with previous algorithms or strategies. Arcuri et al. compared their whole test suite generation with single target approach, even more, the WTS generation was also compared to Random generation as in [55]

I plan to analyze the behavior of IBEA algorithms and check their performance in the field of test-cases generation. Moreover, I plan to use a modified version of basic IBEA algorithm and examine if it can suit the test generation problem. Add to this; a generic comparison will be conducted to compare IBEA with other MOEAs, specifically the NSGA-II.

The algorithms employed in this study will be compared to each other using the following effectiveness metrics,

- Test case characteristics
 - Achieved branch coverage

- Test case execution
 - Test case size
- Evolution time of an Algorithm

1.4 Research Overview

The rest of this thesis is divided into the following:

- **Background:** A description of the theory behind this study and the tools used. Meta-heuristic optimization is explained in the context of this thesis study describing non-dominated solutions and their relation to the solutions that can be provided for the decision makers.
- **Related Work:** Intensive literature review investigating the studies which focus on test suite generation especially those studies that used the evolutionary algorithms.
- **Research Methodology And Experiment Setup:** Describing the different data set used in this study in addition to the setup used. This includes the algorithms configuration.
- **Experiments Results and Analysis:** I will present the results in this section taking into consideration the domain in which the classes under test belongs.

- **Conclusion:** summarizing the results of this study while presenting the threats to validity. Future work is suggested in this chapter based on this study results.

1.5 Research Activities

I will collect different classes from different domains and application to ensure the diversity and emit any biasing probabilities. The experiment will be conducted on the same machine/environment for each algorithm and under same circumstances. Moreover, The following activities are conducted:

- **Detailed *EVOSUITE* experiment:** I will extend *EVOSUITE* and implement the algorithms from scratch and only then start our experiment. Moreover, I contributed to the tool by adding two objective functions.
- **Algorithms Comparison:** I will be using our newly added IBEA, modified IBEA and NSGA-II algorithm that was developed inside *EVOSUITE* and do the comparison.
- **Algorithm minimal run time:** I will be taking into consideration the execution time of the subject algorithms and check which one gives the best result within a certain time budget.

Chapter 2

Background

Test case generation aims to ease the software development process where developers should right quality tests for their code. Several attributes are taken into consideration when adopting such a tool, such as the correctness of a test case, execution time and code coverage (either branch or statement). Developers tend to use tools that generate small test cases that achieves higher code coverage with a reasonable amount of time. It's worth noting here, that coders favor small test cases or suites because it needs less effort in writing test oracles comparing to a larger one.

Having said that, test case generation involves competing objectives such as test case length, the execution time of a test case, statement or branch coverage and resource consumption. For example, the number of statements could compete with the statement coverage, since covering a specific code in SUT requires higher test scenarios. In addition, the number of statements could compete with memory consumption and the execution time; the longer is the test-case the higher would be the execution time and resource usage. Moreover, some of those objectives could

also compete with the configurations set of the generating algorithms, for example, the longer the search budget, the better is the test case coverage.

This chapter presents the theoretical background in detail behind Evolutionary Algorithms, Single Objective Algorithms, multi-objective algorithms, and shed light on the algorithms used for this research. Also, I will shed light and analyze the hypervolume indicator which is used in this research as well as to detailed explanation about the *EVOSUITE* tool.

2.1 Evolutionary Algorithm

Evolutionary Algorithms use a kind of meta-heuristic search technique inspired by the way nature evolves species using a natural selection of the fittest individuals to solve an optimization problem. In our case, a Genetic Algorithm is used to convert the task of test case generation into an optimal problem; this activity can be referred to as Evolutionary Testing. Evolutionary algorithms work with a set of solutions (populations) so that they can provide a set of satisfactory solutions. They also do not use any gradient information but on the contrary, they are based on the random determination.

In Principle, Evolutionary algorithms are family of computational algorithms that are inspired by evolution and decodes a solution of a particular problem into a data structure that groups a certain number of

genes called chromosome and then perform manipulation on those chromosomes to preserve critical information. So I consider the space of possible chromosomes as a solution space to search in, and apply the EA approach to search for the right solution. Evolutionary algorithms are usually seen as an objective optimizer, yet it's well known that they applied to a wide range of problems [62]. EA is commonly used to generate high-quality solutions for optimization problems and search problems and Genetic Algorithms (GA) are one of the most used among the evolutionary-based algorithms.

2.1.1 General Design of Genetic Algorithm

Genetic Algorithm (GA) is one of the evolutionary algorithm family. GA has emerged as a practical, robust optimization technique and search method [56]. GA usually starts by generating a random population of chromosomes among the search space; an evaluation is held over the chromosomes via a specific fitness function deduced from the problem's environment. Then, The probabilities of reproductions are determined; the chromosomes which represent a better solution are given more chances to reproduce comparing to those of imperfect solutions. Production of the new off-springs takes place via Genetic operations that will be discussed in section 2.1.2.

In John Holland approach[29], one parent is selected using the selection operator and based on its fitness value, while if a crossover is to be

performed then the other parent is chosen randomly [47]. However, the algorithm can be summarized as illustrated in the following steps,

- Step 1: Randomly initialize populations P
- Step 2: Determine *fitness* of population
- Step 3: Until convergence or a termination condition is satisfied repeat:
 - Step 4: *Select* parents from population
 - Step 5: Generate new population and perform *crossover*
 - Step 6: Perform *mutation* on new population
 - Step 7: *Calculate fitness* for new population
- Step 8: return solution

2.1.2 Genetic Operators

One main step in the GA is the production of new population and performing some genetic operations in a trial of getting offspring that are better than the producing parents. Production of successive offspring are performed using the following operators[56],

1. Selection Operator

Since GA will be producing new generations, then there is a possibility that the size of current population will be twice the size of the

initial population; thus, I need an operator to select individuals in order to maintain the proper population size and offer preferences to the chromosomes with better fitness values and enable them to pass their genes to offspring (successive generations)

2. Crossover Operator

Crossover operator is performed over the selection. In which two individuals (parents) are selected using selection operator, the output of this operator is a couple of chromosomes that inherits the parents' genes. The operation starts by determining the crossover points and next those genes at these points are manipulated either by swapping locations, or flipping bits, or reordering the sequence to create new offspring. Crossover probability is used to determine how often will be crossover performed. If there is no crossover, offspring is exact copy of parents. According to Rähä in [46], there is a relation between crossover and fitness value of the offsprings; this can be explained by applying higher crossover rates in order to get better off-springs from parents. However, this is can't be guaranteed in the evolutionary computation because crossover is made in hope that new chromosomes will have the right parts of old chromosomes, and maybe the offspring would be better.

3. Mutation Operator

by which I try to enhance the solution set with a diverse population

by manipulating random genes in offspring. This could be accomplished by flipping the value between 0 and 1 as in the binary chromosome or by swapping the location of two numbers in an integer chromosome. However, before performing mutation, a validation process should take place in order to make sure that the mutated chromosome is still valid [46]. Since validating the chromosome after mutation could consume computation resources and increase the algorithm execution time, then the rate of the mutation process (i.e. mutation probability) is set to a minimum value to avoid such a computational overhead. Yet, the mutation probability should be greater than zero to make sure I achieve diversity among offspring and prevent GA from sticking into local optima. Moreover, the mutation rate should not occur very often. Otherwise GA will in fact change to random search.[56]

2.2 Single-objective Evolutionary Algorithms

A single objective or parameter optimization problem can be defined as the procedure of optimizing a set of variables to minimize or maximize a particular objective. This kind of problems might be viewed as a black box with a group of control knobs representing different parameters and need to be adjusted to achieve some goal. Thus, the only output of this black box is a single value that yielded by an evaluation function which indicates how well a specific setting of the parameters (knobs) solves the

optimization function. So the aim would be to find the best set of those parameters that better optimizes the output.

In search-based software testing (SBSE), Genetic algorithms are used to search for optimal test parameters or test inputs combinations that meet a predefined test criterion and hence one can say that testing problem is considered as a search problem. For example. In our study, this could be mapped to finding out a set of test cases that maximizes the branch coverage or minimizes the execution time.

In a single objective scheme, GA algorithms will keep developing generations of test cases and measure how good are they in achieving a certain objective using a fitness function; some of the individual coverage goals that are used in SBSE are code coverage of white box testing which finds the areas of the program not exercised by a set of test cases such as (1) Branch Coverage (2) Line Coverage (3) Weak Mutation testing. In the next section, I will be explaining in detail the branch coverage used in this study.

2.3 Multi-objective Evolutionary Algorithms

Evolutionary multi-objective optimization (EMO) is one of the most active research areas in the field of evolutionary computation as Many real-world decision-making problems have several objectives to achieves; the most recent EMO algorithms share the following three common features

among them: (1) Pareto ranking, (2) diversity preserving, (3) and elitism [32]

In the field of software testing, a practitioner would like (for example) to generate test cases that maximize fault detection, different types of code coverage and minimize execution costs. Genetic algorithms also present advantages in multi-objective inverse problems, because they can determine reliably the Pareto frontier in one single simulation run [10].

The main goal of the single-objective GA algorithm is to find the best solution that can actively minimize or maximize a particular objective function by lumping all objectives into one [51]. Such an algorithm can be used to give practitioners an indication or a way of exploration to the problem domain but they can never offer a set of alternative solutions that trade objectives against each other. However, in multi-objective problems where there are several competing objectives, there is no single optimal solution; indeed, there are several conflicting objectives that might interact and yields a set of compromising solutions known as the trade-off, non dominated, non-inferior or Pareto-optimal solutions [51]. Multi-objective algorithms provide three significant improvements to the single one,

- Identifying a wider range of alternative solutions
- Modeling real-world problem in a more realistic way
- Assign more appropriate roles to those who are part of the decision-making process like modelers and decision-makers.

The main goal of MOAs is to find a set of Pareto-optimal solutions called the Pareto Front (PF). This PF will eventually contain a set of solutions that do not dominate each other in the search space and represent the trade-off between different objectives; that set is named *non-dominated* solutions[35]. Having said that, a solution $x^{(1)}$ is said to be dominating $x^{(2)}$ if $x^{(1)}$ is not worse than $x^{(2)}$ in all objectives and $x^{(1)}$ is better than $x^{(2)}$ in one or more objectives [37]. A fitness criterion (fitness function) for each objective is then used to measure how much the objective is good, and this value could be maximized or minimized according to the problem domain.

2.3.1 Indicator based Evolutionary Algorithm

IBEA [66] is one of multi-objective optimization algorithm that is a preference based evolutionary algorithm. The main advantage of the algorithm is that it has the capability to incorporate the preference of user and include that in the search process. The user preference which will be referred to as an indicator is being formed at the beginning of the search process and next will be used in the selection process of next individuals as the solutions evolves. IBEA solves various Multi-Objective Optimizations Problems (MOPs) by applying three main search components (1) fitness assignment, (2) diversity preservation and (3) elitism. The main contribution of IBEA is that it provides a general framework for indicator based Multiple Objective Evolutionary Algorithms (MOEAs), meaning, several quality indicators could be used and integrated within IBEA

main flow to solve a MOPs in any domain.

Figure 2.1 illustrates The selection strategy in IBEA is a binary tournament between randomly chosen individuals. Whereas, the replacement scheme consists of deleting, one-by-one, the worst individuals, as well as to updating the fitness values of the remaining solutions each time there is a deletion; this step is iterated until the needed population size is achieved. In addition to that, an archive stores solutions mapping to potentially non-dominated solutions in order not lose them during the random searching process.

2.3.2 A Modified Indicator-based Evolutionary Algorithm (mIBEA)

This newly proposed algorithm was proposed by Li et al [35], to solve the problem of solutions distributions of IBEA [66] which attempts to guide the solution to the best and true PF using the fitness indicators it associates with the generated solutions. According to the same authors [35], IBEA is found to be trapped in specific areas of the search space and suffers from lack of diversity. The new algorithm mIBEA was just empowered by additional Pareto-dominance component of selection, this component is the non-dominated sorting of NSGA-II [11].

mIBEA is a modification to IBEA in which it eliminates the non-dominated solutions at each generation by embedding the dominance-based sorting

Input: α (population size)
 N (maximum number of generations)
 κ (fitness scaling factor)
Output: A (Pareto set approximation)

Step 1: **Initialization:** Generate an initial population P of size α ; and an initial mating pool P' of size α ; append P' to P ; set the generation counter m to 0.

Step 2: **Fitness assignment:** Calculate fitness values of individuals in P , i.e., for all $\mathbf{x}_1 \in P$ set

$$F(\mathbf{x}_1) = \sum_{\mathbf{x}_2 \in P \setminus \{\mathbf{x}_1\}} -e^{-I(\{\mathbf{x}_2\}, \{\mathbf{x}_1\})/\kappa} \quad (2)$$

Where $I(\cdot)$ is a dominance-preserving binary indicator.

Step 3: **Environmental selection:** Iterate the following three steps until the size of population P does not exceed α :

1. Choose an individual $\mathbf{x}^* \in P$ with the smallest fitness value, i.e., $F(\mathbf{x}^*) \leq F(\mathbf{x})$ for all $\mathbf{x} \in P$.
2. Remove \mathbf{x}^* from the population.
3. Update the fitness values of the remaining individuals, i.e.

$$F(\mathbf{x}) = F(\mathbf{x}) + e^{-I(\{\mathbf{x}_2\}, \{\mathbf{x}_1\})/\kappa} \text{ for all } \mathbf{x} \in P.$$

Step 4: **Termination:** If $m \geq N$ or another stopping criterion is satisfied then set A to the set of decision vectors represented by the nondominated individuals in P . Stop.

Step 5: **Mating selection:** Perform binary tournament selection with replacement on P in order to fill the temporary mating pool P' .

Step 6: **Variation:** Apply recombination and mutation operators to the mating pool P' and add the resulting offspring to P . Increment the generation counter ($m = m + 1$) and go to Step 2.

of NSGA-II. The embedding of the non-dominance sorting of NSGA-II is illustrated in the following mIBEA algorithm Pseudo code .

Algorithm: mIBEA Pseudo Code [35]

Input : α (*populationsize*);

N (*Maximumnumberofgenerations*);

κ (*fitnessscalingfactor*);

Output: A (*Paretosetapproximation*)

1 **Step1** : Initialize population

2 **Step2.1:** Use the fast non-dominated sorting of NSGA-II to get non-dominated solutions in P and use the non-dominated solutions as the new P .

1. rank the solutions in P : $\text{Ranking rankedP} = \text{new Ranking}(P)$;

2. get the non-dominated solutions: $P = \text{rankedP.getSubfront}(0)$;

Step2.2: Fitness Assignment of IBEA as in Fig 2.1

Steps 3-6 : are the same as the original IBEA in Fig 2.1

2.3.3 Non-dominated Sorting Genetic Algorithm II

Non-dominated Sorting Genetic Algorithm II which is abbreviated by (NSGA-II) is a Multi-Objective Optimization algorithm that belongs to the family of Genetic Algorithms from the field of Evolutionary Computation, and in the taxonomy of Evolutionary Algorithms it's one of the MOEAs. There are two versions of the algorithm, the classical NSGA and the updated form NSGA-II which tries to improve the adaptive fit of a population of candidate solutions to a Pareto front restricted by several objective functions [8].

NSGA-II is similar to any Genetic algorithm in terms of generating population and genetic operations. Its strategy depends on sorting the population into a hierarchy of sub-populations (called fronts) based on the ordering of Pareto dominance; the similarity between individuals in the same front is calculated on the Pareto front and the resulting groups and measures are being used to select a diverse front of non-dominated solutions. NSGA-II solved predecessor MOEAs drawbacks and limitations, the first was the computational complexity which leads to an increase in resources used during execution, NSGA-II reduced the complexity from $O(mN^3)$ to $O(mN^2)$ [11]. The second thing is the lack of elitism, NSGA-II has introduced the elitism which improved the GA performance and helped to preserve good solution. The final point is the sharing parameter used to achieve variations in found solutions; NSGA-II has abandoned this and introduced a selection operator to select the best chromosome from parents/children archive referring to its fitness and spread values. NSGA-II was found to add diversity solutions obtained in the Pareto-front outperformed the previous MOEAs: Pareto-archived evolution strategy (PAES) and strength Pareto EA (SPEA) [11]

The multi-objective evaluation of each solution in NSGA-II is based on Pareto ranking and a crowding measure. Figure 2.2 illustrated NSGA-II procedure which can be summarized in the following four steps [1] [11],

- Step 1: R_t population of the size $2N$ is randomly generated, this

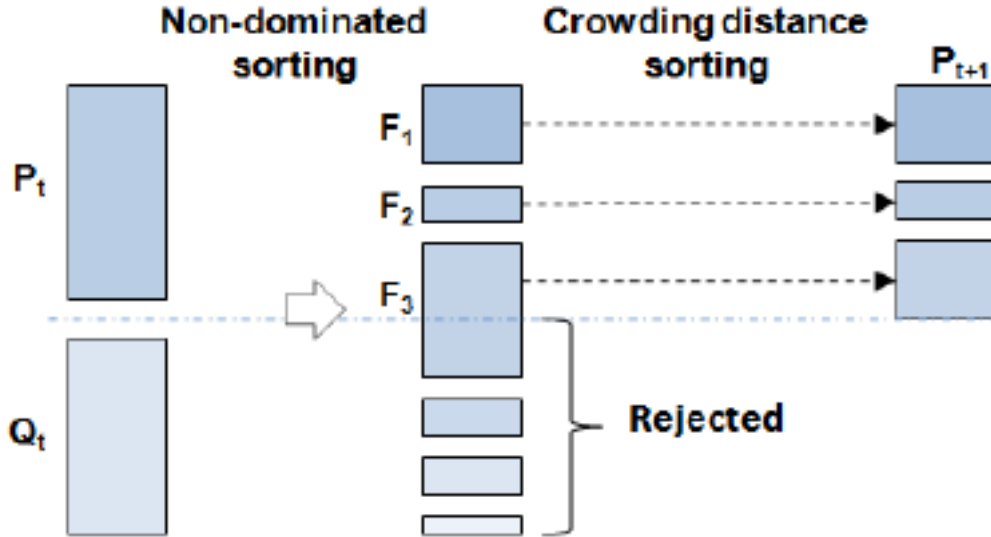


FIGURE 2.2: NSGA-II Algorithm Procedure [1]

population includes parent population P_t and offspring Q_t population, each of size N . Genetic Operators including Binary tournament selection, crossover and mutation are applied to produce the offspring Q_t . Then the population R_t are assigned the fitness value and sorted according to the non-domination level where minimizing the fitness value is needed. The result of this sorting procedure will yield fronts F_i where $i = 1, 2, 3, \dots$

- Step 2: a new population $P_{(t+1)}$ is being constructed from the best F ranked and passed to the next step. If the size of F_1 is less than N , then the population includes chromosomes from $F_2, 3, \dots$, and so on. This procedure is continued until no more sets can be selected
- Step 3: if the size of F_1 is greater than N then to choose exactly N ,

the chromosomes are being sorted using the crowding distance approach in descending order; this is done by measuring the density of the solution from its neighbors and then comparing values between all solutions. Once sorting is done and selected individuals are and pass the selected to the next iteration, then the rest of the solutions in $F_2, 3, \dots$ will be dropped.

- Step 4: Operators including crowded tournament selection, crossover and mutation will be applied to get a new population $Q(t - 1)$ form $P(t + 1)$. In each iteration, a counter i is increased by 1

2.4 EVOSUITE

EVOSUITE is a tool that was constructed in 2010 by Fraser and Arcuri as an output of a research project and is often referred to as one of the main reference tools for search-based software testing [26]. It automatically generates unit tests for Java software using Evolutionary Algorithms. It is well developed sophisticated tool that can be called via command line or can be integrated as a plugin within Maven, Eclipse and IntelliJ; it has been used on more than a hundred of open source projects and found to be effective in revealing bug [22].

EVOSUITE mainly operates on the byte-code level of and collects all needed information from the classes under test (CUT) using Java reflection. thus, it does not require the source code of the CUT and it could be used over any programming language that compiles to java byte-code

such as Scala. *EVOSUITE* for now can only cover code that has no dependencies and assumes that CUT is deterministic, meaning, if the generated unit test was executed ten times, then it will yield the same result [22]. Having said that, its worth noting here that such a tool can't be used in the Test Driven Development (TDD) schemes where automated tests are being written before developing the functional code in small efficient iterations [33, 6]

2.4.1 Unit Testing and Class Under Test

Unit testing is one level of software testing where a single component or unit is being tested. The main goal of unit testing is to examine that a particular component meets the required needs and performs as designed [36]. A unit is considered the smallest piece of the software product that can be tested and in most of the cases, it has several inputs and a single output. A test suite is a group of unit tests for a CUT, and each test-case represents an absolute test scenario. The CUT is formed from a set of methods; each one of them contains a list of statements. Statement types can vary and they could be either a constructor statement, a conditional statement (e.g. if), a method call or a regular statement. A unit test then can be defined as a function that represents a test scenario and when executed, it can call some methods in the CUT and checks that the observed behavior matches the expected one. Moreover, Unit testing is considered another type of code documentation, it can help developers understand the behavior of the software [59].

The following code snippet shows a sample of a test case that was generated during running the experiment of this thesis. This is not the full test-case generated its just a sample for the sake of demonstration purposes.

2.4.2 Types of Code Coverage

There are two main types of testing techniques, black and white box testing. Black box testing is always referred to as Functional testing, where the writing of test cases depends on the requirements and design documents. The second type is white box testing and its referred to as Structural testing, where tests are written based on the implementation since all internal structures of program under test is exposed and available for testers [36].

Here are the mostly used code coverage criteria in software testing literature [12, 30, 63],

- Line (Statement) Coverage: number of statement exercised in CUT when a test is executed
- Function Coverage: number of functions invoked in CUT when a test is executed
- Edge Coverage: number of edges in the control flow graph of CUT that were visited when a test is executed

- Branch coverage: also called DD-Path and is a subset of Edge Coverage which is number of times each branch is being evaluated in CUT.
- Condition coverage: sometimes its called Predicate coverage, and it refers to number of Boolean sub conditions that were evaluated to both True and False.
- Path Coverage: number of times every path in CUT control graph is being executed. number of paths in CUT maybe exponentially related to number of code lines in CUT

There are many other types of coverage like loop, block and state coverage, but we defined the most relative once to our study and explained in detail those that are supported in *EVOSUITE*. This Study will depend on **branch coverage**, since each covering all branches implies covering all possible scenarios [48], Moreover, branch along with path coverage are found to be the best code coverage criteria when being employed in generating effective test suites [31]. Its worth noting here that the branch coverage checks and deals with the full condition statement unlike the condition coverage which checks every sub condition in the main condition statement. The fitness function that is built to evaluate the branch coverage of test-suites is being discussed in details in section 4.2.2.3.

```
@RunWith(EvoRunner.class) @EvoRunnerParameters(
```

```
mockJVMNonDeterminism = true,
useVFS = true, useVNET = true, resetStaticState = true,
separateClassLoader = true, useJEE = true)
public class ComboLeg_ESTest extends ComboLeg_ESTest_scaffolding {

    @Test(timeout = 4000)
    public void test0() throws Throwable {
        ComboLeg comboLeg0 = new ComboLeg();
        ComboLeg comboLeg1 = new ComboLeg();
        assertTrue(comboLeg1.equals((Object)comboLeg0));
        comboLeg0.m_designatedLocation = "k;Py5!PA8DyJo9r";
        boolean boolean0 = comboLeg0.equals(comboLeg1);
        assertFalse(comboLeg1.equals((Object)comboLeg0));
        assertFalse(boolean0);
    }

    @Test(timeout = 4000)
    public void test1() throws Throwable {
        ComboLeg comboLeg0 = new ComboLeg();
        ComboLeg comboLeg1 = new ComboLeg();
        assertTrue(comboLeg1.equals((Object)comboLeg0));
        comboLeg1.m_exchange = null;
        comboLeg1.m_exchange = "k;Py5!PA8DyJo9r";
        boolean boolean0 = comboLeg0.equals(comboLeg1);
        assertFalse(comboLeg1.equals((Object)comboLeg0));
        assertFalse(boolean0);
    }
}
```

```
}  
  
@Test(timeout = 4000)  
public void test2() throws Throwable {  
    ComboLeg comboLeg0 = new ComboLeg();  
    ComboLeg comboLeg1 = new ComboLeg();  
    boolean boolean0 = comboLeg0.equals(comboLeg1);  
    assertEquals(0, comboLeg1.m_conId);  
    assertEquals(0, comboLeg1.m_shortSaleSlot);  
    assertEquals(0, comboLeg1.m_ratio);  
    assertEquals(0, comboLeg1.m_openClose);  
    assertTrue(boolean0);  
}  
  
@Test(timeout = 4000)  
public void test6() throws Throwable {  
    ComboLeg comboLeg0 = new ComboLeg(1212, 0,  
        ",S4jnGwcr4kB5", ",S4jnGwcr4kB5", 0, 0, "");  
    ComboLeg comboLeg1 = new ComboLeg();  
    boolean boolean0 = comboLeg0.equals(comboLeg1);  
    assertEquals(0, comboLeg0.m_shortSaleSlot);  
    assertFalse(boolean0);  
    assertEquals(0, comboLeg0.m_openClose);  
    assertEquals(0, comboLeg0.m_ratio);  
    assertEquals(1212, comboLeg0.m_conId);  
}  
}
```

2.4.3 Whole Test Suite Generation

Before the use of Multi-Objective Evolutionary testing in Search-Based Software Testing, the common approach in the literature was to generate a test case for each objective/goal at a time and then to recombine them in a test suite. The idea of Whole Test Suite (WTS) Optimization has been first proposed by Fraser and Arcuri [22] which was implemented in *EVOSUITE* tool [20] as being stated in Section 3.2. WTS approach tends to produce a set of test-suites that covers all testing goals at the same time instead of deriving test cases to cover a certain objective at a time. The idea of WTS was brought to solve two main issues of the single test case approach:

1. Difficult targets: target varies in the complexity level of being covered. Thus, the single approach could waste a significant amount of the search budget for just trying to achieve a single test goal.
2. Infeasible Targets: some targets are infeasible to cover, and in practice, they could not be covered at all and if our approach were not design to eliminate such a targets then the search budget would be wasted and even if the approach generated a test case for infeasible target, the would not be valuable to the software engineer.

WTS approach which was first proposed by Fraser[18], starts with an initial population of test suites that were randomly constructed. Next, it uses GA to optimize the generated populations towards stratifying certain criteria, while using the test suite size a secondary testing objective.

Finally, the best resulting test suite is being minimized using the following algorithm proposed by [2] which tries to eliminate one statement at a time until the remaining statements contribute to the testing goal. The main advantage of this minimization process is the reduction of both numbers of test cases as well as to their length.

2.4.4 Problem Representation

The first step to applying a search problem in GA to an engineering problem is to define the representation of the valid. In this study I use the same representation as [22] in which they defined the valid solutions for the testing problem as *testsuite* referred to as T of test cases t_i . Given that $|T| = n$ I will be having $T = \{t_1, t_2, \dots, T_n\}$. In a unit testing scope, a unit test t is a program that executes the CUT. Thus a test case requires to be developed in a programming language (in our case Java) and allows research to encode solutions and optimize them to solve the problem, A test case is defined as a sequence of statements $t = s\{s_1, s_2, \dots, s_n\}$ of length l [58]. The length of the test suite is defined as the sum of all its test-cases length i.e. $length(T) = \sum_{t \in T} l_t$

To guide the selection of parents for the sake of producing off-springs, I use a fitness function that evaluates the set of solutions. If two test suites have the same fitness value, then the selection mechanism rewards the test suite with fewer statements, i.e., the shorter one as it would be simpler to write test oracles for small test cases rather than bigger ones. In this study, we focus on the branch coverage as a test criterion, and our

optimal solution T_o would be the solution that covers all feasible branches and methods of CUT, has the minimal number of test statements and has the faster execution time. All this information along with the genetic operators used in *EVOSUITE* will be discussed in chapter 4 .

Chapter 3

Related Work

Testing software under development and ensuring quality gets more complicated as the software evolves. Moreover, the sooner I find the bugs the lower will be the cost and impact on the software. It's been known that the main objective of software testing is finding defects; however, one primary purpose is to secure defect prevention. Defect prevention techniques can be achieved by writing test cases to ensure that the current system meets the requirements specified, responds appropriately to the given input and performs its operation within an acceptable period of time. These test cases can be written at many levels, the first would be functions and classes and that's usually called unit testing and it can be extended to modules level. The second is to write tests to examine the system using its Application Program Interface (APIs) and that includes Smoke, Regression and Stress testing. This study will focus on unit test generation although the *EVOSUITE* tool used in the experiment of this study can be assigned to generating tests on the API level as experimented by Arcuri in [5]

Search-Based Software Testing (SBST) is the field where I apply optimization algorithms to problems in software testing; test case and test oracle generations are among the topics which have been invested in during the past two years. Single objective algorithms have been employed firstly in generating test cases, and they aim to find a test case that achieves a single objective. Many tools have been developed based on a single target strategy such as Test-Gen [15] and Quest [9].

In this chapter, I will illustrate the previous studies thoroughly in the domain of automatic test generation. I will mainly shed light on the studies which adopted a single-objective algorithm to automate the generation of test cases. Moving on, I will consider reviewing studies that utilized Multi Objectives evolutionary algorithms, and last but not least, how our research is distinct from other studies. Finally, the literature will be discussing IBEA in the literature review and how was it adjusted to overcome its limitations.

3.1 Single Structural Target Approach

Tonella [58] used the genetic algorithm to automatically generating a unit testing-cases for java objects in generic usage scenarios. His main objective was to achieve a full branch coverage for the class under test. His result showed that GA is effective when being applied in such an optimization problem, and the generation time was reasonable. All of the previous work was implemented in a research prototype named eToc

((evolutionary Testing of classes). Moreover, his tool was designed to add an assertion to validate the test-cases (test oracles). The test-cases are being generated and executed in the form used by the JUnit testing framework. In addition to the GA, Tonella used the greedy algorithm to select the smallest number of test cases that best achieves the target branch coverage and add them to a test suite. His results were both validated by checking the coverage achieved by his test cases and the number of defects they can detect. In this study, they used the mutation testing to insert a group of defects in the CUT and verify that it can be captured by executing the test suites.

McMin and HolCombe Study [40] addressed the challenges that could be faced by evolutionary algorithms when used in testing scope. They stated that the search algorithm might not achieve good code coverage in terms of statements or branches because they do not take into account the dependencies between objects when generating input for white box testing. Their procedure was to combine the Chaining technique to guide algorithms for achieving better results. The Chaining search is an alternative test data generation technique and based on a local search method known as the “alternating variable method”. Chaining search uses the branch distance calculation of a particular branch from an individual node in order to change the flow of control. Generally, the evolutionary algorithm is used to generate test data initially and the chaining approach is used for targeting complex branches. The main fitness function used in this study for evaluating the individuals evolved were the branch

approach level and the Branch distance. Experiments were performed on different seven programs that include a certain challenge involving flags, special function call return values, enumerations and counters that the evolutionary testing did not succeed to generate test inputs for, but with incorporating Chaining local search, the hybrid approach successfully generated test inputs for those programs.

Study of Wappler and Lammermann [60] focused on validating that the evolutionary algorithm is the best to be used for generating unit tests for the white-box testing of the Object-Oriented programs. Some object-oriented classes need a certain object in a certain state to be used when being executed. Thus, their main challenge was to generate test cases that create objects to be used in the test sequence and how to transition them into a proper state to achieve the test goal. This study employed a universal evolutionary algorithm supported by independent toolboxes to be used as an evolutionary operator. Their main procedure extends the chaining approach starts by encoding the object-oriented test programs into built-in basic types such as int and floats and generate a individuals to be used in the test, in order to generate proper individuals they guide the search algorithm by defining three measure to check the fitness of the individuals and those measures are

- number of errors
- constructor distance
- error evaluation

Their experiment was conducted via Matlab and used to carry out two studies, the first was to validate the feasibility of their approach and the result revealed that their approach managed to outperform the random testing in generating objects in certain states that better describe the application under test. Their main objective function was the code coverage criteria. The second experiment was to examine the use of their fitness measure in the objective function and the results were compared to Boolean objective functions and showed that incorporating more sophisticated measures is essential for a successful evolutionary algorithm.

One of the essential studies was carried out by Wegner et al. [61], in which they used the Evolutionary algorithm to generate test inputs for structural tests for procedural programs written in C language. They divided Structural testing into four categories and each category has its own fitness function and hence the test itself was partitioned into several objectives and as a result, the search becomes more goal-oriented. The experiment was conducted on seven functions with cyclomatic complexity between 2 and 34 and the number of branches between 5 and 153 and the number of loops was up to 4. The evolutionary algorithm was executed with 4 or 5 subpopulations containing 40-60 individuals depending on the complexity of function under testing. The results was compared to random testing for all test objects, and the evolutionary algorithm achieved full coverage whereas random testing was not able to achieve full branch coverage for all functions under test.

The main idea behind the whole test suite approach, which targets

multiple objectives during a search run, is to overcome the waste of search budget while seeking to achieve a single objective. Nevertheless, whole suite approaches have been examined on Object-Oriented programs and never been compared to iterative single target approaches in the context of procedural programs. The recent algorithm that has been implemented was LIPS (Linearly Independent Path-based Search) by Panichella et al. [43], They designed an algorithm for generating test data for programs written in C language, the tool was named OCELOT and its an abbreviation for Optimal Coverage sEarch-based tooL for sOftware Test-ing; the authors have built an iterative single target approach that basically makes use of valuable information from the previous iteration. They used default GA configuration with SBX-Crossover, polynomial mutation, and binary tournament selector. The target selection strategy in LIPS was based on the method proposed by McCabe, which computes a maximal set of linearly independent paths of a program (basis) [38], then to evolve a test-cases that covers a certain branch within the path at a time. Thus if they found a test suites that pass through all branches specified in the basis paths, then that implies that they covered all branches in the control flow graph. The results showed that LIPS had shown comparable or better performance than the whole suite approach and achieved higher branch coverage. Yet, one drawback of LIPS is that it could add redundancy to test suite; this issue was mitigated by using the greedy algorithm to minimize the test suite and for fairness was applied on test suites generated by both LIPS and WTS.

3.2 Multi Structural Target Approach

Fraser and Arcuri [22] have first proposed the idea of Whole Test Suite (WTS) Optimization, which was implemented in *EVOSUITE* tool [20]. WTS approach tends to derive a group of test cases that forms a test suite that covers a set of multi objectives simultaneously. According to Fraser and Arcuri, the test objectives were never been independent and not equally difficult. Even more, some of the coverage goals might be infeasible; thus the search algorithm could be affected by order of goals and how many of them are infeasible. Search algorithms require determining two main things, solution representation and fitness function. The solutions are represented by the test case inputs, and with regard to the fitness function, most of the researchers have made use of branch coverage which is represented by how far (distance) a solution is from the target branch. Fraser and Arcuri [22] have defined their solution as a test-suite whereas their fitness function was summing the branches' distances and approach levels of all targeted branches in order to mitigate the dependency between objectives. Their proposed approach has achieved 83% coverage and outperformed the single-target strategy by 7%, and they also generated test cases that were 62% smaller of the single-target strategy. The drawback of their approach was that more accessible branches are most likely to be covered than the harder ones.

Shamshiri et al. [55] conducted a study to compare the effectiveness

of Genetic Algorithms to generating test suites incrementally using Random search for applications written in Object Oriented. They used three algorithms, GA that generates test suites (whole test suite approach), pure Random search and Random search Enhanced with seeding. This study also sheds light on the branch types and how they affect the performance of each algorithm type, and finally, they analyzed the effect of the search budget on the output of the algorithms. The results showed that GA performed slightly better than Random search and the reason behind this (which was found by the study) is the numerical comparison branch types which result in a smooth gradient of fitness values, and such branches are found to be small in number among the applications. Thus, allowing Enhanced Random search to generate test cases without much relative disadvantage and performs nearly similar to GA. The experiment was conducted using *EVOSUITE* and executed 1000 times on randomly selected java classes from SF100 corpus of open-source projects[17]

Fraser et al. [23] enhanced the Genetic Algorithm in their *EVOSUITE* tool by incorporating local search to the individual statements of method sequences. Unlike other studies that used the local search, this study considered using complex data types like arrays and Strings. This study introduced a new algorithm called Memetic Algorithm in which a hybrid of global and local search is used; specifically, they have combined GA and Hill Climbing for container classes. The result of the experiment showed an increase in branch coverage by 38% over the traditional Genetic Algorithm. Yet, the study observed that the effect is very dependent

on the class on which test generation is applied which in return makes it impossible to figure out the optimal configuration of MA.

Panichella et al. [44] have implemented MOSA (Many-Objective Sorting Algorithm) in which they have defined their solution as a single test cases and evaluated its fitness against a set of branches i.e. different branches are considered as a different objectives to be optimized and the fitness function is evaluated to all branches at the same time including those that were not covered yet. MOSA is simply a multi-objective GA algorithm with an enhanced selection mechanism that depends on determining the test cases with lower fitness function (which is the sum of branch distance and approach level) for each uncovered branch. Their sorting was based on the preference of each solution in relative to the Pareto fronts. Briefly, the workflow of the algorithm could be explained as the following, it first selects those test cases with the best fitness function and guarantees that they will survive to the next generation, so the first front is built with preference sorting. Next, it will form the other fronts by sorting the remaining candidate solution using the traditional non-dominated sorting algorithm used by the NSGAI [11]. Once ranked, the test cases are selected for next-generation from parents and offspring based on their ranking and another heuristic value which is "crowd distance" to ensure diversity among the selected test cases. After each generation, MOSA updates an archive of best test cases that covers new branches (uncovered branch) of the class under test to form the final test

suite. Finally, it forms the test suite by selecting the shortest test case covering a certain branch from the archive. MOSA was found to perform better than a single-objective whole test suite approach by *EVOSUITE* and as a result MOSA algorithm was integrated within *EVOSUITE* tool.

One main disadvantage of the previous MOSA algorithm is that it does not take into consideration the structural dependency between targets of the class under test since some targets can be satisfied if and only if other related targets are satisfied. In other words, to have a control dependency within the same class and thus b2 cannot be covered unless b1 is covered. Panichella et al. [42] has empowered MOSA and implemented an enhanced version named DynaMOSA to deduce which targets are independent of each other and which of those are dependent (has control dependency). So, DynaMOSA starts by checking the control dependency graph and collect independent branches and starts to generate test cases for them, on next-generation formed at each iteration, the list of targets will be updated to add those targets that are uncovered and control dependent on the new targets. This process added dynamicity to the MOSA algorithm. Their experiment was conducted on 346 Java classes and the results proved that DynaMOSA performed better than its predecessor MOSA and Whole test suite approaches in terms of Branch, statement and mutation coverage. Besides, they reported that the convergence to a test-suite is accomplished faster in DynaMOsa since they kept the number of objectives small. It's worth noting here that this study did not incorporate non-coverage criteria within the algorithm like memory

consumption and execution time.

Arcuri proposed a new algorithm in his study [4] named Many Independent Objective (MIO) algorithm and was compared with the Whole test suite generation approach and Many Objective Sorting Algorithm (MOSA) with Random generation as a baseline. This study aims to address the limitation of WTS and MOSA, which can be summarized as follows,

- focusing on the exploration and neglecting the exploitation
- keeping the individuals of covered targets in the next generation which in return negatively affect the tests generations if targets were independent
- in the presence of infeasible targets, some tests can get good fitness score since those algorithms used branch distance in the fitness function
- the limitation of generating populations of fixed size

To overcome all mentioned limitations, MIO was designed to operate with dynamic population, dynamic exploration/exploitation trade-off and selecting the next target using sampled feedback. Moreover, MIO controls how many mutations and fitness evaluations should be performed over an individual. MIO keeps an archive of tests for the targets. Initially the algorithm generates the population randomly, but moving on and with a defined certain probability to generate a new test randomly,

MIO will either select from the archive or generate a new one, if the algorithm is to select individuals, then it will select those that are closer to achieve a targets with higher chances to cover. The experiment proved that MIO achieved better results than the other algorithms and in some cases, it achieved an 80+% coverage improvements.

One of the significant studies that have been conducted in the search-based unit test generation was a study by Rojas1 et al. [49]. In this study, the researches tried to target 9 objectives during unit test generation such as line coverage and output coverage. The authors emphasized that The limitation of Whole test suite generation is a weakness in covering all statements of class under test, and that's because Whole test Suite generation is based on branch coverage when evaluating individuals and assumes that if all branches in the control graph were covered then that implicitly means that all code statements could be covered as well. Yet, this was not the case. Thus, So this study aimed to include more objects in the Whole test suite approach to guarantee higher statement coverage. They extended *EVOSUITE* and conducted their experiment on Java classes. They considered that all 9 objectives are non-conflicting. Thus, they used a linear combination of the different objectives and assigned each objective certain weights so that to avoid overfitting for strong objectives. They found out that optimizing and targeting several objectives at the same time is feasible and does not increase the computational cost. Moreover, They found out that average coverage of test suites decreased

by 0.4% and the test suite size increased by 70%; however, in some specific cases where WTS did not manage to provide full coverage, this approach succeeded to do so.

One of the Major Studies that have been conducted recently is a comparison between the Single Target approach and Whole test suite (WS) and Whole test Suite with Archive approach (WSA) by Rojas et al. [48]. This is a later improvement of the first study by Fraser and his team [22] In which the authors enhanced the search to focus on the uncovered branch only and uses an archive data structure to store test cases that already covered branched that were uncovered. The authors conducted a detail empirical study on 100 java classes using three coverage criteria, (1)Branch Coverage, (2) Weak Mutation Coverage and (3) Line Coverage. The study has checked each of the approaches by using one of the mentioned coverage goals at a time and the authors reported the following,

- The coverage goals that were covered by Single Target GA and were not covered by WS or WSA turned out to be a special case and cannot be generalized.
- In Some For some coverage goals, the WS and WSA led to worse. Yet, those cases are minimal in number when being compared to the better results that were achieved by WS and WSA
- Introducing the archiving mechanism has improved the traditional WS and the authors obtained better results (especially for larger

classes) although they reported some negative side effects because of it since it needs special search operators, especially when being used for smaller classes.

- WS and WSA achieved superior results over the Single Target approach when being used to generate unit tests for complex classes.

Recently, Grano et al., has extended DynaMosa[42] to generate unit tests for Object-oriented classes that require minimum resources to be executed. The main challenges that were addressed by this study are how to generate unit tests that are performance aware without affecting the performance of generation procedure and how to avoid touching the effectiveness of tests in terms of coverage and fault detection. To answer those questions, the authors have introduced a set of 7 performance proxies (such as the number of loops, method calls and object instantiating) that provided insights of test running performance. First, they tried to incorporate the performance score as a secondary Objective in the preference criterion where the first front is selected and in the second trial they add the performance score to the routine where the archive is updated. In both cases, they found out that performance criterion competes with coverage. Thus, the authors devised pDynaMOSA where they decide whether to use the performance proxies depending on the search improvements done during generating unit tests. This adaptive method was introduced because the authors noticed that unit tests with low-performance indicators lead to low coverage goals. pDynaMOSA algorithm used both

the performance proxies and crowding distance as secondary heuristics; when crowding-distance is used then more diverse test are favored but when the proxies is selected then test with lower predicted resources demands are chosen. The toggling between two modes is determined by two things (1) if stagnation is detected on current objective scores (2) which heuristics is being selected at the current iteration. The results of the experiment revealed the following, (1) pDynaMOSA and DynaMoSA is performing similarly for the seven coverage criteria and pDynaMOSA outperforms the Random Search, (2) pDynaMOSA achieved a mutation score similar to DynaMOSA and both performed better than random search and (3) When pDynaMOSA and DynaMOSA achieved the same target coverage, the test suites constructed by pDynaMOSA are lower in execution time and heap memory consumption.

3.3 Adjusting Evolutionary Algorithms

Indicator based Evolutionary Algorithms (IBEA) [66] is the well known indicator-based evolutionary multi-objective algorithm which was proposed by Zitzler and Kunzli in 2004. IBEA allows the incorporation of any performance indicator into the selection mechanism of an MOEA. IBEA was originally tested with the hyper-volume [65] and the binary indicator [66]. The main contribution of IBEA is that it provides a general framework for indicator-based MOEAs, and as described by the authors

of this algorithms, “IBEA is based on quality indicators where a function I assigns each Pareto set approximation a real value reflecting its quality: Then the optimization goal becomes the identification of a Pareto set approximation that minimizes (or maximizes) I ”. Moreover, “ I induce a total order of the set of approximation sets in the objective space, in contrast to the classical aggregation functions like weighted sum that operate on single solutions only and gives rise to a total order of the corresponding objective vectors. The main drawback of the original IBEA is the computational operations needed to select the non dominated solutions which can be summarized as (1) Computing the quality indicator of the solution set and (2) updating the solution set and removes the worst by performing a pairwise comparison. Results of the empirical studies showed that IBEA was superior over two famous MOEAs, NSGA-II and SPEA2, on several bis or three objective benchmark MOPs.

As being stated before, hyper-volume indicator based algorithms are considered effective in multi-objective evolutionary algorithms (MOEAs), yet they suffer from a computational bottleneck and high time complexity when measuring the Hyper volume contribution of different solutions. Jiang et al. [34] proposed a simple and fast hypervolume indicator-based MOEA (FV-MOEA) which quickly updates the HV contributions of each different evolved solution. The main contribution of the proposed FV-MOEA can be summarized as the following,

- , they proposed a simple method for measuring the HV contribution of each solution set by deleting the irrelevant solutions and

transferring HV contributions.

- proposed a method to select offspring in HV indicator based algorithms which can save the computational cost of recalculating the HV indicators when population changes.

The core idea of FV-MOEA is that the HV contribution of a solution is only associated with partial solutions rather than the whole solution set. The empirical experiment demonstrated that the proposed algorithm is in-term of and HV calculations over classical MOEAs such as NSGAI, SPEA2, MOEA/D, IBEA. The experiment was conducted on 44 benchmark MOPs with 2–5 objectives.

Li et al. [35] have proposed a new modification to IBEA named mIBEA, which adds a Pareto-based element to this indicator-based method, analyzing the distribution of non dominated solutions found. The Authors experimented with the behavior of IBEA and found that the search is focused in certain regions in the search space while solving MOPs. Moreover, they found that IBEA suffers from a lack of diversity in generated solutions. The proposed method excludes dominated solutions at each generation after the new population is created by using dominance-based sorting method of NSGA-II, and that because (according to authors) the scaling of the objectives' scores which take place in original IBEA is no longer affected by dominated solutions which resides far away from the best non-dominated ones in the population. So the Basic IBEA algorithm

was empowered in this study by the ranking from Pareto dominance-based multi-objective evolutionary algorithm during the elite solution protection process; specifically, they used fast non-dominated sorting of NSGA-II to get non-dominated solutions in population and used the non-dominated solutions as the new population. The empirical results of the experiment proved that mIBEA is superior IBEA when being examined over the entire DTLZ benchmark functions. Besides, mIBEA results in significant improvement in running time over IBEA.

The above two studies are the major ones in the literature that shed light on IBEA main limitations and drawbacks but yet proved that IBEA could be modified to overcome those and performs much better in terms of running time cost and producing more diverse solutions. In our study I will examine the basic IBEA in the domain of test case generation and then try to improve it by introducing some tempering from the literature

3.4 Thesis Distinction from Other Studies

This thesis differs from the above-related work in some aspects, which presents its contribution to the field of test case generation. The differences can be summarized in the following points:

- Most of the studies tried to improve an already used algorithm by modifying their nature, introducing new testing utilities, or making use of additional data structures like an archive. However, this

study will examine a new algorithm in the domain of test case generation.

- To The best of our knowledge, this study is the first one that tries to investigate IBEA in the field of test case Generation. Most of the researchers avoided analyzing IBEA because of it has a lot of computation by nature.
- This Study will propose some modifications to basic IBEA as a trial of improving its performance in the domain of test case generation
- Currently, *EVOSUITE* framework contains, for example, the implementation of several MOEAs such as SPEA2 and NSGA-II. This Study will be the first to integrate both Basic IBEA and a modified version in *EVOSUITE* tool.

3.5 Literature Review Summary

Table 3.1 summarizes related work done on test case generation using evolutionary algorithms. None of the studies ever used the Indicator-Based Evolutionary Algorithm (IBEA) due to the huge calculation that took place while finding out the best individual. Our Thesis aims to break through this area and find out the performance of IBEA and how it can be adjusted to be useful in unit test generation.

This section should be filled after having a discussion with instructor

Study	Approach	EA Used	Paradigm	# Technology
Search-based testing of procedural programs: iterative single-target or multi-target approach [53]	Iterative single Objective	GA + Greedy for Suite Minimization	Procedural	C
Evolutionary Testing of Classes [38]	single Objective	GA + Greedy for Suite Minimization	Object Oriented	Java
Hybridizing Evolutionary Testing with the Chaining Approach [40]	Single Objective	Hybrid of ET For Global Search + Chaining Technique for local search	Procedural	N/A
Using Evolutionary Algorithms for the Unit Testing of Object-Oriented Software [60]	Single Objective	Universal Evolutionary Algorithm (GA + Improved Chaining technique for local search)	Object Oriented	Java
Evolutionary test environment for automatic structural testing [61]	Single Objective	GA	Procedural	C
Whole Test Suite Generation [22]	Multiple - Whole test Suite	GA	Object Oriented	Java
<i>EVOSETTE</i> : Automatic Test Suite Generation for Object-Oriented Software [20]	Multiple - Whole test Suite	GA	Object Oriented	Java
Reformulating Branch Coverage as a Many-Objective Optimization Problem [44]	Multiple	GA + NSGAII for sorting	Object Oriented	Java
Many Independent Objective (MIO) [4]	Multiple	GA	Object Oriented	Java
Test Suite Generation with Memetic Algorithms [23]	Multiple - Whole test Suite	GA for global search + Hill Climbing for local search	Object Oriented	Java
Random or Genetic Algorithm Search for Object-Oriented Test Suite Generators [55]	Multiple - Whole test Suite	GA	Object Oriented	Java
Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets [42]	Multiple	GA	Object Oriented	Java
Combining Multiple Coverage Criteria in Search-Based Unit Test Generation [49]	Multiple	GA	Object Oriented	Java
A detailed investigation of the effectiveness of whole test suite generation [49]	Single target, Whole Test Suite and Whole Test Suite With Archive	GA	Object Oriented	Java
Testing with Fewer Resources: An Adaptive Approach to Performance-Aware Test Case Generation [25]	Multiple	GA	Object Oriented	Java

TABLE 3.1: Literature Review Summary

Chapter 4

Research Methodology And Experiment Setup

The main contribution of this study is to evaluate the performance of IBEA by comparing it to NSGA-II, report and analyze the results of this algorithm's behavior in the domain of unit tests generation. Another main objective I am willing to achieve is to be the first to integrate IBEA with *EVOSUITE*, a Java framework for generating unit-tests. The tool has several MOEAs implementation and has been used by most of the studies being reviewed in this thesis.

To achieve the intended goals, I need to devise an experiment to obtain the results. A preliminary step for the experiment, is to learn about *EVOSUITE*, implement the algorithm and adjust it to be used within the tool. The genetic Operations such as mutation and crossover, problem representation and chromosomes construction are all inherent to the *EVOSUITE* tool approach and will be detailed in this chapter.

Moreover, in this chapter I will explain the adopted methodology

used to conduct this study, the testing objectives and its relation to the fitness functions used to evaluate the generated solution and guide the algorithm to the best heuristic evaluation. In addition, I will detail the needed information about the data sources.

4.1 Experiment Data Sources

To analyze the behavior of IBEA as an MOEA in the domain of unit test generation, this study will count on conducting the experiment on a group of classes *SF110* that were previously presented by Fraser and Arcuri [17]. The *SF110* was extended by Fraser and his fellows who added ten projects to *SF100*, those added projects were considered the most popular projects. It is worth to note that *SF100* is corpus of classes is a statistically representative sample of 100 Java projects from Sourceforge, which is a popular open source repository (more than 300,000 projects with more than two million registered users).

Demonstrating a scientific approach might be necessarily as demonstrating a practical one, and a technique that found to be working on small or artificial problems might not be able to behave the same on industrial bigger problems which might impose a significant complexity. Having said that, and in order to reduce the external threat of validity associated with our study, I chose a group of classes from *SF110* which was introduced to address the above mentioned problems.

In this empirical study I am planning to use 46 classes that were also used by Panichella et al. [44] to examine his empirical study. Most of the classes chosen by Panichella were also used to evaluate the whole test suite approach [22]. As running the experiment of all classes of SF110 is a time and resource consuming task, I tried to adopt the choice of previous significant studies that tried to select a subset of classes under test to achieve diversity in terms of complexity and functionality. Panichella et al. [44] chose 64 classes from SF110 with only limitations that at least each class should contain 50 branches. As can be seen from Table 4.1, the total number of branches ranges from 50 to 1213 and on average around 215 branches.

4.2 Experiment Setup

In this section I will detail how the solution is represented, the genetic operations of *EVOSUITE* and the experiment metrics.

4.2.1 Chromosome Structure

The first step of applying the Evolutionary Algorithms in any domain is to define how a solution could be presented. The structure of the individuals' chromosomes could be assumed to be quite simple when applying evolutionary testing as it usually consist of a sequence of input values to be provided to the program during test execution. This assumption is valid in procedural programs and relatively in Object Oriented (OO)

No.	Project	Class	Branches
1	Guava	Utf8	63
2	Guava	CacheBuilderSpec	139
3	Guava	BigIntegerMath	133
4	Guava	Monitor	191
5	Tullibee	EReader	306
6	Tullibee	EWrapperMsgGenerator	67
7	Trove	TDoubleShortMapDecorator	59
8	Trove	TShortByteMapDecorator	59
9	Trove	TCharHash	60
10	Trove	TFloatCharHash	87
11	Trove	TFloatDoubleHash	87
12	JSci	LinearMath	262
13	JSci	SpecialMath	196
14	JSci	SimpleCharStream	82
15	CommonsCli	HelpFormatter	142
16	CommonsCli	Option	96
17	CommonsPrimitives	RandomAccessByteList	81
18	CommonsCollections	TreeList	215
19	CommonsCollections	SequencesComparator	89
20	CommonsLang	ArrayUtils	1119
21	CommonsLang	BooleanUtils	271
22	CommonsLang	Conversion	766
23	CommonsLang	NumberUtils	383
24	CommonsLang	StrBuilder	567
25	CommonsLang	DateUtils	314
26	CommonsMath	FunctionUtils	64
27	CommonsMath	TricubicSplineInterpolatingFunction	80
28	CommonsMath	DfpDec	138
29	CommonsMath	MultivariateNormalMixtureExpectationMaximization	66
30	CommonsMath	MatrixUtils	143
31	CommonsMath	SchurTransformer	92
32	CommonsMath	AbstractSimplex	59
33	CommonsMath	BrentOptimizer	65
34	JDom	AttributeList	133
35	JDom	SAXOutputter	89
36	JDom	XMLOutputter	62
37	JDom	Verifier	277
38	JodaTime	BasePeriod	79
39	JodaTime	BasicMonthOfYearDateTimeField	63
40	JodaTime	LimitChronology	112
41	JodaTime	PeriodFormatterBuilder	579
42	Tartarus	englishStemmer	290
43	Tartarus	italianStemmer	228
44	Tartarus	turkishStemmer	514
45	XMLEnc	XMLChecker	1213
46	XMLEnc	XMLEncoder	138

TABLE 4.1: Classes to be used in the Empirical Experiment

```
// an OOP class
class Foo{
    boolean fooMethod (String s, Bar b){
        if(s.equals(b.getName())){
            //target
        }
    }
}

//Test case
class FooTestCase{
    Foo foo = new Foo();
    String s = "foo";
    Bar bar = new Bar("bar");
    foo.fooMethod( s, bar);
    //do assertions
}
```

FIGURE 4.1: Example class and test case

based programs. In fact, in OO programs the representation is more complex because it might imply passing objects (and Objects with a certain state) to a CUT and not just a primitive inputs. The following code snippet 4.1 shows an example of an OO program and its test case.

In order to understand the representation we should digest the difference between a test case and test suite. Thus, a test case in OOP is a sequence of constructor and method invocations, including passing objects and parameter values to some method during the test execution. A test suite is a group or set of test cases. We also should keep in mind that in whole test suite (WTS) generation, the optimization target is not to produce a test that reaches one particular coverage goal, but it is to produce a complete test suite that maximizes coverage, while minimizing the size

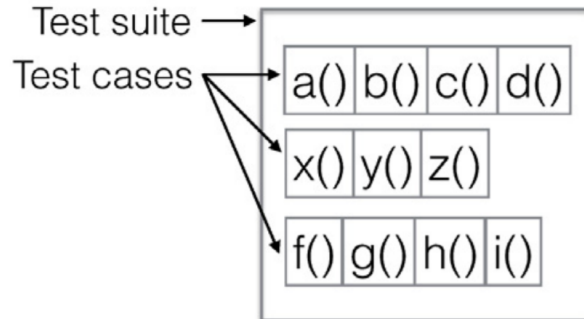


FIGURE 4.2: Chromosome Structure in WTS

at the same time [23]. Thus, Figure 4.2 illustrates and better describes how the chromosome will be structured in Whole test suite generation.

4.2.2 Fitness Function

Meta-heuristic algorithm should be guided during solutions generation; to evaluate the potential found solutions for an optimization problem, we should have a way to evaluate those solutions and determine the level of goodness to the subject problem, the means of such an evaluation is always done via a fitness function. The choice of fitness function is something crucial as it affects the behaviour of the algorithms in the search space [50]. In this study I need to define a fitness function that can measure how good a test suite is with respect to the search optimisation objective; this usually defined as test coverage criterion.

In any type of testing, there should be some pre-determined indicators to measure the efficacy of tests results such as code coverage (statement and branch), weak and string mutation, output coverage, resource

consumption and execution time. Studies in search based software testing have adopted a set of metric which reports the degree of which the source code of the program has been executed when the test run and coverage criteria us the most common used to guide test generation.

In this study I planned to use a standard metric used in search-based testing, the branch coverage as our evaluation criterion. In order to explain the branch coverage we need to understand the branch distance and approach level.

4.2.2.1 Branch distance

Branch distance estimates how close that branch was to evaluating to true or to false. Let us consider the following if condition in figure

```
if (a == 12 ){  
    //do something  
}
```

FIGURE 4.3: Code Snippet to explain Branch distance

If we have a concrete test case where a has the value 12, then the branch distance to make this branch true would be $12 - 8 = 4$ while the branch distance to making this branch false is any number other than 12.

Let $d(b, T)$ be the branch distance of branch b on test suite T can defined according to the following equation,

$$d(b, T) = \begin{cases} 0 & \text{if the branch has been covered,} \\ v(d_{\min}(b, T)) & \text{if the predicate has been} \\ & \text{executed at least twice,} \\ 1 & \text{otherwise.} \end{cases} \quad (4.1)$$

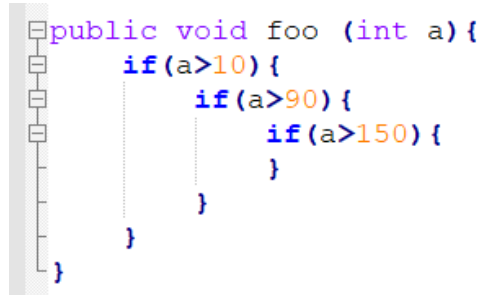
where v is any normalizing function in the $[0,1]$ range; for example $\nu(x) = x/(x + 1)$ [3].

4.2.2.2 Approach Level

In addition to the branch distance and to give more gradient to the search algorithm instead of just reporting a yes/no on whether the branch was covered, I also employ the approach level which is usually used in the SBST by researchers [39]. The approach level $A(t, x)$ for a given test t on a target $x \in X$ is defined as how far is a test case from covering a certain target in terms of control dependent edges in the control dependency graph. So in figure 4.4, if we have a test case t_{15} having the value $a = 15$, then the approach level for the target $x_{a>90}$ is **0** and $x_{a>150}$ is **1**.

4.2.2.3 Fitness Function: Branch Coverage

Our fitness function could be defined as the sum of approach level and branch distance between a test t and a branch coverage goal x . The main idea is to give more gradient to the search (instead of just counting "true" or "false" on whether a goal is covered or not). Having said that, our



```
public void foo (int a){
    if(a>10){
        if(a>90){
            if(a>150){
            }
        }
    }
}
```

FIGURE 4.4: A code Snippet to explain the Branch distance and Approach level

main objective is to minimize this distance. Our fitness function could be represented according to the following equation [3],

$$f(t, x) = A(t, x) + v(d(t, xc)) \quad (4.2)$$

The main objective of the applied search algorithm is to minimize the fitness function value to achieve a valuable test case and cover all targets. To illustrate the fitness value in an example, let us refer again to figure 4.4 and suppose that we have a test case t_{40} having the value $a = 40$, then the approach level for the target $x_{a>150}$ is 1, hence the fitness for this test case is $1 + v(|40 - 90| + 1) = 1 + v(51)$. This test case t_{40} will be for example worse (having higher fitness value) than a test case t_{110} having the value $a = 110$ since the approach level for $x_{a>150}$ is 0 and the fitness value is $0 + v(|110 - 150| + 1) = v(41)$.

4.2.3 Evaluation Metrics

After conducting the experiment and in order to compare the algorithm, this study will use the branch coverage as a metric to indicate how effective was each algorithm, and the search budget consumption to show how efficient was every algorithm. Branch coverage of an algorithm can be computed as the total number of covered branch divided by the total number found in the class. Efficiency will be considered as the time needed to reach the full coverage. If no full coverage was reached, then it will be the number of executed statement. Another metric to be used is the test suite size, its well know from software engineering practitioners that if two test suites have the same coverage, the smallest is desirable. Finally, I will calculate the execution time of the test cases, to give ability for practitioners to choose which test cases to run according to their schedule and resources.

4.3 Algorithms and EvoSuite

This study aims to implement a prototype using *EVOSUITE* tool [21], neither IBEA nor any derived flavor has been implemented before. This Study will be the first one to add the algorithm to be used as part of the tool. The following table 4.2 shows the default setting of the algorithms used in this study.

The experiment associated with this study will conduct a comparison of three algorithms: 1) NSGA-II 2) IBEA 3) mIBEA. The same parameter

Parameter	Value
Population size	50
Crossover rate	0.75
Crossover type	two Point
Mutation rate	0.75
mutation type	test suite manipulation
Independent runs	15
Search budget (statements executed)	1,000,000
Timeout (seconds)	600

TABLE 4.2: EvoSuite PARAMETER SETTINGS

values are used for the three algorithms unless the native nature of the algorithms requires a different configuration. All the algorithm run will target the same objectives on the same environment. Eventually the comparison of this thesis is used to measure the effectiveness and efficiency of IBEA and the possibility of introducing improvements while preserving the main objectives.

4.3.1 Genetic Search Operators of EVOSUITE

This study will prototype will inherit the way *EVOSUITE* tool is representing the problem and all the genetic operations that can be preformed over those chromosomes during executing Genetic Algorithms. Selection operator of the algorithm will be the tournament selection with a tournament size of 1. However, the search operators employed by the tool will different and in the new two section will illustrate how are they implemented in *EVOSUITE*. Moreover, I will explain how randomness is used to generate the initial population.

4.3.1.1 Chromosome Crossover

Crossover between two parents starts by choosing a random value α between $[0,1]$, and the operation occurs on two parents $P1$ and $P2$ to generate two off-springs $O1$ and $O2$. The first off-spring $O1$ will contain $\alpha|P1|$ test cases from the first parent and the last $(1 - \alpha)|P2|$ test cases from $P2$. However, the second off-spring $O2$ will contain $\alpha|P2|$ test cases from the first parent and the last $(1 - \alpha)|P1|$ test cases from $P1$. The following figure shows an example of crossover on two test suites 4.5 [22]

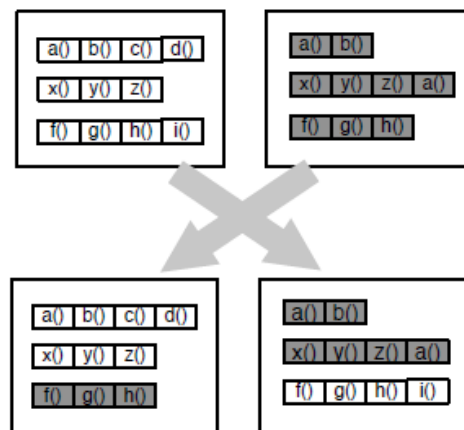


FIGURE 4.5: Test Suite Crossover

Since the test cases are independent between them, then all test suites that the crossover operator will generate will be Valid. Add to this, no offspring test suites will have a size bigger than the largest parent. However, its possible that the total sum of statements in test cases of an offspring test suite would increase in [22]. *EVOSUITE* implements this using a

relative single point crossover, meaning, The splitting point is not an absolute value but a relative value (eg, at position 70% of test – suite). For example, if $size(testSuite1) = 10$ and $size(testSuite2) = 20$ and splitting point is 70%, we would have position 7 in the first and 14 in the second test suites respectively.

4.3.1.2 Chromosome Mutation

Mutation is much complicated comparing to crossover operator because it could occur on the test suite level and on the test cases level as well. Starting with test suite, when a test suite T is mutated then each of its test cases are mutated with a probability of $1/size(T)$. Then a number of new random test cases are generated and added to the suite with probability σ . then the second test suite is added with probability σ^2 and so forth until the i th test case is not added. Its worth noting here that the test cases are added until the size of test suite T does not exceed the limit N [22].

Test cases mutation includes three operation will take place with a probability of $1/3$ (1) remove, (2) change and (3) insert. Thus, on average only one operations will be applied and all of them could be applied with a probability of $(1/3)^2$, the three operations details will be discussed next,

1. **Remove:** For a test case $t = \langle s_1, s_2, \dots, s_l \rangle$ with length l , each statement s_i is deleted with probability $1/l$. [22]. if any successive statements s_j where $1 < j \leq l$ has inputs that depends on the deleted

statement, the dependant statement will be replaced with a new input if possible otherwise it will be deleted.

2. **Change:** For a test case $t = \langle s_1, s_2, \dots, s_l \rangle$ with length l , each statement s_i is changed with probability $1/l$. If a certain statement that will be changed is a primitive value or an array, then those will be changed without invalidating the test-case. but if the input statement was a string, then the string will be mutated similar to the way a test case is mutated. if the statement was an assignment then either the left or the right side value of the assignment will be changed. If the statement is not primitive value and was method, field or a constructor call then those will be randomly changed but the return and input types will be preserved
3. **Insert:** a new statement is inserted With probability σ' , at random place in the test case. If it is added, then a second statement will be added with probability σ'^2 , and so on until the the statement is not inserted. this operation will continue until he limit L is reached,

Figure 4.6 shows an example of the operations on the three test suites. After performing the mutation operations on a test case and the changed test cases appeared to have 0 statements then it will be removed from the test suite. To evaluates a mutated test suite, then we should execute all its underlying statements, however and since on average on test case of a suite would be changed, then there is no need to execute the whole test

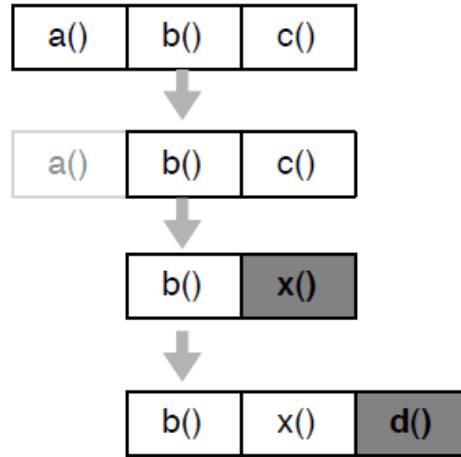


FIGURE 4.6: Test Case Mutation example, the three operations delete, insert and change are shown [22]

suite, only the changed test cases will be evaluated and the fitness of the test suit will be updated accordingly [22].

4.3.2 Random Test Case Generation

To Initialize and construct the first population, *EVOSUITE* randomly generates test cases that are part of a test suites. Those test suites formulates the first population. Test cases are independent and their probability of creation as well. Meaning, the probability of constructing a certain test case is constant and will not be affected by the test cases that are previously generated. Test cases are sampled randomly and thus any possible test case in the search space could be generated.

Since the representation of the problem is not trivial and found to be

index (please refer to sections 2.1.2 and 4.2.1), then its some how impossible to generate test suites and cases according to the uniform distribution because if the limit size of a test case was L , then the possibility of generating a short test cases will be very low.

EVOSUITE choose a random number according to the uniform distribution within a certain range $[l, L]$, and then uses the insertion operator described in section 4.3.1.2 to insert statements in the test case until the size equals to chosen random number.

4.4 Experiment Assumptions

The following assumptions are considered in this study, some of them inherent to the *EVOSUITE* tool.

1. *EVOSUITE* can only cover the CUT that has no environmental dependencies like external service or file systems. Moreover, the tool assumes that the CUT are deterministic, meaning that if a test case was executed ten times, it will yield the same test result [62]
2. The oracles of test cases are not evaluated by this study. The assertions are all being generated automatically by the tool [22].
3. EvoSuite is subject to ongoing enrichment and contribution by researchers who contribute to it via implementing different multi objective algorithms and adding them to EvoSuite. I will take a forked code-base assuming that the latest code of the *EVOSUITE* tool

that was published online and referred to by the tool's official website is valid.

4. Since the subject projects are used in SBST studies and the fact that a lot of studies aimed to evaluate test approaches by measuring the defect detection rate then this study assumes that subject under test are logically reasonable and could have defects which might affect the generation in case failing test cases were generated.
5. During experiment execution I noticed that the tool thrown some errors, I consulted the tools main contributors over email and they said that the reported errors should not affect the coverage goals used. So I assume that those errors are normal and has no effect the final evolved solutions.

Chapter 5

Experiment Results and Analysis

This chapter will illustrate the results of the experiments that I conducted. Besides, it will compare the performance of both IBEA and mIBEA in the scope of generating unit test-cases and how did each algorithm performs when being added to *EVOSUITE* tool. Finally, it will compare IBEA [66] and mIBEA [35] to an already implemented algorithm in *EVOSUITE* NSGA-II.

5.1 Algorithm Comparison

The main purpose of this experiment is to evaluate IBEA in the unit-testing generation problem, and check whether IBEA will generate test suites that cover a set of certain objectives despite its popularity of high computational costs in calculating the indicators [44]. This experiment is intended to achieve the following goals:

1. Implement and add two new algorithms namely IBEA and mIBEA to the well known tool *EVOSUITE*

2. Implement and add two new objectives functions to *EVOSUITE* to compute the test suite size and execution time.
3. Examine the behavior of IBEA and mIBEA and provide analysis on the performance using a base line algorithm NSGA-II which is already integrated within *EVOSUITE*

To achieve the above purpose, I will conduct an empirical evaluation of the feasibility of applying IBEA and mIBEA algorithms in optimizing whole test suites. As being explained in 4.1, I will experiment over well-known software source codes that are used in this research area. As *EVOSUITE* is designed to be extensible, then the main advantage of using this tool is that it saves us from investing efforts and time in performing byte-code instrumentation, as well as to handle static and dynamic analysis on the source code and develop the needed infrastructure to generate test-cases based on the previous activities.

5.1.1 Experiment Results

I have used each of the algorithms, namely NSGA-II, IBEA and mIBEA, to generate test suites for the classes presented in Table B.1. Each algorithm were used for 15 times (i.e. 15 independent runs) upon the subject CUTs. The metrics that were collected for the generated test suites are the following,

1. **Algorithm Evolution Time:** The full-time *EVOSUITE* spent generating the test cases

2. **Test Suite Size:** Number of tests in resulting test suite
3. **Test Suite Length:** Total number of statements in the final test suite
4. **Test Suite Mutation Score:** The obtained score for (strong) mutation testing
5. **Test Suite Branch Coverage:** percentage of covered branches of CUT
6. **Test Suite Execution Time:** The time needed by each test case to examine the CUT
7. **Test Suite Covered Goals:** Total number of covered goals

5.1.1.1 Experiment Results as Box Plots

The following table 5.1 summarizes the total experiment results which are of 2,070 record and shows the mean (average) values of the above metrics.

I am just summarizing the results of our experiment which was left to run for 25 days. Briefly, I can notice that NSGA-II and Modified IBEA outperformed IBEA and covered a huge number of branches. Modified IBEA and NSGA-II have a similar branch coverage with a very tiny advantage for the last. I am not going to depend on this table to analyze the results, since mean cannot reduce the affect of outliers data.

The difference between IBEA and the other two algorithms is not that huge, and if we take the test suite length and size into consideration

TABLE 5.1: Mean Values that summarizes the full Experiment Collected Results

	NSGA II	Modified IBEA	IBEA
Algorithm Evolution Time	554929.87	556548.56	576100.65
Test-suite Branch Coverage	0.8	0.8	0.68
Test-suite Execution Time	36.67	39.48	47.64
Test-suite Length	147.98	145.33	86.21
Test-suite Size	47.76	47.04	27.6
Test-suite Mutation Score	0.34	0.34	0.28
Test-suite Full Coverage	0.93	0.93	0.89

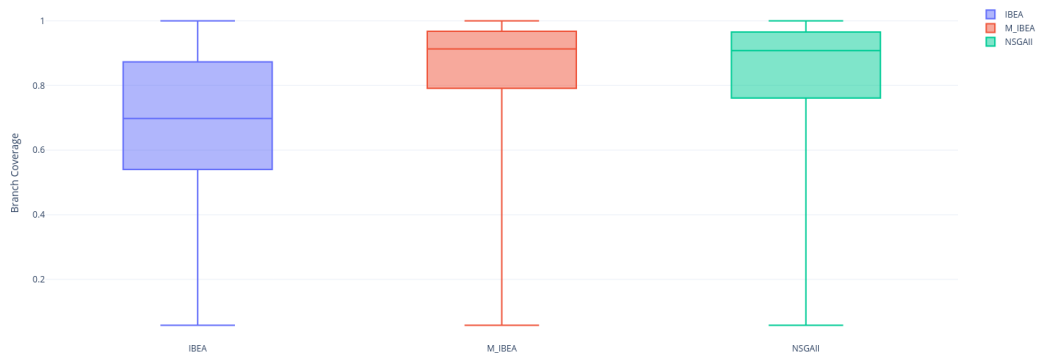


FIGURE 5.1: Generated Test Suite Branch Coverage

which are shown in figure 5.2 and 5.3 , a user has a preference to opt between two group of algorithms. A group that generates a suite with higher branch coverage and larger length, and another algorithm which generates smaller test suites with a minimum lose in the coverage percentage. Again, test suite size is something important to practitioner as the size affects non-functional aspects in testing like maintainability and readability.

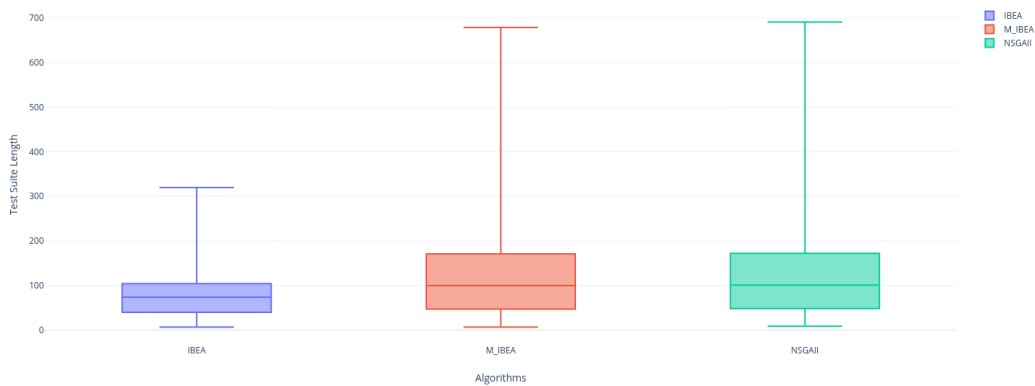


FIGURE 5.2: Generated Test Suite Length

The next box plot 5.4 shows that the three algorithms achieved a similar results. The reason behind this shared behavior is that the three algorithms consumed the full search budget trying to achieve the full set of the target branches. And Since I am using a subject CUTs that have a huge number of branches, it was almost impossible for the three algorithms to hit a full branch coverage.

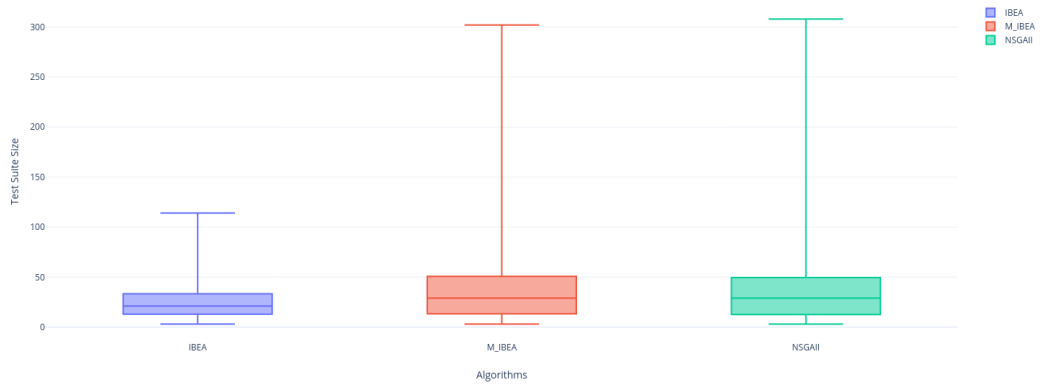


FIGURE 5.3: Generated Test Suite Size

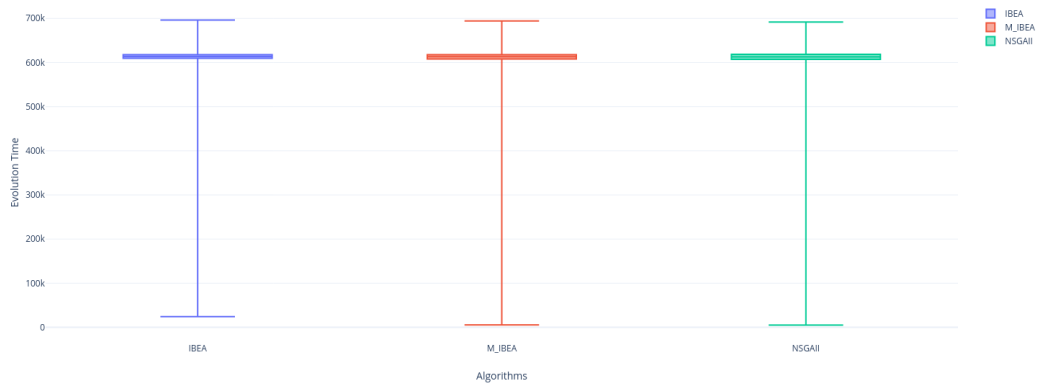


FIGURE 5.4: Algorithm Generation Time

5.1.1.2 Descriptive Statistics of Experiment Results

The following group of tables will show the statistics of the collected data per each algorithm. The first tables which compares the algorithm in terms of branch coverage to test-suite execution time shows that NSGA-II outperforms the other two algorithms in those two metrics.

TABLE 5.2: Add caption

Algorithm	Branch Coverage			Test Suite Execution Time		
	Mean	SD	Median	Mean	SD	Median
IBEA	0.675	0.245	0.696	47.642	497.59	8
M_IBEA	0.796	0.252	0.902	39.475	133.656	13
NSGA II	0.798	0.253	0.907	36.67	195.335	11

The second group of tables 5.3 shows the statistics of test-suite Mutation score and Evolution time of the algorithms. In case of mutation score, we can see that IBEA was surpassed by Modified IBEA and NSGA-II. However, for evolution time, I explained briefly that the number of branches to cover was generally high and thus all algorithms has consumed all search budget in its attempt to chive 100% coverage. I will conduct a detailed analysis on this metric in section 5.3.

Finally yet importantly, I am showing the metrics that are concerned with test-suite size and length in table 5.4, we can clearly notice that IBEA managed to evolve test-suites with minimal size and thus excelled over its competitors algorithms.

TABLE 5.3: Add caption

Algorithm	Mutation Score			Evolution Time		
	Mean	SD	Median	Mean	SD	Median
IBEA	0.283	0.193	0.258	576100.6	149560	613380
M_IBEA	0.344	0.202	0.326	556548.6	196559.7	613333
NSGA II	0.345	0.198	0.329	554929.9	198740.2	613132

TABLE 5.4: Add caption

Algorithm	Test Suite Size			Test Suite Length		
	Mean	SD	Median	Mean	SD	Median
IBEA	27.603	24.242	20	86.206	63.896	75
M_IBEA	47.036	61.893	26	145.328	159.777	97
NSGA II	47.755	63.16	27	147.98	164.181	98

5.2 Branch Coverage Analysis

This section mainly analyzes the results to answer our first research question in which I aim to compare the algorithms in terms of branch coverage. For detailed analysis, I decided to perform the analysis according to the number of branches found in CUTs. Having said that, here is the grouping that I adopted:

1. 22 - 70
2. 71-100
3. 101-200
4. 201-300

5. 301-1215

The first bar chart shows the branch coverage of each algorithm to the number of target branches (goals), its obvious that the three algorithms performs the same.

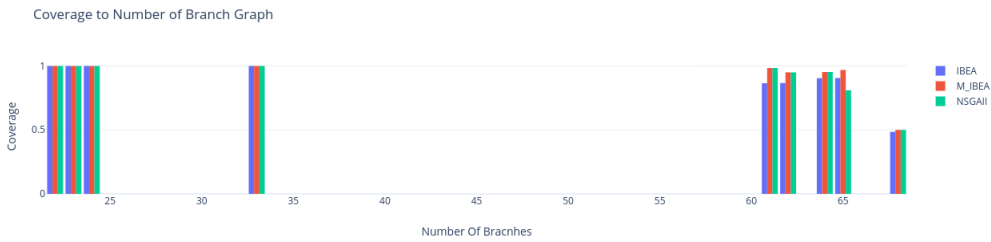


FIGURE 5.5: Branch Coverage when Number of Branches is between 22 and 70

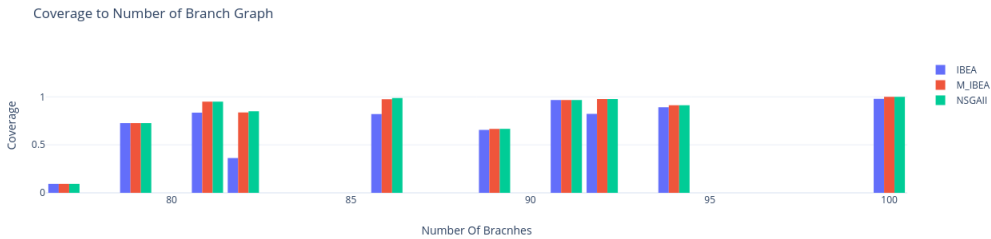


FIGURE 5.6: Branch Coverage when Number of Branches is between 71 and 100

When the number of branches starts to increase beyond 100 branch per CUT, we start to see that NSGA-II and Modified IBEA has a significant advantage over the IBEA. The following group figures illustrates this fact.

The final graph illustrates how the three algorithms perform when the number of branches increases above 300. Apparently, I got similar



FIGURE 5.7: Branch Coverage when Number of Branches is between 101 and 200

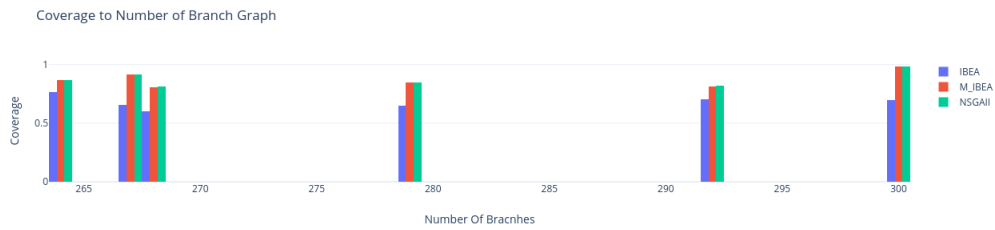


FIGURE 5.8: Branch Coverage when Number of Branches is between 201 and 300

close results to the previous ones, IBEA did not manage to compete with the other two algorithms and achieved a smaller branch coverage. One thing to notice in the next graph 5.9 is that all three algorithms achieved more or less the same branch coverage when the number of branches has exceeded the one thousand. This clearly tells us that no algorithm has any advantage over the other when the number of branches is huge.

One could expect that NSGA-II and Modified IBEA continue to achieve high branch coverage when number of branches to cover exceeds 1200 branch. But, I observed that all algorithms behaved similarly. I think the reason behind that is that all algorithms have consumed all the assigned budget without a managing to cover more branches each generation; and

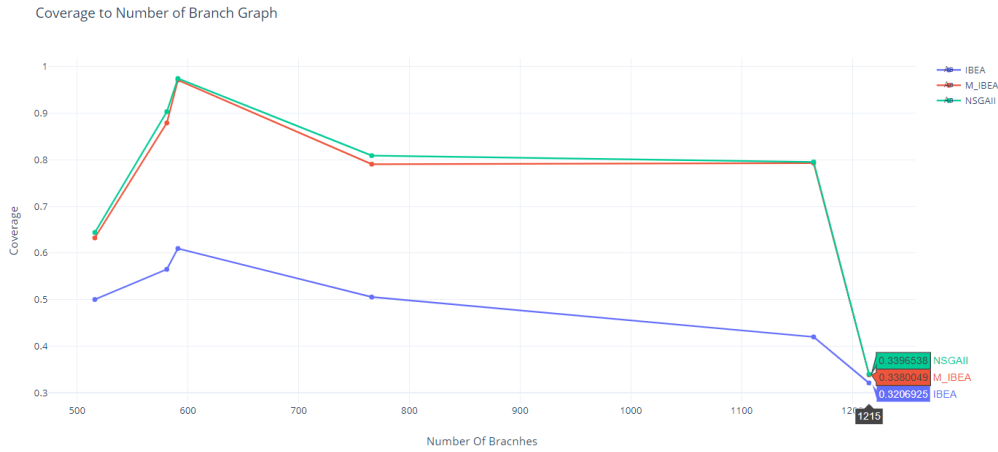


FIGURE 5.9: Branch Coverage when Number of Branches is between 301 and 1215

the coverage seen in this graph is the coverage of first randomly generated population. Future studies should take into consideration analyzing such cases, taking into consideration that they should increase the search budget and timeout assigned to the algorithms. I stated this in the following future work section 6.4.2

To conclude on this and to answer our first research question which is **RQ1: What is the branch coverage of IBEA and Modified IBEA when being compared to other Multi-Objective Algorithms (MOAs) in generating test cases?**, one can say that NSGA-II and Modified IBEA outperform the basic IBEA when targeting CUTs that have branches between 100 and 1000. However, all the three algorithms performed similarly when the number of branches to target was less than 100 and more than 1000. In addition, I can say that the branch coverage achieved by the algorithm is highly dependent on the target branches to cover, so the higher

the set of goals number the lower will be the goal coverage.

5.3 Generation Time Analysis

Execution time analysis is an essential verification activity during comparing the algorithms. To answer our second Research question, which tries to compare the algorithms in terms of Efficiency, I am going to analyze a subset of the data in which all three algorithms managed to achieve full coverage; meaning, all three algorithms covered all the target goals that were assigned to them before starting evolution. The following graph 5.10 illustrates how the evolution time changes as the number of branches increase when all algorithms achieved a 100% coverage, this is another fact that the number of goals really affect the behavior of the algorithms.

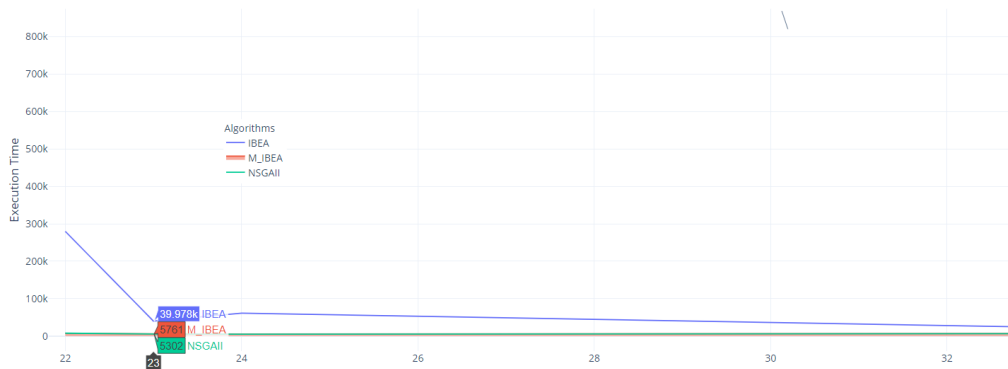


FIGURE 5.10: Evolution Time to Branch Numbers on full Coverage

Generally, in search based software engineering, the execution time is a factor to compare and analyze how efficient are the algorithms to achieve a certain objective. Moreover, its worth to keep in mind that

execution time is a crucial factor for practitioner who will be using the tool to generate test-cases. Its a must for such a tool to have a reasonable generation time.

TABLE 5.5: Data Summary for Full Goals Coverage

	NSGA II	Modified IBEA	IBEA
Algorithm Evolution Time	6789.92	7475.8	117010.12
Test-suite Branch Coverage	1	1	0.99
Test-suite Execution Time	10.68	15.58	14.78
Test-suite Length	33.37	33.17	32.83
Test-suite Size	8.37	8.37	8.3
Test-suite Mutation Score	0.51	0.51	0.52
Test-suite Full Coverage	1	1	1

As shown in the previous table 5.5, NSGA II achieved the smaller execution time, and hence this answers our second research question **RQ2: How is the execution time of IBEA when being compared to other MOAs in generating test cases?**. Modified IBEA has achieved a very close result (less than 1 second difference) and IBEA needed around 2 minutes to achieve the full coverage. On the other hand, IBEA achieved a higher mutation score and generated the smallest test suites in term of length. This is an expected result since IBEA consumes time in doing the hyper-volume calculations between all possible solutions. One more significant thing to shed light on is the enormous reduction in execution time that was achieved by the Modified IBEA.

It's also worth noting here the performance of modified IBEA which has way less execution time, and yet the algorithm still achieves similar Coverage and Mutation score results when being compared to the base IBEA, and is very close to NSGA-II which is used as a reference in comparing the results.

Another important metric to look at is the test suite execution time, I can notice that NSGA-II generated a test suites that are cost effective in terms of allocating resources and execution time. This is the case in all Multi Objective Evolutionary Algorithms which generates a set of non-dominated solutions and give the user a variety of solutions to choose from. So one could favour IBEA in case he wants to get the smallest test suites possible and go for Modified IBEA or NSGA-II if he wanted to expedite generation. Finally, if the user aims for the fastest test suites in terms of execution he would choose NSGA-II; NSGA-II then can be used in generating regression test-cases where the execution resources and budget are limited.

5.4 Testing Metrics Analysis

This section will analyse the results from Quality Assurance developers point of view. Thus I will compare the algorithms using testing metrics like size, length, execution time and mutation score. This section also answer our third questions which is **RQ3: What is the size and execution time of the test suite generated by IBEA when being compared to other**

MOAs in generating test cases?. Execution time and Size of test suite do really concern the QA developers because the size gives indication on how difficult it is to maintain a test suite and add assertions for it, and the execution time gives practitioner the ability to schedule their resources and releases.

Test suite size, which is a test metric that I aimed to minimize. According to the next figure 5.11 one can notice that IBEA generates test cases with the minimal size and beats out other algorithms. Next in the lines are Modified IBEA and NSGA-II with the largest test suites size. There is a slight advantage for Modified IBEA over NSGA-II.

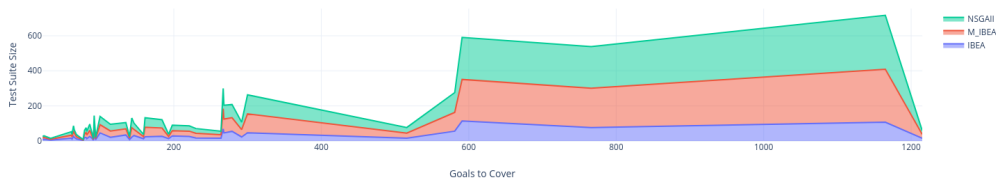


FIGURE 5.11: Test-suite Size to Target Goals

Test suite Length which is related to how many code lines are there in a test suite. Similar to test-suite size, I aim to get this metric minimized during generation. The next figure 5.12 represent algorithms comparison with regard to test suite size. Again IBEA generates test cases with the minimal length and exceed out other algorithms. Modified IBEA and NSGA-II are more or less the same with tiny lead to the first.

Test suite Execution Time is another metric to be used in comparison. As seen in the next figure 5.13 results between algorithms differ as the

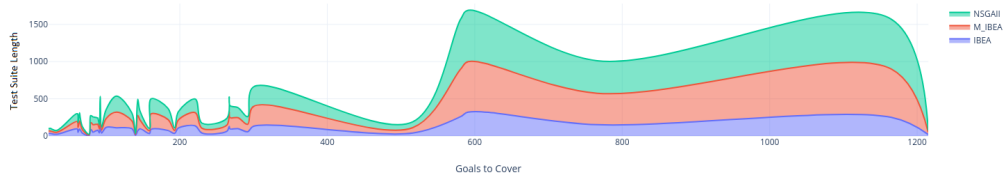


FIGURE 5.12: Test-suite Length to Target Goals

target goals changes, thus I am going to split the reading into two groups according to test goals number:

1. Test goals number lower than 200: I can see that NSGA-II and Modified IBEA are competing but its crystal clear that both beats out IBEA.
2. Test goals greater than 200: IBEA now takes the lead with significant minimization of the execution time, next in line are NSGA-II and finally Modified IBEA.

Test suite Mutation score: one more objective that I did not try to optimize but is used heavily in the literature to compare algorithms is mutation score. Mutation score is one more valuable metric for QA developers, usually developers insert artificial mutants in the code and check whether a test-suite can discover it. This how *EVOSUITE* also measure the mutation score but instead of having the mutants inside the source code they are added at the byte level using byte code instrumentation.

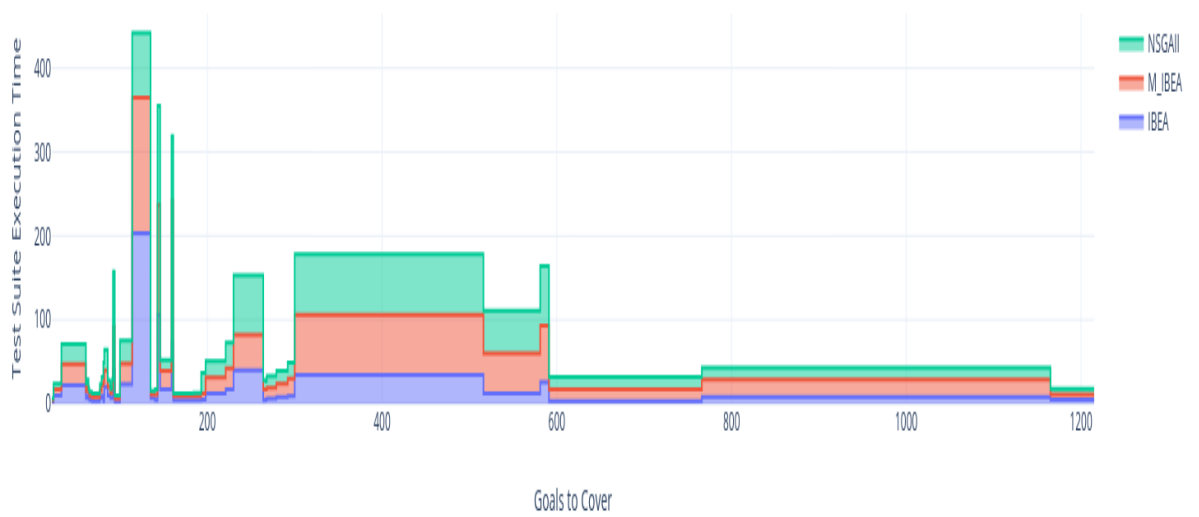


FIGURE 5.13: Test-suite Execution Time to Target Goals

The last figure 5.14 represent the mutation score of the three algorithms. Generally, NSGA-II and Modified IBEA performs the same and crush IBEA in generating test suites with high mutation score.

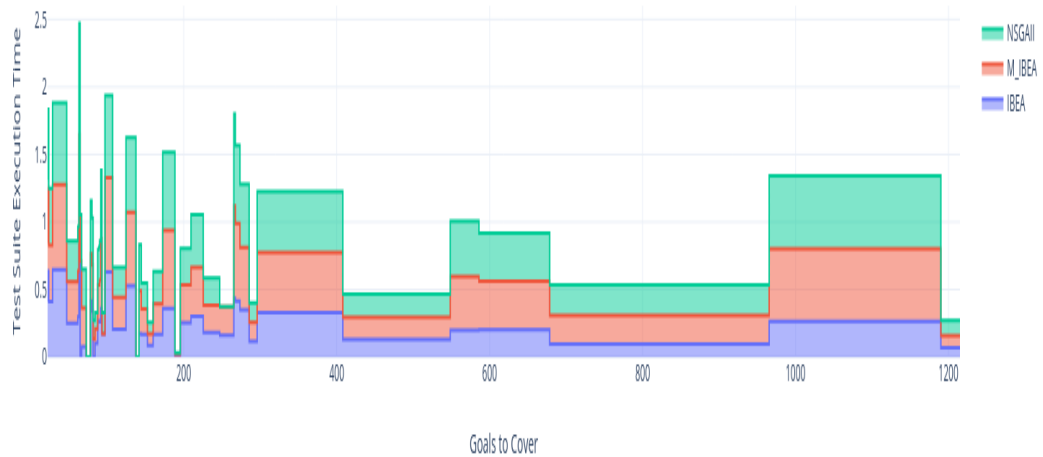


FIGURE 5.14: Test-suite Mutation Score to Target Goals

5.5 Statistical Analysis

In this section, I will be statistically analyzing our collected data and whether there is a significant difference or not. To accomplish that, I have conducted two types of tests. The first is Friedman test which I used to prove that there is indeed a differences between our metrics that were collected and observed across multiple experiment runs. The following

table summarizes the results of Friedman test conducted on our experiments observation.

TABLE 5.6: Friedman test of metrics collected

Metric	p-Value	statistic
Branch-Coverage	1.78E-14	63.32283465
Execution Time	9.53E-08	32.33333333
Total Coverage	1.63E-14	63.5
Mutation Score	1.01E-06	27.61490683
Generation_time	0.004656155	10.73913043
Size	3.66E-11	48.06060606
Length	7.48E-11	46.63218391

As shown in the table 5.6, the p – Value for each of the observed metrics is less than **0.05** and thus we can reject the Idea that all the differences between the collected data are due to stochastic nature of the genetic algorithms. Next, I will show the results of the post Friedman analysis to figure out which algorithms (groups) differ from which algorithms. To accomplish the pairwise comparison between the algorithms, I have used the Wilcoxon signed rank test; this test is used to compare repeated measurements on the same sample and verify that the measured means have different ranks.

The following table shows the pairwise comparison of the Evolution time metric, the table shows Modified IBEA and NSGA-II. Moreover, it shows that there were no significant differences between any other algorithm pairs.

Next, I will show the Wilcoxon test results for the metrics in which I used in calculating the objective (fitness) functions during evolving test

TABLE 5.7: Evolution Time Pair Wise Comparison

Evolution Time			
Algorithm-A	Algorithm-B	p	Significance
M_IBEA	IBEA	1	ns
M_IBEA	NSGA II	0.000804	***
IBEA	NSGA II	0.48	ns

suites.

TABLE 5.8: Evolved Test-suite Branch Coverage

Test-suite Branch Coverage			
Algorithm-A	Algorithm-B	p	Significance
M_IBEA	IBEA	0.000000525	****
M_IBEA	NSGA II	0.007	**
IBEA	NSGA II	0.000000777	****

TABLE 5.9: Evolved Test-suite Length

Test-Suite Length			
Algorithm-A	Algorithm-B	p	Significance
M_IBEA	IBEA	4.68E-08	****
M_IBEA	NSGA II	0.087	ns
IBEA	NSGA II	0.000000108	****

As stated in the Branch coverage table 5.8, there is a significance differences between all Algorithms. Tables 5.9 and 5.7 show that all algorithms differs, but there is no significant statistic difference between Modified IBEA and NSGA-II when measuring test suite length and execution time.

The next two tables 5.11 and 5.12 show the pairwise comparison of mutation score and test suite size, which reflects the number of test cases inside a test suite. In case of Test-suite size there is a difference between

TABLE 5.10: Evolved Test-suite Execution Time

Test Suite Execution Time			
Algorithm-A	Algorithm-B	p	Significance
M_IBEA	IBEA	0.0000192	****
M_IBEA	NSGA II	0.816	ns
IBEA	NSGA II	0.000417	***

all algorithms, however, in case of mutation score I can see that there is a difference between Modified IBEA and IBEA, IBEA and NSGA-II, but there is none between Modified IBEA and NSGA-II.

TABLE 5.11: Evolved Test-suite Size

Test-Suite Size			
Algorithm-A	Algorithm-B	p	Significance
M_IBEA	IBEA	0.00000107	****
M_IBEA	NSGA II	0.043	*
IBEA	NSGA II	0.000000855	****

TABLE 5.12: Evolved Test-suite Mutation Score

Test-Suite Mutation Score			
Algorithm-A	Algorithm-B	p	Significance
M_IBEA	IBEA	0.0000315	****
M_IBEA	NSGA II	1	ns
IBEA	NSGA II	0.000107	***

Chapter 6

Conclusion And Future Work

This chapter will discuss the threats to validity that might exist in our experiment, in addition to the main conclusion, obstacles and complexities that I ran into and finally the future work.

6.1 Threats to Validity

This section analyzes the threats of validity which cares about the possibility of endangering the trustworthiness of the thesis or the existence of biasing factors that could affect the experiment and its results.

6.1.1 Threats to construct validity

I identified two major threats related to the experiment design,

- The first one is related to the choice of our metrics used in comparing the algorithms, which for example does not take into account the severity of faults nor the line coverage. However, our

choice was based on what concerns the software engineers when they are about to write unit tests, or generate them using a similar tools. Moreover, I took into consideration two non functional metrics which aim to reduce the execution time and length of a test suite; minimizing execution yield a utilization in resources and cost, and shrinking length of a test suite would reduce the effort needed by the practitioner to add assertions and maintain it. Since this experiment is repeatable, future studies might count on different metrics to capture the difference between algorithms.

- The second one is related to the nature of meta-heuristic algorithms which could generate different optimized solutions for the same input. This could cause the generation of outliers solutions. I mitigated this by running the algorithms on 46 different CUTs that were chosen from a well known data set. Each algorithm were put to run for 15 independent run. Its worth mentioning here that the data set is taken SF110 corpus of classes which is adopted by the tool designer and used a lot in the Search-based software testing.

6.1.2 Threats to Internal validity

The main two threats I identified in the tool used in our experiment are (1) the experiment effect on the results and (2) whether I have a bug in the code or not. To validate the implementation of the algorithms, I conducted a preliminary experiment over a set of nine classes provided by

the tool authors. The results of this experiment was reported in Appendix B. In addition, I tried to mitigate this by debugging the code several times and running a full mini experiment on 9 CUTs in which each algorithm was left to generate test cases for 20 independent runs. Besides, I analyzed the results, checked the code and verified via the tools logs that everything worked as expected. Finally and to reduce the effect of the platform used to run the experiment, the full experiment was handled on the same AWS Machine for the three algorithms.

6.1.3 Threats to conclusion validity

The choice of the statistical test might not be the best approach to examine the differences between the algorithms performances and affect our conclusion. To mitigate this and since I can't tell if the collected data is normally distributed, I used two types of statistical tests; the Friedman test which is a Non-parametric equivalent that is used to tell if there is a difference among the algorithms used. Moreover, I conducted a post-hoc tests to do a pair wise comparison using Wilcoxon signed rank test to figure out which pairs are different.

6.1.4 Threats to external validity

This type of validity is concerned about the generalization of the study and the representability of the experimental subjects. The first threat

could be related to the classes used and whether they represent a typical code; sizes of the classes used in this experiment are considered huge, the number of code lines and branches is huge, some CUTs have more than 1200+ branches. These days, all programming languages urges developers to have classes that do not exceed a 1500 line of code for the sake of maintainability and readability [57], and if they exceeded that number then they should be splitted into two separated classes. The Second threat is the fitness functions, which concern the branch coverage, length and execution time of a test suite. In future work, one could enhance this fitness function to include other code coverage aspects of a test case like statement and path coverage. Yet, from an industry point of view, it's more important to get a test case that covers a considerable part of the code, and has a reasonable execution time and length. Less time meaning less cost and smaller length could yield a better maintainability

6.2 Conclusion

The main objective of this thesis was to study the IBEA algorithm as a multi objective evolutionary algorithm in the problem of generating and optimizing unit tests. This study aimed to optimize three main objectives, test execution time, branch coverage and test size; three metrics that I consider as the most important for QA developers.

As IBEA is well known for its high cost HV calculations, it was not examined before in unit-tests optimization. To our best knowledge, our

study was the first to examine IBEA in such an optimization problem. In addition, I aimed also to use a modified version of IBEA in which I simplify the complicated computations overhead without affecting the optimization goals. To analyze the two IBEA algorithms, I decided to conduct a comparison with NSGA-II .

Our results revealed that IBEA has outperformed both NSGA-II and Modified IBEA in generating test suites that are small in size and need less execution time. On the other hand, IBEA was outperformed by the other algorithms when it tries to optimize test-cases to cover a certain set of branches. Our study revealed that both Modified IBEA and NSGA-II achieved more or less close scores in those three metrics, with a slight advantage for Modified IBEA in optimizing test case execution length and a tiny lead for NSGA-II when it comes down to the branch coverage and execution time.

This is the main characteristics of MOEA where it provides to the user a set of solutions to choose among. So, I can't conclude that the IBEA did not perform well, on the contrary, it achieved higher scores in test suite execution time and length. Hence, if a decision owner is willing to optimize the resources and efforts of maintaining a test suite, he should choose IBEA, otherwise he can go with NSGA-II or Modified IBEA for having a test suites that achieves a significant high branch coverage

6.3 Difficulties and Obstacles

6.3.1 Mastering *EVOSUITE*

Although *EVOSUITE* is extensible tool, but I faced some issues in implementing the algorithm and integrating it with *EVOSUITE*. Its a complicated tool and a lot of developers and researchers contributed to it, so it was not a trivial task. Moreover, I faced some complexities in adding two fitness functions that I chose to use in this thesis. In addition and due to the fact that its open source tool, I did not manage to add unit tests to test our work; building the tool along with executing its unit tests was impossible, because the build always fails. And thus, I was not very confident that taking the latest code version is the best approach. Nevertheless, I mitigated this issue by conducting a mini preliminary experiment to check the validity of the code added.

The second complexity that I ran into was the Java versions. According to developers of *EVOSUITE*, the tool is compatible with Java 8, 9 and 10. Although this is true, but the tool was not capable to generate unit-tests when invoking it in an environment running Zulu Java implementation. I found that the tool is only compatible with certain Java implementation like Oracle JDK or OpenJDK. So, I setup the experiment environment to run Java OpenJDK.

The third difficulty was exposing a certain metric from *EVOSUITE*. It was strange to us that *EVOSUITE* is not designed to expose the test-case execution time. It was not easy to expose a new non-coverage metric,

but I managed to tweak the code a bit and expose the execution time of a test-case and tests suite.

6.4 Future Work

In this section I will introduce what future studies can contribute to Search-based software testing and IBEA as a Multi Objective Indicator Algorithm, starting from modifying IBEA till contributing to the tool used in this study.

6.4.1 Analyzing different Variations of IBEA

One of the main concerns that this thesis aimed to, is the IBEA penetrating the field of Automating unit test generation. A lot of researchers were not keen to use the IBEA because of its high computational cost. This thesis studied both IBEA and a variation named modified IBEA. Future work would be to analyse and study different flavors of this algorithm. Moreover, there is a room of introducing a specific flavor of IBEA that fits the problem of unit test generation.

6.4.2 Analyzing CUT with large number of objectives

While analyzing the results of our experiment, I have found out that the algorithms behave more or less the same when the CUTs have a large

number of goals to cover. Since this experiment is repeatable, future studies could run larger scale exploratory studies that aim to figure out and analyze the behavior of algorithms when they are put to target large number of objectives and clarify the reasons behind the algorithm's behavior. Algorithms should be assigned larger search budget and longer timeout to the algorithms and examine the branch coverage of each algorithm.

6.4.3 IBEA to cover single branches at a time

As being emphasized in this thesis, the computation cost associated with the exact calculation of the hyper volume indicator in a high-dimensional space (i.e., with more than five objectives) is too expensive [44], making it impracticable to be used in the domain of test case generation where I target the branches individually as a standalone target instead of the Whole Suite approach where I target a group of branches as a single objective.

Having said that, there is a room here to evaluate IBEA (as well as to other variations of IBEA) and analyze how it performs when formulating the problem to target single branches at a time. IBEA then should be compared to MOSA [42, 44] and LIPS [43] and other approaches which focuses on generating a test case that covers a single branch at a time.

In order to improve IBEA performance, future work shall follow the researches in this area and apply some modification to the basic flow of IBEA to reduce its calculation time. The main idea would be around keeping the best test-cases during generation and saving some of the

search budget. To accomplish that, and inspired by the literature of this thesis, I suggest the following:

- The user of Archive to store the test cases which covered a certain branches and make use of the budget to generate test cases for the uncovered branches. This approach should also take into consideration the collateral branches a test case could cover by chance.
- Control the algorithm by enabling local search and pausing global search whenever the algorithm is close to cover a group of branches.
- Another possibility is to have multiple sets of target branches, and the total distance for each set is considered an objective by itself. Each set is reduced and completed again within each iteration.
- Make use of path testing to determine the control flow graph of CUT and considering the dependencies between branches and start covering a certain branch only if its not dependent on a parent branch.

Its worth noting here that future studies could adopt any of the suggested enhancements or a group of them to improve the algorithm's performance.

6.4.4 Contribute to EvoSuite

EVOSUITE is an extensible open source tool, anyone can contribute to this tool through their source code which can be found on GitHub ??,

and there are some testing techniques and search-based tools that can be found in other tools and are missing in *EVOSUITE*. I found that future studies could contribute to *EVOSUITE* through the following:

- Adding more code coverage criteria as well as to non-functional coverage functions such as readability and maintainability.
- Adding the capability to measure the Quality Indicators for the solution sets produced via applying different Algorithms, such as Spread, Epsilon and Hyper-Volumes quality indicators.
- Enhance the tool by adding the capability to conduct Statistical tests to compare the results of the generating algorithms.
- Contribute to the tool by adding different multi objective Genetic Algorithms such as Cellular Genetic Algorithm MOCeLL [41].

Bibliography

- [1] Abdel Rahman Ali M Ahmed, Mohamed H Gadallah, and HeshamA Hegazi. "Multi-Objective Optimization Indices: A Comparative Analysis". In: *Australian Journal of Basic and Applied Sciences* 8.4 (2016), pp. 1–12.
- [2] Andrea Arcuri. "A theoretical and empirical analysis of the role of test sequence length in software testing for structural coverage". In: *IEEE Transactions on Software Engineering* 38.3 (2011), pp. 497–519.
- [3] Andrea Arcuri. "It really does matter how you normalize the branch distance in search-based software testing". In: *Software Testing, Verification and Reliability* 23.2 (2013), pp. 119–147.
- [4] Andrea Arcuri. "Many independent objective (MIO) algorithm for test suite generation". In: *International Symposium on Search Based Software Engineering*. Springer. 2017, pp. 3–17.
- [5] Andrea Arcuri. "RESTful API automated test case generation". In: *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE. 2017, pp. 9–20.

-
- [6] Dave Astels. *Test driven development: A practical guide*. Prentice Hall Professional Technical Reference, 2003.
- [7] Moritz Beller et al. “When, how, and why developers (do not) test in their IDEs”. In: *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 2015, pp. 179–190.
- [8] Jason Brownlee. *Clever algorithms: nature-inspired programming recipes*. Jason Brownlee, 2011.
- [9] Kai H Chang et al. “A performance evaluation of heuristics-based test case generation methods for software branch coverage”. In: *International Journal of Software Engineering and Knowledge Engineering* 6.04 (1996), pp. 585–608.
- [10] Kalyanmoy Deb. *Multi-objective optimization using evolutionary algorithms*. Vol. 16. John Wiley & Sons, 2001.
- [11] Kalyanmoy Deb et al. “A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II”. In: *International conference on parallel problem solving from nature*. Springer. 2000, pp. 849–858.
- [12] Srinivas Devadas, Abhijit Ghosh, and Kurt Keutzer. “An observability-based code coverage metric for functional simulation”. In: *Proceedings of International Conference on Computer Aided Design*. IEEE. 1996, pp. 418–425.
- [13] Chartchai Doungsa-ard et al. “An automatic test data generation from UML state diagram using genetic algorithm.” In: (2007).

-
- [14] Feijoo E Colomine Duran, Carlos Cotta, and Antonio J Fernández-Leiva. “A comparative study of multi-objective evolutionary algorithms to optimize the selection of investment portfolios with cardinality constraints”. In: *European Conference on the Applications of Evolutionary Computation*. Springer. 2012, pp. 165–173.
- [15] Roger Ferguson and Bogdan Korel. “The chaining approach for software test data generation”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 5.1 (1996), pp. 63–86.
- [16] G. Fraser and A. Arcuri. “Whole Test Suite Generation”. In: *IEEE Transactions on Software Engineering* 39.2 (2013), pp. 276–291. DOI: 10.1109/TSE.2012.14.
- [17] Gordon Fraser and Andrea Arcuri. “A large-scale evaluation of automated unit test generation using evosuite”. In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24.2 (2014), p. 8.
- [18] Gordon Fraser and Andrea Arcuri. “Evolutionary generation of whole test suites”. In: *2011 11th International Conference on Quality Software*. IEEE. 2011, pp. 31–40.
- [19] Gordon Fraser and Andrea Arcuri. “EvoSuite at the SBST 2016 Tool Competition”. In: *9th International Workshop on Search-Based Software Testing (SBST’16) at ICSE’16*. 2016, pp. 33–36.
- [20] Gordon Fraser and Andrea Arcuri. “EvoSuite: automatic test suite generation for object-oriented software”. In: *Proceedings of the 19th*

-
- ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM. 2011, pp. 416–419.
- [21] Gordon Fraser and Andrea Arcuri. “EvoSuite: automatic test suite generation for object-oriented software”. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. Szeged, Hungary: ACM, 2011, pp. 416–419. ISBN: 978-1-4503-0443-6. DOI: 10.1145/2025113.2025179. URL: <http://doi.acm.org/10.1145/2025113.2025179>.
- [22] Gordon Fraser and Andrea Arcuri. “Whole test suite generation”. In: *IEEE Transactions on Software Engineering* 39.2 (2012), pp. 276–291.
- [23] Gordon Fraser, Andrea Arcuri, and Phil McMinn. “Test suite generation with memetic algorithms”. In: *Proceedings of the 15th annual conference on Genetic and evolutionary computation*. ACM. 2013, pp. 1437–1444.
- [24] Gordon Fraser and Andreas Zeller. “Generating parameterized unit tests”. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 2011, pp. 364–374.
- [25] Giovanni Grano et al. “Testing with Fewer Resources: An Adaptive Approach to Performance-Aware Test Case Generation”. In: *arXiv preprint arXiv:1907.08578* (2019).

-
- [26] Mark Harman, Yue Jia, and Yuanyuan Zhang. “Achievements, open problems and challenges for search based software testing”. In: *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. 2015, pp. 1–12.
- [27] Mark Harman, S Afshin Mansouri, and Yuanyuan Zhang. “Search-based software engineering: Trends, techniques and applications”. In: *ACM Computing Surveys (CSUR)* 45.1 (2012), pp. 1–61.
- [28] Børge Haugset and Geir Kjetil Hanssen. “Automated acceptance testing: A literature review and an industrial case study”. In: *Agile 2008 Conference*. IEEE. 2008, pp. 27–38.
- [29] John Henry Holland et al. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.
- [30] Marnie L Hutcheson. *Software testing fundamentals: Methods and metrics*. John Wiley & Sons, 2003.
- [31] Laura Inozemtseva and Reid Holmes. “Coverage is not strongly correlated with test suite effectiveness”. In: *Proceedings of the 36th international conference on software engineering*. 2014, pp. 435–445.
- [32] Hisao Ishibuchi, Yusuke Nojima, and Tsutomu Doi. “Comparison between single-objective and multi-objective genetic algorithms: Performance comparison and performance measures”. In: *2006 IEEE International Conference on Evolutionary Computation*. IEEE. 2006, pp. 1143–1150.

-
- [33] David Janzen and Hossein Saiedian. "Test-driven development concepts, taxonomy, and future direction". In: *Computer* 38.9 (2005), pp. 43–50.
- [34] Siwei Jiang et al. "A simple and fast hypervolume indicator-based multiobjective evolutionary algorithm". In: *IEEE Transactions on Cybernetics* 45.10 (2014), pp. 2202–2213.
- [35] Wenwen Li et al. "A modified indicator-based evolutionary algorithm (mIBEA)". In: *2017 IEEE Congress on Evolutionary Computation (CEC)*. IEEE. 2017, pp. 1047–1054.
- [36] Lu Luo. "Software testing techniques". In: *Institute for software research international Carnegie mellon university Pittsburgh, PA 15232.1-19* (2001), p. 19.
- [37] Sumio Masuda and Kazuo Nakajima. "An optimal algorithm for finding a maximum independent set of a circular-arc graph". In: *SIAM Journal on Computing* 17.1 (1988), pp. 41–52.
- [38] Thomas J McCabe. "A complexity measure". In: *IEEE Transactions on software Engineering* 4 (1976), pp. 308–320.
- [39] Phil McMinn. "Search-based software test data generation: a survey". In: *Software testing, Verification and reliability* 14.2 (2004), pp. 105–156.
- [40] Phil McMinn and Mike Holcombe. "Hybridizing evolutionary testing with the chaining approach". In: *Genetic and Evolutionary Computation Conference*. Springer. 2004, pp. 1363–1374.

-
- [41] Antonio J Nebro et al. “Mocell: A cellular genetic algorithm for multiobjective optimization”. In: *International Journal of Intelligent Systems* 24.7 (2009), pp. 726–746.
- [42] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. “Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets”. In: *IEEE Transactions on Software Engineering* 44.2 (2017), pp. 122–158.
- [43] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. “Lips vs mosa: a replicated empirical study on automated test case generation”. In: *International Symposium on Search Based Software Engineering*. Springer. 2017, pp. 83–98.
- [44] 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)*. IEEE. 2015, pp. 1–10.
- [45] Marlllos Paiva Prado and Auri Marcelo Rizzo Vincenzi. “Towards cognitive support for unit testing: A qualitative study with practitioners”. In: *Journal of Systems and Software* 141 (2018), pp. 66–84.
- [46] Outi Räihä. “Applying genetic algorithms in software architecture design”. MA thesis. 2008.
- [47] Colin R Reeves. “A genetic algorithm for flowshop sequencing”. In: *Computers & operations research* 22.1 (1995), pp. 5–13.

-
- [48] José Miguel Rojas et al. "A detailed investigation of the effectiveness of whole test suite generation". In: *Empirical Software Engineering* 22.2 (2017), pp. 852–893.
- [49] José Miguel Rojas et al. "Combining multiple coverage criteria in search-based unit test generation". In: *International Symposium on Search Based Software Engineering*. Springer. 2015, pp. 93–108.
- [50] Omur Sahin and Bahriye Akay. "Comparisons of metaheuristic algorithms and fitness functions on software test data generation". In: *Applied Soft Computing* 49 (2016), pp. 1202–1214.
- [51] Dragan Savic. "Single-objective vs. multiobjective optimisation for integrated decision support". In: (2002).
- [52] Abdel Salam Sayyad, Tim Menzies, and Hany Ammar. "On the value of user preferences in search-based software engineering: a case study in software product lines". In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 492–501.
- [53] Simone Scalabrino et al. "Search-based testing of procedural programs: Iterative single-target or multi-target approach?" In: *International Symposium on Search Based Software Engineering*. Springer. 2016, pp. 64–79.
- [54] Sina Shamshiri et al. "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)".

- In: *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE. 2015, pp. 201–211.
- [55] Sina Shamshiri et al. “Random or genetic algorithm search for object-oriented test suite generation?” In: *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*. ACM. 2015, pp. 1367–1374.
- [56] Chayanika Sharma, Sangeeta Sabharwal, and Ritu Sibal. “A survey on software testing techniques using genetic algorithm”. In: *arXiv preprint arXiv:1411.1154* (2014).
- [57] Michael Smit et al. “Maintainability and source code conventions: An analysis of open source projects”. In: *University of Alberta, Department of Computing Science, Tech. Rep. TR11 6* (2011).
- [58] Paolo Tonella. “Evolutionary testing of classes”. In: *ACM SIGSOFT Software Engineering Notes*. Vol. 29. 4. ACM. 2004, pp. 119–128.
- [59] Joris Van Geet et al. “A lightweight approach to determining the adequacy of tests as documentation”. In: *Proc. of the 2nd Workshop on Program Comprehension through Dynamic Analysis*. 2006, pp. 21–26.
- [60] Stefan Wappler and Frank Lammermann. “Using evolutionary algorithms for the unit testing of object-oriented software”. In: *Proceedings of the 7th annual conference on Genetic and evolutionary computation*. ACM. 2005, pp. 1053–1060.

-
- [61] Joachim Wegener, André Baresel, and Harmen Sthamer. “Evolutionary test environment for automatic structural testing”. In: *Information and software technology* 43.14 (2001), pp. 841–854.
- [62] Darrell Whitley. “A genetic algorithm tutorial”. In: *Statistics and computing* 4.2 (1994), pp. 65–85.
- [63] Manuel W Wik. “Revolution in information affairs: Tactical and strategic implications of Information Warfare and Information Operations”. In: *A. Jones, GL Kovacic h & PG Luzwick (eds.), Global information warfare* (2002), pp. 579–628.
- [64] Randy A Ynchausti. “Integrating unit testing into a software development team’s process”. In: *XP 1* (2001), pp. 84–87.
- [65] Eckart Zitzler. *Evolutionary algorithms for multiobjective optimization: Methods and applications*. Vol. 63. Citeseer, 1999.
- [66] Eckart Zitzler and Simon Künzli. “Indicator-based selection in multiobjective search”. In: *International Conference on Parallel Problem Solving from Nature*. Springer. 2004, pp. 832–842.

Appendices

Appendix A

Thesis External Links

Item	link
<i>EvoSuite</i> GitHub	https://github.com/EvoSuite/evosuite
Experiment Code Base	https://github.com/HadiAwad/evosuite
Thesis Latex Document	https://www.overleaf.com/read/smgssbkchfy
Thesis Source Code	https://github.com/HadiAwad/evosuite
Google	Google.com

Appendix B

Preliminary Experiment Results

To validate the implementation of the algorithms, we conducted a preliminary experiment over a set of nine classes provided by the tool authors. The following table B.1 shows each class as well as to the number of goals (branches) it has.

Class Name	Total_Goals	LoC
ATM	20	80
ATMCard	9	84
Bank	6	35
BankAccount	3	32
Owner	1	12
CurrentAccount	4	43
SavingsAccount	4	40
Company	2	24
Person	3	34

TABLE B.1: *EVOSUITE* Experimental Data-set

B.1 Preliminary Experiment Results

We have used each of the algorithms, namely NSGA-II, IBEA and mIBEA, to generate test suites for the classes presented in Table B.1. Each algorithm were used for 20 times (i.e. 20 runs) upon the subject CUTs. The metrics that were collected for the generated test suites are the following,

1. **Total Time:** The full-time EvoSuite spent generating the test cases
2. **Size: Number:** of tests in resulting test suite
3. **Length:** Total number of statements in the final test suite
4. **Covered Goals :** Total number of covered goals
5. **Mutation Score:** The obtained score for (strong) mutation testing
6. **Branch Coverage :** percentage of covered branched of CUT

The following table ?? summarizes the experiment results and shows the mean values of the above metrics.

Algorithm	NSGA-II	mIBEA	IBEA
Size	3.68	3.68	3.64
Length	17.38	17.22	16.57
MutationScore	0.69	0.69	0.68
Coverage	1	1	1
Covered_Goals	5.78	5.78	5.74
Total_Time	6404.57	7949.37	51194.33

TABLE B.2: Preliminary Experiment Result

B.2 Preliminary Experiment box Plots

The box plots of each metric are as the following, and the following results shows that IBEA has a high execution time when being compared to other algorithms. In fact, that is expected due to the high computational cost when calculating the hyper-volume differences by IBEA. It's also worth noting here the performance of mIBEA which has way less execution time, and yet the algorithm still achieves similar Coverage and Mutation score results when being compared to the base algorithm NSGA-II which is used as a reference in comparing the results.

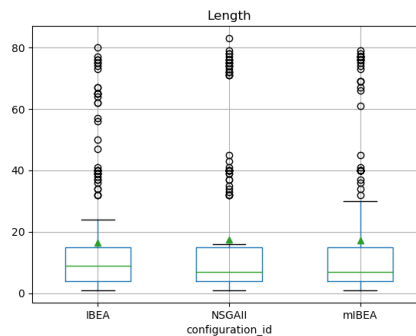


FIGURE B.1: Total Execution Time

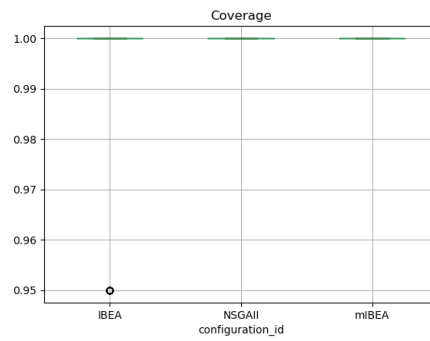


FIGURE B.2: Generated Test Suite Coverage

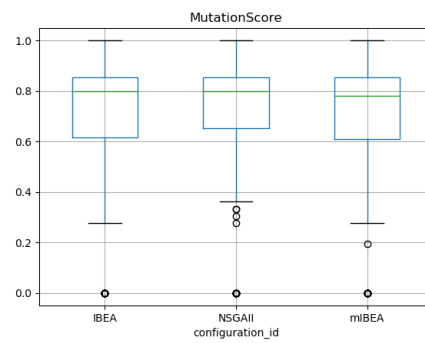


FIGURE B.3: Generated Test Suite Mutation Score

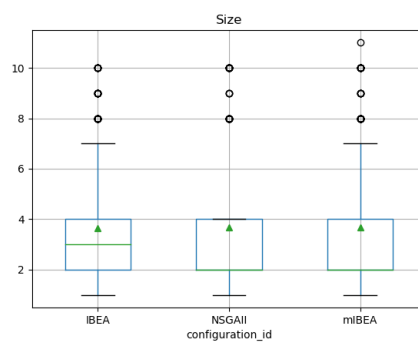


FIGURE B.4: Generated Test Suite Size

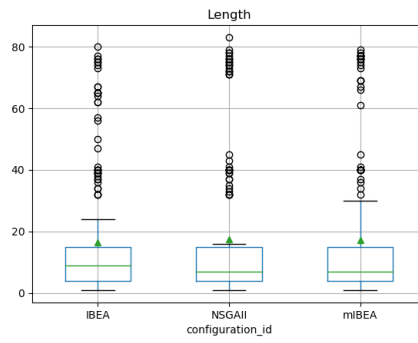


FIGURE B.5: Generated Test Suite Length

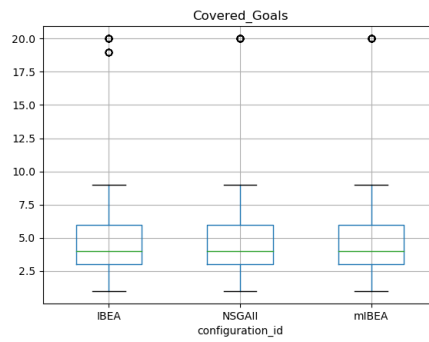


FIGURE B.6: Generated Test Suite Covered Goals