UNIVERSIDADE DE LISBOA

FACULDADE DE CIÊNCIAS

DEPARTAMENTO DE INFORMÁTICA

Bio-inspired Optimization Algorithms for Unit Test Generation

Gonçalo Miguel Inácio Duarte

Mestrado em Engenharia Informática
Especialização em Engenharia de Software

Dissertação orientada por:
Prof. Doutor José Carlos Medeiros de Campos e
Prof. Doutor Alcides Miguel Cachulo Aguiar Fonseca

2021

Acknowledgments

First, I would like to thank my supervisors José Campos e Alcides Fonseca for all the guidance, support and availability throughout this year.

I am very grateful to my parents and my brother, for all the support and encouragement during this phase of my life. Without them, my academic journey would not be possible.

I want to thank Diogo Faustino, Hugo Cristino, Luís Mestre and Francisco Veiga for the friendship, the happy moments and even the crazy ones we shared together. I also wish to thank Rita Costa and João Neto for all their support, help and friendship.

A big thanks to all the other people, friends and family who were not mentioned but helped during this journey.

Lastly, I am thankful to the Foundation for Science and Technology (FCT), project reference CPCA/A0/7402/2020 for the support given to this work.

Dedicatória.

Resumo

Na sociedade atual nós estamos rodeados e usamos todo o tipo de aplicações de software. Problemas no software pode causar todo o tipo de consequências, desde pessoas não conseguirem jogar um jogo como era suposto a uma aeronave despenhar-se matando toda as pessoas a bordo. De modo a que se evite certas consequências, convém que esse software não tenha problemas e funcione como é suposto. Porém, o software é escrito por humanos pelo que está sujeito a ter erros. Para lidar com esta situação, testes de software são feitos, de modo a que se descubra e resolva os problemas no software. Testar software baseado em pesquisa é uma área de teste de software que se tem mostrado bastante bem-sucedida na geração de conjuntos de teste unitários otimizados para cobertura de código. Esta abordagem usa algoritmos meta-heurísticos guiados por critérios de cobertura de código para gerar os testes. Neste estudo, foi utilizado um critério de cobertura múltiplo que é composto por oito critérios diferentes: a cobertura de linhas, cobertura de ramos, cobertura de métodos, cobertura de métodos de nível de topo sem exceção, cobertura de ramos direto, cobertura de output, mutação fraca e cobertura de exceções.

No que diz respeito aos algoritmos meta-heurísticos, os algoritmos evolucionários são o estado da arte atual, tendo apresentado os melhores resultados em estudos anteriores, superando os algoritmos aleatórios. No entanto, serão os algoritmos evolucionários realmente os melhores algoritmos neste contexto? E quanto aos algoritmos de inteligência de grupo, poderão eles também apresentar bons resultados? Poderá o atual estado da arte ser substituído por um algoritmo de inteligência de grupo?

Deste modo, para responder a estas e outras questões, decidimos explorar os algoritmos bio-inspirados, também conhecidos por algoritmos de inteligência de grupo. Estes algoritmos baseiam-se no comportamento de indivíduos que pertencem a grupos na natureza, tais como os enxames de abelhas. Os algoritmos bio-inspirados não são completamente novos na área de testar software. Estudos anteriores mostram que os algoritmos de inteligência de grupo são geralmente melhores que os algoritmos genéticos para testes de estrutura, que na geração de dados para testes o desempenho dos algoritmos depende do tipo de problema e que na geração automática de testes Artificial Bee Algorithm teve o melhor desempenho e o Bat Algorithm é o mais rápido a executar.

Nós escolhemos implementar dez algoritmos de inteligência de grupo que possuem várias características diferentes, com diferentes graus de popularidade e que incluem algoritmos antigos e recentes. Os algoritmos escolhidos são: Genetic Bee Colony (GBC) Algorithm, Fish Swarm Algorithm (FSA), Cat Swarm Optimization (CSO), Whale Optimization Algorithm (WOA), Artificial Algae Algorithm (AAA), Elephant Herding Optimization (EHO), Chicken Swarm Optimization Algorithm (CSOA), Moth Flame Optimization (MFO) Algorithm, Grey Wolf Optimization (GWO) Algorithm and Particle Swarm Optimizer (PSO). Para representar os algoritmos evolucionários e servir de comparação contra os algoritmos de inteligência de grupo, escolhemos o Standard Genetic Algorithm (Standard GA), Many-Objective Sorting Algorithm (MOSA) e o Dynamic Many-Objective Sorting Algorithm (DynaMOSA). Este último é o estado da arte atual. Além destes algoritmos, foi implementado mais um algoritmo que é um híbrido (fusão de algoritmos de inteligência de grupo e evolucionários), o Elephant Dynamic Many-Objective Sorting Algorithm (Elephant-DynaMOSA). O EvoSuite foi a ferramenta de geração de testes escolhida para implementar o híbrido e os dez algoritmos de inteligência de grupo por já possuir diversas otimizações, os algoritmos evolucionários já estão implementados e a natureza modular da ferramenta permite facilmente adicionar novos algoritmos ao módulo dos algoritmos.

O estudo empírico realizado consiste em duas experiências: a calibração dos parâmetros e a avaliação dos algoritmos. Na primeira experiência, escolhemos vários parâmetros e testámos vários valores destes para cada algoritmo. Foi selecionado um subconjunto de 34 classes e testou-se em 30 seeds diferentes durante 60 segundos para se obter os resultados de cada configuração. De seguida, aplicámos o método estatístico Vargha-Delaney de modo a encontrar a melhor configuração de cada algoritmo. A segunda experiência consistiu em correr a melhor configuração de cada algoritmo em 312 classes com 30 seeds durante 60 segundos. Depois, com o intuito de interpretar os resultados obtidos e conseguir ver qual o melhor algoritmo de inteligência de grupo, se os algoritmos de inteligência de grupo são melhores que os três algoritmos evolucionários e quão boa é a performance do algoritmo híbrido, foram usados os métodos estatísticos de Vargha-Delaney e teste de Friedman. Também se observou a relação entre diversos aspetos dos resultados: a cobertura e o número de gerações, cobertura e a pontuação de mutação, cobertura e diversidade e cobertura e tamanho dos testes.

Os nossos resultados revelam que EHO foi o melhor algoritmo de inteligência de grupo e que também superou o Standard GA. Porém, tanto DynaMOSA e MOSA mostram-se superior ao EHO. Em relação ao Elephant-DynaMOSA, que é o híbrido do melhor algoritmo de inteligência de grupo e evolucionário, os resultados foram melhores que o EHO, visto que tem um desempenho semelhante ao MOSA. No final, DynaMOSA foi o algoritmo com maior cobertura média e com os melhores resultados estatísticos nos dois métodos usados.

Posteriormente, decidimos discutir outras particularidades dos resultados e propusemos três hipóteses: o melhor algoritmo é superior em todas as classes, qualquer algoritmo consegue atingir pelo menos 50% de cobertura em todas as classes e o desempenho aumenta se o tempo de execução aumentar. A primeira hipótese provou-se falsa visto que houve seis algoritmos estatisticamente melhor que os outros em certas classes: Standard GA, MOSA, DynaMOSA, EHO, Elephant-DynaMOSA e FSA. Isto foi provado ao mostrar-se os valores médios de vários aspetos obtidos nas execuções (número de gerações e testes, tamanho dos testes e cobertura), os resultados do método estatístico Vargha-Delaney e o desempenho de cada algoritmo por critério de cobertura de código. A segunda hipótese também se provou falsa porque 17.5% das classes obtiveram menos de 50% de cobertura independentemente do algoritmo usado. Uma das principais razões é a limitação do EvoSuite como ferramenta de testes, por exemplo não conseguir gerar os inputs necessários para correr a classe. A última hipótese foi a única que se provou ser verdadeira. Para responder a esta hipótese, selecionados a melhor configuração por algoritmo e correu-se 312 classes em uma seed durante uma hora. A cobertura média de todos os algoritmos subiu cerca de 7% e 13 dos 14 algoritmos melhoraram a sua cobertura. Também observámos a evolução dos algoritmos durante a execução e apenas uma minoria dos algoritmos mostrou uma melhoria significativa no desempenho após 60 segundos. Por isso, concluiu-se que apesar da melhoria geral no desempenho, tal melhoria poderá não valer a pena devido ao aumento de recursos necessários com um maior orçamento de tempo.

Com isto podemos concluir que apesar do DynaMOSA manter-se como o estado da arte, ele não é o melhor em todas as situações. E que os algoritmos de inteligência de grupo mostraram um certo grau de potencial, principalmente o algoritmo híbrido, Elephant-DynaMOSA. Por isso, nós sugerimos que para trabalho futuro se teste mais algoritmos de inteligência de grupo e algoritmos de múltiplos objetivos, com foco em algoritmos híbridos que combinem os melhores aspetos dos vários algoritmos. Outra iniciativa que pode ser realizada é analisar que algoritmos são melhores para cada critério de cobertura e criar um algoritmo múltiplo capaz de se adaptar e otimizar a procura tendo em conta os critérios de cobertura escolhidos.

Palavras-chave: testes de software baseados em pesquisa, algoritmos de inteligência de grupo, estudo empírico

Abstract

Search-based software testing is an area of software testing that has shown to be quite successful in generating unit test suites optimized for code coverage. This approach uses meta-heuristic algorithms guided by code coverage criteria (e.g., branch coverage) to generate the tests.

When it comes to meta-heuristic algorithms, evolutionary algorithms are the current state-of-the-art, having presented the best results in previous studies. However, are evolutionary algorithms truly the best algorithms in this context? What about bio-inspired algorithms, can they also present good results? Will the current state-of-the-art be replaced with a bio-inspired algorithm?

In order to answer these and other questions, we performed an empirical study where we evaluated ten bio-inspired algorithms, three evolutionary algorithms and one hybrid algorithm (a mix of bio-inspired and evolutionary algorithms) on a selection of non-trivial open-source classes. EvoSuite was the test generation tool chosen to implement the ten bio-inspired algorithms and the hybrid since it already has several optimizations and the evolutionary algorithms implemented.

Our results show that the Elephant Herding Optimization has the best performance among the bio-inspired algorithms and has surpassed the Standard Genetic Algorithm. However, both the Many-Objective Sorting Algorithm (MOSA) and the Dynamic Many-Objective Sorting Algorithm (DynaMOSA) showed superior efficiency compared to all ten bio-inspired algorithms. When it comes to the hybrid algorithm, Elephant Dynamic Many-Objective Sorting Algorithm (Elephant-DynaMOSA), it ended up with a similar performance to MOSA but still worse than the current state-of-the-art, DynaMOSA. We also discussed three hypotheses about the results obtained.

Although DynaMOSA remains the state-of-the-art algorithm, it is not the best in all classes. Not only so, but the bio-inspired algorithms showed some potential, especially in the case of the hybrid, Elephant-DynaMOSA. Thus, we suggest future work on hybrid algorithms that fuse the best aspects of several algorithms.

Keywords: search-based software testing, bio-inspired algorithms, empirical study

Contents

List of Figures

List of Tables

Acronyms

AAA	Artificial Algae Algorithm.
BIA	Bio-inspired Algorithm.
CI	Confidence Interval.
CSO	Cat Swarm Optimization.
CSOA	Chicken Swarm Optimization Algorithm.
CUT	Class Under Test.
DynaMOSA	Dynamic Many-Objective Sorting Algorithm.
EHO	Elephant Herding Optimization.
EIA	Evolution-inspired Algorithm.
Elephant-DynaMOSA	Elephant Dynamic Many-Objective Sorting Algorithm.
FCT	Foundation for Science and Technology.
FSA	Fish Swarm Algorithm.
GA	Genetic Algorithm.
GB	Giga Bytes.
GBC	Genetic Bee Colony.
GHz	Giga Hertz.
GWO	Grey Wolf Optimization.
JDK	Java Development Kit.
MFO	Moth Flame Optimization.

MOA	Many-objective Algorithm.
MOSA	Many-Objective Sorting Algorithm.
PSO	Particle Swarm Optimizer.
RAM	Random Access Memory.
RQ	Research Question.
RT	Random Testing.
SBST	Search-based Software Testing.
SE	Symbolic Execution.
WOA	Whale Optimization Algorithm.

Chapter 1

Introduction

1.1 Motivation

Nowadays, software is everywhere, e.g., smartphones, computers, cars and all kinds of different machines. It has become something of utmost importance for today's society and it is responsible for all kinds of technological advances.

Unfortunately, all software programs have bugs. This is something inevitable since code is written by humans and we are not perfect. Bugs can be mistakes when writing code, failure to satisfy certain requirements, etc. A few examples of bugs that have happened in the last decade are: Toyota's electronic throttle control system could cause sudden unintended acceleration, Valve's Steam client for Linux could accidentally delete all the user's files in every directory on the computer and the aeroplane Boeing 787 Dreamliner would have an integer overflow bug that shut down all electrical generators if the aircraft was on for more than 248 days [1]. These bugs caused all sorts of consequences, in the case of the customers/users that used those products, they had their properties damaged or destroyed and some even lost their lives. In the case of the companies responsible for the defective software, they lost trust from the customers, experienced stock market drops, lost lawsuits against them, etc.

What can be done if all software has bugs? To ensure that the software is as "bug-free" as possible, software testing is used, since it is the main approach for quality assurance. Just by performing simple testing on error handling code, the majority of catastrophic failures can be avoided [2]. However, when developing software, software testing usually costs around 50% of the resources, e.g., time, of the project [3].

Nowadays, it has already been shown that automatic generation of test cases is superior to manually writing them since they usually are computationally cheap, faster and more complete, as they are generated in a systematically way [4].

Search-based software testing (SBST) consists of partially or fully automate a testing task, e.g. automatic generation of test suites, using meta-heuristic optimization techniques, like genetic algorithms (GAs). In order to reach an optimal or near-optimal solution within a practical time limit, a problem-specific fitness function, which evaluates how good a solution is, is used to guide the search in the enormous search space to promising solutions, through the evaluation of the solutions found [5, 6].

In recent years, there has been a trend in the optimization of coverage criteria, in that single coverage criteria loses significance over multiple coverage criteria [7]. This happens because single coverage criteria are often insufficient to make a good test unit suite since its desired properties are multi-faceted. The fact that multiple coverage criteria are computationally cheap also helps in their favour. According to Rojas et al. [7], combining 9 different criteria lead to an average decrease of only 0.4% of the constituent coverage criteria and an up to 70% increase in the size of test suites. In the end, even though the average coverage has a slight decrease, it is worth it because the quality of the test suites is better.

The term nature-inspired algorithm (NIA), as the name implies, includes all kinds of algorithms inspired by nature, making it a very broad term. In Tang et al. [8], NIAs are further divided into four major groups of algorithms depending on their inspiration: evolution-inspired algorithms (EIAs), bio-inspired algorithms (BIAs), physics-inspired algorithms and human-behaviour-inspired algorithms. Molina et al. [9] add another two groups to these four, plant-inspired algorithms and miscellaneous (those that do not fit in the other groups).

Regarding the automatic generation of tests suites optimized for code coverage, the search-based approach of EIAs has shown to be quite effective, having a better performance than random approaches [10].

However, the EIAs are just a type of NIAs. There exists an enormous amount of NIAs, and most of them remain largely unexplored. In this thesis, we will focus on a different type of NIAs, the BIAs [8, 9].

1.2 Objectives

In this thesis, the objective is to answer these next questions:

RQ1: Which bio-inspired algorithm performs best?

RQ2: How does swarm-based search compare to traditional evolutionary search?

RQ3: How does swarm-based search compare to many-objective optimization algorithms?

RQ4: How does a hybrid that combines swarm-based search with many-objective optimization performs?

In order to answer these questions, we used the state-of-the-art generation tool EvoSuite, since it has been shown to have good performance in similar contexts [7, 10].

In this thesis, we explored ten BIAs: Genetic Bee Colony (GBC) Algorithm, Fish Swarm Algorithm (FSA), Cat Swarm Optimization (CSO), Whale Optimization Algorithm (WOA), Artificial Algae Algorithm (AAA), Elephant Herding Optimization (EHO), Chicken Swarm Optimization Algorithm (CSOA), Moth Flame Optimization (MFO) Algorithm, Grey Wolf Optimization (GWO) Algorithm and Particle Swarm Optimizer (PSO). Several of them have already shown promising results in a similar context in previous studies [11]. We also decided it would be interesting to fuse bio-inspired and genetic algorithms and created Elephant Dynamic Many-Objective Sorting Algorithm (Elephant-DynaMOSA).

We evaluated the performance of these 11 algorithms by comparing them with three EIAs: the Standard GA, Many-Objective Sorting Algorithm (MOSA) and the Dynamic Many-Objective Sorting Algorithm (DynaMOSA) [10, 12]. We believe these algorithms are representative of EIAs, as one of the best performing single-objective EIA is Standard GA, despite having no optimization towards the particular goal of generating tests. While MOSA and DynaMOSA are evolutionary algorithms optimized to generate tests, with the latter being the state-of-art algorithm for unit test generation [10].

1.3 Contributions

This thesis has several contributions related to the use of BIAs for unit test generation.

We explained and presented several alterations necessary to adapt the BIAs to the context of software testing. We also created a hybrid algorithm, Elephant-DynaMOSA, that combines the characteristics of EHO and DynaMOSA. The idea behind Elephant-DynaMOSA is to take the concepts of a BIA and insert them in an algorithm optimized to test generation.

We performed an empirical study of ten BIAs, three EIAs and one hybrid algorithm that optimize multiple criteria for the generation of unit tests. This study shows the performance of the 11 algorithms implemented and compares it with the performance of EIAs specialised in software testing or not specialised.

1.4 Thesis Outline

This thesis is organized into seven chapters. Chapter 2 presents the related work where BIAs and the three EIAs used in this study are used in software testing. Chapter 3 introduces the field of search-based software testing, covering local search algorithms, global search algorithms, fitness functions and the state-of-art tool EvoSuite. Chapter 4 explains the adaptations done during the implementation of the BIAs. Chapter 5 describes the experimental setup, presents the threats to validity for this study and shows the results of the experiments. Chapter 6 analyses three hypotheses about the results. Chapter 7 are the conclusions and future work.

Chapter 2

Related Work

Swarm intelligence is a sub-field of artificial intelligence, that consists of solving real-world problems through simulating the behaviours and interactions of individuals that belong to biological swarms, e.g., colonies of ants, flocks of birds and swarms of fishes. Hence, the BIAs are also commonly known as swarm intelligence algorithms. BIAs are used in all kinds of applications, e.g., network routing, power systems, parameter optimizations, image processing and software testing [8, 9, 13].

In the context of software testing, Windisch et al. [11] applies PSO to structural testing and compares its performance with GAs. Their results show that PSO has as good or even better performance (mostly in complex problems) than GAs and it is much faster in most cases. Sahin and Akay [14] evaluate PSO, Differential Evolution, Artificial Bee Colony, Firefly Algorithm and Random Search Algorithm on software test data generation. They conclude that the best algorithm depends on the type of problem, e.g., Artificial Bee Colony is effective in problems that have multimodality and the random approach has a good performance in simple problems with a small search space. Bruce et al. [15] proposes a novel Ant Colony Optimization Algorithm to object-oriented unit test generation and develops a tool that implements the said algorithm. The results were promising, with the tool that implements the ant algorithm beating Randoop, a testing tool that has been under development for around ten years. However, the other testing tool used in the study, EvoSuite, obtained the best results which show there is still room for improvement. Khari et al. [16] test Hill Climbing Algorithm, PSO, Firefly Algorithm, Cuckoo Search Algorithm, Bat Algorithm and Artificial Bee Colony for automated test suite generation. Their results show that Artificial Bee Colony is the most optimal algorithm follow by Bat Algorithm and PSO. Considering execution time, Bat Algorithm is the fastest followed by Hill Climbing and Artificial Bee Colony. Campos et al. [10] despite not using any BIA, performed an empirical study that evaluates 13 EIAs (including Standard GA, MOSA and DynaMOSA) and two random algorithms for unit test generation. They conclude that random approaches are inferior to the evolutionary ones, that DynaMOSA is the state-of-the-art algorithm and that there is not an algorithm superior in all classes.

Chapter 3

Search-based Software Testing

Software testing is a key part of the developing process of a product of software and aims to improve the software quality of the said product. This is done with the verification and validation of software [17]. The former serves to check if the developers have fulfilled the software specifications, while the latter sees if the software does what is supposed to do. Software testing cannot prove the absence of bugs, but it can expose their existence. This happens because it is not feasible to try every possible input and/or configuration in the vast majority of software products.

Currently, there are three main areas of study in software testing: random testing (RT), symbolic execution (SE) and SBST.

RT consist of generating random inputs from the whole input domain to test the software. There is some disagreement about its coverage and effectiveness, e.g., Shamshiri et al. [18] demonstrated that RT can be better than GAs, while Campos et al. [10] explored GAs with several optimizations and they end up being better than random approaches. Some enhancements to RT such as Adaptive Random Testing have been proposed, with Adaptive Random Testing having less than 1% chance of finding bugs [19].

SE uses symbolic values instead of concretes inputs and that computes symbolic expressions representing the software variables [17]. In general, SE and its variants can produce high-coverage tests but can have several problems with more complex programs and object-oriented programs. This is due to the problem of path explosion, i.e., the number of paths can grow exponentially with the size of the software. And also the fact that object-oriented programming is more complicated than just optimizing certain inputs to cover several specific paths, i.e., it is necessary to have in account certain sequences of statements to invoke and interact with objects. Another problem with SE is the modulation of side effects that exist in several programming languages, such as Java. This modulation of side effects consists in the results of functions be only known during runtime, and SE analyses the source code without running it. So, this means that SE is not a reliable tool when this happens.

SBST is the application of meta-heuristic algorithms to software testing. Meta-heuristic algorithms are inspired by natural evolution and have been successfully used to address many kinds of optimisation problems, e.g., at generating test suites (i.e., set of unit test cases) optimized for code coverage in object-oriented classes [10]. In meta-heuristic algorithms, a solution is encoded "genetically" as an individual ("chromosome"), and a set of individuals is called a population. The population is gradually optimised using genetic-inspired operations such as crossover, which merges genetic material from at least two individuals to yield new offspring, and mutation, which independently changes the elements of an individual with a low probability. SBST is further explored in the following sections, in particular: representation of a solution (Section 3.1), evaluation of a solution (Section 3.2), and optimization of a solution (Section 3.3 and Section 3.4). Finally, Section 3.5 presents the SBST tool used in this thesis.

3.1 Representation of a solution

In the context of SBST, a solution (also known as an individual) is test data that aim to exercise most of the program under test.

For example, assuming we have developed the program in Figure 3.1, a solution can be represented by a test suite, i.e, a set of test cases. Each test case has four integer numbers that are used as input of method `my_func`, e.g., $T = \{t1, t2\}$ where $t1 = \{a = 3, b = 5, c = 1, d = 8\}$ and $t2 = \{a = 6, b = 4, c = 2, d = 7\}$. Although T might

```
void my_func(int a, int b, int c, int d) {
    if (a >= b) {
        if (b >= c) {
            if (c >= d) {
                // target
            }
        }
    }
...
```

Figure 3.1: Example of program under test.

exercise/test a part of `my_func`'s source code, is it the most effective test data? Is there a better one? In the following sections, we describe how a solution can be evaluated and we then describe two types of meta-heuristic algorithms (local and global search) that aim to optimize one or more solutions.

3.2 Fitness Functions

A fitness function computes and assigns a fitness value to a solution. A fitness function is also used in order to provide a gradient to local/global search algorithms [6, 7, 10, 17] (more on this in the next section).

One of the most known fitness functions in SBST is *branch coverage* [7, 17, 20]. Usually, it is often interpreted as the number of branches of conditional statements that are covered by a test suite. Thus, a test suite is said to fully satisfy the branch coverage criterion if and only if for every branch statement in the class under test (CUT), it contains at least one test case whose execution evaluates the branch predicate to *true*, and at least one test case whose execution evaluates the branch predicate to *false*. As a software bug can only be triggered if the buggy code is executed, a test with a higher branch coverage is more likely to trigger a bug than a test with low code coverage [21].

The fitness value of a test suite T is measured by executing all its test cases, keeping track of the branch distances $d(b, T)$ for each branch b in the program under test. Hence, the equation is a follows [7, 10, 17]:

$$fitness(T, B) = \sum_{b \in B} d(T, b) \tag{3.1}$$

where $d(T, b)$ is be defined as:

$$d(T, b) = \begin{cases} 0 & \text{if branch } b \text{ has been covered,} \\ v(d_{min}(t \in T, b) & \text{if the predicate has been executed at least twice,} \\ 1 & \text{otherwise.} \end{cases} \tag{3.2}$$

The branch distance, $d(T, c)$, is a heuristic that quantifies how far a branch (i.e., the control flow edge resulting from a true/false evaluation of an if condition) is from being evaluated to true or to false. Table 3.1 shows how branch distance is applied on a few predicates of a test case depending on the evaluation of *true* and *false*. There are several normalising functions v that can be used, e.g., $bd(x) = 1 - \alpha^{-x}$, where $\alpha > 1$ [22]. A typical value for α is 1.001.

Expression	**Distance True**	**Distance False**
x == y	|x – y|	1
x != y	1	|x – y|
x > y	y – x + 1	x - y
x >= y	y - x	x – y + 1
x < y	x – y + 1	x – y
x <=y	x - y	x – y + 1

Table 3.1: Example of *branch distance* applied on several predicates adapted from Fraser [23].

In order to show an example of how to calculate the fitness of branch coverage in a test suite, we will consider the program under test in Figure 3.1, which has three branches: lines 2, 3, and 4; and the $T = \{t1, t2\}; t1 = \{a = 3, b = 2, c = 4, d = 8\}, t2 = \{a =$

$6, b = 4, c = 2, d = 7\}$. In this case, $t1$ evaluates the first branch as *true*, the second as *false* and does not execute the last branch. While $t2$ evaluates the first two branches as *true* and the last one as *false*. We can see that the first branch has not been covered and has been executed more than once, the second branch has been covered (at least one test evaluates as *true* and other as *false*) and the last branch has not been covered and only executed once. Now we have to assign the values for each branch according to the previous equations: for the first branch we will use $v(d_{min}(t \in T, b)$, i.e, calculate the distance to false for the two test cases ($d_{t1} = 2$ and $d_{t2} = 3$), select the smaller one (d_{t2}) and use it for the normalising function ($1 - 1.001^{-2} = 0.001997$). The second branch is covered so it is assigned the value 0, while the last branch has 1 assigned to it. With this we can see that the fitness value of T is $0.001997 + 0 + 1 = 1.001997$.

When using multiple coverage criteria, a weighted linear combination of the different fitness functions could be used. That is, each fitness function has a weight assigned to it and the more relevant it is for the testing, the more weight it has [7]. In this thesis, we explored multiple coverage criteria, using the seven different fitness functions described below in addition to branch coverage. All fitness functions have the same weight assigned to them.

- Method coverage - checks if a method in the CUT is called at least once by a test suite. This call can be a direct or indirect one.
- Exception-free Top-level Method coverage - checks if a method in the CUT is directly called at least once by a test suite and does not terminate with an exception.
- Line coverage - checks if every line in the source code is covered by at least one test. Comments lines and lines that do not contain code, do not count for this coverage.
- Direct Branch coverage – checks if each branch of a public method in the CUT is covered by a direct call. There is not a restriction on branches in private methods.
- Output coverage – whether the different output of each public method is covered, e.g, a boolean method has to have at least a test that returns true and another that returns false to cover all possible outputs.
- Weak Mutation – the test generation creates small changes to the source code, i.e., mutations, and then checks if the tests can distinguish the mutants from the originals. A mutant is considered ‘killed’ if the test that executes it ends up in a different state than it was supposed to in the original source code.
- Exception coverage - counts how many exceptions were caught in the CUT. Since it is impossible to know in advance of many exceptions there are, no percentage of the total number of exceptions can be given.

3.3 Local Search

In local search optimization algorithms, only *one solution* is optimized at a time and when they have to consider other solutions and see if they are better than the present one, the algorithms only consider the neighbours [6]. These neighbours are the closest solutions to the current solution when looking through the search space.

3.3.1 Hill Climbing

Hill climbing is a well-known local search algorithm. It consists of selecting a random solution of the search space, then compares it with its neighbours, e.g., in a 1-dimensional search space, the existing solution is compared with its left and right neighbours. The solution with the best score, i.e., best fitness value, replaces the current one. This comparison is repeated until the existing solution is better than its neighbours.

This algorithm is simple and fast, but it can give sub-optimal results. This happens because the fitness function only explores the neighbours of the existing solution and do not take the whole space search into consideration, which can lead to a solution that is better than its neighbours, but when taking into account the global of the search space is not the best solution. As a way to improve this, there are several extensions of Hill climbing that incorporate several “restarts”, i.e., use several different starting solutions, to explore more of the search space.

3.3.2 Simulated Annealing

Simulated annealing is a local search algorithm similar to Hill climbing. But unlike the previous algorithm, is capable of accepting worse solutions depending on an ever decreasing probability. This probability is represented by a parameter called temperature.

The search starts with a high temperature so that the starting position is no longer so decisive for the algorithm and allow greater freedom to explore the search space. As the search continues, the value of the temperature decreases until it reaches zero. At this point, Simulated annealing works just like Hill climbing.

The rate at which the temperature decreases, is important since if the temperature reaches zero very quickly, the algorithm might stop at a sub-optimal solution. On the other end, if the temperature decreases very slowly, the amount of time the algorithm takes to run can become unreasonable long.

3.4 Global Search

In global search optimization, unlike local search optimization, the algorithms consider *several* solutions at once and they are not restricted to the neighbourhood of said solutions [6]. This way the search is no longer bound by the local solutions and encompasses the whole search space when trying to find the optimal solution.

3.4.1 Genetic Algorithms

GAs are very well known among the NIAs, mostly because they are the target of studies in the past decades and usually have good results. Not only that, but they are also easy to implement and understand. GAs are based on the principles of the Darwinian theory of evolution.

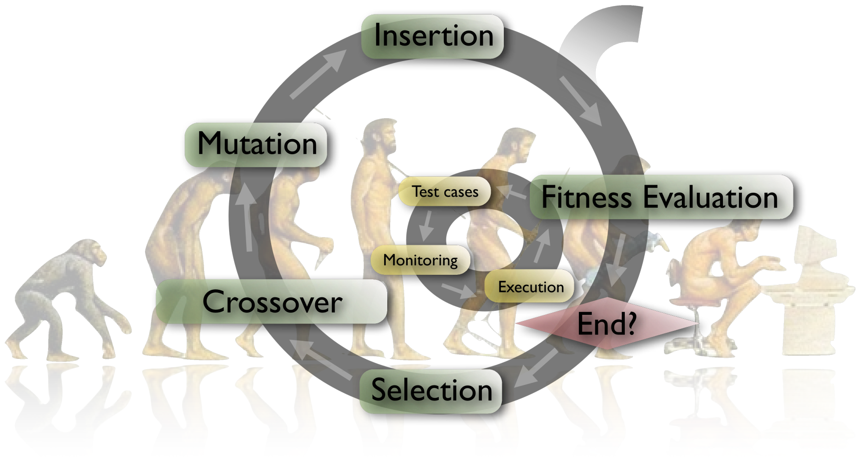


Figure 3.2: Diagram of a genetic algorithm taken from Fraser [23].

A few terms that are used with GAs are individuals, population, selection, crossover and mutation. An individual is a solution that represents a genetically encoded chromosome, while a population is a set of individuals. The other three terms are operations that help optimize the population of individuals. The case of selection consists of choosing individuals for reproduction, wherein fitter/better individuals have a higher chance of being chosen. On the other hand, the crossover is the merging of the genes of at least two individuals, in order to create a new individual. The mutation is the operation that changes/mutate the genes of an individual, according to a low probability. Finally, after all the operations are over, the GA ends up with a new population of individuals and repeats the previous operations or it terminates with the new population being the final population. This termination can be determined by executing time or by fulfilling several problem-specific criteria. The flow of a GA is depicted in Figure 3.2.

An important fact to mention is that the three genetic operations have several types, i.e., each operation has several possible techniques that can be used. In the case of selection, three of the most popular operators are rank, tournament and roulette wheel selection [24]. Rank selection is usually used when the fitness values of the individuals are very similar (more common at the end of the run) and basically, it ranks the individuals according to the fitness values and uses that absolute ranking as the factor to select the individuals, instead of the fitness values. In tournament selection, a random subset of individuals of the population is selected and then they compete to find the individual with the best fitness value among them. Roulette wheel selection is a method where the probability of an individual being chosen is proportional to its fitness, i.e., the better the fitness value the larger chance of being selected.

One-point and two-point crossovers are the most popular recombination operators [24]. The one-point crossover consists of a single random point being selected on the chromosome of the parents, then that point serves as a mark where the genes of the parents are swapped when the parents reproduce and create the children. This way, the children are different from their parents and have their genetic information. In the two-point crossover, two points are randomly selected instead of one, and the genes between those two points are the ones swapped when the parents reproduce.

Two relevant mutation operators are swap and scramble mutation. Swap mutation consists of randomly selecting two genes of an individual and interchange their values. While scramble mutation selects a subset of genes and then shuffles it at random.

We will now apply these genetic operations in the context of SBST using the program under test in Figure 3.1 and three test suites:

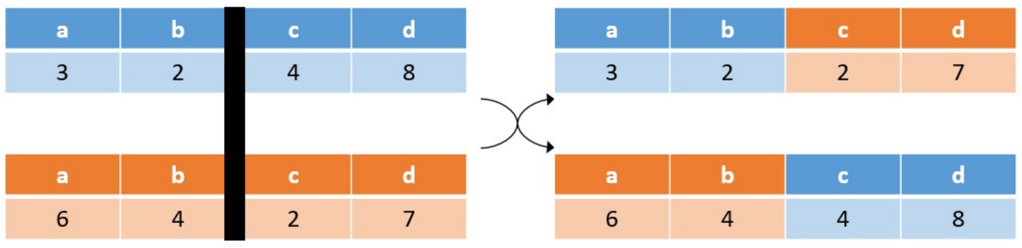
$T1 = \{t1, t2\}; t1 = \{a = 3, b = 2, c = 4, d = 8\}, t2 = \{a = 3, b = 2, c = 2, d = 1\}$
$T2 = \{t3, t4\}; t3 = \{a = 6, b = 3, c = 4, d = 8\}, t4 = \{a = 6, b = 4, c = 2, d = 7\}$
$T3 = \{t5, t6\}; t5 = \{a = 1, b = 4, c = 7, d = 8\}, t6 = \{a = 2, b = 4, c = 5, d = 9\}$

Test Suite	Fitness	Rank Selection	Roulette Wheel Selection
$T1$	1.001997	1st	$\approx 40.008\%$
$T2$	1.002994	2nd	$\approx 39.988\%$
$T3$	2.001997	3rd	$\approx 20.004\%$

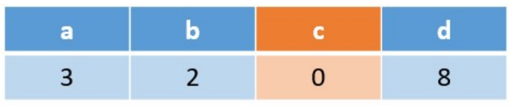
Table 3.2: Example of how selection works.

In Table 3.2, we can see the fitness values of the three individuals used in this example, along with the application of two selection techniques. In rank selection, the fittest individual is placed in the first position, while the worst individual is placed in the last position. In the roulette wheel selection, each individual has assigned to it a certain percentage that defines how probable it is to being selected.

We use rank selection to choose the two fittest individuals ($T1$ and $T2$). Then, we apply the crossover operation that consists of selecting a random test case from $T1$ and $T2$, followed by a single-point technique as we can see in Figure 3.3a. In Figure 3.3b, we can see an example of swap mutation on test case $t1$ from individual $T1$, that consists of selecting a random input, read it in byte format and swap the value of a single bit ($4 = 0100$ changes to $0 = 0000$).



(a) Crossover example between test case $t1$ from individual $T1$ (blue) and test case $t4$ from individual $T2$ (orange).



(b) Mutation example on test case $t1$ from individual $T1$.

Figure 3.3: Examples of genetic operations in the context of software testing.

Standard Genetic Algorithm

Standard GA (Algorithm 1) is a well-known algorithm that uses the three genetics operations (selection, crossover and mutation). In the beginning, a random population of individuals is generated and evaluated according to their fitness value (lines 1-2). Then, following a certain selection technique (e.g., rank selection), two individuals are selected (line 7). After that, according to a probability value, the individuals may go through a crossover operation, where according to a certain function (e.g, single-point), are recombined and create two offsprings (line 8). Then, the two offspring have a certain probability to suffer a mutation, according to a mutation function and are joined together with the next population (lines 9-11). In the end, the fitness values of the individuals of the new population are computed and the archive is updated (lines 14-15). This archive saves the tests that have better coverage values and also taking into account their size (tests with fewer lines of code are preferred over large tests with the same coverage value).

MOSA/DynaMOSA

MOSA and DynaMOSA are many-objective algorithms (MOAs) and like other MOAs they use test cases as individuals. On the other hand, most GAs like Standard GA uses test suites as individuals. This happens because, unlike the Standard GA that tries to cover all goals at the same time, the MOAs aim to optimize each test case for a distinct coverage goal [10].

Algorithm 2 illustrates how DynaMOSA works. At the beginning of the search process, through the use of G, the algorithm perceives which coverage goals are free of control dependencies ($U*$) and the ones that can only be covered after satisfying other targets (line 1). Then it generates a random population of test cases, computes their fit-

Algorithm 1 Standard Genetic Algorithm as described in [10]

```
Input: Stopping condition $C$, Fitness function $\lambda$, Population size $p_s$, Selection function
    $s_f$, Crossover function $c_f$, Crossover probability $c_p$, Mutation function $m_f$, Mutation
    probability $m_p$
Output: Archive of optimised individuals $A$
 1: $P \leftarrow$ GENERATERANDOMPOPULATION($p_s$)
 2: PERFORMFITNESSEVALUATION($\lambda, P$)
 3: $A \leftarrow \{\ \}$
 4: while $\neg C$ do
 5:     $N_P \leftarrow \{\ \} \cup$ ELITISM($P$)
 6:     while $|N_P| < p_s$ do
 7:         $p_1, p_2 \leftarrow$ SELECTION($s_f, P$)
 8:         if $c_p$ then
 9:             $o_1, o_2 \leftarrow$ CROSSOVER($c_f, p_1, p_2$)
10:         else
11:             $o_1, o_2 \leftarrow p_1, p_2$
12:         end if
13:         MUTATION($m_f, m_p, o_1$)
14:         MUTATION($m_f, m_p, o_2$)
15:         $N_P \leftarrow N_P \cup \{o_1, o_2\}$
16:     end while
17:     $P \leftarrow N_P$
18:     PERFORMFITNESSEVALUATION($\lambda, P$)
19:     UPDATEARCHIVE($A, P$)
20: end while
21: return $A$
```

ness evaluation only considering the targets in $U*$ and updates the archive and the targets to be covered (lines 3-6). The last one serves as a way to include any uncovered target that is control dependent on newly covered targets. After this, the offspring is created using the genetic operations selection, crossover and mutation. Followed by their fitness evaluation using $U*$ (lines 8-11). Selection sorts the combination of parents and offspring, making use of both preference sorting criterion and non-dominance relation (line 14). The preference fitness sorting criterion assigns the best individuals for each non-covered target the highest rank (F_0) to give them a better chance to survive in the next generation (elitism). If the numbers of test cases that belong to F_0 are less than p_s then the remaining tests are ranked according to their non-dominance relation for U^*, else they are just put in F_1. Later, individuals are selected, starting from those with rank F_0 until the p_s is reached (lines 16-20). If the p_s is not reached, then tests of the last rank previously used (F_r) are selected according to crowding distance, in which those with higher distance values are chosen. This happens until p_s is reached (lines 21-22).

DynaMOSA is an extension of MOSA that was specially designed to handle test generation with coverage optimizations [12]. They are very similar algorithms, but there

Algorithm 2 DynaMOSA as described in [12]

```
Input: Stopping condition $C$, Fitness function $\lambda$, Population size $p_s$, Selection function $s_f$, Crossover function $c_f$, Crossover probability $c_p$, Mutation probability $m_p$, Control dependency graph $G$, Partial map between edges and targets $\phi$, Set of coverage targets $U$
Output: Archive of optimised individuals $A$
 1: $U^* \leftarrow$targets in $U$ with no control dependencies
 2: $A \leftarrow \{ \}$
 3: $P \leftarrow$ GenerateRandomPopulation($p_s$)
 4: PerformFitnessEvaluation($\lambda, P$)
 5: UpdateArchive($A, P$)
 6: UptateTargets($U^*, G, \phi$)
 7: while $\neg C$ do
 8:     $N_o \leftarrow$ GenerateOffspring($s_f, c_f, c_p, m_p, P$)
 9:     PerformFitnessEvaluation($\lambda, N_o$)
10:     UpdateArchive($A, N_o$)
11:     UptateTargets($U^*, G, \phi$)
12:     $R_t \leftarrow P \cup N_o$
13:     $r \leftarrow 0$
14:     $F_r \leftarrow$ PreferenceSorting($R_t, U^*$)
15:     $N_P \leftarrow \{ \}$
16:     while $|N_P| + |F_r| \leq p_s$ do
17:         CalculateCrowdingDistance($F_r, U^*$)
18:         $N_P \leftarrow N_P \cup F_r$
19:         $r \leftarrow r + 1$
20:     end while
21:     DistanceCrowdingSort($F_r$)
22:     $N_P \leftarrow N_P \cup F_r$ with size $p_s - |N_P|$
23: end while
24: return $A$
```

is one difference, DynaMOSA only considers the targets with no control dependencies. This gives DynaMOSA the edge in SBST since there can be structural dependencies between targets that should be considered when deciding which targets to optimize, e.g., in Figure 3.1 the branch in line 3 can only be reached after the branch in line 2 has been covered.

3.5 EvoSuite

EvoSuite is a state-of-the-art search-based tool that automatically generates executable test suites, i.e, a set of test cases for Java code [10, 17]. These test cases can be seen as a sequence of calls with a non-fixed length and that can have dependencies between statements. A statement can be a method call on an instantiated object, a definition of primitive variables (strings, integers, booleans, ...), among others. So, a dependency in this context can be something as simple as a definition of a string that will be used as a parameter in a future method call.

EvoSuite already includes implementations of several meta-heuristics algorithms, like the Standard GA and DynaMOSA, that were used in this thesis [10]. Along with these algorithms, it has the implementation of several techniques for each genetic operation used by the GAs (selection, crossover and mutation). The majority of these algorithms use test suites as a representation of individuals.

In the case of selection, EvoSuite supports several standard techniques, e.g., elitism selection and rank selection. However, due to the inconsistent size of the test suites and test cases (number of test cases and number of statements are variable, respectively), there can be problems with bloat [10, 17]. Bloat is a complex phenomenon in EAs, where the length of the individuals grows abnormally over time, which can lead to a larger execution time and/or consumption of all memory of the device that is running the test suites [20]. Therefore, as a way to counter bloat, after rank selection (standard selection technique of EvoSuite) is applied, if there is a draw in the first position, then the factor size (number of lines of code) is used to sort the best individuals and the smaller one is selected.

In EvoSuite, crossover consists of the exchange of test cases between test suites, i.e., amongst the individuals [10, 17]. On the other hand, mutation can be divided into two: adding or modifying tests cases to test suites and adding, removing or changing statements inside test cases [10, 17]. This last type of mutation requires special attention since we have to guarantee that the test cases are still valid after the mutation. For example, if a statement is removed, we have to guarantee that that are not pendant dependencies with that statement. A few examples of techniques already implemented in EvoSuite for these last two genetic operations are single-point crossover, mutation binomial e uniform.

Meta-heuristic algorithms rely on fitness functions and EvoSuite supports several of them. It also has implemented several optimizations to improve the performance of the algorithms. In Section 3.2 it is possible to see a few examples of this.

Another major aspect of EvoSuite is the fact that it has a modular architecture. This makes it easier to extend functionalities to satisfy our needs. In our case, we extended the algorithms' module and implemented ten new BIAs. EvoSuite is a black-box testing tool, meaning that it only needs the byte code (no source code required) to create and run the tests. It supports several plugins and can be used on the command line, Maven plugin, Jenkins (interacts with the Maven plugin), Eclipse and IntelliJ development environments [25, 26].

It is worth mentioning that EvoSuite has won seven out of eight editions of the annual competition of search-based software testing tools, only being bested on the 3rd edition where it ended up as the 2nd best tool [26].

3.6 Summary

In this chapter, we saw the explanation of how SBST works, the EIAs used for this study and a description of the testing tool EvoSuite. Next chapter, we will show a different kind of algorithms, the BIAs. We will explain how they work, present their implementation including the adaptions introduced that were necessary to our context and display their pseudo-code. Later on, the performance of all algorithms will be evaluated through several experiments.

Chapter 4

Bio-inspired Algorithms

In this chapter, we describe how the implemented BIAs work and show their pseudo-code. All BIAs implemented had to be adapted to the context of SBST, i.e., the formulas that could not be directly used had to be replaced with operations that represent the same process. The most common adaptations are the random movement from an individual in the search space turning into a mutation and the movement in a 2D/3D physical space in a straight line from an individual towards a certain target (food, best individual, etc) being replaced by a crossover operation between said individual and target. In our adaptions, we only consider one D-dimensional space since in SBST there are no physical coordinates to determine the position in the search space. Our algorithms consider test suites as individuals, except Elephant-DynaMOSA which use test cases.

4.1 Particle Swarm Optimization

PSO is inspired by the movements of social animal groups, particularly fish swarms and bird flocks [11, 27]. The individuals of PSO are called particles and their whole is called a swarm. Individuals can gain information in two ways: through a social part shared by all (global memory) and through the individual own experience within the swarm (local memory) [27].

PSO starts with the creation of a random population and its fitness evaluation (lines 1-2). Right after that, the local memory is initialized with the population values (line 3). To update the position of each particle in the swarm we do a crossover operation having both the local and global memories as a reference. After which the particle suffers a mutation (lines 8-15). Then, the local memory is updated, the new population is evaluated and finally, the archive is updated (lines 16-19).

Our adaption of PSO uses genetic operations to update the particle's positions, the local memory saves the best particle associated with a population index and the global memory is the best particle of the swarm. Originally, the update position phase uses a velocity variable to calculate the velocity of a particle in a certain dimension and then uses

Algorithm 3 PSO as described in [27]

```
Input: Stopping condition $C$, Fitness function $\lambda$, Population size $p_s$, Crossover function $c_f$, Mutation function $m_f$
Output: Archive of optimised individuals $A$
1: $P \leftarrow$ GENERATERANDOMPOPULATION($p_s$)
2: PERFORMFITNESSEVALUATION($\lambda, P$)
3: $lm \leftarrow$ INITIALIZELOCALMEMORY($P$)
4: $A \leftarrow \{\ \}$
5: while $\neg C$ do
6:     $N_P \leftarrow \{\ \} \cup$ ELITISM($P$)
7:     $p_b \leftarrow$ PARTICLE WITH THE BEST FITNESS VALUE
8:     for $i \leftarrow 0, p_s$ do                              ▷ 1-Update position phase
9:         if $P[i] \neq lm[i]$ then
10:            $p \leftarrow$ CROSSOVER($c_f, P[i], lm[i]$)
11:        end if
12:        $p \leftarrow$ CROSSOVER($c_f, p, p_b$)
13:        MUTATION($m_f, p$)
14:        $N_P \leftarrow N_P \cup \{p\}$
15:    end for
16:    UPDATELOCALMEMORY($N_P, lm$)                       ▷ 2-Update local memory phase
17:    $P \leftarrow N_P$
18:    PERFORMFITNESSEVALUATION($\lambda, P$)
19:    UPTADEARCHIVE($A, P$)
20: end while
21: return $A$
```

it to change the current particle's position. This velocity depends on several coefficients, random numbers and uses the local memory and global memory as references [27].

4.2 Genetic Bee Colony Algorithm

GBC (Algorithm 4) is a meta-heuristic algorithm that combines crossover and mutation operations of GAs with the artificial bee colony algorithm [13]. In this algorithm, the individuals are represented as food sources while the bees are the variables and operations that run the GBC itself. The GBC is divided into three main phases: employer, onlooker and scout bees phase. GBC begins with the generation of a population of random individuals and their fitness evaluation (lines 1-2). Then comes the employer bees phase where each bee is assigned a food source of the population and tries to discover a new food source better than the current one (lines 6-9). This process (Algorithm 1) consists of selecting two random neighbours (other population members) and using them as a reference for a crossover operation with the individual to create children. Then, these children suffer a mutation to create grandchildren. In the end, we compare the food source, two children and two grandchildren and select the best solution (the one with

Algorithm 4 Genetic Bee Colony Algorithm as described in [13]

```
Input: Stopping condition $C$, Fitness function $\lambda$, Population size $p_s$, Selection function
    $s_f$, Crossover function $c_f$, Mutation function $m_f$, Limit $L$, Onlooker Rate $o_r$, Number
    of Scouts $n_s$
Output: Archive of optimised individuals $A$
 1: $P \leftarrow$ GENERATERANDOMPOPULATION($p_s$)
 2: PERFORMFITNESSEVALUATION($\lambda, P$)
 3: $A \leftarrow \{ \}$
 4: while $\neg C$ do
 5:     $N_P \leftarrow \{ \}$
 6:     for all $food \in P$ do                                  ▷ 1-Employee bee phase
 7:         $food \leftarrow$DISCOVERNEWFOOD($P, \lambda, c_f, m_f, food$)
 8:         $N_P \leftarrow N_P \cup \{food\}$
 9:     end for
10:     $N_{P_s} \leftarrow N_P$ SIZE
11:     for $j \leftarrow 0, N_{P_s} * o_r$ do                   ▷ 2-Onlooker bee phase
12:         $food \leftarrow$ SELECTION($s_f, N_P$)
13:         $food_2 \leftarrow$DISCOVERNEWFOOD($N_P, \lambda, c_f, m_f, food$)
14:         if $food \neq food_2$ then
15:             $N_P \leftarrow N_P \setminus \{food\}$
16:             $N_P \leftarrow N_P \cup \{food_2\}$
17:         end if
18:     end for
19:     $scouts \leftarrow 0$
20:     $food_{age} \leftarrow$VALUE OF $p$ WHEN THE $food$ WAS CREATED
21:     for all $food \in N_P$ do                                ▷ 3-Scout bee phase
22:         if $p - food_{age} > L$ then
23:             $N_P \leftarrow N_P \setminus \{food\}$
24:             $food \leftarrow$GENERATERANDOMINDIVIDUAL()
25:             PERFORMFITNESSEVALUATION($\lambda, food$)
26:             $N_P \leftarrow N_P \cup \{food\}$
27:             $scouts \leftarrow scouts + 1$
28:             if $scouts \geq n_s$ then
29:                 end for
30:             end if
31:         end if
32:     end for
33:     $P \leftarrow N_P$
34:     UPTADEARCHIVE($A, P$)
35: end while
36: return $A$
```

the best fitness value). The next step of the algorithm is the onlooker bees phase which is similar to the employer phase (lines 11-18). The differences are that the number of onlooker bees is a percentage of the population and the individuals that are explored are

Method 1 Discover new food source in GBA as described in [13]

```
Input: Population $P$, Fitness function $\lambda$, Crossover function $c_f$, Mutation function $m_f$,
    Individual $food$
Output: Individual $food_{new}$
1: $food_{r1} \leftarrow$ GetRandomNeighbour($P$)    ▷ 1-Select Random Neighbours
2: $food_{r2} \leftarrow$ GetRandomNeighbour($P$)
3: $food_{c1} \leftarrow$ Crossover($c_f, food, food_{r1}$)    ▷ 2-Generate Children
4: $food_{c2} \leftarrow$ Crossover($c_f, food, food_{r2}$)
5: $food_{gc1} \leftarrow$ Mutation($m_f, food_{c1}$)    ▷ 3-Generate Grandchildren
6: $food_{gc2} \leftarrow$ Mutation($m_f, food_{c2}$)
7: $food_{new} \leftarrow$GetBestIndividual($\lambda, food, food_{c1}, food_{c2}, food_{gc1}, food_{gc2}$)    ▷
    4-Choose Best Individual
8: return $food_{new}$
```

selected according to a selection function. The scout bees phase consists of discarding the individuals that did not get updated a certain number of generations and generate new individuals to replace them (lines 21-32). Lastly, the GBC ends with the update of the archive (line 34).

The number of bees in bee algorithms is variable [28], so we decided to have a fixed number of employees equal to the population size to give each individual a chance to improve in each iteration, while the number of onlookers and scouts can vary depending on the emphasis we want to give to exploitation and exploration respectively.

4.3 Cat Swarm Optimization

Cats are animals that spend most of their time resting and being attentive to their surroundings. They also have moments where they are more active, e.g, hunting preys. CSO (Algorithm 5) uses these behaviours as its inspiration.

CSO starts with the same procedures as the previous algorithms (lines 1-2). Then, depending on a random value and the crossover probability it can have one of two behaviours: tracing or seeking phase. The tracing phase simulates the behaviour of cats when they are more active, so the cat will try to move towards its target, which in this case it is a crossover operation with the best solution in the swarm (lines 9-10). The seeking phase represents the behaviour of cats when they are saving energy in which they can move to better positions. This phase consists of creating a list of clones of the cat, their mutation and selecting one of them for the next generation. This list can include the current cat or only its mutated clones (whether the self-position consideration is true or false) and its size depends on the seeking memory pool (lines 11-16). After the new population is filled, it goes through a fitness evaluation and the archive is updated (lines 19-21).

By origin, CSO's tracing phase has a similar equation to the update position of Algorithm 3. The difference lies in the fact that in CSO the velocity variable has only 1

Algorithm 5 Cat Swarm Optimization as described in [13, 29]

```
Input: Stopping condition $C$, Fitness function $\lambda$, Population size $p_s$, Mixture ratio $mr$,
    Seeking memory pool $smp$, Self position consideration $spc$, Selection function $S_f$,
    Crossover function $c_f$, Crossover probability $c_p$, Mutation function $m_f$
Output: Archive of optimised individuals $A$
 1: $P \leftarrow$ GENERATERANDOMPOPULATION($p_s$)
 2: PERFORMFITNESSEVALUATION($\lambda, P$)
 3: $A \leftarrow \{ \}$
 4: while $\neg C$ do
 5:     $cat_b \leftarrow$ CAT WITH THE BEST FITNESS VALUE
 6:     $N_P \leftarrow \{ \} \cup$ ELITISM($P$)
 7:     for all $cat \in P$ do
 8:         $r1 \leftarrow$ RANDOM VALUE [0,1]
 9:         if $r1 < c_p$ then                                  ▷ 1-Tracing phase
10:             $cat_{new} \leftarrow$ CROSSOVER($c_f, cat, cat_b$)
11:         else                                                ▷ 2-Seeking phase
12:             $list \leftarrow$ CREATELISTOFCLONES($spc, smp, cat$)
13:             MUTATION($m_f, list$)
14:             PERFORMFITNESSEVALUATION($\lambda, list$)
15:             $cat_{new} \leftarrow$ SELECTION($S_f, list$)
16:         end if
17:         $N_P \leftarrow N_P \cup \{cat_{new}\}$
18:     end for
19:     $P \leftarrow N_P$
20:     PERFORMFITNESSEVALUATION($\lambda, P$)
21:     UPTADEARCHIVE($A, P$)
22: end while
23: return $A$
```

coefficient, one random number and only considers the best solution in the swarm as a reference. As for the seeking phase in CSO, we replaced the random alterations to the copies of the cat for a mutation operation.

4.4 Whale Optimization Algorithm

WOA (Algorithm 6) is inspired by the hunting method of the humpback whales and consists of three different behaviours: encircle the prey using the shrinking encircling mechanism or the spiral updating position and search for food.

In WOA, in every iteration, each whale can do one of the three phases depending on the calculated variables (line 8) and the inputs given. In the shrinking encircling mechanism, the whale tries to move towards the best solution since it assumes it is close to the optimum. This is done with a crossover operation with the best whale (line 11). The search for food behaviour explores the search space using a random member of the pop-

Algorithm 6 Whale Optimization Algorithm as described in [13, 30]

```
Input: Stopping condition $C$, Fitness function $\lambda$, Population size $p_s$, Crossover function $c_f$, Crossover probability $c_p$, Mutation function $m_f$, Mutation probability $m_p$, Shinrinking probability $s_p$
Output: Archive of optimised individuals $A$
 1: $P \leftarrow$ GENERATERANDOMPOPULATION($p_s$)
 2: PERFORMFITNESSEVALUATION($\lambda, P$)
 3: $A \leftarrow \{ \}$
 4: while $\neg C$ do
 5:     $whale_b \leftarrow$ WHALE WITH THE BEST FITNESS VALUE
 6:     $N_P \leftarrow \{ \} \cup$ ELITISM($P$)
 7:     for all $whale \in P$ do
 8:         $v_1, v_2, r_1 \leftarrow$ CALCULATEBEHAVIORALVARIABLES($C$)
 9:         if $r_1 < c_p$ then
10:             if $|v_1| < 2 * s_p$ then                ▷ 1a-Shrinking Encircling Mechanism
11:                 $whale_{new} \leftarrow$ CROSSOVER($c_f, whale, whale_b$)
12:             else                                      ▷ 1b-Search For Prey
13:                 $whale_r \leftarrow$ RANDOM WHALE OF $P$
14:                 $whale_{new} \leftarrow$ CROSSOVER($c_f, whale, whale_r$)
15:             end if
16:             if $v_2 \leq 2 * m_p$ then
17:                 $whale_{new} \leftarrow$ MUTATION($m_f, whale_{new}$)
18:             end if
19:         else                                          ▷ 2-Spiral Updating Position
20:             $whale_{new} \leftarrow$ MUTATION($m_f, whale$)
21:         end if
22:         $N_P \leftarrow N_P \cup \{whale_{new}\}$
23:     end for
24:     $P \leftarrow N_P$
25:     PERFORMFITNESSEVALUATION($\lambda, P$)
26:     UPTADEARCHIVE($A, P$)
27: end while
28: return $A$
```

ulation as a reference, hence it is represented with a crossover operation with a random whale from the population (line 13-14). Both these phases can also suffer a mutation to add more randomness to the behaviours (line 16-17). Finally, there is the spiral updating position phase. Originally in this phase, the whale also tries to move towards the best solution, but it simulates a spiral movement instead of a straight line. Since we don't have a way to move individuals in a spiral movement in the context of SBST and we used a crossover operation and a possible mutation in the shrinking encircling mechanism phase, we decided to represent the spiral updating position with a mutation (line 20).

Algorithm 7 Moth-flame Optimization Algorithm as described in [13, 31]

```
Input: Stopping condition $C$, Fitness function $\lambda$, Population size $p_s$, Crossover function
    $c_f$, Mutation function $m_f$
Output: Archive of optimised individuals $A$
 1: $P \leftarrow$ GENERATERANDOMPOPULATION($p_s, d_s$)
 2: PERFORMFITNESSEVALUATION($\lambda, P$)
 3: $A \leftarrow \{ \}$
 4: while $\neg C$ do
 5:     $N_P \leftarrow \{ \}$
 6:     $F \leftarrow$ UPDATEFLAMES($P, C$)
 7:     SORTFLAMES($F$)
 8:     for all $moth \in P$ do                                  ▷ 1-UpdatePosition
 9:         $v_1 \leftarrow$ CALCULATEBEHAVIORALVARIABLE($C$)
10:         if $v_1 < 0$ then                              ▷ 1a-Exploitation behaviour
11:             $flame \leftarrow$ GETFLAME($F, p_s$)
12:             $moth_{new} \leftarrow$ CROSSOVER($c_f, moth, flame$)
13:         else                                           ▷ 1b-Exploration behaviour
14:             $moth_{new} \leftarrow$ MUTATION($m_f, moth$)
15:         end if
16:         $N_P \leftarrow N_P \cup \{moth_{new}\}$
17:     end for
18:     $P \leftarrow N_P$
19:     PERFORMFITNESSEVALUATION($\lambda, P$)
20:     UPTADEARCHIVE($A, P$)
21: end while
22: return $A$
```

4.5 Moth-flame Optimization Algorithm

MFO takes the spiral movements of the moths around a source of light as inspiration. MFO has more than one set of solutions: moths and flames. The moths are the “standard” individuals while the flames are the best individuals found so far.

Algorithm 7 begins with the generation of the population of moths and their fitness evaluation (lines 1-2). Then, the number of flames is updated according to how close the execution is to the end (line 6). Initially, the number of flames is the same as the number of months to promote exploration and avoid a local optimum. As the execution advances, the number of flames diminishes accordingly, to promote the exploitation of the best solutions found during the execution. Following that, each moth in the population can go through an exploitation or exploration behaviour to update its position. The former consists of a crossover with a flame (lines 11-12), while the latter is a mutation (line 14). Lastly, the new population of moths is evaluated according to the fitness functions and the archive is updated (lines 19-20).

In the original MFO, the update position is an equation that simulates a spiral move-

Algorithm 8 Grey Wolf Optimization Algorithm as described in [13, 32]

```
Input: Stopping condition $C$, Fitness function $\lambda$, Population size $p_s$, Crossover function
    $c_f$, Crossover probability $c_p$, Mutation function $m_f$, Mutation probability $m_p$
Output: Archive of optimised individuals $A$
 1: $P \leftarrow$ GenerateRandomPopulation($p_s$)
 2: PerformFitnessEvaluation($\lambda, P$)
 3: $A \leftarrow \{ \}$
 4: while $\neg C$ do
 5:     $N_P \leftarrow \{ \} \cup$ Elitism($P$)
 6:     $wolf_\alpha \leftarrow$ best solution $\in P$
 7:     $wolf_\beta \leftarrow$ 2ND best solution $\in P$
 8:     $wolf_\delta \leftarrow$ 3RD best solution $\in P$
 9:     for all $wolf \in P$ do                                  ▷ 1-Hunting behaviour
10:         $v_1, v_2 \leftarrow$ CalculateBehavioralVariables($C$)
11:         if $v_1 < 2 * c_p$ then                     ▷ 1.1-Attacking Prey - exploitation
12:             $wolf_{new} \leftarrow$ Crossover($c_f, wolf, wolf_\alpha, wolf_\beta, wolf_\delta$)
13:         end if
14:         if $v_1 \geq 2 * c_p \vee v_2 \leq 2 * m_p$ then      ▷ 1.2-Search for Prey - exploration
15:             $wolf_{new} \leftarrow$ Mutation($m_f, wolf_{new}$)
16:         end if
17:         $N_P \leftarrow N_P \cup \{wolf_{new}\}$
18:     end for
19:     $P \leftarrow N_P$
20:     PerformFitnessEvaluation($\lambda, P$)
21:     UptadeArchive($A, P$)
22: end while
23: return $A$
```

ment of a moth around a flame. In this case, the exploitation and exploration are both considered part of this equation depending on whether the final position of the moth is between the initial position and the flame or if it is "around" the flame.

4.6 Grey Wolf Optimization Algorithm

GWO algorithm (Algorithm 8) is inspired by the social behaviour of the grey wolves, more specifically the hunting of these apex predators. GWO imitates the behaviour of real grey wolves by assigning a hierarchy to the pack: the alpha is the best solution, the beta is the 2nd best solution, the delta is the 3rd best solution and all the other wolves are the omegas (lines 6-8). As for the hunting behaviour of the wolves, it is influenced by several variables (line 10) and it is composed of two parts: attack or search for the prey. When attacking the prey, real-life wolf packs are guided by the alpha and the beta and the delta can also participate. When adapting this to an algorithm, these three wolves are used as a reference for the pack to move towards the prey since it is assumed they

Algorithm 9 Artificial algae algorithm as described in [33]

Input: Stopping condition C, Fitness function λ, Population size p_s, Selection function s_f, Adaptation probability a_p, Max amount of energy E_{max}, Loss of energy constant l_e, Crossover function c_f

Output: Archive of optimised individuals A

```
1: $P \leftarrow$ GenerateRandomPopulation($p_s, d_s$)
2: $A \leftarrow \{ \}$
3: while $\neg C$ do
4:     $N_P \leftarrow \{ \} \cup$ Elitism($P$)
5:     for all $algae \in P$ do
6:         $E \leftarrow$ CalculateEnergy($E_{max}, algae$)
7:         while $E > 0 \wedge \neg C$ do                    ▷ 1-Helical Movement phase
8:             $algae_s \leftarrow$ Selection($s_f, P$)
9:             $algae_{new} \leftarrow$ Crossover($c_f, algae, algae_s$)
10:            LossOfEnergy($E, l_e$)
11:            PerformFitnessEvaluation($algae_{new}, \lambda$)
12:            if $algae_{new}$ is better than $algae$ then
13:                $algae \leftarrow algae_{new}$
14:            else
15:                LossOfEnergy($E, l_e$)
16:            end if
17:        end while
18:        $N_P \leftarrow N_P \cup \{algae\}$
19:    end for
20:    $algae_{best} \leftarrow$ best solution $\in N_P$
21:    $algae_{worst} \leftarrow$ worst solution $\in N_P$
22:    Reproduction($algae_{best}, algae_{worst}$)                ▷ 2-Reproduction phase
23:    $r_1 \leftarrow$ random number [0,1]
24:    if $r_1 < a_p$ then                                      ▷ 3-Adaptation phase
25:        $algae_{best} \leftarrow$ best solution $\in N_P$
26:        $algae_{oldest} \leftarrow$ oldest solution $\in N_P$
27:        $N_P \leftarrow N_P \setminus \{algae_{oldest}\}$
28:        $algae_{new} \leftarrow$ Crossover($c_f, algae_{best}, algae_{oldest}$)
29:        $N_P \leftarrow N_P \cup \{algae_{oldest}\}$
30:    end if
31:    $P \leftarrow N_P$
32:    PerformFitnessEvaluation($\lambda, P$)
33:    UptadeArchive($A, P$)
34: end while
35: return $A$
```

know where the food is. We adapted this to a crossover operation between the wolf and the three references (line 12). However, during the pursuit of prey, several obstacles can appear (rocks, trees, ...) and to simulate these random hurdles it is also possible for the attacking wolf to suffer a mutation operation (line 15). Searching for prey represents the real-life behaviour of wolves in which they disperse to find prey and it also is depicted as a mutation (line 15).

The equations we replaced with the crossover and mutation operations have the purpose of exploiting the best solutions and explore the search space and avoid local optimum respectively.

4.7 Artificial algae algorithm

AAA (Algorithm 9) uses the characteristics of real algae to find optimal solutions in the search space. AAA focus on three algae behaviours: movement, evolutionary process and adaption. Each iteration of the algorithm goes through these three phases in this order. The algae can develop better the closer to the water surface they are, which gives them more energy to move around. In the movement phase (lines 5-19), each alga is assigned an energy value according to its fitness value. Then, while the said energy is above zero, the alga goes through a crossover with one individual of the population previously selected (target to move towards), loses energy and saves the best solution to the new population. Following that, there is the evolutionary/reproduction process that consists of replacing a random test case of the worst individual with the copy of a random test case from the best individual (lines 20-22). This represents the death of a portion of the cells of the worst colony and the reproduction of the biggest colony. Finally, the adaption phase replaces the oldest member of the population with the individual that we get from a crossover between the best and oldest algae (lines 24-30). It simulates the algae behaviour in which they copy the fittest individuals to better adapt to the environment.

Our adaption of the AAA has several changes from the original algorithm: the equations that decide the helical movement in a 3D space and the variables used are simplified to a single crossover operation, in the reproduction phase we select a random test case from each individual that represent the algae cells of the algae colonies and the adaptation phase equation was replaced by a crossover operation.

4.8 Chicken Swarm Optimization Algorithm

CSOA (Algorithm 10) uses chickens as its inspiration, more specifically their social and gather food behaviours. In CSOA, the population is divided into different groups and hierarchical ranks (lines 5-8): the roosters (best solutions), the chicks (worst solutions), the hens and the mother-hens (randomly selected among the hens). Each group is composed of one rooster, a random number of hens and if at least one hen is a mother-hen her chicks. After that, the CSOA simulates the behaviour of the chicken swarms in searching for food: in each group, the hens will follow the rooster and compete with other hens, hence the crossover operation using the rooster and another random hen from the group as

Algorithm 10 Chicken Swarm Optimization as described in [34]

```
Input: Stopping condition $C$, Fitness function $\lambda$, Population size $p_s$, Number of roosters
    $N_r$, Number of chicks $N_c$, Number of mother hens $N_m$, Groups update constant $g_c$,
    Crossover function $c_f$, Mutation function $m_f$
Output: Archive of optimised individuals $A$
 1: $P \leftarrow$ GENERATERANDOMPOPULATION($p_s$)
 2: PERFORMFITNESSEVALUATION($\lambda, P$)
 3: $A \leftarrow \{\ \}$
 4: while $\neg C$ do
 5:     if $p\%g_c == 0$ then                                    ▷ 1-Update Groups Hierarchy
 6:         $G \leftarrow$UPDATEGROUPS($P, N_r, N_h, N_c, N_m$)
 7:         $G \leftarrow$ GROUPS OF CHICKS, HENS AND ROOSTERS
 8:     end if
 9:     $N_P \leftarrow \{\ \}$
10:     for all $g \in G$ do
11:         $N_g \leftarrow \{\ \}$
12:         $rooster \leftarrow$ BEST SOLUTION $\in g$
13:         for all $chicken \in g \wedge chicken \neq rooster$ do
14:             if CHICKEN IS HEN then                           ▷ 2-Update Hen Position
15:                 $hen_{new} \leftarrow$ CROSSOVER($c_f, hen, rooster$)
16:                 $hen_r \leftarrow$ RANDOM HEN, $hen_r \in g \wedge (hen_r \neq rooster \wedge hen_r \neq hen)$
17:                 $hen_{new} \leftarrow$ CROSSOVER($c_f, hen_{new}, hen_r$)
18:                 $N_g \leftarrow N_g \cup \{hen_{new}\}$
19:             else                                             ▷ 3-Update Chick Position
20:                 $hen_m \leftarrow$ MOTHER OF CHICK
21:                 $chick_{new} \leftarrow$ CROSSOVER($c_f, chick, hen_m$)
22:                 $N_g \leftarrow N_g \cup \{chick_{new}\}$
23:             end if
24:         end for
25:         $rooster \leftarrow$ MUTATION($m_f, rooster$)                ▷ 4-Update Rooster Position
26:         $N_g \leftarrow N_g \cup \{rooster\}$
27:         $N_P \leftarrow N_P \cup N_g$
28:     end for
29:     $G \leftarrow N_P$
30:     $P \leftarrow N_P$
31:     PERFORMFITNESSEVALUATION($\lambda, P$)
32:     UPTADEARCHIVE($A, P$)
33: end while
34: return $A$
```

references (line 14-18). The chicks will follow their mother on their quest for food, which is expressed as a crossover using the mother as a reference (lines 19-23). The movement of the rooster is represented by a mutation (line 25). Finally, CSOA ends with the creation of the new population with the new roosters, hens and chicks, evaluating their fitness and updating the archive (lines 29-32).

Algorithm 11 Elephant Herding Optimization as described in [13, 35]

```
Input: Stopping condition $C$, Fitness function $\lambda$, Population size $p_s$, Number of clans
    $N_{clan}$, Number of male per clan $N_{male}$, Crossover function $c_f$, Mutation function $m_f$,
    Select from Archive $S_A$
Output: Archive of optimised individuals $A$
 1: $P \leftarrow$ GENERATERANDOMPOPULATION($p_s$)
 2: $C_{clans} \leftarrow$ GENERATECLANS($P, N_{clan}$)
 3: PERFORMFITNESSEVALUATION($\lambda, P$)
 4: $A \leftarrow \{\,\}$
 5: while $\neg C$ do
 6:     $N_P \leftarrow \{\,\}$
 7:     $N_{CLANS} \leftarrow \{\,\}$
 8:     for all $clan \in C_{clans}$ do
 9:         $N_{clan} \leftarrow \{\,\}$
10:         $ele_{best} \leftarrow$ BEST SOLUTION $\in clan$
11:         for all $ele \in clan \wedge ele \neq ele_{best}$ do        ▷ 1.1-Update Elephant Position
12:             $ele_{new} \leftarrow$ CROSSOVER($c_f, ele, ele_{best}$)
13:             $N_{clan} \leftarrow N_{clan} \cup \{ele_{new}\}$
14:         end for
15:         $ele_{best} \leftarrow$ MUTATION($m_f, ele_{best}$)        ▷ 1.2-Update Matriarch Position
16:         $N_{clan} \leftarrow N_{clan} \cup \{ele_{best}\}$
17:         $N_{CLANS} \leftarrow N_{CLANS} \cup N_{clan}$
18:     end for
19:     for all $clan \in N_{CLANS}$ do        ▷ 2-Male Separation phase
20:         $male \leftarrow 0$
21:         while $male < N_{male}$ do
22:             $ele_{male} \leftarrow$ WORST SOLUTION $\in clan$
23:             $clan \leftarrow clan \setminus \{ele_{male}\}$
24:             $ele \leftarrow$GENERATERANDOMINDIVIDUAL($S_A, A$)
25:             $clan \leftarrow clan \cup \{ele\}$
26:             $male \leftarrow male + 1$
27:         end while
28:     end for
29:     $N_P \leftarrow N_P \cup N_{CLANS}$
30:     $P \leftarrow N_P$
31:     PERFORMFITNESSEVALUATION($\lambda, P$)
32:     UPTADEARCHIVE($A, P$)
33: end while
34: return $A$
```

In the original CSOA, the roosters are divided into two groups and their equations depend on a Gaussian distribution, while the equations of the movement of the hens and chicks are similar to the equations of the previous algorithms that use a reference to move. The reason we only use one type of rooster is that we assign one rooster per group, so there is not a need to differentiate between the roosters.

Algorithm 12 Fish Swarm Algorithm as described in [13, 36, 37]

```
Input: Stopping condition $C$, Fitness function $\lambda$, Population size $p_s$, Neighbour range $n_r$, Fish concentration $\gamma$, Number of attempts $N_{at}$, Crossover function $c_f$, Mutation function $m_f$
Output: Archive of optimised individuals $A$
 1: $P \leftarrow$ GenerateRandomPopulation($p_s$)
 2: PerformFitnessEvaluation($\lambda, P$)
 3: $A \leftarrow \{\ \}$
 4: while $\neg C$ do
 5:     $N_P \leftarrow \{\ \} \cup$ Elitism($P$)
 6:     for all $fish \in P$ do
 7:         $nei \leftarrow$ CreateNeighbourhood($P, fish, n_r, \lambda$)
 8:         $fish_1 \leftarrow$ SwarmPhase($P, fish, nei, \gamma, N_{at}, c_f, m_f$)    ▷ 1-Swarm behavior phase
 9:         $fish_2 \leftarrow$ FollowPhase($P, fish, nei, \gamma, N_{at}, c_f, m_f$)    ▷ 2-Following behavior phase
10:         $fish_{new} \leftarrow$ ChooseBestFish($fish_1, fish_2, \lambda$)    ▷ 3-Best behaviour phase
11:         $N_P \leftarrow N_P \cup \{fish_new\}$
12:     end for
13:     $P \leftarrow N_P$
14:     PerformFitnessEvaluation($\lambda, P$)
15:     UpdateArchive($A, P$)
16: end while
17: return $A$
```

4.9 Elephant Herding Optimization

EHO (Algorithm 11) uses elephant clans, how they are organised and move, as its inspiration. EHO begins with the generation of a random population of elephants and their fitness evaluation (lines 1-3). Like real elephants, this population is organised in clans that are lead by a matriarch (the best solution in the clan). Each clan will move towards their matriarch through a crossover (line 11-14), while her movement is expressed by a mutation (line 15). Then, comes the separation phase where the male elephants (worst solutions in each clan) leave the clan (lines 19-28). This phase replaces the males with new individuals that can be from the archive or newly generated. Lastly, all the new clans are joined together to create the new population, which then goes through a fitness evaluation and the archive is updated (lines 29-32).

In this algorithm, we use the crossover operation between the elephants and the matriarch the same way we did with movement towards a target in other algorithms. The mutation of the matriarch however is different, originally it moves toward the central position of the clan, but we do not have that concept. Finally, the separating equation that represents the males leaving the clans after reaching puberty was replaced with the replacement of a new individual.

Method 2 Swarm and Follow phases of FSA as described in [13, 36, 37]

```
Input: Population $P$, Individual $fish$, Neighbourhood $nei$, Fish concentration $\gamma$, Number of attempts $N_{at}$, Crossover function $c_f$, Mutation function $m_f$
Output: Individual $fish_{new}$
 1: $fish_{new} \leftarrow \{ \}$
 2: if $nei = \{ \}$ then                                        ▷ 1-Swarm/Follow phase
 3:     $fish_n \leftarrow$GETNEIGHBOUR$(P, nei)$
 4:     $\lambda_{fish_n} \leftarrow$FITNESS OF $fish_n$
 5:     $nei_s \leftarrow$ NEIGHBOURHOOD SIZE
 6:     if $\lambda_{fish_n}/nei_s < \lambda_{fish_n} * \gamma$ then
 7:         $fish_{new} \leftarrow$ CROSSOVER$(c_f, fish, fish_n)$
 8:     end if
 9: end if
10: if $fish_{new} = \{ \}$ then
11:     if $nei = \{ \}$ then                                    ▷ 2-Prey phase
12:         for $i \leftarrow 0, N_{at}$ do
13:             $fish_r \leftarrow$GETRANDOMNEIGHBOUR$(P, nei)$
14:             if ChooseBestFish$(fish, fish_r, \lambda) = fish$ then
15:                 $fish_{new} \leftarrow$ CROSSOVER$(c_f, fish, fish_r)$
16:                 end for
17:             end if
18:         end for
19:     end if
20:     $fish_{new} \leftarrow$ MUTATION$(m_f, fish)$            ▷ 3-Random phase
21: end if
22: return $fish_{new}$
```

4.10 Fish Swarm Algorithm

FSA (Algorithm 12) is inspired by the pattern behaviours of fish swarms, such as forming swarms to avoid danger and search for food. For each fish in the population, we create a neighbourhood according to its fitness value and the range in the input (line 7). We decided that the neighbours are those fishes that are closer to the individual from a fitness value perspective since in our context we do not have a physical space to check which fishes are physically closer to each other. Then, we generate two fishes from the original fish, one is from the swarming phase while the other is from the following phase (lines 8-9). These phases are very similar to each other, depending on the parameters, we will have a crossover (with the best neighbour in the following behaviour and with the middle neighbour in the swarming one) or go through the prey phase. The prey phase consists of selecting a random neighbour and if the said neighbour is better then the fish do a crossover with it. A certain number of attempts can be done at this stage. If no crossover is done, the fish goes through the random phase where it does a random movement which we represented with a mutation. The two fishes generated end up in the best behaviour

Algorithm 13 Elephant-DynaMOSA - fusion between EHO and DynaMOSA

Input: Stopping condition C, Fitness function λ, Population size p_s, Crossover function c_f, Crossover probability c_p, Mutation probability m_p, Control dependency graph G, Partial map between edges and targets ϕ, Set of coverage targets U, Number of clans N_{clan}, Number of male per clan N_{male}, Select from Archive S_A

Output: Archive of optimised individuals A

```
 1: $U^* \leftarrow$targets in $U$ with no control dependencies
 2: $P \leftarrow$ GenerateRandomPopulation($p_s$)
 3: PerformFitnessEvaluation($\lambda, P$)
 4: $C_{clans} \leftarrow$ GenerateClans($P, N_{clan}$)
 5: $A \leftarrow \{\ \}$
 6: while $\neg C$ do
 7:     $N_{CLANS} \leftarrow \{\ \}$
 8:     $N_{p+1} \leftarrow \{\ \}$
 9:     for all $clan \in C_{clans}$ do
10:         $N_o \leftarrow$ GenerateOffspring($c_f, c_p, m_p, clan$)
11:         PerformFitnessEvaluation($\lambda, N_o$)
12:         $R_t \leftarrow clan \cup N_o$
13:         $r \leftarrow 0$
14:         $F_r \leftarrow$ PreferenceSorting($R_t, U^*$)
15:         $N_{clan} \leftarrow \{\ \}$
16:         while $|N_{clan}| + |F_r| \leq p_s$ do
17:             CalculateCrowdingDistance($F_r, U^*$)
18:             $N_{clan} \leftarrow N_{clan} \cup F_r$
19:             $r \leftarrow r + 1$
20:         end while
21:         DistanceCrowdingSort($F_r$)
22:         $N_{clan} \leftarrow N_{clan} \cup F_r$ with size $p_s - |N_{clan}|$
23:         $male \leftarrow 0$
24:         while $male < N_{male}$ do                       ▷ Male Separation phase
25:             $ele_{male} \leftarrow$ WORST SOLUTION $\in N_{clan}$
26:             $N_{clan} \leftarrow N_{clan} \setminus \{ele_{male}\}$
27:             $ele \leftarrow$GenerateRandomIndividual($S_A, A$)
28:             $N_{clan} \leftarrow N_{clan} \cup \{ele\}$
29:             $male \leftarrow male + 1$
30:         end while
31:         $N_{CLANS} \leftarrow N_{CLANS} \cup N_{clan}$
32:     end for
33:     $N_{p+1} \leftarrow N_{p+1} \cup N_{CLANS}$
34:     UpdateArchive($A, N_{p+1}$)
35:     UptateTargets($U^*, G, \phi$)
36: end while
37: return $A$
```

phase, where the best one is selected for the new population (lines 10-11).

4.11 Elephant-DynaMOSA

As we are going to see in chapter Chapter 5, EHO was the BIA that stood out the most and showed the greatest potential of becoming the new state-of-the-art. So, we decided to explore his properties in more detail and see if his performance would allow it to become the best algorithm. However, DynaMOSA still remained as the state-of-the-art. This made us think of the possibility of fusing both algorithms. Not only we thought it would be interesting, but we also concluded it would be relevant to create Elephant-DynaMOSA. This hybrid combines elements of swarm-based search with many-objective optimization and has the potential of becoming the new state-of-the-art algorithm.

This new algorithm uses DynaMOSA (Algorithm 2) and introduces a few concepts from EHO (Algorithm 11). The differences from the DynaMOSA are that the algorithm works with clans instead of the whole population. In the generate offspring phase (line 10) the concept of the matriarch is introduced and we removed the step where newly generated test cases are added to the offspring and the male separation phase from EHO was added (lines 24-32).

4.12 Summary

After learning about the BIAs implemented in this chapter we will go on to the next chapter to see how the experiments and evaluation were done. With that, we can analyse the performance of the algorithms, allowing us to rank them accordingly. Will the state-of-the-art be updated?

Chapter 5

Empirical Study

In this chapter, we first describe the experiments we performed to evaluate the performance of the algorithms described in Section 5.2 and analyse the results obtained.

5.1 Research Questions

In this section, we list the RQs that will be answered in this chapter by the analysis of our results.

RQ1: Which bio-inspired algorithm performs best?

RQ2: How does swarm-based search compare to traditional evolutionary search?

RQ3: How does swarm-based search compare to many-objective optimization algorithms?

RQ4: How does a hybrid that combines swarm-based search with many-objective optimization performs?

5.2 Experimental Setup

5.2.1 Classes Under Test

The selection of classes under test is one of the most important aspects to evaluate techniques that aim to automatically generate tests for. The reason is that, on one hand, if the classes under test are too simple then all algorithms should have a great performance. On the other hand, if classes are too complicated then all algorithms are expected to perform badly. Both these cases make it harder to distinguish the algorithms' performance and show their differences in performance. Thus, we choose the classes under test from a set of classes used in previous studies [10, 12] [1].

[1]`https://github.com/jose/non-trivial-java-classes-to-study-search-based-software-testing-approaches`

The set is composed of 117 open-source Java projects and 346 Java classes. These classes have several degrees of complexity: the number of lines varies between 5 and 4940 the number of branches can go from 2 to 7938. The average number of lines is 226 and the average number of branches is 176.

5.2.2 Experimental Infrastructure

All experiments were executed on a cluster composed of 600 compute nodes where each node has two Intel 8-core "Sandy Bridge" generation Xeon processors at 2.7 GHz with 32 GB of RAM and a shared storage with 1.5 PB [2]. The operating system installed on these nodes was CentOS Linux 7. We used Java Development Kit (JDK) 8 on our experiments as it is required by the classes under test. The implementation of the algorithms described in Chapter 4 was done in the EvoSuite [25, 38] [3] tool. As it was explained previously, one of the main reasons for choosing EvoSuite is the fact that the algorithms used as a benchmark in RQ2, RQ3 and RQ4 are already implemented.

We wish to affirm our commitment to open science, and make our scripts and our detailed analysis available for others to use. Likewise, we make all of our own research data produced in our experiments available to the research community to assist in future research. Data and scripts are available at `https://github.com/gmiduarte/swarm-study-data` (commit 8305c13).

5.2.3 Experimental procedure

In our study, we performed two experiments to access the performance of all 14 algorithms. First, we did a tuning experiment and then we ran a comparison experiment.

The tuning experiment served to find the best configuration for each algorithm. To do this, we selected a subset of 34 classes [4] from several different projects, we ran each algorithm in 30 different seeds (due to the underline randomness of each algorithm) with a budget of 60 seconds and varied the values of several properties of the algorithms.

In the comparison experiment, we ran the best configuration of each algorithm on the remaining 312 classes [5] in the same conditions (i.e., 30 seeds and a search budget of 60 seconds).

[2]Despite having a total of 9600 cores available, we were only allowed to use 256 nodes as part of the CPCA/A0/7402/2020 project supported by the Foundation for Science and Technology (FCT).

[3]Fork of the EvoSuite's official repository with our changes in here `https://github.com/gmiduarte/evosuite/tree/develop_SIAs`, commit 04bdc8b.

[4]2 classes were excluded due to EvoSuite bugs.

[5]4 classes were excluded due to EvoSuite bugs, configuration issues and instrumentation limitation.

5.2.4 Experimental metrics

In our experiments, we used eight different criteria to measure overall coverage (see Section 3.2 for more information), as well as other metrics to measure the performance of the algorithms: the number of test cases and lines of code of the test suites, number of iterations, diversity and mutation score [39, 40].

Diversity

Diversity measures how different the individuals of a population are from each other. To compute the diversity value it is necessary to compare each individual with the rest of the population.

Each comparison consists of comparing all the statements of each individual, i.e., the type and the underlying type of the statements. The types of statements are constructor, method, field and primitive depending on whether the statement calls on a method, initialize a primitive variable, etc. The underlying types go into more detail inside the type, e.g., the primitive statements can be int primitive statement, long primitive statement, among others. The value of each comparison can vary between -1.0 and 1.0, where:

- -1.0, the two individuals do not share a single statement that has a common type and underlying type.
- 0.0, the same number of differences and similarities between the statements of the two individuals.
- 1.0, the two individuals have the same number of statements and can associate each statement with a statement of the other that shares the same type and underlying type.

Then the following equation is used to compute the diversity value of a population:

$$D = 1.0 - \frac{c}{n_c} \tag{5.1}$$

where D is the Diversity, c is the sum of the values of all the comparisons between the individuals and n_c is the number of comparisons. The diversity varies between 0.0 and 2.0, where 0.0 means all the comparisons have the value of 1.0 and 2.0 is the case where the comparisons have all the value of -1.0. To help demonstrate how diversity is computed let's consider the following individuals represented in Figure 5.1:

test1: one int primitive statement.

test2: two int primitive statements.

test3: no statements.

```java
public class TestClass1 {
  @Test
  public void test1() {
    int i = 10;
  }
}
```

(a) Individual 1.

```java
public class TestClass2 {
  @Test
  public test2() {
    int i = 10;
    int l = 42;
  }
}
```

(b) Individual 2.

```java
public class TestClass3 {
  @Test
  public test3() {
    // NO-OP
  }
}
```

(c) Individual 3.

Figure 5.1: Example of three different individuals.

Comparing individual 1 with individual 2 the value obtained is 0.0 while individual 1 with individual 3 and individual 2 with individual 3 results in -1.0 in both cases. The sum of the three comparisons is -2.0, resulting in a diversity of approximately 1.67.

Note that in EvoSuite, this metric only works on algorithms that consider a test suite as an individual. Due to this, the diversity for MOSA, DynaMOSA and Elephant-DynaMOSA was not computed.

Relative Coverage

When it comes to measuring code coverage, we do not use absolute coverage values. Instead, we use relative coverage. This is due to the fact that there are several pieces of code that are always covered regardless of the algorithm and/or configuration, the easy code [10]. One example of that is a constructor of an object that only has an integer as a parameter. The initial populations in EvoSuite, most of the time already have the easy code covered, giving them a certain level of coverage despite the algorithm has not even started. The relative coverage, *δ(c,r)*, is computed as the values of the absolute coverage of class *c* in a run *r* along with the maximum and minimum values of the coverage of *c* of all runs, *max(cov(c))* and *min(cov(c))*:

$$\delta(c, r) = \frac{cov(c, r) - min(cov(c))}{max(cov(c)) - min(cov(c))} \tag{5.2}$$

To get the average relative coverage from a set of runs *R* the following equation is used:

$$\triangle(c) = \frac{1}{|R|} \sum_{r \in R} \delta(c, r) \tag{5.3}$$

Finally, the relative of an algorithm *A* in a set of classes *C* is defined with this formula:

$$cov_A = \frac{1}{|C|}\sum_{c\in C}\triangle(c) \tag{5.4}$$

To show a practical example of how the relative coverage works let's assume:

- Class A with algorithm X has a coverage value of 0.50.
- Class A with algorithm Y has a coverage value of 0.75.

Using Equation (5.2), we compute a relative coverage of 0.00 using algorithm X and 1.00 using algorithm Y since the relative coverage focus on the values where the algorithms are relevant (between 50% and 75% of coverage).

Although both absolute and relative coverages have the same algorithm as the better one (Y in the case above), the relative coverage allows us to highlight the differences between algorithms, making it clearer where the algorithms are relevant and the magnitude of their impact. Thus, the relative coverage treats all classes the same, regardless of their size and complexity.

5.2.5 Statistical Analysis

Along with the computation of the coverage values, we present the standard deviation σ, confidence intervals (CI) at 95%, and statistical results.

For the statistical analysis of comparing different randomized algorithms over a set of subjects, we followed the same procedures as others have used (e.g., Campos et al. [10]). In detail, we used the Wilcoxon-Mann-Whitney U-test to determine whether there is a statistically significant difference between two algorithms and the Vargha-Delaney $\hat{A}_{12}$ effect size to measure this difference (if any), and the Friedman test.

The Vargha-Delaney compares two algorithms at a time, to see which one is better statistically. It consists of pairwise tournaments, where we compare the 30 executions of each class (one per seed) and return an effect size that varies between 0.0 and 1.0. When comparing algorithm X with Y, if the value is >0.5 it means X has better performance, 0.5 both algorithms have the same performance and <0.5 means X has worse performance than Y. The largest the distance to 0.5, the larger the difference in the algorithms' performance.

The Friedman test is a non-parametric test for multiple-problem analysis and we used it to compute the ranking between algorithms over multiple independent problems, i.e., Java classes in our case. A significant *p-value* (i.e., < 0.05) indicates that the null hypothesis (i.e., no algorithm in the tournament performs significantly different from the others) has to be rejected in favour of the alternative one (i.e., the performance of algorithms is significantly different from each other). If the null hypothesis is rejected, we use the post hoc Conover's test for pairwise multiple comparisons. Such a test is used to detect pairs of algorithms that are significantly different. The Friedman test allows us to compare all algorithms at the same time and rank them accordingly. We take one class at a time, sort the algorithms for each class and calculate the mean of all the ranks. In the end, the final ranking is determined by that mean.

5.3 Threats to Validity

5.3.1 Internal Validity

As in all empiric studies, there are threats to its validity. When it comes to internal validity, we thoroughly tested the test generation tool, to reduce the likelihood of finding faults. Although this does not mean that we can consider that there are no defects. Also, to coupe with the randomized aspects of the algorithms, we executed each experiment 30 times and used meticulous statistical methods.

To have a fair comparison between algorithms, they were all implemented in the same testing tool, EvoSuite, and we run all executions on the same machine. Additionally, in our tuning experiments besides testing several specific parameters of the algorithms, we also tested several shared parameters. The latter used the same set of values regardless of the algorithm. All other relevant parameters used the default values of EvoSuite.

To evaluate the performance of the algorithms, we used several metrics as explained in Section 5.2.4. It is possible that there are several errors in the implementation and analysis of the results and that a few adaptions done to the BIAs are not correct and do not capture what the original algorithm is supposed to do. To cope with this issue all authors reviewed the work done in all phases of this study.

Another threat that exists is the fact that a BIA we did not evaluate could potentially invalidate our conclusions. This comes from the fact that there are countless algorithms and it is not feasible to evaluate all of them. To avoid this threat we selected several BIAs among the most famous ones, we choose BIAs of several ages and we also make sure that they are diverse in the ways they optimize the population.

5.3.2 External Validity

As for the external validity, there is the threat of a lack of generalization of the subjects (i.e., classes under test) used in our empirical study. To handle this issue we used 346 classes of 117 open source projects, with several degrees of complexity and different sizes (see Section 5.2.4). Despite this is not being enough to prove that our study generalises for all classes, it is still a best effort that mitigates this particular threat.

5.4 Tuning Results

In the tuning experiment, we evaluated five different properties each with different values:

- Population size: 10, 25, 50, and 100 for all 14 algorithms.
- Elitism: 0 and 1 for Whale, Cat, Fish, Algae, and Particle; and 0 and 3 for Wolf.
- Number of clans of 10%, 20%, and 50% of the population size for Elephant and Elephant-DynaMOSA.
- Number of male elephants per clan of 10%, 25%, 50%, and 75% of the clan size, 0 and all except the matriarch for Elephant and Elephant-DynaMOSA.
- Whether to select elephants from the archive or not for Elephant and Elephant-DynaMOSA.

This means that algorithms such as Standard GA, DynaMOSA, MOSA, Bee, Moth, and Chicken have 4 configurations; Wolf, Whale, Cat, Fish, Algae, and Particle have 8 configurations; Elephant and Elephant-DynaMOSA have 144 configurations.

We choose population number since it is a common property among all algorithms and its optimum value can easily change depending on the algorithm. We choose to use the same values as a previous study [10]. Elitism was chosen because several BIAs use the best individuals as a reference and depending on the number of references so does the value of the property. The other three properties were evaluated after the preliminary results showed that Elephant was the most promising BIA. More information on this can be found in Section 4.11. The values of the number of clans have two aspects in consideration: the algorithms are supposed to run with clans and not a population (one clan), and each clan should have at least two individuals. When it comes to the number of males we explored all kinds of possibilities, since having no males per clan all the way to a clan of only males except the matriarch. Both properties use a percentage of the population and clan size so that the values scale accordingly.

All other properties had fixed values. In the case of the properties that were part of EvoSuite prior to the implementation done in Chapter 4 (e.g., crossover rate, mutation rate, etc), we used EvoSuite's default values. The only exception being the selection function, which we decided not to use the default of EvoSuite (rank selection) in the algorithms that have a specific selection function in their descriptions: tournament in Algae and roulette wheel in both Bee and Cat algorithms [13, 29, 33]. As for the properties that were implemented with the BIAs we assigned them to the following values:

- Number of scouts 1, onlooker bee rate of 100% of population size and maximum number of iterations without improvement of 5 for Bee.

- Self position consideration with a value of false and seeking memory pool of 6 for Cat.
- Shrinking encircling mechanism rate of 50% for Whale.
- Chicken swarm update interval of 5 and number of roosters, chicks and mother hens of 1 for Chicken.
- Maximum energy of 100, energy loss rate of 20% and adaption rate of 10% for Algae.
- Number of attempts of 1, fish neighbourhood of 0.0025, and fish concentration of 0.04 for Fish.

Tables 5.1 to 5.3 report the Vargha-Delaney statistical method which consists of a pairwise tournament where for each class, a configuration of an algorithm competes with another configuration of the same algorithm. Each win in a comparison is given a point. The configuration with the most points in 30 seeds is considered better in that class and wins the tournament. After calculating the difference between tournaments won and lost, the configuration with the highest value is considered to be the best. In the case of a draw, the winner is the configuration with the highest effect size. The best configuration per algorithm is annotated with a grey background in Tables 5.1 to 5.3. The configuration column shows the values of the properties altered throughout the experiments: population, population and elitism or in the case of the Elephant, population, number of clans, number of males per clan, and whether the males can be replaced with solutions from the archive. The $\hat{A}_{12}$ and p-value columns show how good the performance of the configuration is having into account all configurations of each algorithm. Table 5.1 reports the three best configurations per algorithm, whereas Tables 5.2 and 5.3 show the ten best configurations of the Elephant and Elephant-DynaMOSA in detail.

				Tournaments Won			Tournaments Lost		
Algorithm	Configuration	$\hat{A}_{12}$	p-value	#	$\hat{A}_{12}$	p-value	#	$\hat{A}_{12}$	p-value
Standard GA	(10)	0.51	0.21	22 / 384	0.79	**< 0.01**	19 / 384	0.27	**0.01**
Standard GA	(50)	0.51	0.22	17 / 384	0.71	**0.01**	15 / 384	0.26	**0.01**
Standard GA	(25)	0.50	0.26	15 / 384	0.77	**< 0.01**	14 / 384	0.27	**0.01**
MOSA	(50)	0.52	0.24	15 / 384	0.72	**0.01**	6 / 384	0.31	**0.02**
MOSA	(25)	0.51	0.27	13 / 384	0.72	**0.01**	11 / 384	0.30	**< 0.01**
MOSA	(100)	0.49	0.21	15 / 384	0.76	**< 0.01**	19 / 384	0.28	**0.01**
DynaMOSA	(100)	0.55	0.32	22 / 384	0.79	**< 0.01**	4 / 384	0.34	**0.02**
DynaMOSA	(50)	0.52	0.40	10 / 384	0.75	**< 0.01**	4 / 384	0.25	**0.01**
DynaMOSA	(25)	0.51	0.39	7 / 384	0.73	**< 0.01**	9 / 384	0.23	**< 0.01**
Wolf	(100, 0)	0.54	0.24	50 / 1792	0.72	**0.01**	21 / 1792	0.28	**0.01**
Wolf	(100, 3)	0.53	0.24	49 / 1792	0.74	**< 0.01**	25 / 1792	0.29	**0.01**
Wolf	(50, 0)	0.52	0.35	30 / 1792	0.71	**0.01**	9 / 1792	0.29	**0.01**
Bee	(50)	0.51	0.33	9 / 384	0.72	**0.01**	4 / 384	0.29	**0.01**
Bee	(100)	0.51	0.27	15 / 384	0.75	**< 0.01**	11 / 384	0.29	**0.02**
Bee	(25)	0.50	0.35	7 / 384	0.72	**0.01**	9 / 384	0.29	**0.01**
Whale	(50, 1)	0.52	0.34	29 / 1792	0.70	**0.01**	15 / 1792	0.31	**0.02**
Whale	(50, 0)	0.51	0.33	26 / 1792	0.70	**0.01**	12 / 1792	0.32	**0.02**
Whale	(100, 0)	0.51	0.25	38 / 1792	0.72	**< 0.01**	29 / 1792	0.26	**< 0.01**
Cat	(50, 0)	0.51	0.37	22 / 1792	0.68	**0.02**	10 / 1792	0.32	**0.02**
Cat	(100, 1)	0.50	0.33	31 / 1792	0.72	**0.01**	25 / 1792	0.27	**0.01**
Cat	(50, 1)	0.50	0.38	17 / 1792	0.68	**0.02**	12 / 1792	0.32	**0.03**
Elephant	(10, 5, 1, false)	0.73	0.09	18 / 64	0.87	**< 0.01**	1 / 64	0.26	**< 0.01**
Elephant	(50, 5, 1, false)	0.27	0.09	1 / 64	0.74	**< 0.01**	18 / 64	0.13	**< 0.01**
Chicken	(100)	0.68	0.11	57 / 384	0.85	**< 0.01**	10 / 384	0.22	**< 0.01**
Chicken	(50)	0.54	0.11	36 / 384	0.77	**< 0.01**	25 / 384	0.24	**< 0.01**
Chicken	(25)	0.41	0.15	14 / 384	0.71	**0.01**	36 / 384	0.19	**< 0.01**
Moth	(100)	0.55	0.21	25 / 384	0.75	**< 0.01**	11 / 384	0.30	**0.02**
Moth	(50)	0.52	0.32	16 / 384	0.71	**0.01**	6 / 384	0.30	**0.01**
Moth	(25)	0.48	0.32	6 / 384	0.68	**0.02**	13 / 384	0.26	**< 0.01**
Fish	(50, 0)	0.55	0.29	49 / 1792	0.74	**< 0.01**	11 / 1792	0.28	**0.02**
Fish	(50, 1)	0.54	0.28	49 / 1792	0.74	**< 0.01**	15 / 1792	0.25	**0.01**
Fish	(100, 0)	0.54	0.24	58 / 1792	0.78	**< 0.01**	31 / 1792	0.22	**< 0.01**
Algae	(100, 1)	0.77	0.13	157 / 1792	0.87	**< 0.01**	0 / 1792	N/A	N/A
Algae	(100, 0)	0.77	0.13	155 / 1792	0.88	**< 0.01**	0 / 1792	N/A	N/A
Algae	(50, 1)	0.60	0.14	104 / 1792	0.84	**< 0.01**	42 / 1792	0.21	**< 0.01**
Particle	(100, 0)	0.52	0.22	54 / 1792	0.75	**< 0.01**	35 / 1792	0.24	**< 0.01**
Particle	(100, 1)	0.52	0.23	52 / 1792	0.75	**< 0.01**	34 / 1792	0.26	**0.01**
Particle	(50, 1)	0.51	0.28	31 / 1792	0.71	**0.01**	24 / 1792	0.30	**0.01**

Table 5.1: Top-3 best configurations per algorithm.

			Tournaments Won			Tournaments Lost		
Configuration	$\hat{A}_{12}$	p-value	#	$\hat{A}_{12}$	p-value	#	$\hat{A}_{12}$	p-value
(10, 2, 4, false)	0.72	0.11	1905 / 304192	0.86	**< 0.01**	97 / 304192	0.24	**< 0.01**
(10, 1, 9, false)	0.71	0.12	1828 / 304192	0.86	**< 0.01**	93 / 304192	0.26	**< 0.01**
(25, 6, 3, false)	0.70	0.12	1799 / 304192	0.85	**< 0.01**	99 / 304192	0.27	**0.01**
(25, 2, 11, false)	0.70	0.12	1783 / 304192	0.85	**< 0.01**	106 / 304192	0.27	**< 0.01**
(10, 5, 1, false)	0.70	0.12	1791 / 304192	0.85	**< 0.01**	124 / 304192	0.23	**< 0.01**
(50, 5, 9, false)	0.70	0.12	1769 / 304192	0.86	**< 0.01**	117 / 304192	0.29	**0.01**
(10, 1, 7, false)	0.69	0.12	1732 / 304192	0.85	**< 0.01**	144 / 304192	0.26	**< 0.01**
(10, 2, 3, false)	0.69	0.12	1742 / 304192	0.85	**< 0.01**	183 / 304192	0.23	**< 0.01**
(50, 25, 1, false)	0.69	0.13	1718 / 304192	0.85	**< 0.01**	159 / 304192	0.26	**< 0.01**
(50, 12, 3, false)	0.69	0.14	1656 / 304192	0.85	**< 0.01**	118 / 304192	0.28	**0.01**

Table 5.2: Top-10 best configurations of the Elephant algorithm.

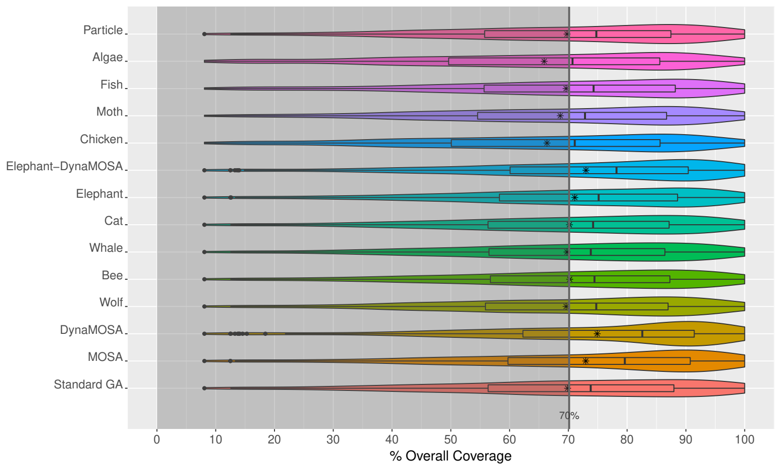
			Tournaments Won			Tournaments Lost		
Configuration	$\hat{A}_{12}$	p-value	#	$\hat{A}_{12}$	p-value	#	$\hat{A}_{12}$	p-value
(25, 2, 11, false)	0.68	0.17	1661 / 304192	0.83	**< 0.01**	88 / 304192	0.33	**0.02**
(100, 10, 9, false)	0.68	0.18	1638 / 304192	0.84	**< 0.01**	84 / 304192	0.34	**0.02**
(100, 10, 7, false)	0.67	0.18	1595 / 304192	0.83	**< 0.01**	82 / 304192	0.34	**0.02**
(50, 5, 7, false)	0.68	0.18	1566 / 304192	0.83	**< 0.01**	68 / 304192	0.31	**0.02**
(50, 5, 9, false)	0.68	0.19	1537 / 304192	0.84	**< 0.01**	68 / 304192	0.31	**0.02**
(10, 1, 9, false)	0.66	0.19	1562 / 304192	0.82	**< 0.01**	96 / 304192	0.26	**0.01**
(10, 1, 7, false)	0.66	0.18	1561 / 304192	0.82	**< 0.01**	100 / 304192	0.29	**0.01**
(25, 2, 6, false)	0.65	0.18	1509 / 304192	0.82	**< 0.01**	107 / 304192	0.29	**0.01**
(100, 10, 5, false)	0.66	0.19	1481 / 304192	0.82	**< 0.01**	84 / 304192	0.29	**0.02**
(25, 2, 9, false)	0.66	0.21	1450 / 304192	0.84	**< 0.01**	89 / 304192	0.28	**0.02**

Table 5.3: Top-10 best configurations of the Elephant-DynaMOSA algorithm.

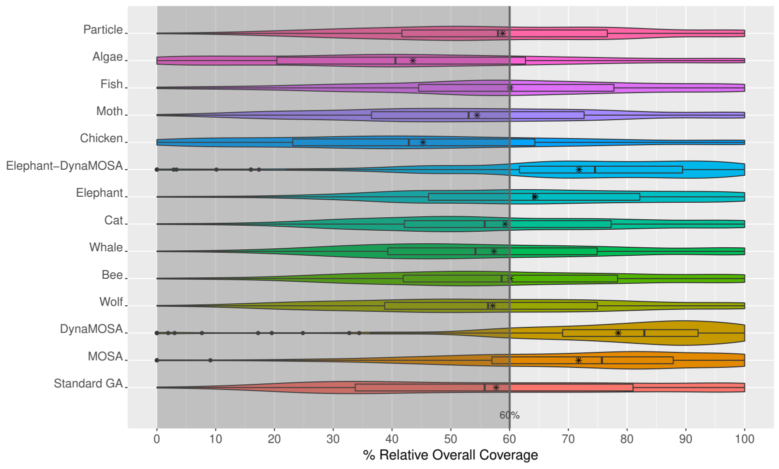
5.5 RQ1: Which bio-inspired algorithm performs best?

Figure 5.2 shows the distribution of the overall coverage and relative coverage per algorithm. On one hand, the distribution of absolute values shows that most BIAs perform similarly. On the other hand, the distribution of the relative coverage shows a different result and it is easier to see the BIAs' performance differences. Recall Section 5.2.4 for the reason why we computed relative coverage. Out of the ten BIAs, Elephant is the only one above the average, Fish and Bee are very close to the average, and the other seven BIAs perform below the average. Figure 5.3a, clearly shows that the Algae and Chicken have the worst performance as they have the highest density of classes for low percentage values. That is, both algorithms achieved a low coverage (less than 10%) for a larger portion of classes under test.

Similar conclusion if we take into account mutation score. In Figure 5.4, we can see

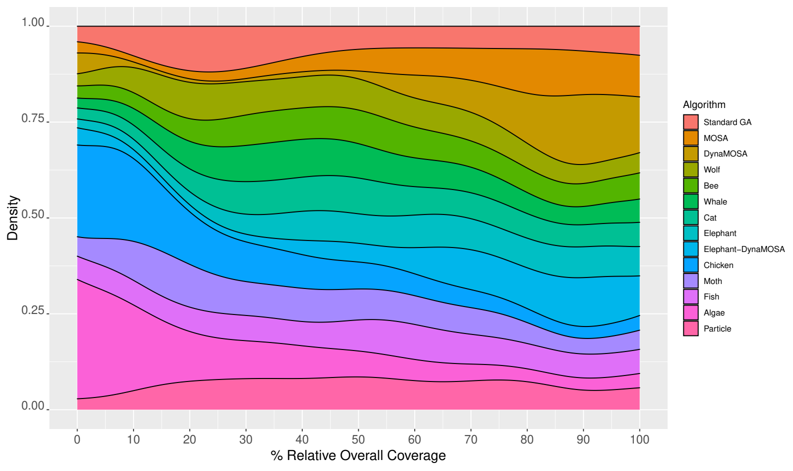


(a) Distribution of overall coverage.

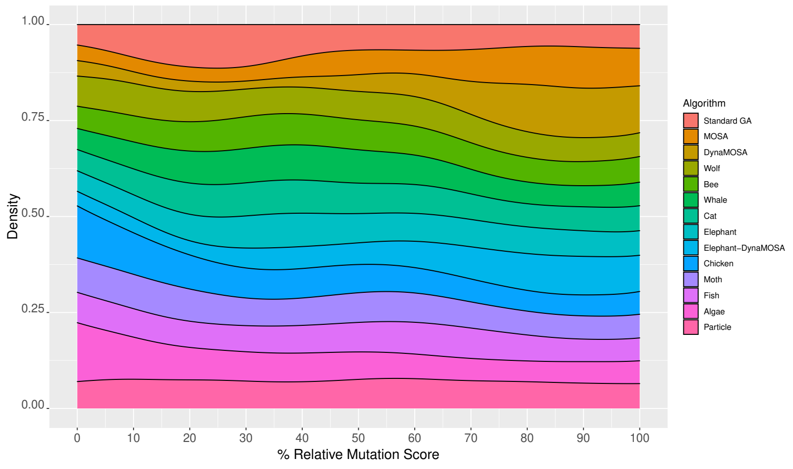


(b) Distribution of relative overall coverage.

Figure 5.2: Overall Coverage distribution by each algorithm. The grey line represents the mean of all algorithms, while the symbol * is the mean of each algorithm.



(a) Density of relative overall coverage

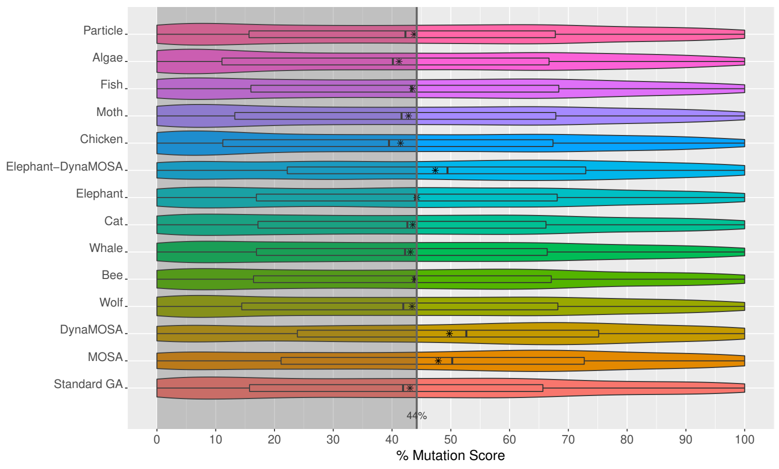


(b) Density of relative mutation score

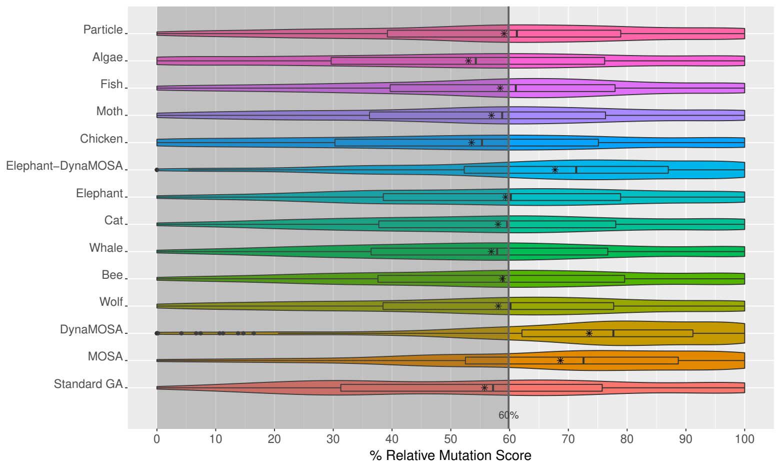
Figure 5.3: Density of the relative coverage and mutation score.

that almost all BIAs have similar values. The Chicken and the Algae are the ones that stand out, being the only BIAs below 55%. Also in Figure 5.3b, both algorithms have a slightly higher density of classes with lower percentages of coverage.

Table 5.4 reports the absolute and relative coverage and mutation score, the number of generations each algorithm ran, the number of test cases of the final solution and their average size in terms of the number of lines. Note that EvoSuite performs a minimization at the end of execution, i.e., discards all generated tests and/or lines of code within tests that are considered redundant—tests/lines that do not contribute to exercise more code of the class under test. As we can see, the BIAs have similar performances if we look at the absolute values: 9.02% and 8.58% difference between the highest and lowest coverage and mutation score, respectively. The relative values show a maximum difference of 34.94% and 20.54%, making it much easier to compare and see the impact the algorithms have in terms of overall values. The Elephant algorithm achieved the highest relative coverage (64.38%), followed by Bee, Fish and Cat (60.12%, 60.10%, and 59.26%, respectively).



(a) Distribution of mutation score



(b) Distribution of relative mutation score

Figure 5.4: Mutation Score distribution by each algorithm. The grey line represents the mean of all algorithms, while the symbol * is the mean of each algorithm.

The Chicken and the Algae algorithms stand out with their low values of 45.28% and 43.55%, respectively. As for the relative mutation score, all BIAs except Chicken and Algae achieved scores between 56.88% and 59.32%. Once again, these two algorithms have the lowest values: 53.54% and 53.01%.

High coverage means more code of the class under test is explored, which in turn increases the chances of detecting and killing mutants (alterations to the source code). As we can in Figure 5.5, all BIAs follow this rule and show a significant improvement in performance the higher the values of both coverage and mutation score.

There is a large difference between a few algorithms in terms of the number of generations, e.g., Elephant has 81 and Algae has 5. This happened even though all algorithms had the same time budget of 60 seconds. The two main reasons for this are the population size and the number and type of operations done per individual in each generation (e.g., crossovers and mutations) which affect the execution time of generation. Although the

		% Relative Overall			% Relative Mutation				
Algorithm	# G	Coverage	σ	CI	Score	σ	CI	# T	L
Standard GA	380	57.72 (69.87)	31.06	[57.08, 58.39]	55.74 (43.06)	32.67	[55.06, 56.42]	31	109
MOSA	63	71.75 (72.96)	25.79	[71.19, 72.27]	68.65 (47.87)	29.39	[68.05, 69.23]	41	145
DynaMOSA	51	78.49 (74.91)	23.19	[78.02, 78.98]	73.55 (49.77)	27.70	[72.99, 74.06]	46	168
Wolf	9	57.14 (69.62)	27.45	[56.58, 57.68]	58.08 (43.41)	31.40	[57.40, 58.76]	37	120
Bee	10	60.12 (70.23)	27.06	[59.58, 60.68]	58.78 (43.84)	30.94	[58.19, 59.42]	36	118
Whale	28	57.35 (69.77)	27.43	[56.77, 57.90]	56.88 (43.13)	31.11	[56.20, 57.53]	35	116
Cat	23	59.26 (70.18)	27.08	[58.69, 59.84]	58.04 (43.53)	31.03	[57.41, 58.73]	36	118
Elephant	81	64.38 (71.09)	26.28	[63.86, 64.85]	59.32 (44.21)	31.03	[58.67, 59.92]	35	127
Elephant-DynaMOSA	146	71.82 (72.99)	25.43	[71.32, 72.32]	67.74 (47.36)	29.18	[67.15, 68.35]	43	184
Chicken	11	45.28 (66.36)	30.26	[44.68, 45.88]	53.54 (41.45)	33.24	[52.85, 54.20]	35	109
Moth	9	54.47 (68.62)	28.03	[53.83, 55.02]	56.92 (42.80)	31.87	[56.23, 57.60]	36	117
Fish	43	60.10 (69.61)	27.53	[59.50, 60.68]	58.41 (43.47)	31.10	[57.79, 59.07]	37	125
Algae	5	43.55 (65.89)	30.62	[42.99, 44.21]	53.01 (41.19)	33.56	[52.33, 53.69]	34	108
Particle	7	58.83 (69.77)	27.09	[58.28, 59.38]	59.07 (43.76)	31.23	[58.43, 59.72]	37	123

Table 5.4: *# G* represents the average number of generations, the values between parentheses represents the absolute values, *# T* represents the average number of test cases generated by each algorithm across all classes under test, and *L* represents the average length (i.e., number of lines) of the generated test cases.

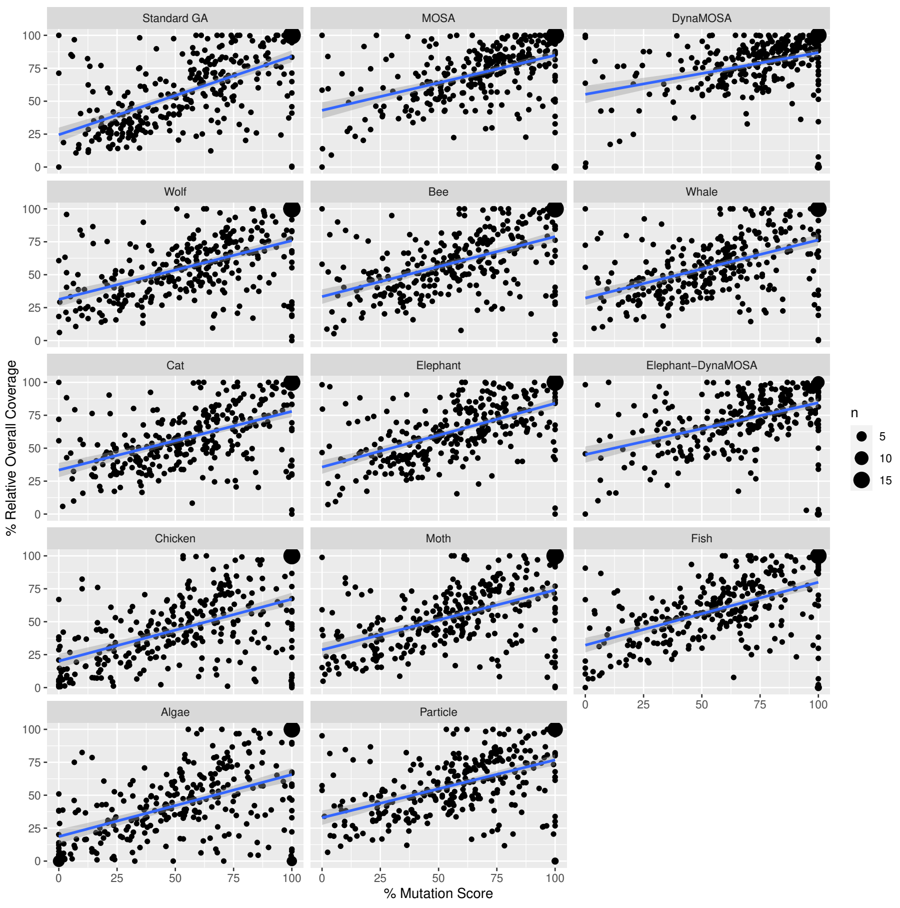


Figure 5.5: Scatter plot of % Relative Overall Coverage vs Mutation Score.

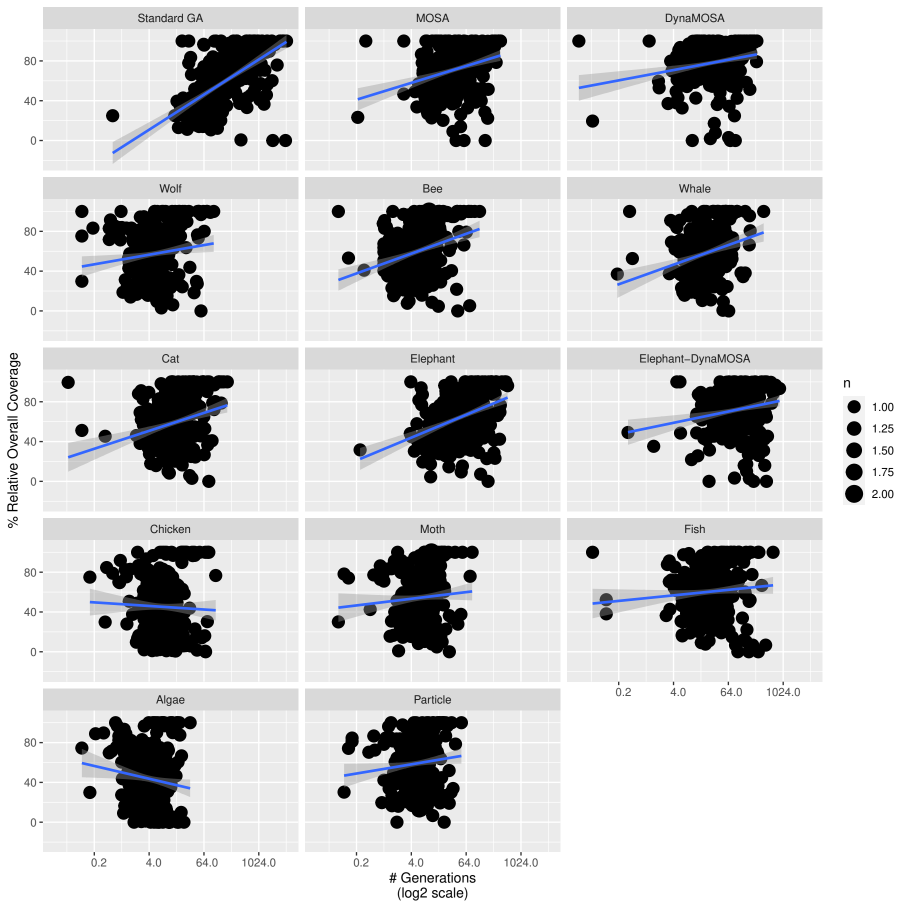


Figure 5.6: Scatter plot of % Relative Overall Coverage vs Generations

BIA with the highest coverage and mutation score values has the most generations and the BIA with the lowest values has the least generations, having more generations does not necessarily mean that the algorithm will achieve a better coverage/mutation score at the end of the execution. Each algorithm explores a different amount of search space per generation. This can be observed in the case of the Bee and Whale, where the Bee achieved higher coverage and mutation score than the Whale algorithm while having the number of generations nearly three times lower. When considering Figure 5.6 we see that the BIAs can be divided into three groups. Algae and Chicken tend to have worse coverage values the higher the number of generations. Moth, Particle and Fish have a slight increase in coverage values when the generations increase. Wolf, Bee, Whale, Cat and Elephant have a clear increase in performance the higher number of generations go. In order to explain why these groups exist, we will explain a few aspects in the figure that lead to the execution of the classes to be like that. High coverage and a low number of generations mean the classes are so simple that the initial/first few populations can already achieve a good performance at the end of the execution. High coverage with a high number of generations signifies that the algorithm was able to efficiently explore the search space and find good solutions. The reason for that can be either the classes were

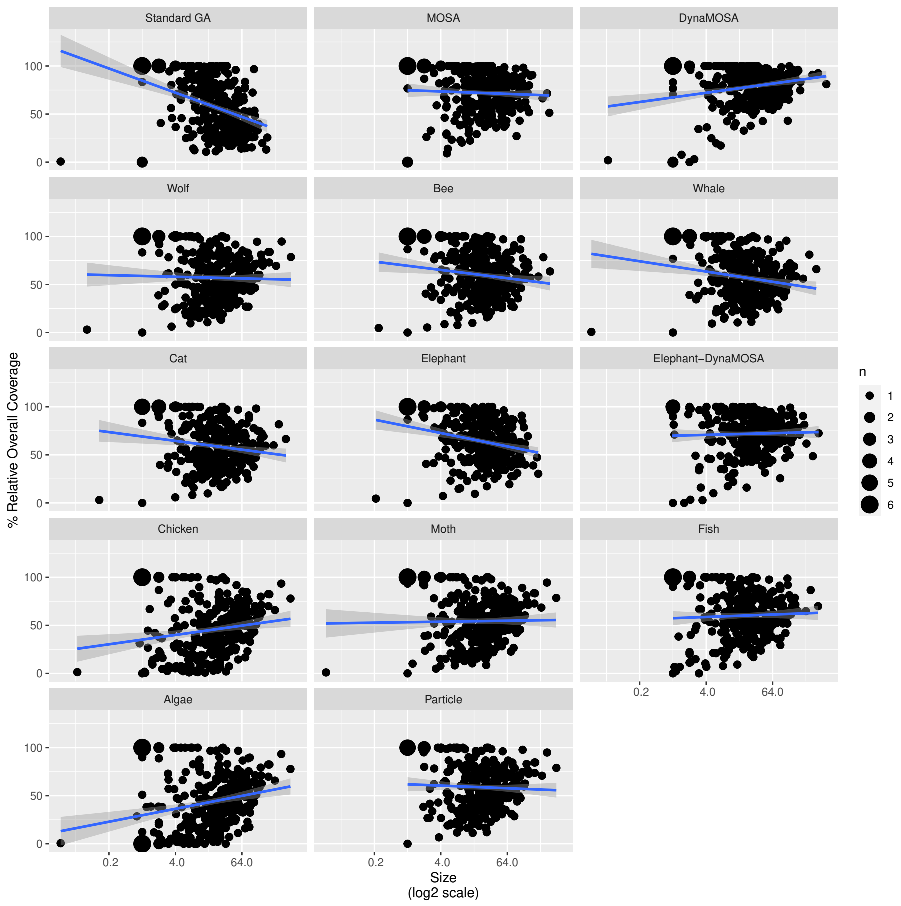


Figure 5.7: Scatter plot of % Relative Overall Coverage vs Size

simple and/or the algorithm was good enough to explore the more complex classes. Low coverage with a low number of generations implies that the budget was not enough for the algorithms to explore the classes that have a certain degree of complexity. Low coverage with a high number of generations can be attributed to the algorithm being bad for those classes. The algorithm has many useless operations that do not help it achieve high coverage for those classes.

Lastly, the number of test cases of the final solution and their average size is similar among all BIAs. As we can see in Table 5.4, the Algae algorithm generated the lowest number of test cases (34) and the smallest tests (108 lines of code on average). In theory, fewer tests and/or smaller tests means tests would take less time to run, they could also mean less code being exercised. However, this does not hold to our results. As we can see, the Bee and Moth algorithms generated tests with nearly the same number of lines of code, yet Bee has the 2nd highest coverage value and Moth has the 8th. In Figure 5.7, we can see that Chicken and Algae clearly benefit from larger sizes, allowing them to reach higher coverage values. Bee, Cat, Whale, and Elephant are the opposite, i.e., their performance gets worst the larger the final solution is.

As explained in Section 5.2.4, diversity shows the degree of difference between the

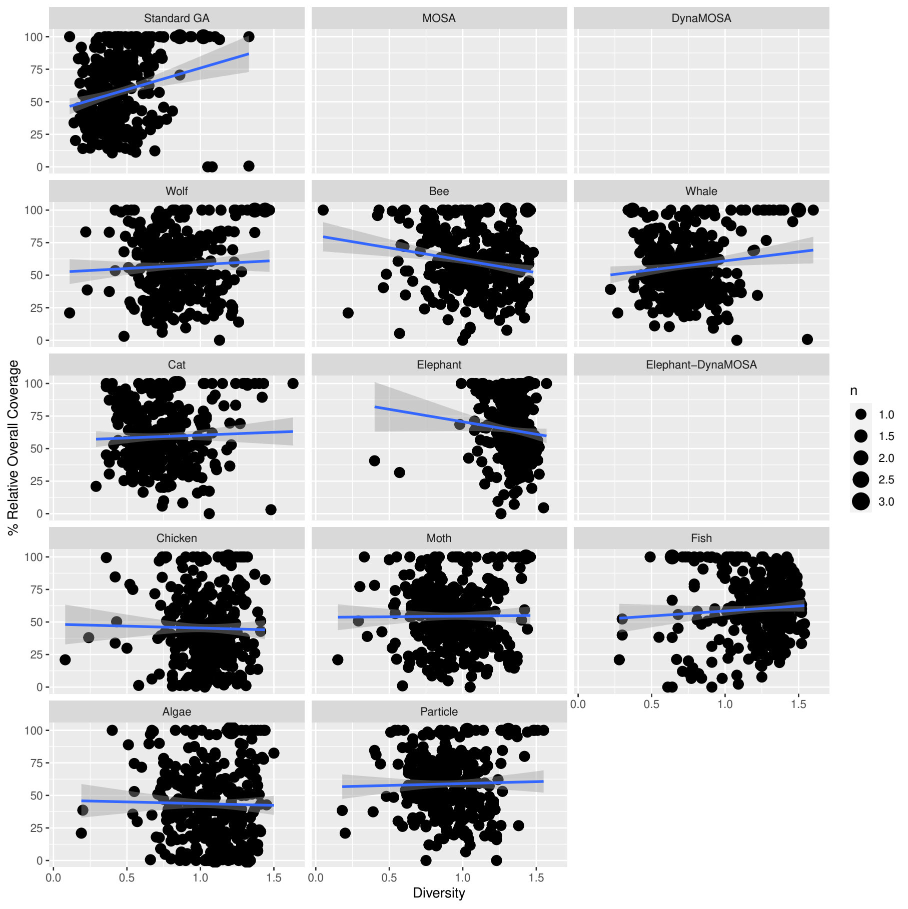


Figure 5.8: Scatter plot of % Relative Overall Coverage vs Diversity. Due to the limitations of EvoSuite, it is not possible to obtain the values of diversity of DynaMOSA, MOSA and Elephant-DynaMOSA.

individuals of a population. Achieving a high or low diversity in the final solution should not affect the coverage by itself, it only shows if the algorithms tend to converge towards a certain point in the search space or favour exploration. In Figure 5.8 we can observe the influence diversity has on coverage. From the slopes, the BIAs where diversity seems to have a certain degree of influence are Bee, Elephant and Whale. However, if we pay attention to the uncertainty (grey area) it is possible to see that no conclusion can be reached for all BIAs. This happens because in all BIAs the values of diversity are concentrated in a small range of values and outside said range there are too few results to make the size of the uncertainty reasonable. Due to the limitations of EvoSuite, DynaMOSA, MOSA and Elephant-DynaMOSA do not have values of diversity during their execution.

Table 5.5 illustrates how the Vargha-Delaney method ranks the BIAs in terms of relative coverage and mutation score. Each tournament consists in comparing one algorithm with another in a single class, and the one who wins more comparisons in the 30 executions wins. The columns with the tournament won and lost only considers the statistically relevant comparisons in which the algorithm is better or worse than another. The ranking

				Tournaments Won			Tournaments Lost		
Algorithm	Rank	$\hat{A}_{12}$	p-value	#	$\hat{A}_{12}$	p-value	#	$\hat{A}_{12}$	p-value
% Relative Overall Coverage									
Wolf	6	0.52	0.22	663 / 2772	0.81	**< 0.01**	538 / 2772	0.22	**< 0.01**
Bee	4	0.55	0.20	841 / 2772	0.82	**< 0.01**	464 / 2772	0.22	**< 0.01**
Whale	7	0.49	0.21	551 / 2772	0.82	**< 0.01**	700 / 2772	0.23	**< 0.01**
Cat	5	0.54	0.23	713 / 2772	0.82	**< 0.01**	483 / 2772	0.23	**< 0.01**
Elephant	1	0.64	0.12	1423 / 2772	0.85	**< 0.01**	351 / 2772	0.18	**< 0.01**
Chicken	9	0.33	0.18	194 / 2772	0.78	**< 0.01**	1400 / 2772	0.14	**< 0.01**
Moth	8	0.47	0.20	542 / 2772	0.81	**< 0.01**	780 / 2772	0.20	**< 0.01**
Fish	2	0.58	0.17	1064 / 2772	0.81	**< 0.01**	408 / 2772	0.21	**< 0.01**
Algae	10	0.31	0.17	164 / 2772	0.79	**< 0.01**	1477 / 2772	0.13	**< 0.01**
Particle	3	0.56	0.21	854 / 2772	0.80	**< 0.01**	408 / 2772	0.23	**< 0.01**
% Relative Mutation Score									
Wolf	3	0.52	0.33	402 / 2772	0.76	**< 0.01**	228 / 2772	0.25	**< 0.01**
Bee	5	0.51	0.30	436 / 2772	0.76	**< 0.01**	335 / 2772	0.25	**< 0.01**
Whale	8	0.48	0.32	263 / 2772	0.77	**< 0.01**	400 / 2772	0.26	**0.01**
Cat	6	0.50	0.32	345 / 2772	0.76	**< 0.01**	327 / 2772	0.26	**< 0.01**
Elephant	2	0.53	0.23	692 / 2772	0.79	**< 0.01**	456 / 2772	0.23	**< 0.01**
Chicken	9	0.44	0.29	232 / 2772	0.75	**< 0.01**	654 / 2772	0.20	**< 0.01**
Moth	7	0.50	0.33	337 / 2772	0.75	**< 0.01**	321 / 2772	0.25	**< 0.01**
Fish	4	0.52	0.28	477 / 2772	0.78	**< 0.01**	333 / 2772	0.27	**0.01**
Algae	10	0.44	0.29	229 / 2772	0.75	**< 0.01**	686 / 2772	0.19	**< 0.01**
Particle	1	0.55	0.32	508 / 2772	0.75	**0.01**	181 / 2772	0.27	**0.01**

Table 5.5: Pairwise tournament: Swarm-based algorithms.

reported in this table is done by subtracting the tournament lost from the tournament won and sorting that value. Looking at the relative coverage, the ranks are consistent with the number of tournaments won and lost, i.e, the rankings are the same as if we were to order the algorithms according to the tournaments won (descending order) and the tournaments lost (ascending order). The best BIA is the Elephant with 51.33% of the tournaments won and 12.66% lost. The worst BIAs are clearly the Chicken and Algae which won 7.00% and 5.91% and lost 50.50% and 53.28%, respectively.

Considering relative mutation score, the best BIA is Particle (won 18.32% and lost 6.52%), followed by the Elephant (won 24.96% and lost 16.45%). The worst BIAs are once again Chicken (won 8.36% and lost 23.59%) and Algae (won 8.26% and lost 24.74%).

RQ1: Elephant is the bio-inspired algorithm that achieves the highest coverage and the second highest mutation score (in line with the Particle algorithm).

5.6 RQ2: How does swarm-based search compare to traditional evolutionary search?

Figure 5.2a reports that the Standard GA has similar absolute coverage as most BIAs, around 70%. While in Figure 5.2b, Standard GA is below the mean of all algorithms as are most BIAs (Particle, Algae, Moth, Chicken, Cat, Whale and Wolf). Similarly, Figure 5.4 reports that the absolute and relative mutation score of the Standard GA is very similar to all BIAs.

When it comes to the density values (see Figure 5.3), Standard GA achieved the highest value ($\sim$25%), which is different from the values achieved by each BIA.

Looking at Table 5.4, we can see that the Standard GA achieved a relative coverage of 57.72%, i.e., performing worse than Elephant, Bee, Fish, Particle, and Cat. As for the mutation score, the value is 55.74%, only higher than Chicken and Algae. One interesting fact is the sheer amount of generations made by the Standard GA, 380. This value is more than four times larger than the number of generations of Elephant (the BIA with the highest number of generations). The reason for this value is that Standard GA has a small population and a low number of operations done per generation compared to other algorithms. This makes the time used to run each generation very small compared to the BIAs. Standard GA generated the lowest number of test cases (31) with a low average number of lines of code (109).

Looking at Figures 5.5 to 5.8, we can see that Standard GA shows a tendency to achieve higher coverage values the higher the mutation score, diversity, and generations are. We can also ascertain that a smaller number of tests benefits the performance.

In Tables 5.6 and 5.7, we can see the pairwise tournament of the BIAs and the Standard GA. The former uses all 11 algorithms in the comparisons, while the latter is focused on an individual comparison between each BIA and the Standard GA.

Table 5.6 shows that Standard GA is ranked 6th in terms of relative coverage, despite being the 3rd with the most tournaments won (34.12%). This is because it is the 3rd algorithm that lost more tournaments (30.75%). From this, we can conclude that Standard GA in some executions achieved higher coverage values than most BIAs while in others achieved lower coverage values than almost all BIAs.

Standard GA is almost the worst algorithm when it comes to relative mutation score, ranked 9th. The reason for this low rank is because it is the algorithm that lost the highest amount of tournaments (29.22%), while the number of tournaments won is not far from the numbers of most BIAs (17.31%).

When comparing BIAs and Standard GA (see Table 5.7 and Figure 5.9), we can see that Bee, Cat, Elephant, Fish, and Particle performed, overall, better in terms of relative coverage as the effect size is higher than 0.5. If we look at the relative mutation score values, only Chicken and Algae performed worse in general than Standard GA.

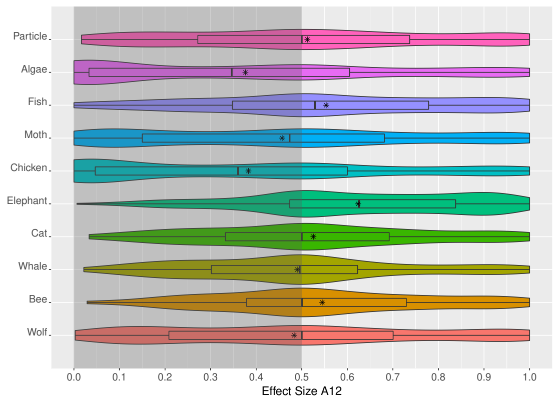
				Tournaments Won			Tournaments Lost		
Algorithm	Rank	$\hat{A}_{12}$	p-value	#	$\hat{A}_{12}$	p-value	#	$\hat{A}_{12}$	p-value
				% Relative Overall Coverage					
Standard GA	6	0.50	0.11	1051 / 3080	0.83	**< 0.01**	947 / 3080	0.15	**< 0.01**
Wolf	7	0.52	0.21	757 / 3080	0.82	**< 0.01**	657 / 3080	0.21	**< 0.01**
Bee	4	0.55	0.20	938 / 3080	0.82	**< 0.01**	537 / 3080	0.22	**< 0.01**
Whale	8	0.49	0.21	620 / 3080	0.83	**< 0.01**	799 / 3080	0.23	**< 0.01**
Cat	5	0.54	0.22	808 / 3080	0.82	**< 0.01**	572 / 3080	0.23	**< 0.01**
Elephant	1	0.64	0.12	1565 / 3080	0.85	**< 0.01**	397 / 3080	0.19	**< 0.01**
Chicken	10	0.33	0.17	268 / 3080	0.81	**< 0.01**	1555 / 3080	0.13	**< 0.01**
Moth	9	0.47	0.18	631 / 3080	0.81	**< 0.01**	906 / 3080	0.19	**< 0.01**
Fish	2	0.58	0.17	1180 / 3080	0.81	**< 0.01**	493 / 3080	0.21	**< 0.01**
Algae	11	0.32	0.16	237 / 3080	0.82	**< 0.01**	1635 / 3080	0.13	**< 0.01**
Particle	3	0.56	0.20	952 / 3080	0.81	**< 0.01**	509 / 3080	0.22	**< 0.01**
				% Relative Mutation Score					
Standard GA	9	0.46	0.20	533 / 3080	0.79	**< 0.01**	900 / 3080	0.21	**< 0.01**
Wolf	3	0.53	0.32	503 / 3080	0.77	**< 0.01**	292 / 3080	0.25	**< 0.01**
Bee	5	0.52	0.29	524 / 3080	0.77	**< 0.01**	365 / 3080	0.25	**< 0.01**
Whale	8	0.48	0.31	331 / 3080	0.77	**< 0.01**	446 / 3080	0.26	**0.01**
Cat	6	0.51	0.31	429 / 3080	0.77	**< 0.01**	370 / 3080	0.26	**< 0.01**
Elephant	2	0.53	0.23	790 / 3080	0.79	**< 0.01**	484 / 3080	0.23	**< 0.01**
Chicken	10	0.45	0.28	316 / 3080	0.76	**< 0.01**	733 / 3080	0.20	**< 0.01**
Moth	7	0.51	0.31	431 / 3080	0.76	**< 0.01**	382 / 3080	0.24	**< 0.01**
Fish	4	0.52	0.27	567 / 3080	0.78	**< 0.01**	374 / 3080	0.26	**< 0.01**
Algae	11	0.44	0.27	312 / 3080	0.77	**< 0.01**	773 / 3080	0.19	**< 0.01**
Particle	1	0.55	0.31	618 / 3080	0.76	**< 0.01**	235 / 3080	0.26	**< 0.01**

Table 5.6: Pairwise tournament: Swarm-based algorithms and Standard GA.

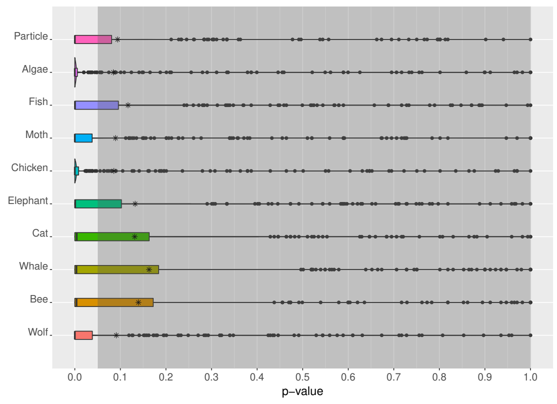
RQ2: Five out of ten bio-inspired algorithms performed better (not significantly better) than the Standard GA at coverage level, and eight out of ten bio-inspired algorithms performed better (not significantly better) than the Standard GA at mutation level.

	vs Standard GA	
Algorithm	$\hat{A}_{12}$	p-value
% Relative Overall Coverage		
Wolf	0.48	0.09
Bee	0.55	0.14
Whale	0.49	0.16
Cat	0.53	0.13
Elephant	0.62	0.13
Chicken	0.38	0.09
Moth	0.46	0.09
Fish	0.55	0.12
Algae	0.38	0.08
Particle	0.51	0.09
% Relative Mutation Score		
Wolf	0.55	0.18
Bee	0.56	0.25
Whale	0.53	0.23
Cat	0.55	0.22
Elephant	0.57	0.21
Chicken	0.49	0.15
Moth	0.53	0.18
Fish	0.56	0.21
Algae	0.49	0.14
Particle	0.56	0.19

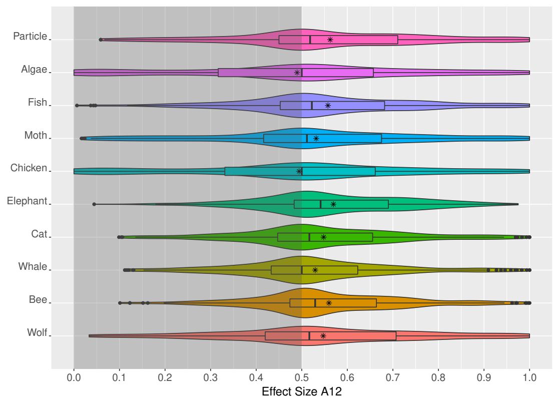
Table 5.7: Swarm-based algorithms vs. Standard GA.



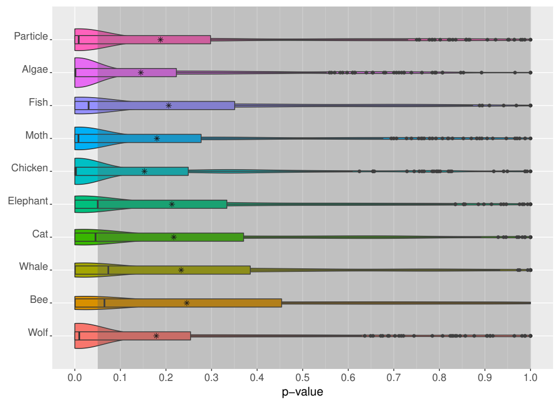
(a) Effect size (% Relative Overall Coverage).



(b) p-value (% Relative Overall Coverage).



(c) Effect size (% Relative Mutation Score).



(d) p-value (% Relative Mutation Score).

Figure 5.9: Distribution of effect sizes and p-values: Swarm-based algorithms vs. Standard GA.

5.7 RQ3: How does swarm-based search compare to many-objective optimization algorithms?

In Figures 5.2 and 5.4, we can see that the absolute values achieved by MOSA and DynaMOSA are clearly above the mean and higher than any BIAs' values. When we look at the relative coverage and mutation score the difference between the MOSAs and the BIAs is even higher.

Figure 5.3 shows that for a larger number of classes, DynaMOSA and MOSA achieved 80% to 100% coverage and mutation score. This supports the observation made above, that both MOSA and DynaMOSA achieved higher values than the BIAs.

Table 5.4 shows that DynaMOSA and MOSA achieved the highest values for relative coverage (78.49% and 71.75%) and mutation score (73.55% and 68.65%) than the BIAs. As for the number of generations (51 and 63), both have more than all BIAs, except the Elephant (81). The generated solutions are clearly larger than the ones of the BIAs, having more test cases and a higher average number of lines. DynaMOSA has an average of 46 test cases with an average of 168 lines while MOSA has 41 test cases with an average size of 145.

Looking at Figures 5.5 and 5.6, we can see that both MOSA and DynaMOSA achieve higher coverage values the higher the mutation score and number of generations is. When it comes to Figure 5.7, DynaMOSA achieves higher coverage the larger the number of tests, while MOSA's performance was hardly influenced by it.

Table 5.8 reports the comparisons of 13 algorithms and Table 5.9 reports two direct comparisons: BIAs vs. MOSA and BIAs vs. DynaMOSA. DynaMOSA is clearly the best algorithm follow by MOSA in both relative coverage and mutation score. Both MOSAs won the most tournaments and lost the fewest tournaments. As for the BIAs, they are clearly below DynaMOSA and MOSA. In detail, for relative coverage, DynaMOSA with 74.11% tournaments won and 3.72% lost while MOSA has 61.74% won and 10.89% lost. As for mutation score, DynaMOSA has 60.42% won and 2.13% lost while MOSA has 49.76% won and 8.35% lost.

When comparing bio-inspired algorithms with MOSA and DynaMOSA (see Table 5.9 and Figures 5.10 and 5.11), there are several statistically relevant results. MOSA performed statistically better than Chicken and Algae, while DynaMOSA performed statistically better than Cat, Chicken, and Algae. In terms of mutation score, neither MOSA nor DynaMOSA performed statistically better than any BIA.

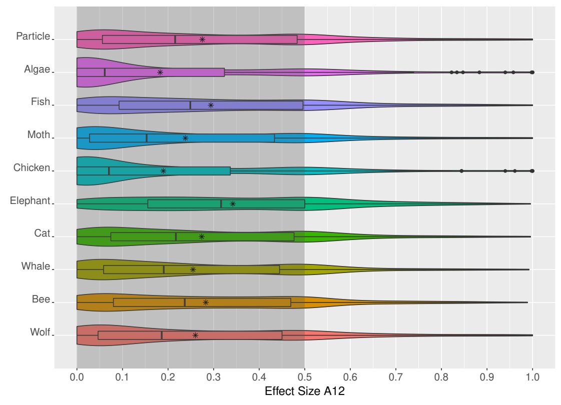
RQ3: Many-objective optimization algorithms perform better than any single-objective bio-inspired algorithm but only statistically better on three out of ten algorithms.

				Tournaments Won			Tournaments Lost		
Algorithm	Rank	$\hat{A}_{12}$	p-value	#	$\hat{A}_{12}$	p-value	#	$\hat{A}_{12}$	p-value
% Relative Overall Coverage									
MOSA	2	0.71	0.08	2092 / 3388	0.88	**< 0.01**	369 / 3388	0.20	**< 0.01**
DynaMOSA	1	0.80	0.06	2511 / 3388	0.91	**< 0.01**	126 / 3388	0.20	**< 0.01**
Wolf	8	0.47	0.19	702 / 3388	0.81	**< 0.01**	983 / 3388	0.16	**< 0.01**
Bee	6	0.50	0.18	876 / 3388	0.82	**< 0.01**	896 / 3388	0.17	**< 0.01**
Whale	9	0.44	0.19	577 / 3388	0.82	**< 0.01**	1151 / 3388	0.18	**< 0.01**
Cat	7	0.48	0.20	746 / 3388	0.82	**< 0.01**	926 / 3388	0.18	**< 0.01**
Elephant	3	0.58	0.12	1475 / 3388	0.84	**< 0.01**	730 / 3388	0.16	**< 0.01**
Chicken	11	0.30	0.15	218 / 3388	0.79	**< 0.01**	1886 / 3388	0.12	**< 0.01**
Moth	10	0.42	0.17	575 / 3388	0.81	**< 0.01**	1236 / 3388	0.16	**< 0.01**
Fish	4	0.52	0.15	1099 / 3388	0.81	**< 0.01**	836 / 3388	0.16	**< 0.01**
Algae	12	0.28	0.15	183 / 3388	0.80	**< 0.01**	1969 / 3388	0.11	**< 0.01**
Particle	5	0.50	0.19	894 / 3388	0.80	**< 0.01**	840 / 3388	0.16	**< 0.01**
% Relative Mutation Score									
MOSA	2	0.65	0.12	1686 / 3388	0.84	**< 0.01**	283 / 3388	0.24	**< 0.01**
DynaMOSA	1	0.73	0.11	2047 / 3388	0.87	**< 0.01**	72 / 3388	0.29	**0.01**
Wolf	5	0.48	0.29	435 / 3388	0.76	**< 0.01**	589 / 3388	0.19	**< 0.01**
Bee	7	0.47	0.27	452 / 3388	0.76	**< 0.01**	688 / 3388	0.20	**< 0.01**
Whale	10	0.44	0.28	283 / 3388	0.77	**< 0.01**	769 / 3388	0.20	**< 0.01**
Cat	8	0.46	0.28	367 / 3388	0.76	**< 0.01**	691 / 3388	0.20	**< 0.01**
Elephant	4	0.49	0.21	712 / 3388	0.79	**< 0.01**	792 / 3388	0.20	**< 0.01**
Chicken	11	0.41	0.25	262 / 3388	0.75	**0.01**	1039 / 3388	0.17	**< 0.01**
Moth	9	0.46	0.29	366 / 3388	0.75	**< 0.01**	694 / 3388	0.19	**< 0.01**
Fish	6	0.48	0.25	497 / 3388	0.78	**< 0.01**	686 / 3388	0.21	**< 0.01**
Algae	12	0.41	0.25	261 / 3388	0.75	**< 0.01**	1075 / 3388	0.17	**< 0.01**
Particle	3	0.50	0.28	540 / 3388	0.75	**0.01**	530 / 3388	0.19	**< 0.01**

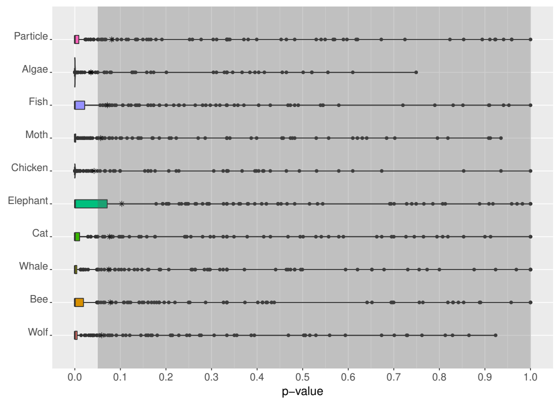
Table 5.8: Pairwise tournament: Swarm-based algorithms, MOSA, and DynaMOSA.

	vs MOSA		vs DynaMOSA	
Algorithm	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value
% Relative Overall Coverage				
Wolf	0.26	0.06	0.19	0.05
Bee	0.28	0.08	0.21	0.06
Whale	0.25	0.07	0.19	0.05
Cat	0.27	0.08	0.20	**0.05**
Elephant	0.34	0.10	0.26	0.08
Chicken	0.19	**0.04**	0.14	**0.04**
Moth	0.24	0.06	0.17	0.05
Fish	0.29	0.07	0.21	0.06
Algae	0.18	**0.04**	0.13	**0.04**
Particle	0.28	0.08	0.20	0.06
% Relative Mutation Score				
Wolf	0.33	0.10	0.27	0.08
Bee	0.33	0.14	0.27	0.12
Whale	0.31	0.11	0.25	0.10
Cat	0.32	0.12	0.26	0.10
Elephant	0.34	0.12	0.29	0.12
Chicken	0.30	0.07	0.24	0.07
Moth	0.32	0.10	0.26	0.09
Fish	0.34	0.11	0.27	0.10
Algae	0.29	0.07	0.23	0.07
Particle	0.34	0.12	0.27	0.10

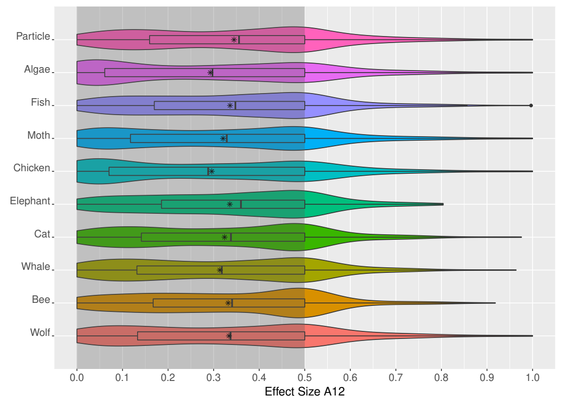
Table 5.9: Swarm-based algorithms vs. MOSA and Swarm-based algorithms vs. DynaMOSA.



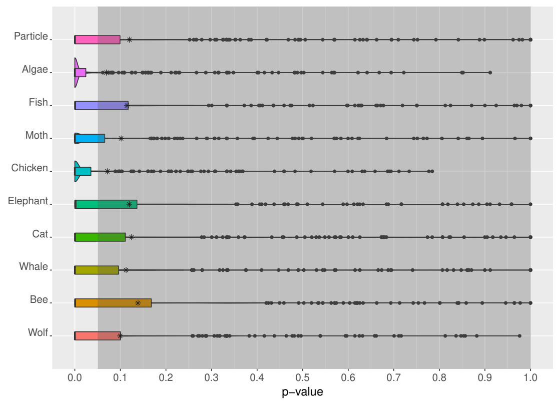
(a) Effect size (% Relative Overall Coverage).



(b) p-value (% Relative Overall Coverage).

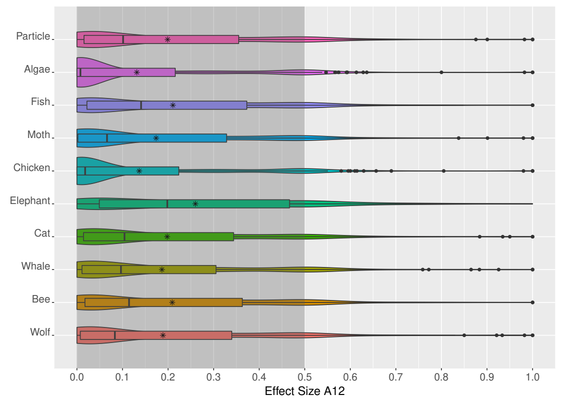


(c) Effect size (% Relative Mutation Score).

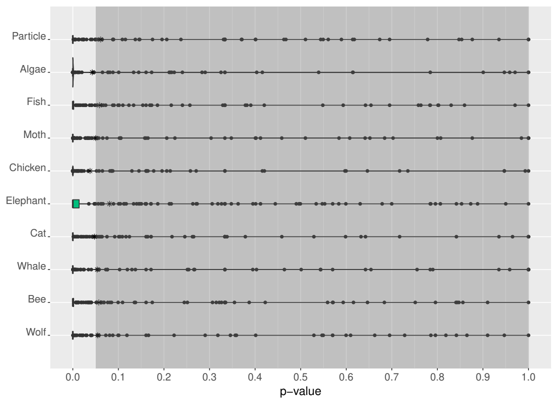


(d) p-value (% Relative Mutation Score).

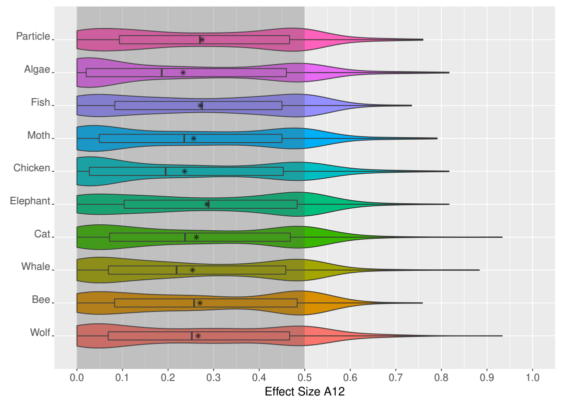
Figure 5.10: Distribution of effect sizes and p-values: Swarm-based algorithms vs. MOSA.



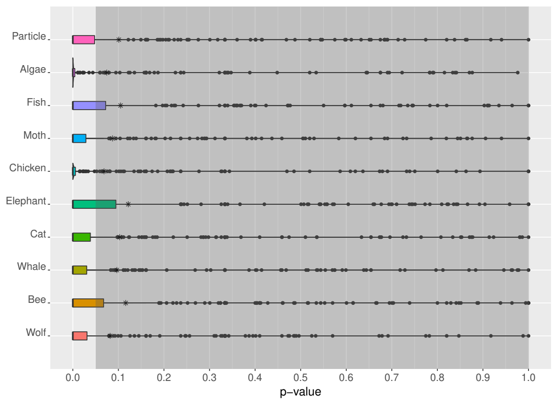
(a) Effect size (% Relative Overall Coverage).



(b) p-value (% Relative Overall Coverage).



(c) Effect size (% Relative Mutation Score).



(d) p-value (% Relative Mutation Score).

Figure 5.11: Distribution of effect sizes and p-values: Swarm-based algorithms vs. DynaMOSA.

5.8 RQ4: How does a hybrid that combines swarm-based search with many-objective optimization performs?

When we look at Figures 5.2 to 5.4, we can see that Elephant-DynaMOSA has similar values to both MOSAs. This is further shown in Table 5.4 where Elephant-DynaMOSA has very close values to the ones of MOSA: 71.82% of relative coverage and 67.74% of relative mutation score. When it comes to the number of generations, Elephant-DynaMOSA has over double the value of the MOSAs, being the 2nd algorithm with the highest amount of generations (146). This can be attributed to the low population number. As for the number of test cases of the final solution, it is around the same as the other MOSAs, but the average size of the tests is the biggest of all algorithms (184).

Figures 5.5 and 5.6 shows that Elephant-DynaMOSA is another algorithm that has higher coverage values the higher the mutation score and number of generations is. Lastly, in Figure 5.7, the performance of Elephant-DynaMOSA shows us that size is not a relevant factor to its performance.

Table 5.10 shows that the performance of all algorithms according to the Vargha-Delaney method. Elephant-DynaMOSA ranks 2nd in relative coverage behind DynaMOSA, having won 60.66% of the tournaments and lost only 10.89%. For comparison purposes, we will show the MOSAs and best BIA percentages: DynaMOSA won 71.65% and lost 4.34%, MOSA won 58.81% and lost 11.81% and Elephant won 40.76% and lost 23.78%. Elephant-DynaMOSA ranks 3rd in relative mutation score being worse than the two MOSAs with 46.75% of tournaments won and lost only 8.32%. Looking at the MOSAs and best BIA we have: DynaMOSA won 58.81% and lost 2.27%, MOSA won 48.10% and lost 8.52% and Particle won 16.63% and lost 18.43%.

When considering more direct comparisons in Table 5.11 and Figure 5.12, we have a few statistically relevant results in the relative coverage. We can see that Elephant-DynaMOSA is statistically better than Chicken and Algae. As for the other algorithms, we can see that in general, the BIAs are worse than the hybrid, DynaMOSA is better and MOSA is equal or better (coverage and mutation score, respectively).

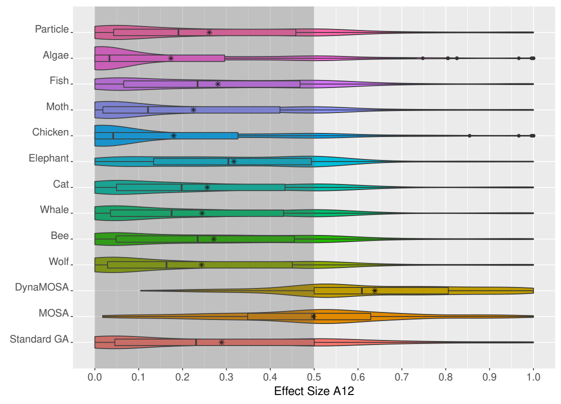
RQ4: Elephant-DynaMOSA performances better than all the bio-inspired algorithms (statistically better than Chicken and Algae) and the Standard GA; it has a similar performance to MOSA (effect size equal to 0.50), and performs worse than DynaMOSA.

				Tournaments Won			Tournaments Lost		
Algorithm	Rank	$\hat{A}_{12}$	p-value	#	$\hat{A}_{12}$	p-value	#	$\hat{A}_{12}$	p-value
% Relative Overall Coverage									
Standard GA	9	0.45	0.11	1133 / 4004	0.83	**< 0.01**	1539 / 4004	0.13	**< 0.01**
MOSA	3	0.69	0.09	2355 / 4004	0.88	**< 0.01**	473 / 4004	0.20	**< 0.01**
DynaMOSA	1	0.78	0.07	2869 / 4004	0.90	**< 0.01**	174 / 4004	0.21	**< 0.01**
Wolf	10	0.45	0.18	815 / 4004	0.82	**< 0.01**	1319 / 4004	0.15	**< 0.01**
Bee	7	0.48	0.17	994 / 4004	0.82	**< 0.01**	1172 / 4004	0.17	**< 0.01**
Whale	11	0.43	0.18	661 / 4004	0.83	**< 0.01**	1469 / 4004	0.17	**< 0.01**
Cat	8	0.47	0.19	860 / 4004	0.82	**< 0.01**	1228 / 4004	0.17	**< 0.01**
Elephant	4	0.56	0.12	1632 / 4004	0.84	**< 0.01**	952 / 4004	0.17	**< 0.01**
Elephant-DynaMOSA	2	0.70	0.10	2429 / 4004	0.89	**< 0.01**	436 / 4004	0.19	**< 0.01**
Chicken	13	0.30	0.14	311 / 4004	0.81	**< 0.01**	2280 / 4004	0.11	**< 0.01**
Moth	12	0.41	0.16	683 / 4004	0.82	**< 0.01**	1581 / 4004	0.15	**< 0.01**
Fish	5	0.51	0.15	1235 / 4004	0.81	**< 0.01**	1120 / 4004	0.16	**< 0.01**
Algae	14	0.28	0.14	272 / 4004	0.82	**< 0.01**	2369 / 4004	0.11	**< 0.01**
Particle	6	0.49	0.17	1012 / 4004	0.81	**< 0.01**	1149 / 4004	0.16	**< 0.01**
% Relative Mutation Score									
Standard GA	12	0.42	0.18	574 / 4004	0.78	**< 0.01**	1440 / 4004	0.19	**< 0.01**
MOSA	2	0.65	0.13	1926 / 4004	0.84	**< 0.01**	341 / 4004	0.24	**< 0.01**
DynaMOSA	1	0.72	0.12	2355 / 4004	0.87	**< 0.01**	91 / 4004	0.30	**0.01**
Wolf	6	0.48	0.27	551 / 4004	0.77	**< 0.01**	814 / 4004	0.19	**< 0.01**
Bee	8	0.47	0.26	551 / 4004	0.76	**< 0.01**	866 / 4004	0.20	**< 0.01**
Whale	11	0.44	0.26	361 / 4004	0.77	**< 0.01**	991 / 4004	0.20	**< 0.01**
Cat	9	0.46	0.27	461 / 4004	0.76	**< 0.01**	898 / 4004	0.20	**< 0.01**
Elephant	5	0.49	0.20	817 / 4004	0.79	**< 0.01**	981 / 4004	0.20	**< 0.01**
Elephant-DynaMOSA	3	0.64	0.16	1872 / 4004	0.83	**< 0.01**	333 / 4004	0.23	**< 0.01**
Chicken	13	0.41	0.23	362 / 4004	0.76	**< 0.01**	1298 / 4004	0.16	**< 0.01**
Moth	10	0.46	0.27	475 / 4004	0.76	**< 0.01**	918 / 4004	0.18	**< 0.01**
Fish	7	0.48	0.24	600 / 4004	0.78	**< 0.01**	883 / 4004	0.21	**< 0.01**
Algae	14	0.40	0.23	364 / 4004	0.76	**< 0.01**	1343 / 4004	0.16	**< 0.01**
Particle	4	0.50	0.27	666 / 4004	0.76	**< 0.01**	738 / 4004	0.20	**< 0.01**

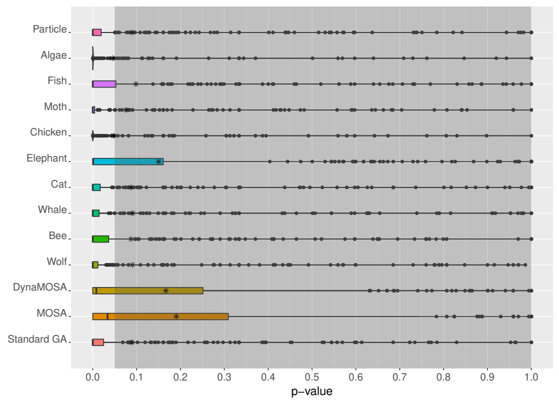
Table 5.10: Pairwise tournament: All algorithms.

	vs Elephant-DynaMOSA	
Algorithm	$\hat{A}_{12}$	p-value
% Relative Overall Coverage		
Standard GA	0.29	0.09
MOSA	0.50	0.19
DynaMOSA	0.64	0.17
Wolf	0.24	0.09
Bee	0.27	0.09
Whale	0.24	0.09
Cat	0.26	0.09
Elephant	0.32	0.15
Chicken	0.18	**0.05**
Moth	0.22	0.08
Fish	0.28	0.10
Algae	0.17	**0.05**
Particle	0.26	0.09
% Relative Mutation Score		
Standard GA	0.32	0.13
MOSA	0.52	0.25
DynaMOSA	0.62	0.23
Wolf	0.34	0.15
Bee	0.34	0.17
Whale	0.32	0.14
Cat	0.33	0.15
Elephant	0.34	0.16
Chicken	0.30	0.11
Moth	0.32	0.14
Fish	0.34	0.15
Algae	0.29	0.11
Particle	0.35	0.16

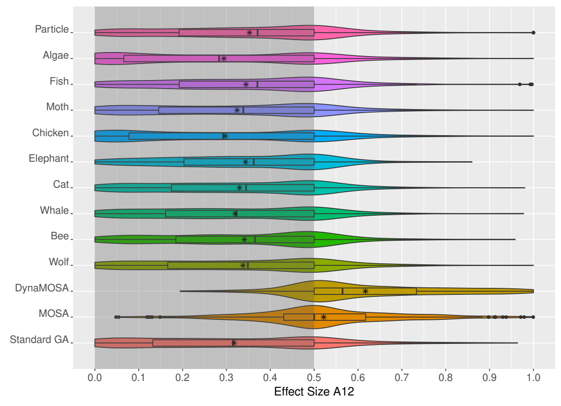
Table 5.11: All algorithms vs. Elephant-DynaMOSA.



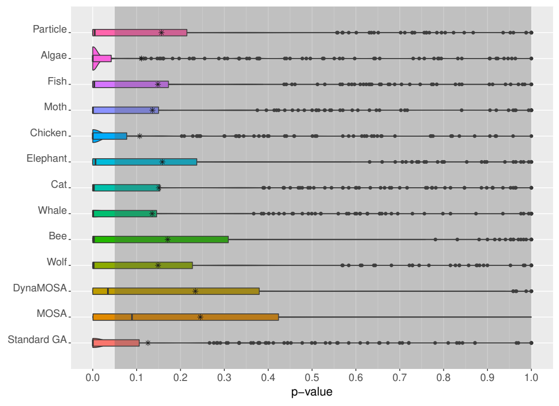
(a) Effect size (% Relative Overall Coverage).



(b) p-value (% Relative Overall Coverage).



(c) Effect size (% Relative Mutation Score).



(d) p-value (% Relative Mutation Score).

Figure 5.12: Distribution of effect sizes and p-values: All algorithms vs. Elephant-DynaMOSA.

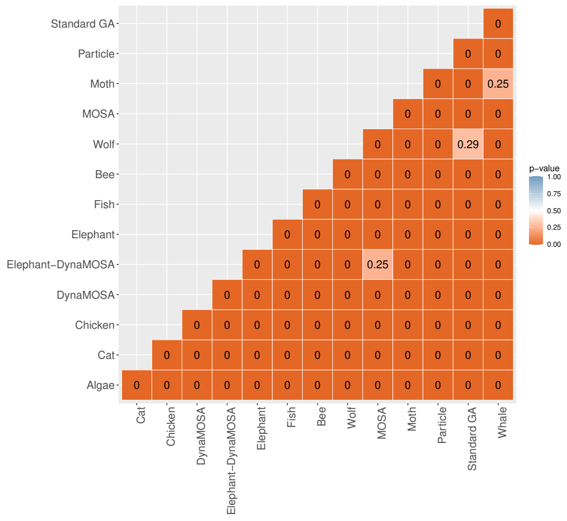
5.9 Summary

In this chapter, we saw how the experiments were set up, how they were performed and their results. We provided answers to the RQs presented in Section 5.1, report the average values of several metrics, and the statistical analysis described in Section 5.2.5.

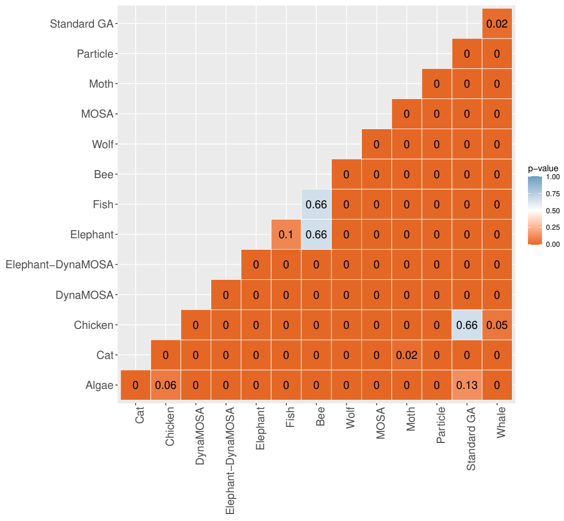
Table 5.12 reports another statistical method to rank the algorithms, the Friedman ranking test. It compares all algorithms at the same time and the lower the ranking value the better. As we have seen before, the MOSA, DynaMOSA, and the Elephant-DynaMOSA algorithms are the best ranking algorithms. There are some differences from the pairwise tournament ranking, e.g., MOSA is ranked higher than Elephant-DynaMOSA in overall coverage. These differences happen due to the way the two methods rank the algorithms: Vargha-Delaney compares two algorithms at a time while Friedman ranking test compares all algorithms at once (more information in Section 5.2.4).

Algorithm	Value	σ	CI	Rank	σ	CI
% Relative Overall Coverage ($\chi^2 = 1443.62$, p-value **< 0.01**)						
Standard GA	57.72	27.21	[54.82, 60.68]	8.31	4.63	[7.81, 8.83]
MOSA	71.75	21.63	[69.31, 74.31]	4.38	3.35	[4.01, 4.75]
DynaMOSA	78.48	19.38	[76.15, 80.62]	2.84	3.00	[2.51, 3.16]
Wolf	57.14	24.14	[54.50, 59.85]	8.25	2.70	[7.95, 8.57]
Bee	60.12	23.85	[57.67, 62.69]	7.57	2.72	[7.27, 7.86]
Whale	57.35	23.65	[54.80, 59.81]	9.04	2.64	[8.76, 9.33]
Cat	59.26	23.54	[56.74, 61.77]	7.99	2.39	[7.74, 8.27]
Elephant	64.38	22.58	[61.80, 67.00]	6.36	3.59	[5.96, 6.77]
Elephant-DynaMOSA	71.81	21.47	[69.73, 74.37]	4.48	3.63	[4.05, 4.90]
Chicken	45.28	27.71	[41.92, 48.43]	11.02	2.81	[10.72, 11.37]
Moth	54.47	24.89	[51.72, 57.20]	9.13	2.89	[8.80, 9.47]
Fish	60.10	24.41	[57.26, 62.82]	6.91	2.95	[6.57, 7.24]
Algae	43.55	28.70	[40.41, 46.51]	11.51	3.19	[11.17, 11.88]
Particle	58.83	23.66	[56.22, 61.40]	7.20	2.70	[6.91, 7.51]
% Relative Mutation Score ($\chi^2 = 1020.57$, p-value **< 0.01**)						
Standard GA	55.74	27.78	[52.58, 58.91]	9.44	4.33	[8.92, 9.94]
MOSA	68.65	25.07	[65.73, 71.36]	4.51	3.59	[4.09, 4.89]
DynaMOSA	73.55	23.47	[70.88, 75.99]	3.18	3.23	[2.81, 3.52]
Wolf	58.08	27.72	[55.20, 61.27]	7.58	3.11	[7.25, 7.94]
Bee	58.77	26.99	[55.89, 61.75]	7.92	2.98	[7.56, 8.24]
Whale	56.88	26.60	[54.23, 59.71]	9.25	2.75	[8.93, 9.58]
Cat	58.04	26.93	[54.94, 61.16]	8.45	2.80	[8.15, 8.77]
Elephant	59.32	26.53	[56.60, 62.40]	7.84	3.80	[7.38, 8.25]
Elephant-DynaMOSA	67.74	25.11	[65.07, 70.52]	4.75	3.50	[4.35, 5.13]
Chicken	53.54	30.32	[50.21, 56.95]	9.41	3.42	[9.02, 9.81]
Moth	56.92	28.29	[54.07, 60.08]	8.27	2.87	[7.96, 8.59]
Fish	58.41	27.08	[55.36, 61.64]	7.98	3.16	[7.62, 8.36]
Algae	53.01	31.07	[49.69, 56.47]	9.56	3.61	[9.17, 9.98]
Particle	59.07	27.33	[56.08, 62.27]	6.85	2.80	[6.54, 7.16]

Table 5.12: Friedman ranking test.



(a) % Relative Overall Coverage.



(b) % Relative Mutation Score.

Figure 5.13: Conover posthoc test's p-values of the Friedman ranking test.

Figure 5.13 shows the p-values of the Friedman test and we can see which comparisons are statistically relevant (i.e., p-value less than 0.05). This shows whether that the ranking of the algorithms is consistent throughout all classes, e.g., DynaMOSA is always better than Cat as the p-value is 0. If the p-value is 1, that means two algorithms are always changing the ranks between each other. In Figure 5.13, the cases that stand out the most are the three 0.66 values in mutation score, which show that the algorithms in the following pairs, Bee and Fish, Bee and Elephant, and Standard GA and Chicken, change the ranking among themselves quite frequently.

Although we have studied in detail the performance of bio-inspired algorithms vs. state-of-the-art algorithms, there are still several questions that need to be answered. In the following chapter, we are going to discuss several aspects and insights of our results that were not covered in this chapter: the performance of the algorithms throughout the execution, what makes an algorithm the best in some classes, generalisation of the results, among others.

Chapter 6

Discussion

In this chapter, we propose and discuss three hypotheses (one in each section). To answer these hypotheses we analysed the results of the previous chapter from another point of view.

In Section 6.1 we examine the reasons that allow an algorithm to reach its level of coverage in certain classes. While in Section 6.2 we explain the causes behind the limitations of the coverage values in several classes, regardless of the algorithm. Lastly, in Section 6.3 we interpret the results of a 3rd experiment with a larger budget, which show how all 14 algorithms evolve during their execution allowing us to see whether the results would be the same as in the previous study.

6.1 The algorithm with the highest overall coverage is best in all classes

In the previous chapter, we saw how DynaMOSA has the highest overall coverage and it is also statistically better than all the other algorithms. However, does that mean that DynaMOSA is the best in all classes? No, as we can see in Table 6.1 six algorithms are considered statistically better in a certain number of classes. As expected DynaMOSA is the algorithm with the highest number of classes. On the other hand, Standard GA which had a relatively poor performance is actually the 3rd algorithm with more classes and MOSA has a very low number of classes where it is statistically the best algorithm.

In Table B.1, we can see detailed information about the best class out of all classes selected in Table 6.1 per algorithm. This means that these classes are the ones where the algorithms have the best results in comparison to all the other algorithms. They are the classes that best highlight the algorithms in Table 6.1. Table B.1 has the name of the class, the number of generations, the number of test cases and their average size, the absolute and relative values of coverage, and the effect size and p-value. An effect size of 1 means the algorithm was the best in all criteria in all 30 runs.

In Figures C.1 to C.6, we can see all criteria used to measure the coverage, the number

Algorithm	# C	% Relative Overall Coverage	$\hat{A}_{12}$	p-value
Standard GA	7	57.72 (69.87)	0.89	**< 0.01**
MOSA	2	71.75 (72.96)	0.92	**< 0.01**
DynaMOSA	90	78.49 (74.91)	0.95	**< 0.01**
Elephant	1	64.38 (71.09)	0.92	**< 0.01**
Elephant-DynaMOSA	13	71.82 (72.99)	0.90	**< 0.01**
Fish	1	60.10 (69.61)	0.92	**< 0.01**

Table 6.1: # C represents the number of classes where the algorithms were statistically better than all others, the values between parentheses represents the absolute values, and $\hat{A}_{12}$ is the effect size of the algorithm in the selected classes.

of targets in each criterion (columns), the number of targets that were covered by more than half of the runs and how many runs the algorithm managed to covered the targets (colours). This allows us to see how each algorithm can reach the values of coverage in Table B.1 and what makes them the best or worse for the class. The higher the number of runs each algorithm is able to cover the targets in all criteria, the higher coverage will be. One cannot forget that all criteria have the same weight, so the value of each target takes that into account, i.e., the more targets a criterion has, the fewer coverage percentage those targets represent on an individual level.

In Figure C.1 we can see that Elephant-DynaMOSA is clearly the best algorithm since it is the algorithm with the most targets covered in at least 16 out of 30 runs in every single criterion. Looking at the colours we can also see that it is the algorithm that is the more consistent in all criteria. In Method and No Method Exception, it has similar performance to other algorithms and it is in the other criteria that the difference is clear. On the other hand, we can also identify Standard GA as the worst algorithm looking at both the colours and numbers beside the names.

In Figure C.5, we can see exactly why Fish, Standard GA, MOSA and DynaMOSA have the highest coverage and the likes of Algae and Chicken have the lowest values. Ignoring the few criteria where all algorithms have the same performance, we can see that Algae and Chicken are the algorithms we the lowest number of targets covered throughout the executions, in many cases having very low values. On the other hand, the four best algorithms are the ones with the highest number of targets covered in more than half the executions. Though there are several differences among them. E.g., DynaMOSA and MOSA have better performance in branch and cbranch (difference in the colours). While Standard GA and Fish are better in output. The reason why Fish is the best algorithm in this class is that it is the most consistent taking into account all criteria.

The algorithm with the highest overall coverage is best in all classes False, the algorithm with the highest overall coverage is not the best in all classes. There are a total of six algorithms that are statistically better than all others in certain classes.

6.2 Any algorithm can execute more than 50% of the classes

Ideally, to catch as many bugs in the source code as possible we want 100% coverage in all criteria. This means that the more code is executed by the test suites, the more likely it is to find and get rid of bugs. The algorithms used in this thesis were implemented to create test suites capable of reaching high coverage values in all kinds of classes.

However, out of all 308 classes used in the main experiment, there were 54 (17.5%) that regardless of the algorithm always had low absolute coverage. We can see in Figure 6.1 the classes that, regardless of the algorithm, have a maximum of 50% coverage. As we have already seen in Table 5.4, the algorithm with the lowest overall coverage achieved 65.89%, which is a much higher value than the mark of 50% used in this section.

So, why is it that the best algorithms for the classes presented in Figure 6.1 have significantly lower values than the lowest overall value?

One reason may be the limitations of the algorithms and/or operations used, e.g., a new algorithm could be able to explore more of the search space. Or the time budget was not enough for the algorithms to explore these 54 classes and reach higher percentages of coverage.

We also analysed the source of several of the classes present in Figure 6.1 and believe that the limitations of EvoSuite as a testing tool are also a reason for the low coverage. In the case of class *Predicates* from project *guava*, all that its methods do is create instances of its nested classes and interact with the enumerate that is inside the class. This shows that EvoSuite had trouble exploring the nested classes in detail, achieving coverage of around 40%. On the other hand, on class *ConsumerGUI* from project *water-simulator*, EvoSuite only achieved coverage of around 10%. With this class, we can see that EvoSuite has problems handling graphical user interfaces. The coverage value is not 0% due to exception coverage and method coverage criteria (EvoSuite caught the single exception and called the constructor). In class *WikipediaInfo* from project *wikipedia*, the majority of the public methods require at least one of the same two objects as parameters. The other public methods do not require parameters. From the fact that the coverage achieved is around 40%, we can assume that most of the methods were called. This allows the coverage of several lines of code at the beginning of the methods that are just the initialization of simple variables. However, once we look past those “easy” lines, we have calls for methods from project-specific classes. We believe that the reason for the low coverage is the fact EvoSuite is unable to execute these methods. Another case is class *MP3* from project *celwars2009*, where despite its simplicity, it only reached 40% coverage. Looking at the source code, we saw that the problem in this class is in the fact EvoSuite is not able to generate valid music files. EvoSuite can initialize the variables but cannot read

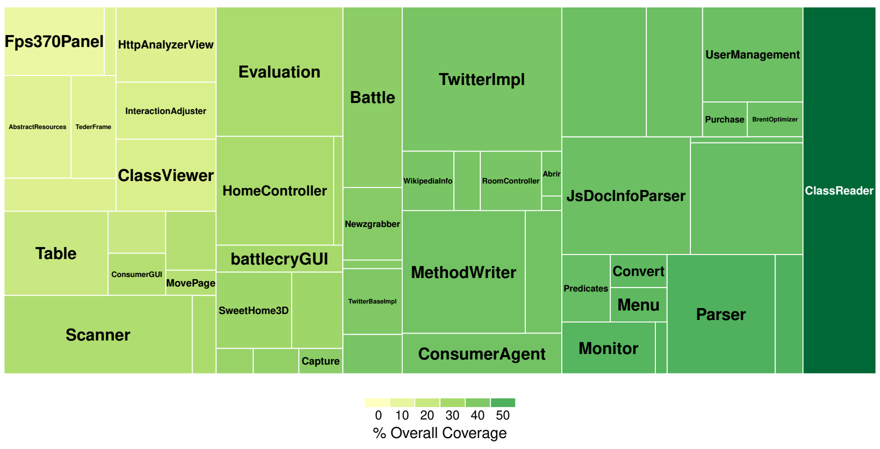


Figure 6.1: Classes with a maximum of 50% absolute coverage

the information of the music file created because its content is null. This causes both line coverage and branch coverage criteria to have low values. Nonetheless, due to its simplicity, several coverage criteria always have 100% coverage, e.g., method coverage (constructor initializes some variables with null values and calls the only other method) and output coverage (has no goals, so EvoSuite automatically assigns 100% coverage).

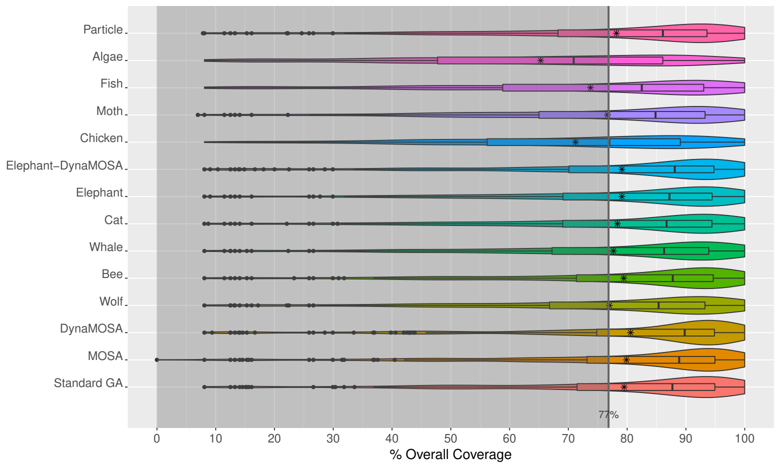
After this analysis, we can see that although EvoSuite is a state-of-the-art testing tool there is still room for improvement. These improvements can be extending EvoSuite so that it can generate more types of files or making EvoSuite able to work with graphical user interfaces.

Any algorithm can execute more than 50% of the classes: False, there are 54 classes where the algorithms could not achieve 50%. The reasons for this are the limitations of EvoSuite, the time budget was too small or the algorithms have trouble optimizing these classes in specific.

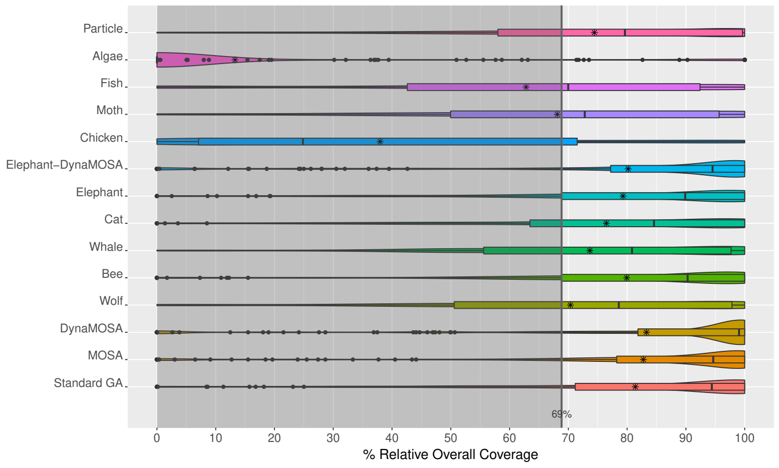
6.3 The performance of the algorithms increases with a larger budget

After we analysed the results in Chapter 5, we wondered if the values obtained would change if we had a different time budget. Is it possible to increase the performance if the algorithms had more time to optimize the solution? Maybe that is the reason why 100% coverage was not achieved.

So we decided to perform another experiment in which we had all 14 algorithms running their best configurations in all 312 classes during one hour. In this experiment, we decided to only use one seed since it takes a great amount of time to finish the execution and we did not have enough time to run several seeds.



(a) Distribution of overall coverage.



(b) Distribution of relative overall coverage.

Figure 6.2: Overall Coverage distribution by each algorithm in the one hour experiment. The grey line represents the mean of all algorithms, while the symbol * is the mean of each algorithm.

Looking at Figure 6.2, we can see the distribution of the overall coverage and relative coverage during the one hour experiment. In terms of absolute coverage, we have three outliers that are the only algorithms clearly below average: Algae, Chicken and Fish. All the other algorithms are around or above the average value. Comparing the average with Figure 5.2a from the comparison experiment, we can see an increase of around 7%. When it comes to the distribution of the relative coverage, it is very easy to see several differences among the algorithms. Algae has very low values, follow by Chicken. Fish and Moth are the other two algorithms below average. The algorithms that stand out with the highest values are Standard GA, DynaMOSA, MOSA and Elephant-DynaMOSA.

In Figure 6.3, the most noticeable aspect is the high-density values for the low relative coverage values by Algae and Chicken. Just like the previous experiment, most of the classes tested by these two algorithms ended up with low coverage values in comparison

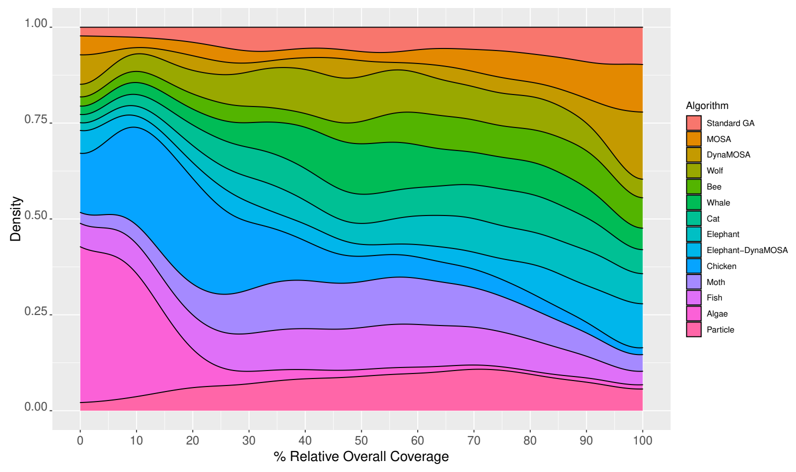


Figure 6.3: Density of the relative coverage in the one hour experiment.

to the other algorithms.

Table 6.2 shows the summary of the values obtained during the one hour experiment. Looking at the differences in absolute values between DynaMOSA (highest values) and Algae (lowest values), we obtained 15.3% and 15.35% for coverage and mutation score. If we look at the relative values, the difference is massive, with 70.01% for coverage and 49.42% for mutation score. Another thing of note is the fact that all algorithms except Algae and Chicken show an increase in the number and length of the test cases. Also, when it comes to the length Elephant-DynaMOSA has a much larger number than all the other algorithms.

Comparing Table 6.2 to Table 6.2 from the previous experiment, we can observe that the difference in relative values is clearly larger. This proves that the difference in the performance of the best and worst algorithms is much larger with the increase in time budget. Looking closer to the coverage, we can see that there are two major differences between the results: Standard GA is the 3rd algorithm with the highest coverage (was the 9th) and Fish ended up in 12th (was the 6th). With these coverage values is almost certain that the answers to the RQ2 would change since Standard GA is one of the algorithms with the highest coverage. As for the answer to RQ1, the best algorithm would be a close call between Elephant and Bee. Finally, the answer to RQ3 and RQ4 has a fair chance to change because DynaMOSA, MOSA, Standard GA, Elephant-DynaMOSA, Elephant and Bee have all achieved a relative coverage close to 80%. One interesting fact is that Algae has lower values during the one hour experiment than the comparison experiment. The reason behind this fact may also be the case of the two major differences between the results of the experiments. This reason is that in the experiment that ran each class for one hour a total of 54 classes were discarded (the comparison experiment only had 4). Perhaps the 50 extra classes excluded would increase the coverage achieved by the Fish and do the opposite for Standard GA. We conjecture these classes would increase

		% Relative Overall			% Relative Mutation				
Algorithm	# G	Coverage	σ	CI	Score	σ	CI	# T	L
Standard GA	57394	81.39 (79.50)	26.28	[78.15, 84.60]	66.91 (51.64)	34.03	[62.88, 71.28]	49	217
MOSA	7809	82.76 (79.89)	27.13	[79.78, 86.42]	73.92 (53.66)	32.06	[69.66, 77.69]	60	289
DynaMOSA	6832	83.29 (80.59)	29.44	[79.72, 86.90]	78.69 (55.24)	30.50	[75.17, 82.61]	60	277
Wolf	1288	70.35 (76.99)	29.16	[66.70, 73.86]	63.12 (50.73)	34.21	[59.34, 67.19]	44	169
Bee	1799	79.93 (79.42)	25.52	[76.96, 82.95]	69.23 (52.25)	32.46	[65.01, 73.33]	49	195
Whale	3148	73.65 (77.66)	26.64	[70.63, 76.94]	61.94 (49.86)	33.94	[57.99, 66.15]	45	187
Cat	2269	76.46 (78.35)	26.81	[73.26, 79.88]	65.36 (51.42)	33.06	[61.31, 69.74]	47	202
Elephant	9241	79.30 (79.11)	25.72	[76.31, 82.28]	66.16 (51.52)	33.32	[62.04, 70.38]	50	248
Elephant-DynaMOSA	15785	80.18 (79.11)	29.50	[76.70, 83.48]	75.74 (54.22)	29.72	[72.21, 79.37]	65	452
Chicken	2748	37.98 (71.22)	35.62	[33.53, 42.18]	40.43 (43.53)	40.04	[35.44, 45.02]	33	113
Moth	883	68.12 (76.67)	28.75	[64.51, 71.83]	59.90 (49.99)	35.30	[55.58, 64.45]	43	170
Fish	4680	62.82 (73.76)	32.87	[58.50, 67.18]	56.41 (47.86)	35.73	[52.27, 61.03]	44	183
Algae	1206	13.28 (65.29)	30.81	[9.66, 17.19]	29.27 (39.89)	40.59	[24.11, 34.25]	30	94
Particle	916	74.43 (78.18)	26.82	[71.05, 77.91]	65.45 (51.51)	31.81	[61.54, 69.32]	45	180

Table 6.2: *# G* represents the average number of generations, the values between parentheses represents the absolute values, *# T* represents the average number of test cases generated by each algorithm across all classes under test, and *L* represents the average length (i.e., number of lines) of the generated test cases.

the coverage value of Algae because it is very unusual for the performance to decrease with such a big increase in the time budget. Though more unlikely, there is a chance that the Algae is just not suitable for this task and ended up blocked in a local optimum. This reflects an inefficiency to explore the search space.

Analysis throughout the execution

To not only see the final values of the experiment, we decided to analyse the evolution of the values throughout the execution. Here we have the overall coverage and overall diversity values of all the algorithms. This not only allows us to see how the values evolved during the execution but also compare the difference in the evolution of the values in the first minute and the other 59 minutes.

In Figure 6.4 we can see the evolution of coverage, with the black points referring to the first minute and the red ones to time after that. All algorithms evolve the coverage the fastest in the black points and after that, there are some differences between the algorithms: Algae seems to not evolve any further, others show a small increase of around 5% (e.g., MOSA) and the final group are the algorithms with a substantial increase (e.g., Standard GA with almost 20%). This reflects the different capability between the algorithms to not be stuck in a local optimal solution and always try to reach a better solution.

Figure 6.5 also has different colours to distinguish the first minute from the rest. In the first minute, all algorithms except Elephant and Standard GA have a big variation in the diversity values. They share the same pattern between them, starting with very low values and at around 15 seconds a steep increase in values start until it reaches values close to 1 at around 30 seconds. On the other hand, Elephant and Standard GA start with

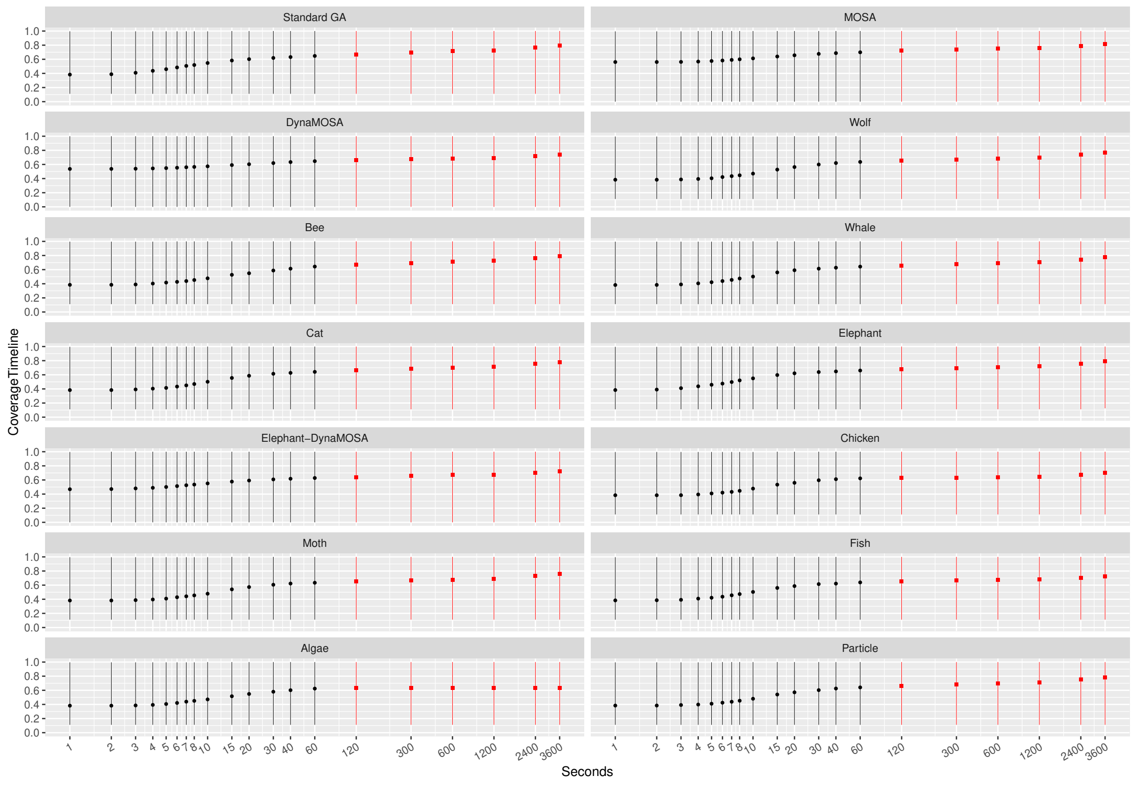


Figure 6.4: Evolution of the overall coverage in the one hour experiment.

higher values and then, show a slight increase and steep decrease, respectively. The reason for the difference in the starting points, even though the initial populations are the same between all algorithms, is the fact that only Elephant and Standard GA does not need several seconds to create the initial population. This happens because they have a smaller population number making the start much faster. For the remaining 59 minutes, we have several algorithms (e.g., Elephant) that maintain a similar value of diversity while others have a gradual decrease (e.g., Bee). Higher values of diversity mean the algorithm is favouring exploration over exploitation, while lower values mean the opposite. E.g., Standard GA selects the two best individuals per generation and uses them as references for all new individuals. This makes the population converge towards the best individuals. Elephant always maintains a high diversity since it replaces all members of the clan (except matriarch) with new random individuals. Thus, we can see the behaviours of the algorithms throughout the execution. One interesting fact is that in general, algorithms that focus the most on exploitation have better coverage values than the other algorithms. Moth and Elephant are the algorithms that go against this tendency.

The performance of the algorithms increases with a larger budget: True, almost all algorithms manage to obtain higher values with a larger budget. However, the increase in performance is small for most cases and the increase in resources (e.g., time) is very large, making it needless in most cases.

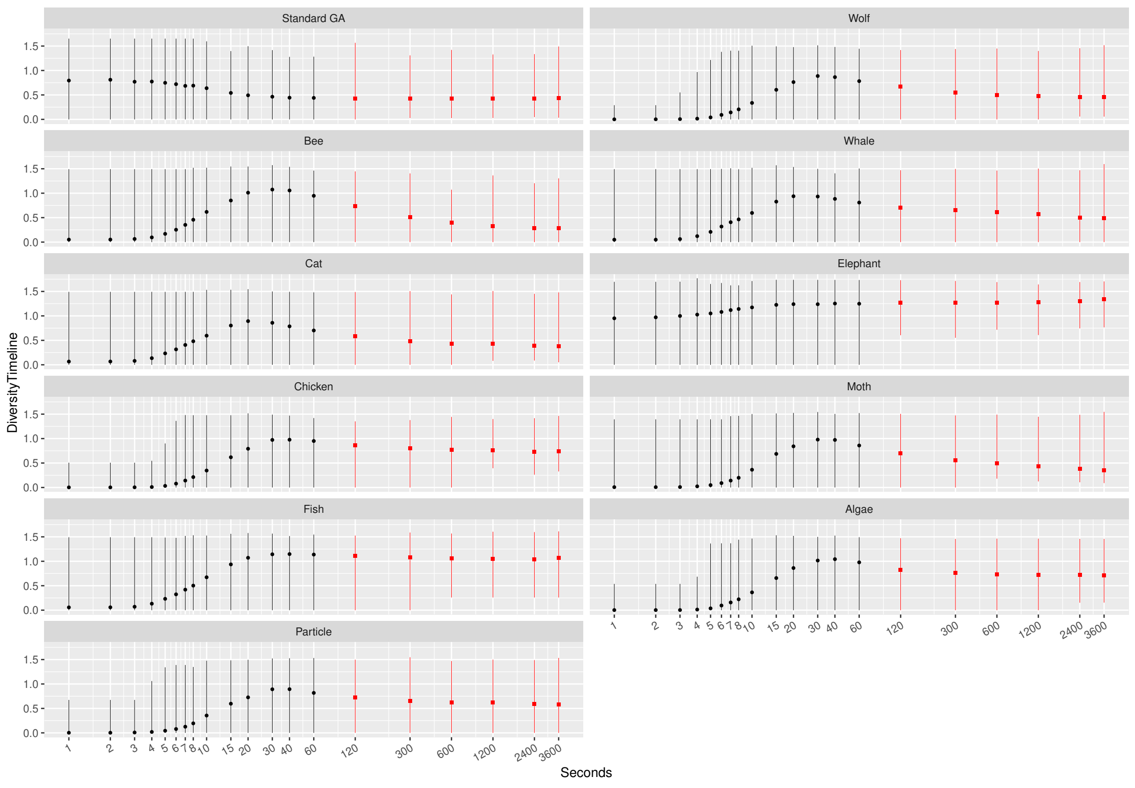


Figure 6.5: Evolution of the diversity in the one hour experiment. Due to the limitations of EvoSuite, we could not obtain the values of diversity of DynaMOSA, MOSA and Elephant-DynaMOSA.

6.4 Summary

In this chapter, we answered the three hypotheses we proposed at the beginning of the chapter. We explain why the best overall algorithm is not the best in all classes, we saw the reasons that make several classes have low coverage values regardless of the algorithm used and in the end, we analysed the values obtained in our 3rd experiment. This experiment allows us to see in more detail the impact the time budget has on the performance of the algorithms.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Meta-heuristic algorithms have several applications, e.g., SBST where these algorithms are responsible for the automatic generation of test suites optimized for code coverage. The EIAs were shown to be the best algorithms for SBST, being superior to random algorithms. However, there are groups of algorithms that remain largely unexplored in this context, e.g., the BIAs. This leads to the questions such as: How good are the BIAs in SBST?

The work described in this study focus on the implementation and evaluation of BIAs in SBST. The implementation of BIAs in EvoSuite required several adaptions in each algorithm. As for the evaluation, it consists of an empirical study divided into two experiments: tuning and comparison experiments. In the first one, the parameters of the algorithms were tweaked to find the best configuration for each algorithm. In the comparison experiment, the best configuration of each algorithm executed 312 classes for 60 seconds with 30 seeds.

The results obtained through the use of Vargha-Delaney and Friedman test statistical methods show that Elephant is the BIA with the best performance. And also that Standard GA is inferior in comparison to Elephant. Nonetheless, Elephant was not the best algorithm, as it was surpassed by both MOSA and DynaMOSA. As for the performance Elephant-DynaMOSA, it was similar to that of the MOSA and worse than the performance of DynaMOSA. With this DynaMOSA maintained the position of the state-of-the-art algorithm, as it was the algorithm with the best performance.

Afterwards, three hypotheses were proposed and discussed: The algorithm with the highest overall coverage is the best in all classes, any algorithm can execute more than 50% of the classes and the performance of the algorithms increases with a larger budget. DynaMOSA was not the best in all classes. In total, six algorithms were statistically better than all others in a certain number of classes. So, the first hypothesis is false. The second hypothesis is also false since 17.5% of classes achieved less than 50% coverage value.

One of the reasons is the fact that EvoSuite was unable to generate the required inputs to test the classes. To check the validity of the third hypothesis, another experiment was run where the best configurations of the algorithms were executed during one hour with one seed. The results of this experiment were that the average coverage increased by 7% and that the improvement of coverage past the 60 seconds mark was small in the majority of algorithms. With this, it was proven that the hypothesis is true but the number of extra resources necessary makes this improvement in performance unnecessary for most cases.

7.2 Future Work

In this study, BIAs showed some potential in SBST but this is just a small sample of BIAs. There are many BIAs that remain unexplored so we suggest adapting them to SBST and evaluate their performance.

Our results showed that the hybrid Elephant-DynaMOSA was superior to the BIAs and was among the top three algorithms. This hybrid combines the aspects of BIAs with MOAs, which leads us to suggest further experimentation with hybrid algorithms. Another observation taken is the fact that the best three algorithms consider test cases as individuals while the other algorithms use test suites as a representation of the individual. Perhaps optimizing algorithms using test cases as individuals might lead to better results.

In the future, we intend to evaluate the enhanced Multiple-Searching Genetic Algorithm which has already shown promising results in SBST, surpassing six GAs, Chemical Reaction Optimization and Random Search.

Looking at Section 6.1, where it is possible to see the performance of the algorithms per fitness criterion, we thought about seeing which algorithms are the best per criteria and create a multi-algorithm with those algorithms. The idea is for the multi-algorithm to select the algorithms within and adapt to the criteria chosen.

We also suggest future work in extending EvoSuite to solve several of the problems that make several classes impossible or very hard to get good coverage values, e.g., increase the number of types of files that EvoSuite can generate.

Appendix A

Details of the Java projects and classes used in the empirical evaluation

This table presents all 117 open-source Java projects and 346 Java classes used in this study. It shows the name of the project and class, whether the class was used in tuning (type equals training) or comparison experiment (type equals testing), the number of lines (column #L) and branches (column #B).

Project	Class	Type	# L	# B
tullibee	c.i.c.ExecutionFilter	testing	20	20
tullibee	c.i.c.ComboLeg	testing	23	21
water-simulator	s.C.g.ConsumerGUI	testing	163	59
water-simulator	s.C.ConsumerAgent	testing	301	137
jgaap	jgaapGUI	testing	237	23
netweaver	c.s.m.s.a.s.HeapInfo	testing	70	73
netweaver	c.s.m.s.a.s.J2EEApplicationAlias	testing	56	69
squirrel-sql	n.s.s.c.p.GlobalPreferencesSheet	testing	149	51
squirrel-sql	n.s.s.f.g.CascadeInternalFramePositioner	testing	31	21
sweethome3d	c.e.s.v.HomeController	testing	928	618
sweethome3d	c.e.s.v.RoomController	testing	283	154
sweethome3d	c.e.s.SweetHome3D	testing	374	228
sweethome3d	c.e.s.j.Room3D	testing	360	353
vuze	c.a.a.u.s.s.SWTSkinObjectContainer	testing	154	119
vuze	c.a.a.u.s.f.FeatureManagerUIListener	testing	196	134
freemind	f.e.ExportHook	training	76	29
freemind	a.p.t.JDayChooser	testing	400	208
weka	w.c.Memory	training	48	29
weka	w.c.Evaluation	testing	1378	809
weka	w.c.b.n.s.c.ICSSearchAlgorithm	testing	188	267
weka	w.c.FindWithCapabilities	testing	387	247
liferay	c.l.p.k.s.f.i.FilterMapping	testing	81	93
liferay	c.l.p.d.m.DLSyncWrapper	testing	128	78
pdfsam	j.HelpCmdLineHandler	training	56	41
pdfsam	o.p.g.b.t.c.JPodThmbnailCallable	testing	50	17
imsmart	c.i.s.HTMLFilter	testing	18	15
firebird	o.f.g.i.w.AbstractJavaGDSImpl	testing	1717	1040
firebird	o.f.j.FBProcedureCall	testing	117	98
firebird	o.f.e.EncodingFactory	testing	242	195
dsachat	d.s.Handler	testing	186	98

dsachat	d.c.g.InternalChatFrame	testing	189	69
jdbacl	o.d.j.SQLUtil	testing	251	188
jdbacl	o.d.j.DBUtil	testing	324	197
omjstate	u.m.j.j.s.Transition	testing	36	30
beanbin	n.s.b.r.MethodReflectionCriteria	testing	61	47
beanbin	n.s.b.r.ReflectionSearch	testing	52	41
inspirento	c.a.i.u.XmlElement	testing	156	95
inspirento	c.a.i.u.MainMenu	testing	54	27
jsecurity	o.j.w.f.a.BasicHttpAuthenticationFilter	testing	51	36
jsecurity	o.j.w.DefaultWebSecurityManager	training	97	67
jsecurity	o.j.u.AntPathMatcher	testing	172	170
jmca	c.s.C.J.J.JavaCharStream	testing	239	216
jmca	c.s.C.J.J.JavaParserTokenManager	training	981	1707
jmca	c.s.C.J.J.JavaParser	testing	4940	7938
jmca	c.s.C.J.JMCAAnalyzer	testing	222	199
a4j	n.k.a.D.Product	testing	62	31
a4j	n.k.a.f.FileUtil	training	357	125
geo-google	g.g.d.GeoStatusCode	testing	18	23
geo-google	g.g.m.AddressToUsAddressFunctor	testing	37	30
byuic	c.y.p.y.c.ScriptOrFnScope	testing	79	39
byuic	c.y.p.y.c.YUICompressor	training	127	87
byuic	c.y.p.y.c.JavaScriptCompressor	training	475	561
byuic	o.m.j.Parser	testing	1211	735
jwbf	n.s.j.m.a.e.MovePage	testing	64	52
jwbf	n.s.j.m.a.q.TemplateUserTitles	testing	39	45
saxpath	o.s.Axis	testing	43	55
saxpath	c.w.s.XPathLexer	testing	250	484
jipa	j.Main	testing	161	134
jipa	j.Variable	testing	37	23
gangup	g.AudioManager	testing	89	32
apbsmem	a.Main	training	815	388
apbsmem	j.p.PlotAxis	testing	88	41
bpmail	c.b.e.c.t.MessageList	testing	44	27
xisemele	n.s.x.i.OperationsHelperImpl	testing	31	27
httpanalyzer	h.HttpAnalyzerView	testing	960	200
httpanalyzer	h.Password	testing	120	56
javaviewcontrol	c.p.j.t.TokenMgrError	testing	35	32
javaviewcontrol	c.p.j.t.JVCParser	testing	355	223
javaviewcontrol	c.p.j.t.JVCParserTokenManager	testing	1350	2380
corina	c.f.TucsonSimple	testing	55	55
schemaspy	n.s.s.m.x.TableMeta	testing	27	17
schemaspy	n.s.s.m.Table	testing	438	380
petsoar	o.p.o.CreditCardInfo	testing	14	16
petsoar	o.p.s.l.DefaultLuceneDocumentFactory	testing	94	47
javabullboard	f.u.StringUtils	testing	148	98
javabullboard	f.u.j.JDBCUtils	testing	214	141
diffi	d.b.s.h.IndexedString	testing	30	20
diffi	d.b.s.StringIncrementor	testing	57	35
rif	c.d.r.s.t.WebServiceDescriptor	testing	31	21
glengineer	g.b.Block	testing	42	23
glengineer	g.a.GroupAgent	testing	157	115
follow	g.f.g.TabbedPane	testing	31	20
follow	g.f.s.SearchableTextPane	training	81	35
asphodel	o.a.DefaultRepositoryManager	testing	73	42
lilith	d.h.l.d.a.AccessEvent	training	86	134
summa	o.a.l.s.e.c.NamedCollatorComparator	testing	42	30

summa	o.a.l.s.e.ExposedTimSort	testing	376	372
summa	d.s.s.s.d.StatementHandler	training	84	108
summa	d.s.s.s.h.InteractionAdjuster	testing	373	279
lotus	l.c.p.Phase	testing	21	28
lotus	l.c.Game	testing	30	24
nutzenportfolio	c.b.e.n.c.AuswertungGrafik	testing	97	21
nutzenportfolio	c.b.e.n.s.f.NaOpNuDaoService	testing	174	48
dvd-homevideo	Capture	testing	148	29
dvd-homevideo	Convert	testing	223	52
dvd-homevideo	Menu	testing	185	84
resources4j	c.m.r.i.AbstractResources	testing	289	176
diebierse	b.m.Drink	testing	220	81
diebierse	b.c.DefaultSettingsController	testing	52	27
templateit	o.t.Region	testing	15	31
templateit	o.t.WorkbookParser	training	77	54
biff	Scanner	testing	1391	817
jiprof	c.m.t.p.r.Profile	testing	166	76
jiprof	o.o.a.j.c.LocalVariablesSorter	testing	110	87
jiprof	o.o.a.j.MethodWriter	testing	1084	824
jiprof	o.o.a.j.ClassReader	testing	891	817
lagoon	n.s.l.LagoonGUI	testing	195	63
lagoon	n.s.l.LagoonCLI	testing	144	65
db-everywhere	c.g.d.u.DBEHelper	testing	303	123
db-everywhere	c.g.d.s.SapdbTableList	training	26	15
lavalamp	n.s.l.d.TimeOfDay	testing	28	18
jhandballmoves	v.h.m.a.CreateMovePdfAction	testing	44	26
jhandballmoves	v.h.m.x.HandballModelReader	testing	93	31
hft-bomberman	c.n.ForwardingObserver	testing	29	13
hft-bomberman	s.ServerGameModel	testing	139	128
fps370	f.Fps370Panel	testing	412	151
fps370	t.TederFrame	testing	186	70
mygrid	m.w.Fail	testing	28	28
mygrid	m.w.AvailableJobsResponse	testing	28	28
sugar	n.s.s.f.FSPathResult	testing	37	24
sugar	n.s.s.f.c.FSPathExplorer	testing	75	51
noen	f.v.n.t.o.f.DaikonFormatter	testing	113	71
noen	f.v.n.m.b.p.s.ProbeInformation	training	40	67
objectexplorer	d.p.e.e.ExplorerFrameEventConverter	testing	240	175
objectexplorer	d.p.e.m.AttributeModelComparator	testing	12	17
jtailgui	f.p.j.g.a.IndexFileAction	testing	40	13
jtailgui	f.p.j.g.v.JTailPanel	testing	65	23
gsftp	c.g.f.FtpApplet	training	48	29
gsftp	m.s.SSHSCPGUIThread	testing	253	91
openjms	o.e.j.n.c.DefaultConnectionPool	testing	214	99
openjms	o.e.j.n.s.SocketRequestInfo	testing	54	34
biblestudy	b.o.Verse	testing	79	32
biblestudy	b.u.Queue	testing	62	37
lhamacaw	m.u.DisplayableListPanel	testing	203	70
lhamacaw	m.p.MacawWorkBench	testing	112	23
sfmis	c.h.s.c.Base64	testing	75	32
ext4j	n.s.e.l.l.ExtrasPatternParser	testing	14	11
ext4j	n.s.e.t.b.Request	testing	154	139
battlecry	b.bcGenerator	testing	284	281
battlecry	b.battlecryGUI	testing	312	78
fim1	o.o.s.a.u.UpdateUserPanel	training	223	73
fim1	o.o.s.c.u.u.FontChooserDialog	testing	61	25

fixsuite	o.f.m.Library	testing	51	35
fixsuite	o.f.m.v.TreeView	testing	155	72
openhre	c.b.a.User	testing	192	97
openhre	c.b.o.h.i.r.ExpressionImpl	testing	62	50
io-project	i.s.n.ClientGroup	testing	53	66
caloriecount	c.l.e.SimpleKeyListenerHelper	testing	23	25
caloriecount	c.l.s.c.SimpleComboBox	testing	43	26
caloriecount	c.l.i.DirectoryScanner	testing	307	232
twfbplayer	d.o.f.s.BattleStatistics	testing	236	156
twfbplayer	d.o.f.m.i.SimpleSector	testing	105	74
gfarcegestionfa	f.u.g.d.OracleIdentiteDao	testing	111	76
gfarcegestionfa	f.u.g.i.ModifTableStockage	testing	184	123
wheelwebtool	w.u.DynamicSelectModel	training	75	40
wheelwebtool	w.a.FieldWriter	testing	85	58
wheelwebtool	w.a.ClassReader	testing	868	817
wheelwebtool	w.a.ClassWriter	testing	345	174
javathena	o.j.l.p.FromClient	testing	60	49
javathena	o.j.l.UserManagement	testing	711	329
javathena	o.j.l.Login	testing	433	255
javathena	o.j.u.ConfigurationManagement	training	237	190
ipcalculator	i.BinaryCalculate	testing	243	103
ipcalculator	i.WhoIS	testing	229	55
xbus	n.s.x.b.c.MessageHandler	testing	71	27
xbus	n.s.x.b.c.r.XBUSClassLoader	testing	29	17
ifx-framework	o.s.i.b.IFXObject	testing	94	72
shop	u.c.s.JSState	testing	80	41
shop	u.c.s.JSPredicateForm	testing	165	87
shop	u.c.s.JSTerm	testing	318	192
shop	u.c.s.JSJshop	testing	225	114
at-robots2-j	n.v.a.g.r.RobotRenderer	testing	83	37
at-robots2-j	n.v.a.r.Robot	testing	212	123
jaw-br	j.e.Salvar	testing	61	46
jaw-br	j.e.Abrir	testing	50	26
jopenchart	d.p.c.DefaultChart	testing	50	20
jopenchart	d.p.c.CoordSystemUtilities	testing	196	92
jiggler	j.i.o.l.Clip	testing	67	33
jiggler	j.i.o.Shift	testing	57	56
jiggler	j.i.o.ConnectedComponents	testing	75	74
jiggler	j.i.l.LevelSetNudge	testing	162	118
dcparseargs	d.d.p.ArgsParser	testing	83	80
classviewer	c.j.v.ClassViewer	testing	599	235
classviewer	c.j.v.SAXDirParser	training	87	60
classviewer	c.j.v.ClassInfo	testing	209	168
jcvi-javacommon	o.j.j.a.c.p.DefaultPhdReadTag	testing	61	61
jcvi-javacommon	o.j.j.a.c.f.Distance	testing	51	28
quickserver	o.q.n.s.i.BlockingClientHandler	testing	327	228
quickserver	o.q.n.s.i.NonBlockingClientHandler	testing	723	501
quickserver	o.q.u.x.QuickServerConfig	testing	96	69
quickserver	o.q.n.c.m.HostMonitoringService	training	131	74
jclo	e.m.c.j.JCLO	testing	183	133
celwars2009	MP3	testing	30	11
celwars2009	Client	training	836	350
heal	o.h.u.InterfaceUtilitiesBean	testing	99	38
heal	o.h.m.s.AdvSearchDAO	testing	266	175
feudalismgame	s.VassalRebellion	testing	16	11
feudalismgame	s.Purchase	testing	73	56

feudalismgame	s.Battle	testing	730	788
trans-locator	J.F.FoxHuntFrame	testing	127	34
trans-locator	J.F.HuntDisplay	testing	50	23
newzgrabber	N.BatchDriver	testing	90	32
newzgrabber	Newzgrabber	testing	222	78
newzgrabber	N.Downloader	training	339	268
checkstyle	c.p.t.c.a.AbstractLoader	testing	27	9
checkstyle	c.p.t.c.a.AutomaticBean	testing	76	18
checkstyle	c.p.t.c.a.FileContents	testing	79	50
checkstyle	c.p.t.c.a.FileText	testing	82	26
checkstyle	c.p.t.c.u.AnnotationUtility	testing	32	28
checkstyle	c.p.t.c.u.ScopeUtils	testing	92	110
commons-cli	o.a.c.c.HelpFormatter	testing	198	144
commons-cli	o.a.c.c.Option	training	110	98
commons-codec	o.a.c.c.l.DoubleMetaphone	testing	404	498
commons-collections	o.a.c.c.p.ArrayByteList	testing	66	28
commons-collections	o.a.c.c.l.TreeList	testing	359	219
commons-collections	o.a.c.c.s.SequencesComparator	testing	82	89
commons-lang	o.a.c.l.t.t.NumericEntityUnescaper	testing	39	48
commons-lang	o.a.c.l.ClassUtils	testing	279	266
commons-lang	o.a.c.l.ArrayUtils	testing	1263	1163
commons-lang	o.a.c.l.t.ExtendedMessageFormat	testing	183	138
commons-lang	o.a.c.l.t.StrBuilder	testing	747	589
commons-lang	o.a.c.l.t.FastDateFormat	testing	45	40
commons-lang	o.a.c.l.LocaleUtils	testing	81	99
commons-lang	o.a.c.l.t.DurationFormatUtils	testing	239	159
commons-lang	o.a.c.l.t.DateUtils	testing	299	298
commons-lang	o.a.c.l.BooleanUtils	testing	187	265
commons-lang	o.a.c.l.b.CompareToBuilder	testing	253	249
commons-lang	o.a.c.l.Validate	testing	138	134
commons-lang	o.a.c.l.Conversion	testing	521	764
commons-lang	o.a.c.l.b.HashCodeBuilder	testing	148	116
commons-math	o.a.c.m.f.ProperFractionFormat	testing	63	26
commons-math	o.a.c.m.o.d.CMAESOptimizer	testing	460	266
commons-math	o.a.c.m.d.HypergeometricDistribution	testing	66	38
commons-math	o.a.c.m.l.RectangularCholeskyDecomposition	testing	56	31
commons-math	o.a.c.m.o.n.EmbeddedRungeKuttaIntegrator	testing	108	62
commons-math	o.a.c.m.o.AbstractIntegrator	testing	124	82
commons-math	o.a.c.m.g.e.t.Rotation	testing	316	123
commons-math	o.a.c.m.u.MultidimensionalCounter	testing	68	41
commons-math	o.a.c.m.o.g.LevenbergMarquardtOptimizer	testing	371	207
commons-math	o.a.c.m.o.l.SimplexTableau	testing	167	149
commons-math	o.a.c.m.f.Fraction	testing	155	114
commons-math	o.a.c.m.u.MathUtils	training	40	66
commons-math	o.a.c.m.a.i.TricubicSplineInterpolatingFunction	testing	102	80
commons-math	o.a.c.m.o.n.s.n.AbstractSimplex	training	87	59
commons-math	o.a.c.m.l.MatrixUtils	testing	253	159
commons-math	o.a.c.m.g.e.o.IntervalsSet	testing	66	52
commons-math	o.a.c.m.d.DfpDec	training	134	138
commons-math	o.a.c.m.d.f.MultivariateNormalMixtureExpectationMaximization	testing	119	66
commons-math	o.a.c.m.a.FunctionUtils	testing	214	157
commons-math	o.a.c.m.o.u.BrentOptimizer	testing	113	75
commons-math	o.a.c.m.l.SchurTransformer	testing	189	92
compiler	c.g.j.j.CheckGlobalThis	testing	56	73
compiler	c.g.j.j.ExploitAssigns	testing	80	87
compiler	c.g.j.j.CollapseProperties	testing	365	273

compiler	c.g.j.j.PeepholeSubstituteAlternateSyntax	testing	741	615
compiler	c.g.j.j.ControlFlowAnalysis	testing	402	413
compiler	c.g.j.j.ScopedAliases	training	197	140
compiler	c.g.j.r.j.RecordType	testing	107	79
compiler	c.g.j.j.p.JsDocInfoParser	testing	1042	757
compiler	c.g.j.j.ReferenceCollectingCallback	training	204	196
guava	c.g.c.m.BigIntegerMath	testing	139	133
guava	c.g.c.c.CacheBuilderSpec	testing	142	141
guava	c.g.c.u.c.Monitor	testing	248	191
guava	c.g.c.b.Joiner	testing	94	60
guava	c.g.c.b.Predicates	testing	140	107
guava	c.g.c.b.SmallCharMatcher	testing	46	27
guava	c.g.c.b.Splitter	testing	111	86
guava	c.g.c.b.Suppliers	testing	71	45
guava	c.g.c.b.Utf8	testing	57	63
guava	c.g.c.b.Objects	training	52	37
guava	c.g.c.b.CharMatcher	testing	347	325
hibernate	o.h.s.u.l.i.LoggerFactory	testing	9	3
javaml	n.s.j.c.AbstractInstance	testing	65	38
javaml	n.s.j.c.Complex	testing	33	12
javaml	n.s.j.c.DefaultDataset	testing	95	52
javaml	n.s.j.c.DenseInstance	testing	76	49
javaml	n.s.j.c.Fold	testing	61	48
javaml	n.s.j.c.SparseInstance	training	104	56
javaml	n.s.j.t.d.ARFFHandler	testing	29	16
javex	o.j.Expression	testing	225	173
jdom	o.j.u.NamespaceStack	testing	140	84
jdom	o.j.t.JDOMResult	testing	123	50
jdom	o.j.o.SAXOutputter	testing	138	90
jdom	o.j.o.XMLOutputter	testing	164	62
jdom	o.j.Verifier	testing	236	277
jfree-chart	o.j.c.r.c.AbstractCategoryItemRenderer	testing	422	216
jfree-chart	o.j.d.c.DefaultIntervalCategoryDataset	testing	143	103
jfree-chart	o.j.c.p.MultiplePiePlot	testing	182	93
jfree-chart	o.j.d.t.TimeSeries	testing	271	157
jfree-chart	o.j.d.g.DatasetUtilities	testing	507	335
jfree-chart	o.j.c.p.ValueMarker	testing	22	13
jfree-chart	o.j.c.r.c.MinMaxCategoryRenderer	testing	129	60
jfree-chart	o.j.c.r.GrayPaintScale	testing	25	15
jfree-chart	o.j.c.r.c.StatisticalBarRenderer	testing	160	79
jfree-chart	o.j.c.a.Axis	testing	286	156
jfree-chart	o.j.c.p.XYPlot	testing	1353	822
jfree-chart	o.j.d.t.TimePeriodValues	testing	149	74
joda	o.j.t.b.BaseSingleFieldPeriod	testing	69	53
joda	o.j.t.t.ZoneInfoCompiler	testing	382	232
joda	o.j.t.f.PeriodFormatterBuilder	testing	665	579
joda	o.j.t.f.DateTimeFormatterBuilder	testing	1026	746
joda	o.j.t.DateTimeZone	testing	349	205
joda	o.j.t.MutableDateTime	testing	237	140
joda	o.j.t.Partial	testing	249	134
joda	o.j.t.Period	testing	281	129
joda	o.j.t.f.DateTimeFormatter	testing	208	113
joda	o.j.t.b.BasePeriod	testing	173	79
joda	o.j.t.c.BasicMonthOfYearDateTimeField	testing	110	63
joda	o.j.t.c.LimitChronology	testing	236	112
joda	o.j.t.MutablePeriod	testing	147	76

jsci	J.m.s.SimpleCharStream	testing	174	84
jsci	J.m.s.ExpressionParser	testing	565	435
jsci	J.m.SpecialMath	testing	321	196
jsci	J.m.LinearMath	training	584	280
scribe	o.s.m.OAuthConfig	testing	15	12
scribe	o.s.m.OAuthRequest	testing	10	8
scribe	o.s.m.Request	testing	73	37
scribe	o.s.m.Response	testing	21	15
scribe	o.s.m.Token	testing	13	7
scribe	o.s.m.Verifier	testing	5	2
tartarus	o.t.s.e.turkishStemmer	testing	1008	514
tartarus	o.t.s.e.italianStemmer	testing	358	228
tartarus	o.t.s.e.englishStemmer	testing	417	290
trove	g.t.d.TDoubleShortMapDecorator	testing	90	77
trove	g.t.d.TShortByteMapDecorator	testing	90	77
trove	g.t.i.h.TCharHash	training	96	60
trove	g.t.i.h.TFloatCharHash	testing	142	87
trove	g.t.i.h.TFloatDoubleHash	testing	142	87
trove	g.t.i.h.TShortHash	testing	96	60
trove	g.t.l.l.TDoubleLinkedList	testing	466	281
trove	g.t.m.h.TFloatObjectHashMap	testing	384	254
trove	g.t.m.h.TByteObjectHashMap	testing	384	254
trove	g.t.m.h.TByteFloatHashMap	testing	476	297
twitter4j	t.ExceptionDiagnosis	testing	37	27
twitter4j	t.GeoQuery	testing	76	65
twitter4j	t.Paging	testing	104	54
twitter4j	t.TwitterException	testing	129	123
twitter4j	t.TwitterBaseImpl	testing	198	131
twitter4j	t.OEmbedRequest	testing	81	69
twitter4j	t.TwitterImpl	testing	442	319
wikipedia	d.t.u.w.a.Title	testing	41	19
wikipedia	d.t.u.w.a.CategoryDescendantsIterator	testing	54	27
wikipedia	d.t.u.w.a.WikipediaInfo	testing	155	72
wikipedia	d.t.u.w.a.CycleHandler	testing	43	21
xmlenc	o.z.x.XMLChecker	training	102	1213
xmlenc	o.z.x.XMLEncoder	testing	119	182

Appendix B

Class under test for each algorithm X performed statistically better than any another algorithm

Algorithm	# G	# T	# L	% Relative Overall Coverage	$\hat{A}_{12}$	p-value
			org.apache.commons.lang3.Conversion			
Standard GA	51	149	275	19.75 (67.30)	—	—
MOSA	29	336	3749	71.53 (83.19)	—	—
DynaMOSA*	31	426	1491	92.51 (89.63)	1.00	**< 0.01**
Wolf	5	266	500	69.32 (82.51)	—	—
Bee	4	234	440	58.29 (79.13)	—	—
Whale	15	228	429	56.01 (78.43)	—	—
Cat	12	240	452	59.94 (79.63)	—	—
Elephant	14	188	350	38.99 (73.20)	—	—
Elephant-DynaMOSA	22	435	3608	72.51 (83.49)	—	—
Chicken	8	253	477	65.80 (81.43)	—	—
Moth	8	265	496	68.44 (82.24)	—	—
Fish	22	256	477	64.72 (81.10)	—	—
Algae	4	253	476	65.72 (81.41)	—	—
Particle	6	272	509	70.80 (82.97)	—	—
			weka.core.FindWithCapabilities			
Standard GA	58	42	152	17.02 (71.12)	—	—
MOSA	32	56	244	36.75 (74.75)	—	—
DynaMOSA	25	68	322	58.93 (78.82)	—	—
Wolf	3	56	243	44.88 (76.24)	—	—
Bee	3	51	207	28.83 (73.29)	—	—
Whale	12	51	204	32.51 (73.97)	—	—
Cat	8	52	212	31.55 (73.79)	—	—
Elephant	22	57	248	41.65 (75.65)	—	—
Elephant-DynaMOSA*	106	76	397	83.92 (83.41)	0.99	**< 0.01**
Chicken	6	54	233	40.41 (75.42)	—	—
Moth	6	56	243	44.16 (76.11)	—	—
Fish	13	56	236	40.97 (75.52)	—	—

Algae	3	55	234	40.36 (75.41)	—	—
Particle	4	57	250	48.70 (76.94)	—	—
net.sourceforge.jwbf.mediawiki.actions.queries.TemplateUserTitles						
Standard GA*	1553	20	131	97.49 (89.27)	0.93	**< 0.01**
MOSA	70	18	85	89.75 (83.36)	—	—
DynaMOSA	75	17	77	91.64 (84.80)	—	—
Wolf	27	15	85	89.83 (83.42)	—	—
Bee	34	18	103	94.87 (87.27)	—	—
Whale	50	14	83	88.40 (82.33)	—	—
Cat	63	17	104	92.41 (85.39)	—	—
Elephant	231	16	87	90.32 (83.79)	—	—
Elephant-DynaMOSA	262	16	76	91.25 (84.50)	—	—
Chicken	24	4	16	26.69 (35.23)	—	—
Moth	15	9	51	70.50 (68.67)	—	—
Fish	183	8	43	50.62 (53.50)	—	—
Algae	11	2	5	0.00 (14.86)	—	—
Particle	14	12	65	82.34 (77.70)	—	—
geo.google.mapping.AddressToUsAddressFunctor						
Standard GA	628	8	98	86.43 (87.87)	—	—
MOSA*	178	11	134	93.36 (90.87)	0.93	**< 0.01**
DynaMOSA	115	10	119	86.48 (87.89)	—	—
Wolf	21	4	26	24.26 (60.93)	—	—
Bee	24	7	76	74.05 (82.50)	—	—
Whale	64	6	53	60.14 (76.47)	—	—
Cat	58	6	54	55.88 (74.63)	—	—
Elephant	234	7	70	65.75 (78.91)	—	—
Elephant-DynaMOSA	438	7	75	64.61 (78.41)	—	—
Chicken	23	3	17	5.99 (53.01)	—	—
Moth	16	4	23	15.09 (56.95)	—	—
Fish	192	6	46	52.00 (72.95)	—	—
Algae	13	3	16	5.27 (52.69)	—	—
Particle	19	5	44	44.74 (69.80)	—	—
com.puppycrawl.tools.checkstyle.utils.AnnotationUtility						
Standard GA	330	14	28	79.13 (64.51)	—	—
MOSA	116	14	29	80.58 (64.65)	—	—
DynaMOSA	130	15	30	82.49 (64.84)	—	—
Wolf	10	11	18	15.17 (58.26)	—	—
Bee	14	12	22	49.60 (61.63)	—	—
Whale	33	12	21	34.17 (60.12)	—	—
Cat	31	12	20	31.63 (59.87)	—	—
Elephant	218	13	24	56.65 (62.31)	—	—
Elephant-DynaMOSA	380	12	23	31.01 (59.81)	—	—
Chicken	13	10	16	1.25 (56.91)	—	—
Moth	10	11	17	12.73 (58.03)	—	—
Fish*	59	14	29	88.88 (65.46)	0.92	**< 0.01**
Algae	10	10	16	1.25 (56.91)	—	—
Particle	10	12	20	33.36 (60.04)	—	—

org.apache.commons.math3.optim.univariate.BrentOptimizer						
Standard GA	426	13	54	86.20 (46.01)	—	—
MOSA	44	14	35	80.68 (45.68)	—	—
DynaMOSA	50	14	35	82.81 (45.81)	—	—
Wolf	12	10	29	60.88 (44.49)	—	—
Bee	9	11	34	68.69 (44.96)	—	—
Whale	44	12	47	75.76 (45.38)	—	—
Cat	27	11	34	69.40 (45.00)	—	—
Elephant*	126	13	92	90.65 (46.28)	0.92	**< 0.01**
Elephant-DynaMOSA	184	13	35	81.38 (45.72)	—	—
Chicken	11	7	19	8.33 (41.34)	—	—
Moth	11	9	24	43.69 (43.46)	—	—
Fish	38	12	50	75.33 (45.36)	—	—
Algae	7	8	18	6.24 (41.21)	—	—
Particle	10	11	34	69.05 (44.98)	—	—

Table B.1: # G represents the average number of generations, the values between parentheses represents the absolute values, # T represents the average number of test cases generated by each algorithm across all classes under test, and L represents the average length (i.e., number of lines) of the generated test cases. The best algorithm per class has a * in front of the name.

Appendix C

Detailed coverage achieved by each algorithm as a heatmap

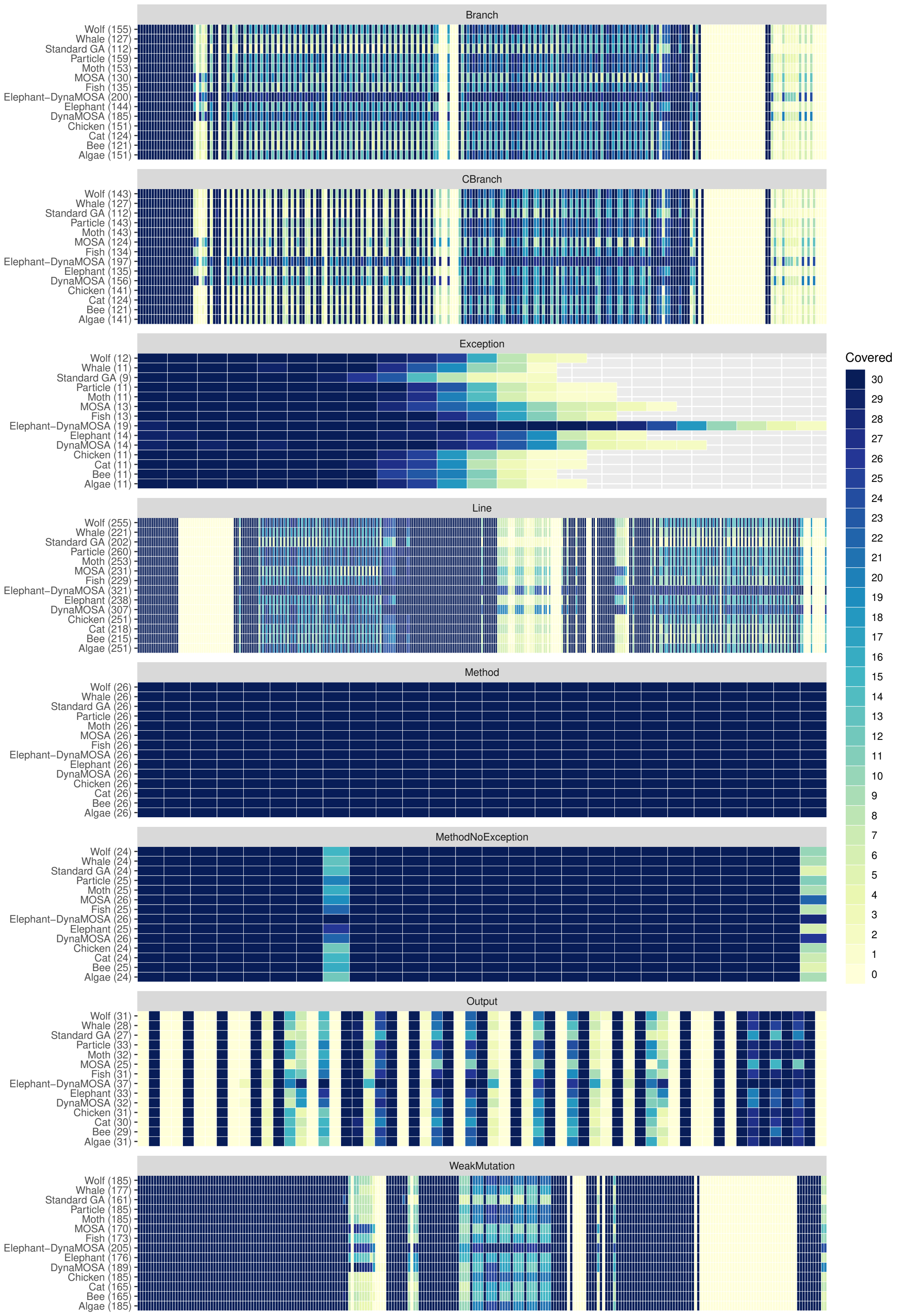


Figure C.1: Heatmap of class weka.core.FindWithCapabilities. The number besides the algorithm is the number of occurrences where the targets were covered by more than 15 runs. The best algorithm is Elephant-DynaMOSA.

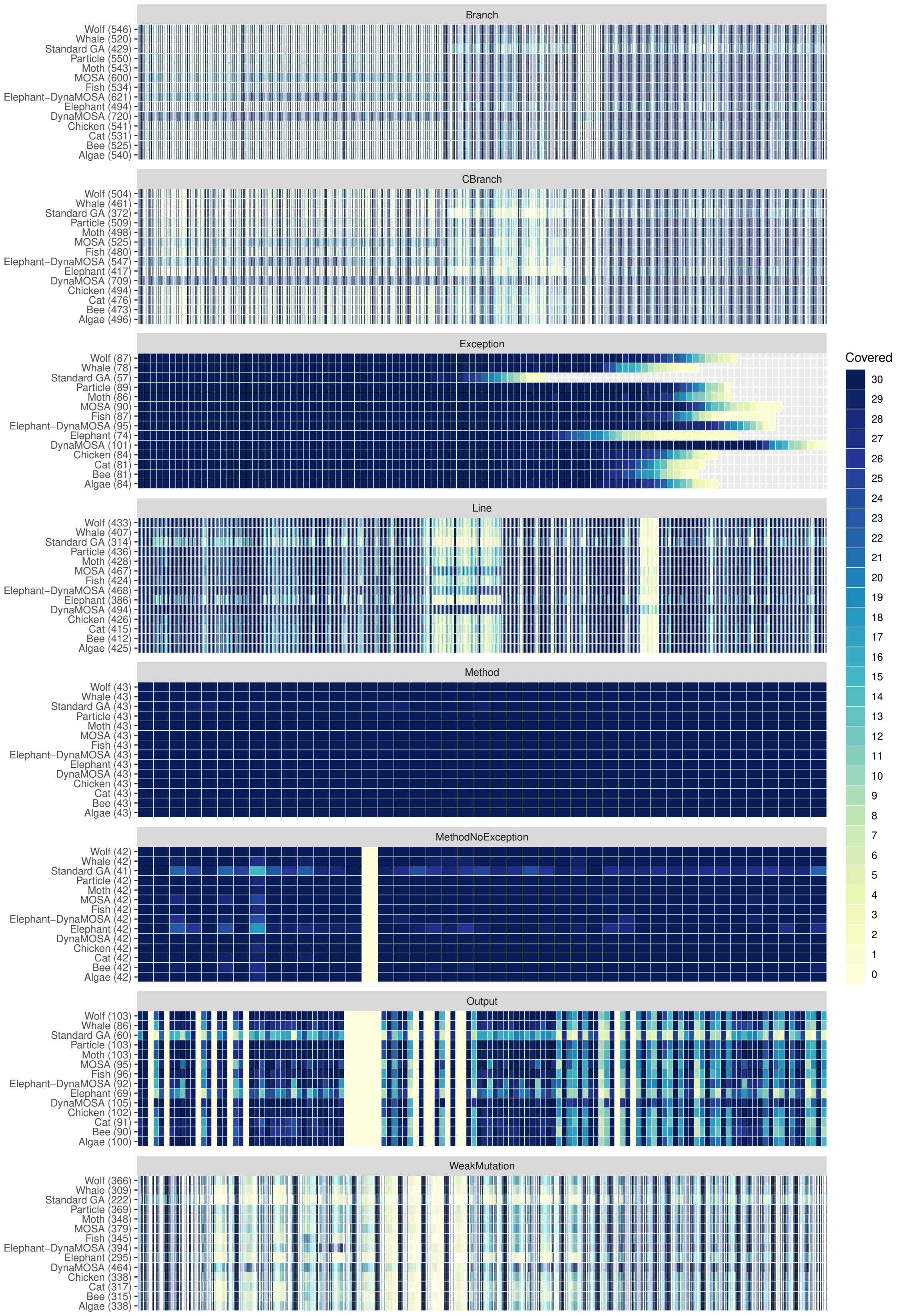


Figure C.2: Heatmap of class org.apache.commons.lang3.Conversion. The number besides the algorithm is the number of occurrences where the targets were covered by more than 15 runs. The best algorithm is DynaMOSA.

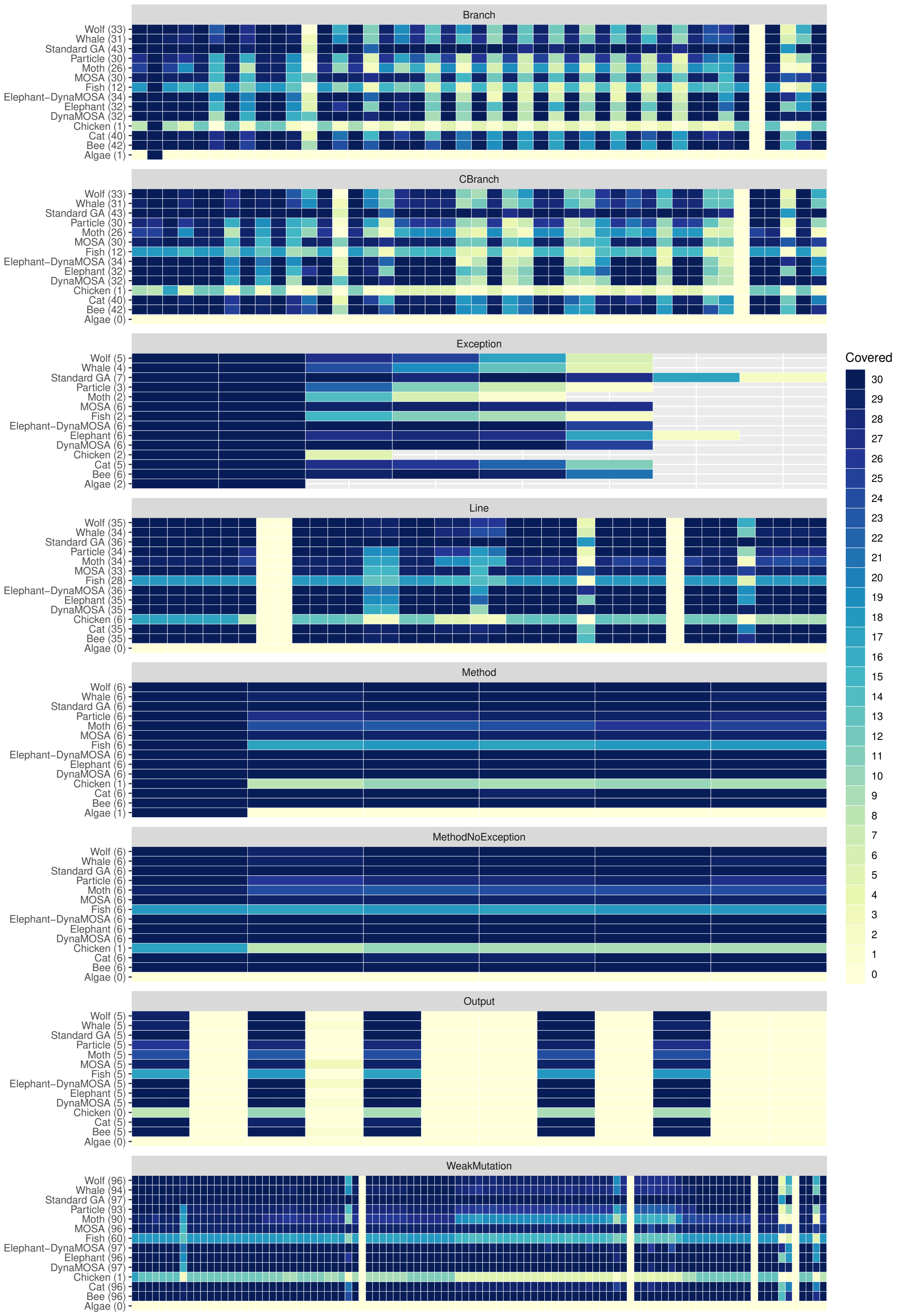


Figure C.3: Heatmap of class net.sourceforge.jwbf.mediawiki.actions.queries.TemplateUserTitles. The number besides the algorithm is the number of occurrences where the targets were covered by more than 15 runs. The best algorithm is Standard GA.

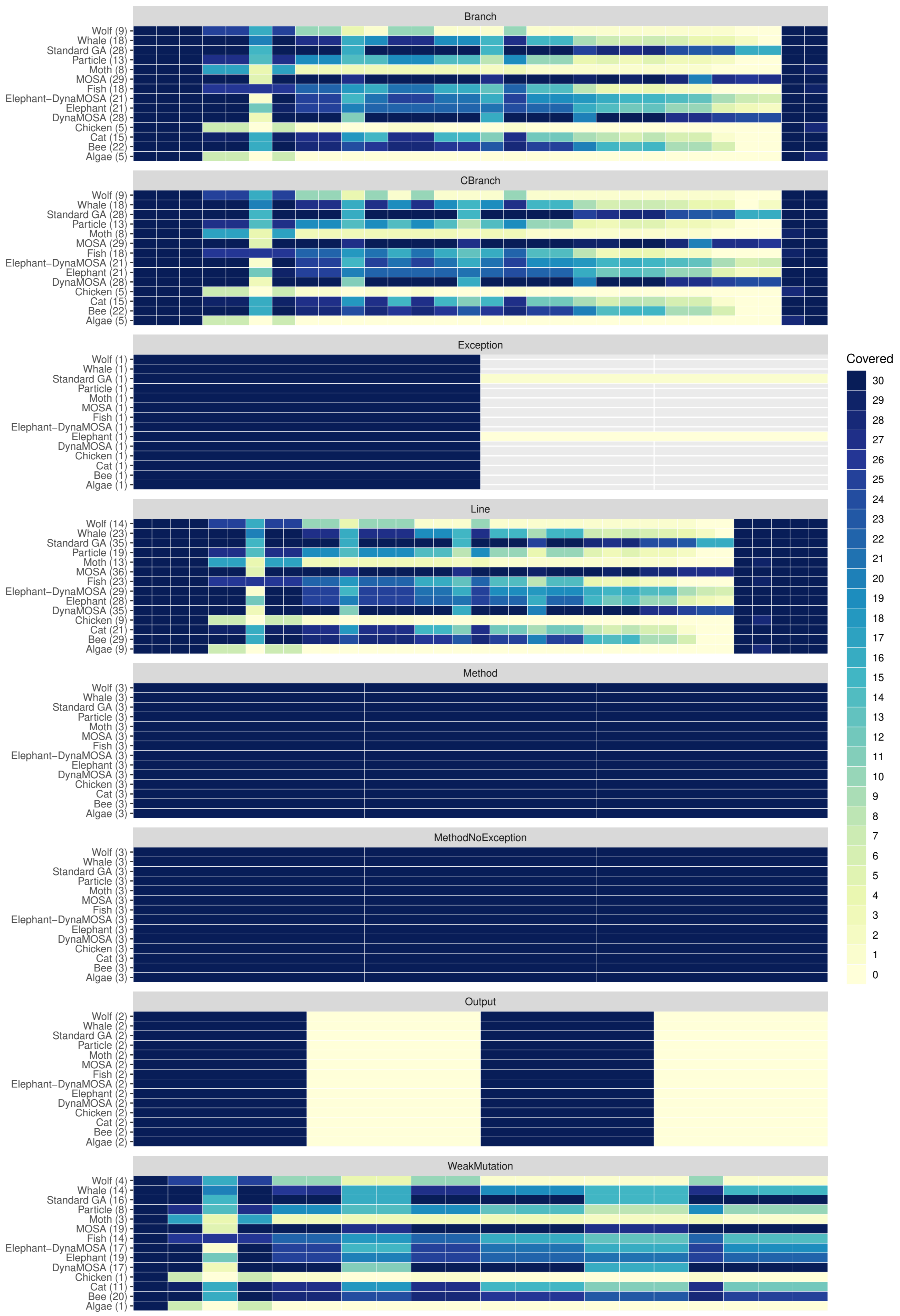


Figure C.4: Heatmap of class geo.google.mapping.AddressToUsAddressFunctor. The number besides the algorithm is the number of occurrences where the targets were covered by more than 15 runs. The best algorithm is MOSA.

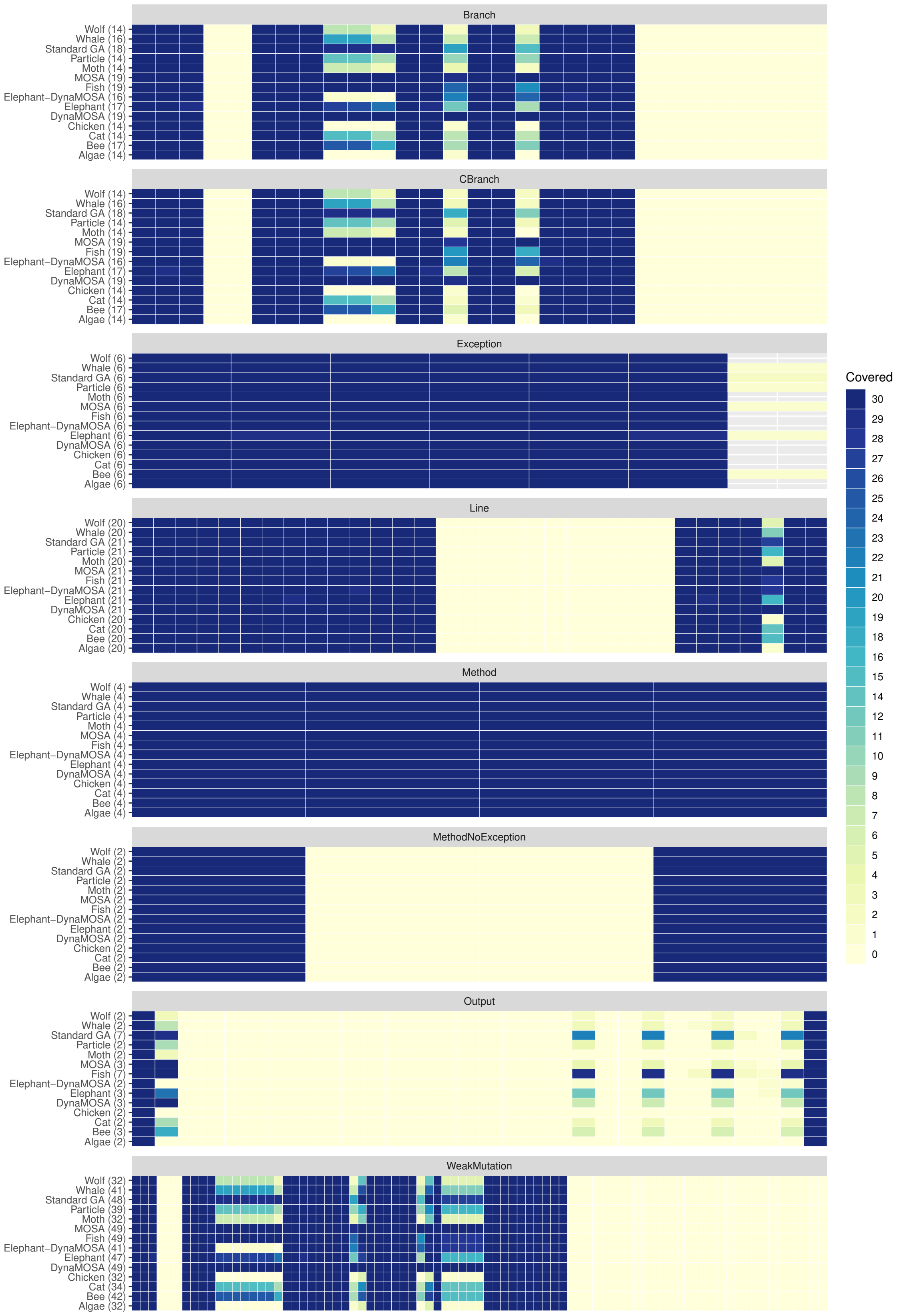


Figure C.5: Heatmap of class com.puppycrawl.tools.checkstyle.utils.AnnotationUtility. The number besides the algorithm is the number of occurrences where the targets were covered by more than 15 runs. Best algorithm is Fish.

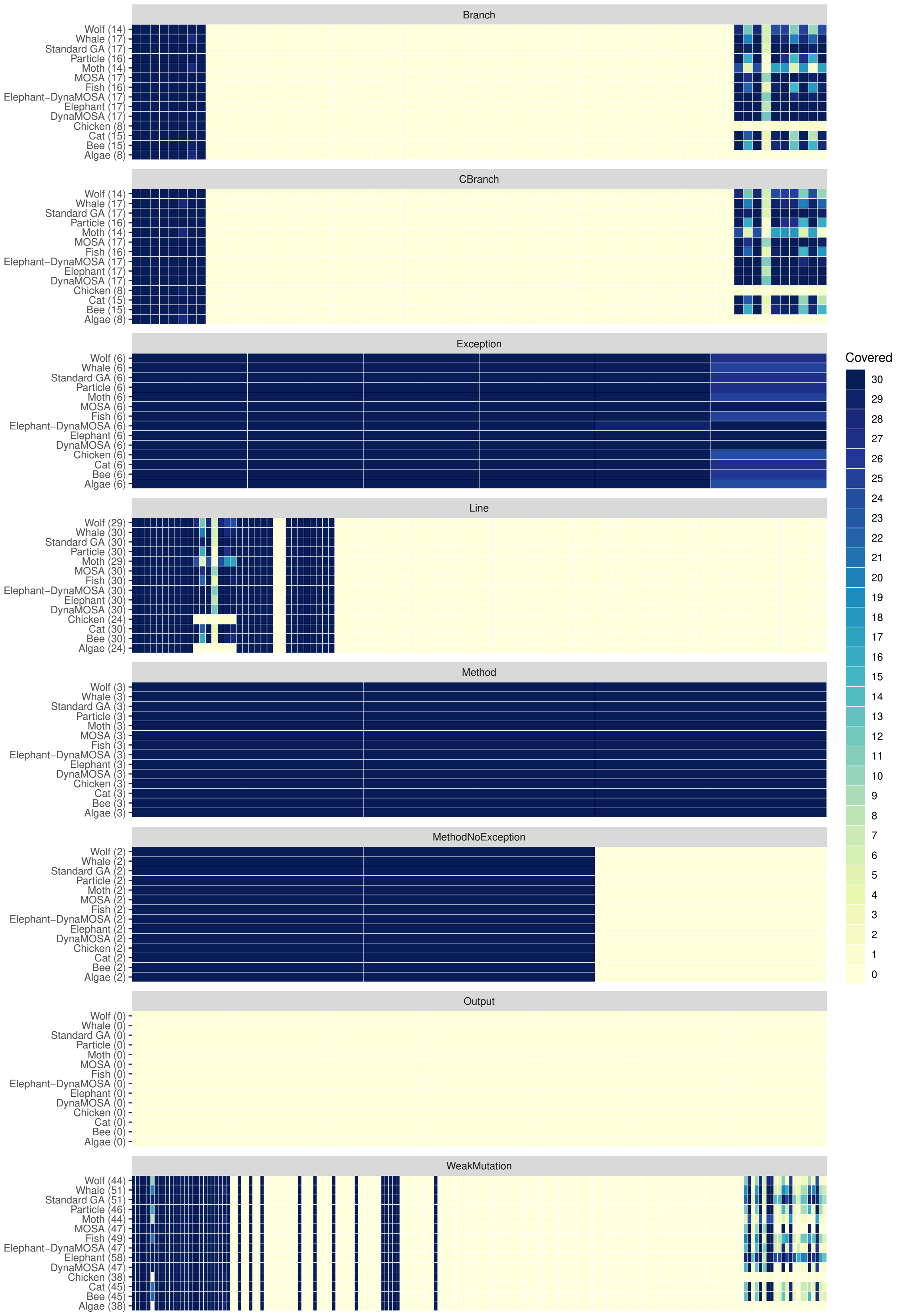


Figure C.6: Heatmap of class org.apache.commons.math3.optim.univariate.BrentOptimizer. The number besides the algorithm is the number of occurrences where the targets were covered by more than 15 runs. Best algorithm is Elephant.

Bibliography

[1] Wikipedia. *List of software bugs*. Accessed: 07-10-2020. URL: `https://en.wikipedia.org/wiki/List_of_software_bugs`.

[2] Ding Yuan, Yu Luo, Xin Zhuang, Guilherme Renna Rodrigues, Xu Zhao, Yongle Zhang, Pranay U. Jain, and Michael Stumm. "Simple Testing Can Prevent Most Critical Failures: An Analysis of Production Failures in Distributed Data-Intensive Systems". In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 249–265. ISBN: 978-1-931971-16-4. URL: `https://www.usenix.org/conference/osdi14/technical-sessions/presentation/yuan`.

[3] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*. Vol. 2. Wiley Online Library, 2004.

[4] Gordon Fraser and José Miguel Rojas. "Software Testing". In: *Handbook of Software Engineering*. Springer, 2019, pp. 123–192.

[5] Mark Harman, Phil McMinn, Jerffeson Teixeira De Souza, and Shin Yoo. "Search based software engineering: Techniques, taxonomy, tutorial". In: *Empirical software engineering and verification*. Springer, 2010, pp. 1–59.

[6] Phil McMinn. "Search-based software testing: Past, present and future". In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE. 2011, pp. 153–163.

[7] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. "Combining multiple coverage criteria in search-based unit test generation". In: *International Symposium on Search Based Software Engineering*. Springer. 2015, pp. 93–108.

[8] Jun Tang, Gang Liu, and Qingtao Pan. "A Review on Representative Swarm Intelligence Algorithms for Solving Optimization Problems: Applications and Trends". In: *IEEE/CAA Journal of Automatica Sinica* 8.10 (2021), pp. 1627–1643. DOI: `10.1109/JAS.2021.1004129`.

[9] Daniel Molina, Javier Poyatos, Javier Del Ser, Salvador García, Amir Hussain, and Francisco Herrera. "Comprehensive Taxonomies of Nature-and Bio-inspired Optimization: Inspiration versus Algorithmic Behavior, Critical Analysis and Recommendations". In: *arXiv preprint arXiv:2002.08136* (2020).

[10] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. "An empirical evaluation of evolutionary algorithms for unit test suite generation". In: *Information and Software Technology* 104 (2018), pp. 207–235.

[11] Andreas Windisch, Stefan Wappler, and Joachim Wegener. “Applying particle swarm optimization to software testing”. In: *Proceedings of the 9th annual conference on Genetic and evolutionary computation*. 2007, pp. 1121–1128.

[12] 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 Trans. Software Eng.* 44.2 (2018), pp. 122–158. DOI: `10.1109/TSE.2017.2663435`. URL: `https://doi.org/10.1109/TSE.2017.2663435`.

[13] Ashraf Darwish. “Bio-inspired computing: Algorithms review, deep analysis, and the scope of applications”. In: *Future Computing and Informatics Journal* 3.2 (2018), pp. 231–246.

[14] 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.

[15] Dan Bruce, Héctor D Menéndez, Earl T Barr, and David Clark. “Ant Colony Optimization for Object-Oriented Unit Test Generation”. In: *International Conference on Swarm Intelligence*. Springer. 2020, pp. 29–41.

[16] Manju Khari, Anunay Sinha, Enrique Herrerra-Viedma, and Rubén González Crespo. “On the Use of Meta-Heuristic Algorithms for Automated Test Suite Generation in Software Testing”. In: *Toward Humanoid Robots: The Role of Fuzzy Sets: A Handbook on Theory and Applications* (2021), pp. 149–197.

[17] José Carlos Medeiros de Campos. “Search-based Unit Test Generation for Evolving Software”. PhD thesis. University of Sheffield, 2017.

[18] Sina Shamshiri, José Miguel Rojas, Gordon Fraser, and Phil McMinn. “Random or genetic algorithm search for object-oriented test suite generation?” In: *Proceedings of the 2015 annual conference on genetic and evolutionary computation*. 2015, pp. 1367–1374.

[19] Andrea Arcuri and Lionel Briand. “Adaptive random testing: An illusion of effectiveness?” In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 2011, pp. 265–275.

[20] Gordon Fraser and Andrea Arcuri. “Handling test length bloat”. In: *Softw. Test. Verification Reliab.* 23.7 (2013), pp. 553–582. DOI: `10.1002/stvr.1495`. URL: `https://doi.org/10.1002/stvr.1495`.

[21] Wikipedia. *Code coverage*. Accessed: 23-10-2020. URL: `https://en.wikipedia.org/wiki/Code_coverage`.

[22] Andrea Arcuri. “It Does Matter How You Normalise the Branch Distance in Search Based Software Testing”. In: *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*. IEEE Computer Society, 2010, pp. 205–214. DOI: `10.1109/ICST.2010.17`. URL: `https://doi.org/10.1109/ICST.2010.17`.

[23] Gordon Fraser. *Search-based Test Generation at International Summer School on Training And Research On Testing (TAROT)*. 2014.

[24] Ahmad B. A. Hassanat, Khalid Almohammadi, Esra'a Alkafaween, Eman Abunawas, Awni Mansoar Hammouri, and V. B. Surya Prasath. "Choosing Mutation and Crossover Ratios for Genetic Algorithms - A Review with a New Dynamic Approach". In: *Inf.* 10.12 (2019), p. 390. DOI: 10.3390/info10120390. URL: https://doi.org/10.3390/info10120390.

[25] Andrea Arcuri, José Campos, and Gordon Fraser. "Unit Test Generation During Software Development: EvoSuite Plugins for Maven, IntelliJ and Jenkins". In: *2016 IEEE International Conference on Software Testing, Verification and Validation, ICST 2016, Chicago, IL, USA, April 11-15, 2016*. IEEE Computer Society, 2016, pp. 401–408. DOI: 10.1109/ICST.2016.44. URL: https://doi.org/10.1109/ICST.2016.44.

[26] Annibale Panichella, José Campos, and Gordon Fraser. "EvoSuite at the SBST 2020 Tool Competition". In: *ICSE '20: 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*. ACM, 2020, pp. 549–552. DOI: 10.1145/3387940.3392266. URL: https://doi.org/10.1145/3387940.3392266.

[27] Adnan Acan and Ahmet Ünveren. "A memory-based colonization scheme for particle swarm optimization". In: *Proceedings of the IEEE Congress on Evolutionary Computation, CEC 2009, Trondheim, Norway, 18-21 May, 2009*. IEEE, 2009, pp. 1965–1972. DOI: 10.1109/CEC.2009.4983181. URL: https://doi.org/10.1109/CEC.2009.4983181.

[28] Dervis Karaboga and Bahriye Basturk. "A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm". In: *Journal of global optimization* 39.3 (2007), pp. 459–471.

[29] Shu-Chuan Chu, Pei-Wei Tsai, et al. "Computational intelligence based on the behavior of cats". In: *International Journal of Innovative Computing, Information and Control* 3.1 (2007), pp. 163–173.

[30] Seyedali Mirjalili and Andrew Lewis. "The Whale Optimization Algorithm". In: *Adv. Eng. Softw.* 95 (2016), pp. 51–67. DOI: 10.1016/j.advengsoft.2016.01.008. URL: https://doi.org/10.1016/j.advengsoft.2016.01.008.

[31] Seyedali Mirjalili. "Moth-flame optimization algorithm: A novel nature-inspired heuristic paradigm". In: *Knowledge-based systems* 89 (2015), pp. 228–249.

[32] Seyedali Mirjalili, Seyed Mohammad Mirjalili, and Andrew Lewis. "Grey wolf optimizer". In: *Advances in engineering software* 69 (2014), pp. 46–61.

[33] Sait Ali Uymaz, Gülay Tezel, and Esra Yel. "Artificial algae algorithm (AAA) for nonlinear global optimization". In: *Appl. Soft Comput.* 31 (2015), pp. 153–171. DOI: 10.1016/j.asoc.2015.03.003. URL: https://doi.org/10.1016/j.asoc.2015.03.003.

[34] Xianbing Meng, Yu Liu, Xiao Zhi Gao, and Hengzhen Zhang. “A New Bio-inspired Algorithm: Chicken Swarm Optimization”. In: *Advances in Swarm Intelligence - 5th International Conference, ICSI 2014, Hefei, China, October 17-20, 2014, Proceedings, Part I*. Ed. by Ying Tan, Yuhui Shi, and Carlos A. Coello Coello. Vol. 8794. Lecture Notes in Computer Science. Springer, 2014, pp. 86–94. DOI: `10.1007/978-3-319-11857-4\_10`. URL: `https://doi.org/10.1007/978-3-319-11857-4\_10`.

[35] Gai-Ge Wang, Suash Deb, and Leandro dos S Coelho. “Elephant herding optimization”. In: *2015 3rd International Symposium on Computational and Business Intelligence (ISCBI)*. IEEE. 2015, pp. 1–5.

[36] Hsing-Chih Tsai and Yong-Huang Lin. “Modification of the fish swarm algorithm with particle swarm optimization formulation and communication behavior”. In: *Appl. Soft Comput.* 11.8 (2011), pp. 5367–5374. DOI: `10.1016/j.asoc.2011.05.022`. URL: `https://doi.org/10.1016/j.asoc.2011.05.022`.

[37] AR Moradi, Y Alinejad-Beromi, and K Kiani. “Artificial Fish Swarm Algorithm for solving the Economic Dispatch with Valve-Point Effect”. In: *International Journal of Engineering and Technology* 2.3 (2014), pp. 299–313.

[38] 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`.

[39] Gordon Fraser and Andrea Arcuri. “Achieving Scalable Mutation-based Generation of Whole Test Suites”. In: *Empirical Software Engineering* (2014), pp. 1–30. ISSN: 1382-3256. DOI: `10.1007/s10664-013-9299-z`. URL: `http://dx.doi.org/10.1007/s10664-013-9299-z`.

[40] Gordon Fraser and Andreas Zeller. “Mutation-driven Generation of Unit Tests and Oracles”. In: *IEEE Transactions on Software Engineering* 38.2 (2012), pp. 278–292. ISSN: 0098-5589. URL: `http://doi.ieeecomputersociety.org/10.1109/TSE.2011.93`.