A Comparison Between Hierarchical and Non-Hierarchical Software Clustering

Léo C. R. Antunes
Federal University of the State of Rio de Janeiro (UNIRIO)
Rio de Janeiro, Brazil
leo.antunes@edu.unirio.br

Márcio de O. Barros
Federal University of the State of Rio de Janeiro (UNIRIO)
Rio de Janeiro, Brazil
marcio.barros@uniriotec.br

ABSTRACT

Heuristic search algorithms have been applied in various areas of Software Engineering, such as requirements prioritization, software architecture improvement, and time and cost planning. The research field of Search-based Software Engineering (SBSE) proposes the use of heuristic search techniques to solve Software Engineering challenges, which are described as optimization problems. One of the challenges frequently analyzed from the SBSE perspective is the Software Module Clustering (SMC) problem. This problem involves distributing a software project's component modules (or classes) among larger structures called clusters (or packages). Practice and experimental studies have shown a strong correlation between poor distributions and the presence of component failures. The main factors used to assess the quality of distributions are coupling and cohesion. However, research shows that optimization based on metrics that capture structural characteristics does not result in an adequate distribution of classes in packages from the point of view of software developers. One of the reasons that might have led to such limited results is that the algorithms produce non-hierarchical distributions of classes into packages. In this study, we executed the same greedy algorithm using two different fitness functions: one that does not consider hierarchy and the other that does. We wanted to measure whether they produced different results regarding authoritativeness. We found indications that a hierarchy-based approach produces solutions closer to those software developers propose.

KEYWORDS

Software Engineering, Software Clustering, Hierarchical Design

1 INTRODUCTION

Heuristic search algorithms have been applied in various areas of Software Engineering, such as requirements prioritization [4, 15, 41], software architecture improvement [5, 8, 35], and time and cost planning [1, 2, 10, 29]. Traditional search techniques, such as brute force or branch-and-bound [24], are used in some areas. However, as the addressed problems grow in size and complexity, a search procedure based on the exhaustive examination of all possible solutions may require too much processing power to find the optimal solution. Thus, using thorough searches in some situations, such as those requiring quick decision-making responses or making use of frequently updated data, becomes unfeasible.

The research field of Search-based Software Engineering (SBSE) [3, 7, 16, 21] proposes the use of heuristic search techniques to solve Software Engineering challenges which are described as optimization problems. These techniques are more efficient in processing power than traditional search, although they cannot be guaranteed to find optimal solutions to the analyzed problem. In scenarios where a complete search is not feasible, the goals of optimization techniques are relaxed, and the search looks for reasonable (instead of optimal) solutions to the problem. Research has shown that this approach is promising for various Software Engineering problems.

One of the challenges frequently analyzed from the SBSE perspective is the Software Module Clustering (SMC) problem [31]. This problem deals with distributing the modules comprising a software project among larger structures called clusters. The modules of the software project might be classes, methods, or variables. The relationships between these modules represent the import of classes, the invocation of methods, or the access to variables. Clusters are groupings of modules and represent packages, namespaces, or software subsystems. They are structures with a higher level of abstraction than software modules. This manuscript assumes that object-oriented programming is used; therefore, modules depict **classes**, and clusters denote **packages**. Dependencies between modules represent importing classes, calling methods, or accessing variables. The pairs of terms *module/class* and *cluster/package* will be used for the same purpose in this manuscript.

A good distribution of classes into packages helps to identify the classes responsible for implementing a feature, provides more straightforward navigation between the software components [17], and makes it easier to understand the structure of the software [23, 28]. Therefore, proper distribution of classes into packages tends to make software easier to develop and maintain. In addition, practice and experimental studies have shown a strong correlation between poor distributions and the presence of failures [9].

The most frequently used metrics to assess the quality of a given distribution of software modules into clusters are coupling and cohesion [42]. Coupling measures dependency between software modules (or the modules within a cluster) within a system. At the same time, cohesion measures how much the module members (or the modules within a cluster) are dedicated to a single objective. Usually, coupling is measured by the number of dependencies between modules belonging to different clusters, while cohesion is calculated through the dependencies between modules belonging to the same cluster. In general, the goal of clustering is to distribute the software modules in a given number of clusters to minimize coupling and maximize cohesion – with the restriction that each module must belong to a single cluster.

However, research shows that optimization based on metrics that capture structural characteristics does not result in an adequate distribution of modules into clusters from the developers' point of view [6]. One reason that might have led to such limited results is the search for non-hierarchical distributions, that is, clusters can contain only modules. In such distributions, clusters cannot contain other clusters. On the other hand, most programming languages allow the creation of clusters within clusters, and programmers broadly use this resource. We, therefore, intend to explore the concept of hierarchical distributions of modules into clusters.

Wood [40] developed a complexity measure for the hierarchical clustering of software projects based on a message-passing metaphor called the Minimum Description Length (MDL) principle. The metric makes no explicit mention of the concepts of cohesion and coupling. Still, it aims to permit trade-offs between engineering attributes (such as cohesion, coupling, and module size) to achieve a better structure in a hierarchical framework. It assumes the best distribution of modules into clusters for a software project is the "simplest" according to the MDL principle.

In this paper, we performed an experiment on real-world projects using a greedy algorithm applied to the software clustering problem and driven by two distinct fitness functions: one that does not consider hierarchy (Modularization Quality - *MQ*) and the other that does (Minimum Description Length - *MDL*). We wanted to determine whether the results yielded by an optimization guided by the second function produce distributions of modules into clusters with higher authoritativeness than an optimization driven by the former metric. By authoritativeness, we mean distributions of modules to clusters closer to the one developers have proposed in the software.

We compared both approaches using distinct metrics, such as the average of classes per package and the percentage of commits that contained classes belonging to a single package. We found that the authoritativeness of solutions produced by an optimization guided by *MDL* is better on average than solutions produced by *MQ* on the chosen metrics. Also, while *MDL* produces distributions that are still outperformed by the *DEV* distribution on what concerns the concentration of changes, such solutions are far better than those produced by *MQ*.

This paper is structured as follows: Section 2 introduces the theoretical foundation of information theory, which is required to understand the MDL principle and the metric proposed by Wood [40], as well as the related work. Section 3 presents the proposed approach, while Section 4 depicts the experimental study setup and results. Finally, section 5 presents the threats to this work's validity, and Section 6 highlights the conclusions and future work directions.

2 BACKGROUND

2.1 SMC and the Minimum Description Length

In conversational language, we identify information about an individual object as a set of descriptions of its characteristics. We can formalize this concept computationally by defining the amount of information required to describe a finite object (such as a string) as the size of the smallest program (in bits) that, starting with a blank memory, outputs the object and then terminates. This concept of handling the quantity of information needed to describe an individual object is called "Algorithmic Information Theory" or "Kolmogorov Complexity" [25].

Kolmogorov Complexity started as a research field in probability theory, combinatorics, and conceptions of randomness and gained momentum with the theory of algorithms. It is based on Shannon's classical information theory [37] and aims to define a function that calculates the quantity of information needed to represent an ensemble of possible messages. Considering that all messages in the ensemble are equally probable, this quantity is the number of bits needed to count all possibilities to send unique messages from a sender to a receiver. However, the theory cannot be directly used in practice due to the noncomputability[1] of such a count.

Rissanen [36] proposed the Minimum Description Length (MDL) principle to cope with such a limitation. According to Li and Vitányi [25], the Minimum Description Length principle states that the best theory to explain a set of data is the one that minimizes the sum of the length, in bits, of the description of the theory and the length, in bits, of the data when encoded with the help of such theory. Given some data D, the *MDL* principle states that we should pick the theory T that minimizes Equation 1, where $length(T)$ is the number of bits needed to minimally encode the theory T and $length(D|T)$ is the number of bits needed to minimally encode the data D given the theory T. We must be able to decode the message; otherwise, a single bit would be enough.

$$length(T) \; + \; length(D|T) \tag{1}$$

While applying the MDL principle to address the software module clustering problem, Wood [40] proposed a theory to describe a graph representing the design of a software system. In this graph, nodes represent the modules that convey parts of the implementation of the software system, and edges represent dependencies among these modules. Each module is represented by a unique identifier (a positive integer), and each edge is represented by a pair of positive integers representing, respectively, the identifiers of the source and target modules of the dependency.

Wood [40] proposes that the length of a message describing all edges of a graph is a measure of the structural complexity of the graph and that a smaller message length indicates a simpler graph. Subsequently, a simpler graph represents a better design and, therefore, a better distribution of the software modules into clusters. The author proposed the message depicted in Equation 2 to represent a hierarchical graph.

$$\begin{aligned} \textbf{object\_description} &= \text{object\_id} + \\ &\quad |nodes| + \\ &\quad |objects| \; \{\text{object}_1, \ldots, \text{object}_m\} + \\ &\quad |edges| \; \{edge_1, \ldots, edge_e\} \\ \textbf{point} &= \{object\_id\} \, * \, basic\_entity\_id \\ \textbf{edge} &= point_1 \; - \; point_2 \end{aligned} \tag{2}$$

The length of the message describing a graph representing the structure of a software's design using this encoding (Ψ) is calculated by Equation 3, where $\mathcal{M}$ is the set of all clusters, N_m is the number of sub-clusters and modules within cluster m, $N^{'}_m$ is the number of sub-clusters within cluster m ($N_m \geq N^{'}_m$), E_m is the number of edges described in cluster m, and $\mathcal{N}_m$ is the set of sub-clusters and modules within cluster m ($\mathcal{N}_m = |N_m|$). f_n is the frequency of module or cluster n in cluster m, which for modules is given by

[1]Further details about the noncomputability as well as mathematical proof can be found at [25].

the degree[2] of the corresponding module, and for clusters, is the sum of the degrees of the child sub-clusters and modules plus 1 and then $F_n = \sum_n f_n$.

$$\Psi = \sum_{m \in \mathcal{M}} (nodes_m + clusters_m + edges_m - freq_m + E_m) + 1 \quad (3)$$

$$nodes_m = \log_2^* (N_m + 1) \quad (4)$$

$$clusters_m = \log_2^* (N_m^{'} + 1) \quad (5)$$

$$edges_m = \log_2^* (E_m + 1) \quad (6)$$

$$freq_m = \sum_{n \in \mathcal{N}_m} f_n \times \log_2 \left(\frac{f_n}{F_n} \right) \quad (7)$$

2.2 Related Work

Among the experiments performed on the SMC problem, those carried out by the Drexel University group stand out [14, 27]. In several comparative experiments [14, 20, 30], the results obtained by a local search algorithm using the Random Restart Hill Climbing technique were superior to those found by more complex meta-heuristics, such as Genetic Algorithms and Simulated Annealing. Pinto *et al.* [34] presented a heuristic based on *Iterated Local Search* which proved effective for the SMC problem, outperforming the genetic algorithm approaches and using less computational processing time. Later, Monçores *et al.* [32] presented a heuristic based on a constructive algorithm and the *Large Neighborhood Search* algorithm that outperformed the ILS heuristic in 93 out of 124 software projects with a confidence level of 95%.

Although most research work handles the SMC problem using a non-hierarchical approach, some authors considered addressing the SMC problem hierarchically. Wood's work [40], presented in the previous section, is an interesting example of such approaches and was highlighted for serving as the basis for this research. The approach used a narrow beam hill-climbing driven by the defined complexity measure search strategy to look for less complex designs. Initial empirical validation showed potential.

Lutz [26] proposed a variant of the metric proposed by Wood [40] and introduced the term HMD (Hierarchical Modular Decomposition) to refer to a hierarchical structure of modules. He proposed a genetic algorithm that was run on three examples. The author concluded that the HMD produced by the search had its components better organized and is logically better than the organization found by Wood [40] for one instance. We have not based our work on Lutz's work because implementing the metric defined by the author did not produce the same results outlined in the paper. So, we decided to take a step back and implement the original work made by Wood, which was successfully replicated.

Another hierarchical clustering algorithm is ACDC (Algorithm for Comprehension-Driven Clustering), proposed by Tzerpos *et al.* [38]. Unlike algorithms based on structural information that aim to satisfy metrics such as low coupling and high cohesion, the ACDC algorithm aims for an easily understandable output, ensuring that the clusters follow familiar patterns and naming the clusters intelligently. Experiments were carried out with two real projects. It was found that ACDC can be an effective clustering algorithm and a good candidate for reverse engineering projects.

Hall and McMinn [19] analyzed the Bunch tool in the hierarchical approach. The Bunch tool was used to produce hierarchical solutions through successive clusters. A bottom-up approach was used, with each clustering grouping the clusters produced previously, i.e. the clusters acting as modules in the new clustering. The study results indicate an improvement of up to 30% by reducing the hierarchical levels' layering produced by the clustering.

Few studies have proposed hierarchy-based techniques. Some of these studies relied on meta-heuristics, particularly genetic algorithms, and no work we have been able to track has compared the proposed solutions with the developers' distributions. Our study tested a hierarchical-based metric on real-world projects and compared it with the developers' solutions. To our knowledge, no such work has been carried out.

3 PROPOSED APPROACH

As noted in the previous section, many different heuristics have been proposed to solve the SMC problem. Notably, the application of local search was shown to be promising for non-hierarchical clustering, with good results being yielded by algorithms such as the Iterated Local Search (*ILS*) [34] and the Large Neighborhood Search (*LNS*) [33]. In both approaches, the first step of the search process involves generating a promising initial solution to ease the computational time required to execute the local search.

Pinto's implementation of *ILS*, called *ILS_CMS*, uses a constructive approach in which the initial solution is built iteratively using a Greedy algorithm [13, 18]. Greedy algorithms build up a solution in small steps, choosing a decision at each step myopically to optimize some underlying criterion. For some problems, Greedy algorithms can produce a solution that is guaranteed to be close to optimal, even if it does not achieve the precise optimum [22].

Pinto's greedy algorithm is based on an approach applied to clustering large networks [11] and used to identify "community structures" – vertices more densely connected among themselves than connected to vertices in other clusters. The problem of finding "community structures" can be generically represented as identifying groups in large networks, such as the Internet, social networks, and article citation networks. The greedy algorithm is an Agglomerative Clustering (*AC*) algorithm that is fast enough for the hierarchical partitioning of graphs with several thousand vertices.

Our proposed approach is based on Pinto's work and modifies the latter by replacing *MQ* with the *MDL* fitness function to drive the optimization process using the *AC* algorithm. Alongside the new fitness function, we have modified the original algorithm to allow the creation of hierarchical structures. We added a step to attempt to add clusters as sub-modules of other clusters to see if it enhances MDL. Furthermore, at the end of the algorithm, we perform cleanup operations: (i) remove clusters without children and (ii) merge clusters with just one child to their parents to avoid single-node clusters. These modifications have been proposed after some exploratory executions. Besides, the proposal of using cluster

[2]An edge, (n_i, n_j), is said *to be* incident from module n_i and incident to module n_j. The sum of the number of edges incident to and from a module is called the degree of the module [40].

Algorithm 1: Agglomerative Clustering*

```
1: $s* \leftarrow$ generate an initial solution with one module per cluster
2: repeat
3:   $maxDelta \leftarrow 0$
4:   for each $c \in clusters$ do
5:     for each $c' \in clusters$ do
6:       $currentDelta \leftarrow calculateMergeClustersMDL(c, c')$
7:       if $currentDelta > maxDelta$ then
8:         $maxDelta \leftarrow currentDelta$
9:         $target\_c \leftarrow c$
10:        $target\_c' \leftarrow c'$
11:      end if
12:    end for
13:  end for
14:  $s* \leftarrow mergeClusters(target\_c, target\_c')$
15: until does not find clusters to merge
16: repeat
17:  $maxDelta \leftarrow 0$
18:  for each $c \in clusters$ do
19:    for each $c' \in clusters$ do
20:      $currentDelta \leftarrow$
         $calculateAddAsChildClusterMDL(c, c')$
21:      if $currentDelta > maxDelta$ then
22:        $maxDelta \leftarrow currentDelta$
23:        $target\_c \leftarrow c$
24:        $target\_c' \leftarrow c'$
25:      end if
26:    end for
27:  end for
28:  $s* \leftarrow AddClusterAsChild(target\_c, target\_c')$
29: until does not find clusters to add as children
30: $s* \leftarrow removeClustersWithoutChildren(s*)$
31: $s* \leftarrow moveOnlyChildToGrandparent(s*)$
```

movements was essential to form the hierarchical structure; otherwise, the algorithm would suggest a flat solution. The modified algorithm's pseudo-code, Agglomerative Clustering* (*AC**), is presented at Algorithm 1. In the scope of this paper, when we mention *AC* using *MDL* as the fitness function, we are referring to *AC**.

4 EXPERIMENTS

4.1 Project Selection

We have selected systems used in previous studies [32, 34] and applied the following criteria.

(1) *Open-source software*: we chose open-source libraries due to the availability of the source code, which is necessary to execute and evaluate the proposed approach;
(2) *Source code is available through a Git repository*: since we needed to evaluate software evolution history, we chose systems that stored their source code in Git repositories. Git is one of the most popular VCS in use today;
(3) *Source code is primarily written in Java programming language*: we decided to evaluate Java-based systems due to the availability of tools to extract project metadata automatically from the compiled code;
(4) *Different numbers of classes and packages*: to evaluate the proposed approach in different scenarios and scales, we picked systems with varying sizes, measured as the number of classes and packages (from tens to thousands).

Since we have started from 124 open-source Java programs, the criteria (1) and (3) have been automatically applied. We removed those missing a GitHub repository to ensure we could analyze their source-code modification history.

Table 1: Investigated software systems

Software system	Description	Version
AEP Core	A client Java library to manage App Engine Java applications for any project that performs App Engine Java application management.	0.10.0
JavaGeom	JavaGeom is a geometry library for Java.	0.11.3
JUnit	The 5th major version of the programmer-friendly testing framework for Java and the JVM.	5.10.1
JMetal	jMetal is a Java-based framework for multi-objective optimization with meta-heuristics.	6.2.2
JGit	An implementation of the Git version control system in pure Java.	6.8.0

Table 1 presents the chosen systems, their description, and versions. We chose to work with a limited number of classes and packages for the preliminary studies reported in this chapter because of the time needed to execute *ILS_CMS* for large software projects, particularly when the *MDL* fitness function was used.

The proposed algorithm merges and rebuilds the set of packages comprising the system several times during the optimization. Thus, the number of iterations it performs and its execution time are strongly related to the number of classes and relationships. Also, on each iteration, MDL must be recalculated from scratch, an expensive process in terms of computation power that also requires going over all the classes and their relationships. For instance, JGit has 1,613 classes. Considering the execution time observed while optimizing its sub-modules, we estimate that 200 days would be required to optimize the entire system using the same configurations and hardware. On the other hand, MQ is calculated through an optimized version that does not require recalculating the metric from scratch on every round of optimization. Thus, it took seconds to calculate all sub-modules when a change is tested during the optimization. Using MQ, generating the solution for the entire system would take minutes.

Recalculating the metric from scratch is a drawback of the proposed approach, which will be addressed in the next iterations of the research. There are avenues for improving the performance of the MDL metric calculation process. In the current study, we aimed to determine whether the proposed solutions would be closer to those developers created.

Therefore, we extracted sub-modules with less than 200 classes from each of the selected software systems to maintain a reasonable variation in the number of classes and packages. The limit was chosen as a threshold to guarantee that the optimization of each one of the projects could run in a short period (up to 30 minutes).

Table 2 shows the sub-modules selected and their number of classes and packages.

Table 2: Investigated systems/sub-modules ordered by the number of classes.

ID	Software system	Sub-module	Classes	Packages
AEP	AEP Core	-	162	17
JGE	JavaGeom	-	159	20
JUP	JUnit	Platform	135	10
JMC	JMetal	Component	132	43
JUJ	JUnit	Jupiter	129	9
JGP	JGit	PGM	121	5
JUL	JUnit	Launcher	115	7
JGS	JGit	SSH	98	7
JMA	JMetal	Auto	69	7
JGH	JGit	HTTP	61	3

4.2 Data Collection

We executed the *AC* algorithm for each selected system's sub-module with *MQ* and *MDL* fitness functions. Subsequently, the following information has been extracted:

- *Number of packages*: the number of packages in the resulting solution;
- *Average of classes per package*: the average number of classes per package calculated by dividing the number of classes in the project by the number of packages in the resulting solution. Nested packages are counted independently;
- *Max depth*: the longest path from the root package to the deepest package in the hierarchy;
- *MoJoFM*: calculated by comparing the optimization results with the distribution of classes to packages proposed by the developers;
- *Single-package commits ratio*: percentage of commits that contained classes belonging to a single package;
- *Average number of packages affected per commit*: average number of packages with classes contained in each commit to the version control system.

The "number of packages" and the "average number of classes per package" metrics strike a balance between the number of packages and the number of classes on each package. While systems with fewer packages are easier to understand, a package with many classes is usually more complex than one with just a few. So, we must balance the total number of packages and the number of classes per package.

The "max depth" metric has been studied to determine whether the solutions contain hierarchies comparable to those humans generated. The $MojoFM$ metric compares two distributions of classes into packages for the same system. It points out how far a solution generated by an algorithm is from the solution designed by developers.

The "single-package commits ratio" computes how often classes belonging to the same package are committed together. We assume that a commit modifies a single feature provided by the system. So, a large number of single-package commits indicates that the related distribution of classes into packages brings together classes that are changed together for a single purpose. On the other hand, while single-package commits would be desired, commits involving more than one package might involve as few as possible. So, the "average number of packages affected per commit" metric determines if such a concept applies to a given solution.

Since the *AC* algorithm is deterministic, we have executed it once for each instance. We then compared the results with the distribution of classes to packages proposed by the developers of each project. We refer to such distribution as "*DEV*". The distribution resulting from the *MQ*-driven optimization was called "*MQ*", and "*MDL*" was the name given to the distribution proposed by the *MDL*-driven optimization. Next, we present the results and related discussion for each information described above.

4.3 Analysis: Number of Packages

Figure 1 presents the number of packages for each distribution of classes into packages (*DEV*, *MQ*, and *MDL*). Nested packages were counted independently for the *DEV* and *MDL* distributions. For instance, if package A contained packages A1 and A2, the number of packages would be three: A, A1, and A2. This allows comparing the number of packages with the solution proposed by *MQ*, which contains only flat packages.

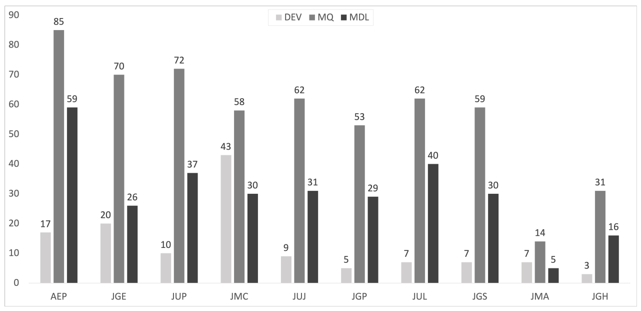

Figure 1: Number of packages (*DEV* x *MQ* x *MDL*).

We present the raw data in Table 3. We observe a large difference in the number of packages on different solutions. *MQ* generated many more packages than the *MDL* and the *DEV* solutions for all software projects under analysis, which we can observe in the MQ/DEV column and visually through the boxplot in Figure 2. The ratio between *MQ* and *DEV* varies from 1.35 to 10.60, with an average of 6.42 and a median of 7.04. For most cases, *MDL* generated a balanced number of packages, with a few projects producing fewer packages than *DEV* (*JMC* and *JMA*), coincidentally both sub-modules of the JMetal project. We can observe this in the MDL/DEV column. The ratio between *MDL* and *DEV* varies from 0.70 to 5.80, with an average of 3.45 and a median of 3.59.

Furthermore, the correlation coefficient (ρ) of the difference between *MQ* and *MDL* ($MQ-MDL$ column) and the number of classes is equal to 0.79, which indicates a strong correlation according to Cohen [12]. In other words, we can affirm that when the number of classes increases, the difference between the solutions proposed by *MDL* and *MQ* also increases, meaning the number of packages in *MDL* grows slower than *MQ*.

Table 3: Number of packages (*DEV* x *MQ* x *MDL*).

ID	#	Packages (*DEV*)	Packages (*MQ*)	Packages (*MDL*)	*MQ*/ *DEV*	*MDL*/ *DEV*	*MQ*− *MDL*
AEP	162	17	85	59	5	3.47	26
JGE	159	20	70	26	3.5	1.3	44
JUP	135	10	72	37	7.2	3.7	35
JMC	132	43	58	30	1.35	0.7	28
JUJ	129	9	62	31	6.89	3.44	31
JGP	121	5	53	29	10.6	5.8	24
JUL	115	7	62	40	8.86	5.71	22
JGS	98	7	59	30	8.43	4.3	29
JMA	69	7	14	5	2	0.71	9
JGH	61	3	31	16	10.33	5.33	15
Min	-	3.0	14.0	5.0	1.35	0.70	9.0
Mean	-	12.8	56.6	30.3	6.42	3.45	26.3
Median	-	8.0	60.5	30.0	7.04	3.59	27.0
Max	-	43.0	85.0	59.0	10.60	5.80	44.0

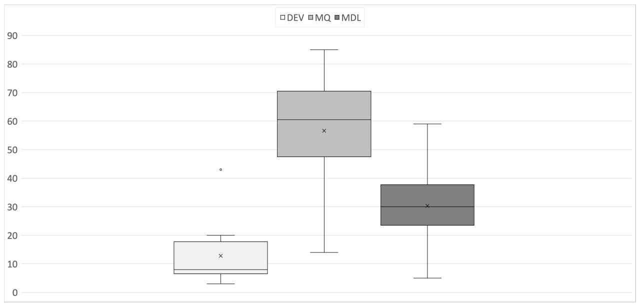

Figure 2: Number of packages (*DEV* x *MQ* x *MDL*) boxplot.

4.4 Analysis: Average of Classes per Package

Figure 3 shows each distribution's average number of classes per package (*DEV*, *MQ*, and *MDL*). We also present the raw data in Table 4. *MQ* has an average of 2.33 and a median of 2.05 classes per package, which is far different than the *DEV* distribution, which has an average of 9.14 and a median of 8.25. In contrast, *MDL* contains, on average, 5.85 classes per package with a median of 4.95 – a balanced solution compared to the previous two.

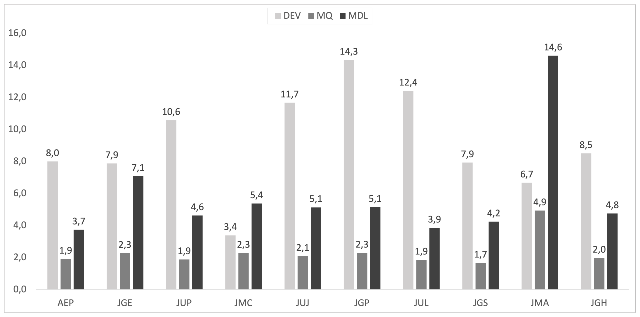

Figure 3: Average of classes per package (*DEV* x *MQ* x *MDL*).

The DEV distributions show larger values for most software projects, from eight to twelve classes per package on average, except for *JMC* and *JMA*, which have fewer classes per package (3.4 and 6.7, respectively) – again sub-modules of the JMetal project. This can be visualized in the boxplot in Figure 4. One interesting observation is that the MDL average is greater than DEV's for those projects (5.4 and 14.6, respectively).

In the *MDL*/*DEV* column, we can observe that the ratio between *MDL* and *DEV* varies from 0.31 to 2.18, with an average of 0.78 and a median of 0.5. This indicates a tendency to produce fewer classes per package than *DEV* but with a smaller variance than *MQ*, which can be observed in the *MQ*/*DEV* column. The ratio between *MQ* and *DEV* varies from 0.15 to 0.73, with an average of 0.31 and a median of 0.23.

The correlation (ρ) of the difference between *MDL* and *MQ* ($MDL - MQ$ column) and the number of classes is equal to -0.41, which indicates a medium inverted correlation according to Cohen [12]. In other words, when the number of classes increases, the difference between the solutions proposed by *MDL* and *MQ* decreases, meaning the number of classes per package in *MDL* grows faster than *MQ*.

Table 4: Average of classes per package (*DEV* x *MQ* x *MDL*).

ID	Classes	Classes (*DEV*)	Classes (*MQ*)	Classes (*MDL*)	*MQ*/ *DEV*	*MDL*/ *DEV*	*MDL*− *MQ*
AEP	162	8	1.9	3.7	0.24	0.46	1.8
JGE	159	7.9	2.3	7.1	0.29	0.9	4.8
JUP	135	10.6	1.9	4.6	0.18	0.43	2.7
JMC	132	3.4	2.3	5.4	0.68	1.59	3.1
JUJ	129	11.7	2.1	5.1	0.18	0.44	3
JGP	121	14.3	2.3	5.1	0.16	0.36	2.8
JUL	115	12.4	1.9	3.9	0.15	0.31	2
JGS	98	7.9	1.7	4.2	0.22	0.53	2.5
JMA	69	6.7	4.9	14.6	0.73	2.18	9.7
JGH	61	8.5	2	4.8	0.24	0.56	2.8
Min	-	3.40	1.70	3.70	0.15	0.31	1.80
Mean	-	9.14	2.33	5.85	0.31	0.78	3.52
Median	-	8.25	2.05	4.95	0.23	0.50	2.80
Max	-	14.30	4.90	14.60	0.73	2.18	9.70

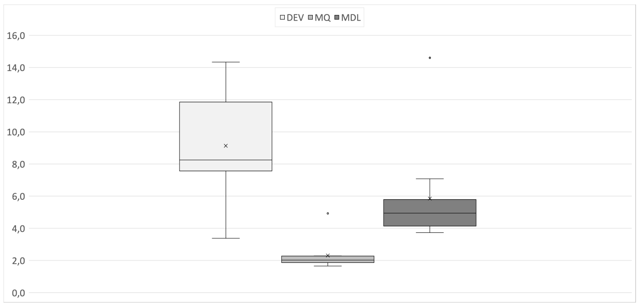

Figure 4: Average of classes per package (*DEV* x *MQ* x *MDL*) boxplot.

4.5 Analysis: Maximum Depth

Figure 5 depicts the maximum depth of the package structure in each distribution (*DEV*, *MQ*, and *MDL*). Also, the raw data is presented in Table 5. *MQ* generates flat packages, so the depth is always constant, equal to 1 - as we can observe in both figure and table. In contrast, *MDL* contains, on average, a maximum of 7 hierarchy levels and values varying from 3 to 11. This is similar to the values obtained at the *DEV* distributions, which vary from 6 to 10 with an average of 7.4 and median of 7. This can also be observed in the boxplot depicted in Figure 6.

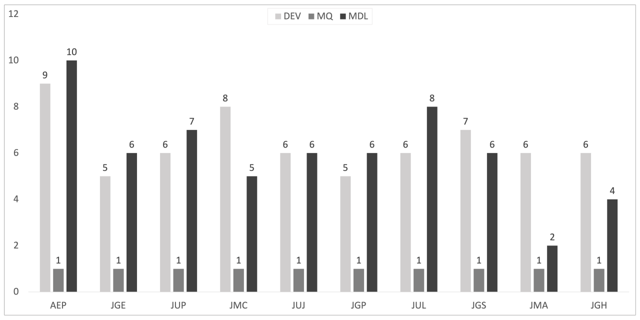

Figure 5: Maximum depth (*DEV* x *MQ* x *MDL*)

In the MDL/DEV column, we can observe that the ratio between MDL and DEV varies from 0.63 to 1.33, with an average of 0.95 and a median of 1.06. This indicates how close one is to the other. The ratio between MQ and DEV (MQ/DEV column) varies according to the DEV distribution since MQ values are constant.

The correlation (ρ) of the difference between MDL and MQ ($MDL - MQ$ column) and the number of classes is equal to 0.73, which indicates a strong correlation according to Cohen [12]. In other words, we can say that the difference between MDL and MQ grows when the number of classes increases. Since MQ proposes only flat solutions and is constant, we can conclude the solutions proposed by MDL increase the maximum depth according to the number of classes.

Table 5: Maximum depth (*DEV* x *MQ* x *MDL*).

ID	Classes	Depth (DEV)	Depth (MQ)	Depth (MDL)	MQ/DEV	MDL/DEV	$MDL-MQ$
AEP	162	9	1	10	0.11	1.11	9
JGE	159	5	1	6	0.20	1.20	5
JUP	135	6	1	7	0.17	1.17	6
JMC	132	8	1	5	0.13	0.63	4
JUJ	129	6	1	6	0.17	1.00	5
JGP	121	5	1	6	0.20	1.20	5
JUL	115	6	1	8	0.17	1.33	7
JGS	98	7	1	6	0.14	0.86	5
JMA	69	6	1	2	0.17	0.33	1
JGH	61	6	1	4	0.17	0.67	3
Min	-	5.0	1	5.0	0.11	0.63	4
Mean	-	6.4	1	6.0	0.16	0.95	5
Median	-	6.0	1	6.0	0.17	1.06	5
Max	-	9.0	1	10.0	0.20	1.33	9

4.6 Analysis: *MojoFM*

Figure 7 shows $MojoFM$ results for MQ and MDL compared to the DEV distribution. Except for JMA, we can observe that MDL values are mostly better than MQ values – values closer to 100 represent more similarity between the distribution yielded by the optimization process and the distribution proposed by the developers. Following the same pattern as previous metrics, the JMA's MQ distribution $MojoFM$ value is better than MDL.

Table 6 shows the $MojoFM$ values for all projects alongside relevant metrics. We can then observe that MQ values vary from

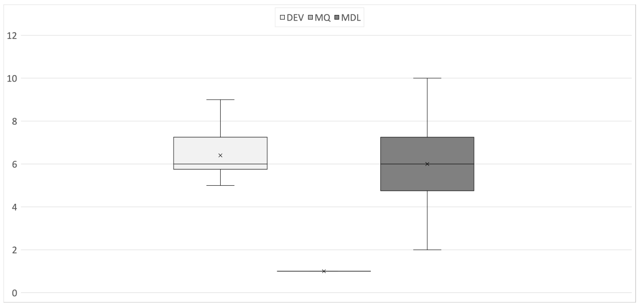

Figure 6: Maximum depth (*DEV* x *MQ* x *MDL*) boxplot.

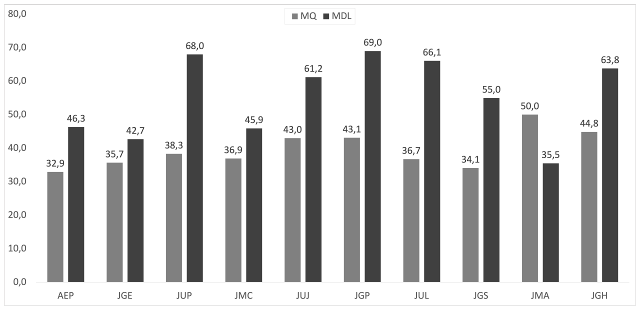

Figure 7: *MojoFM* (*MQ* x *MDL*)

32.9 to 50.0 and have an average of 39.6 and a median of 37.6, which is very different from the MDL values. MDL ranges from 35.5 to 69.0, with an average of 55.35 and a median of 58.10. The boxplot in Figure 8 depicts this visually.

Table 6: *MojoFM* value (*MQ* x *MDL*).

ID	Classes	$MojoFM$ (MQ)	$MojoFM$ (MDL)	$MDL - MQ$
AEP	162	32.9	46.3	13.4
JGE	159	35.7	42.7	7.0
JUP	135	38.3	68.0	29.7
JMC	132	36.9	45.9	9.0
JUJ	129	43.0	61.2	18.2
JGP	121	43.1	69.0	25.9
JUL	115	36.7	66.1	29.4
JGS	98	34.1	55.0	20.9
JMA	69	50.0	35.5	-14.5
JGH	61	44.8	63.8	19.0
Min	-	32.9	35.50	-14.5
Mean	-	39.6	55.35	15.8
Median	-	37.6	58.10	18.6
Max	-	50.0	69.00	29.7

For the JMA instance, the results show that the MQ distribution comes closer to the DEV distribution than the MDL proposal. Looking at the previous sections, we can infer this is due to (i) the average of classes per package being smaller than in other projects, and (ii) the generated MDL solution producing fewer packages and putting more classes in a single package. These widened the chasm between the two solutions.

The correlation (ρ) of the difference between MDL and MQ ($MDL - MQ$ column) and the number of classes is equal to 0.22, which indicates a weak correlation according to Cohen [12]. In

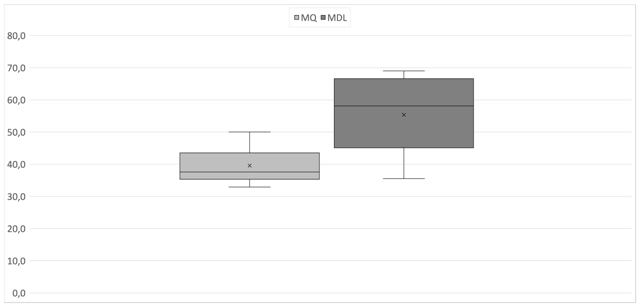

Figure 8: $MojoFM$ value (MQ x MDL) boxplot.

other words, we cannot correlate the number of classes with the difference between the solutions proposed by MDL and MQ. It might indicate that this metric varies according to other solutions' aspects not yet considered, such as the number of dependencies.

4.7 Analysis: Single-package Commits Ratio

Table 7 and Figure 9 present the percentile of commits with a single package involved. We notice a better performance on the DEV distribution (more commits involving changes in classes from a single package), with MDL coming closer than MQ for all software projects. For JMA, we observed that MDL has better results than the DEV distribution. This can indicate that the DEV distribution for this instance might benefit from restructuring.

Table 7: Percentage of one-package commits (DEV x MQ x MDL).

ID	Classes	% (DEV)	% (MQ)	% (MDL)	$MQ/$ DEV	$MDL/$ DEV	$MDL-$ MQ (%)
AEP	162	64.7	52.3	56.9	0.81	0.88	4.59
JGE	159	40.4	23.4	33.6	0.58	0.83	10.22
JUP	135	80.7	64.1	77.1	0.79	0.96	12.95
JMC	132	67.2	54.8	64.0	0.82	0.95	9.17
JUJ	129	79.9	63.6	67.1	0.8	0.84	3.49
JGP	121	86.7	79.7	84.0	0.92	0.97	4.37
JUL	115	75.4	59.7	69.7	0.79	0.93	10.00
JGS	98	50.0	39.5	44.2	0.79	0.88	4.66
JMA	69	61.1	48.7	70.6	0.80	1.16	21.9
JGH	61	84.1	57.9	61.9	0.69	0.74	3.96
Min	-	40.43	23.40	33.62	0.58	0.74	3.49
Mean	-	69.01	54.38	62.91	0.78	0.91	8.53
Median	-	71.27	56.37	65.53	0.80	0.90	6.92
Max	-	86.67	79.67	84.04	0.92	1.16	21.9

In the MDL/DEV column, we observe that the ratio between MDL and DEV varies from 0.74 to 1.16, with an average of 0.91 and a median of 0.90. This indicates a tendency to produce results that are more similar to DEV than MQ regarding the distribution of change, which can be observed in the MQ/DEV column. For JMA, MDL actually outperformed DEV, with 70.6% against 61.1%. The ratio between MQ and DEV varies from 0.58 to 0.92, with an average of 0.78 and a median of 0.80. This can be seen visually through the boxplot in Figure 10.

The correlation (ρ) of the difference between MDL and MQ ($MDL - MQ$ column) and the number of classes is equal to -0.23, which indicates a weak inverted correlation according to Cohen

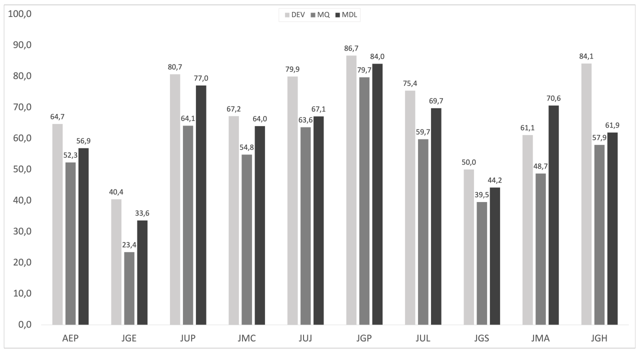

Figure 9: Single-package commits ratio (DEV x MQ x MDL)

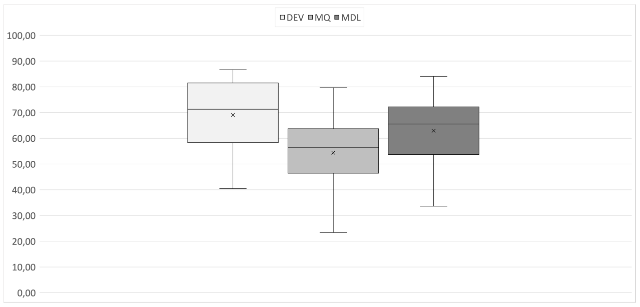

Figure 10: Single-package commits ratio (DEV x MQ x MDL) boxplot.

[12]. We wanted to determine whether the number of classes influenced the quality of the MQ and MDL solutions for single-package commits. We might say that, as the number of classes increases, the MQ solutions come slightly closer to the MDL ones.

4.8 Analysis: Average Packages per Commit

Table 8 and Figure 11 show the average number of packages per commit on each distribution (DEV, MQ, and MDL). This metric complements the previous metric, giving us a better view of how distant some software projects are from a single-package commit, i.e., the bigger the value, the worse the performance.

Table 8: Average of clusters per commit (DEV x MQ x MDL).

ID	#	Clusters (DEV)	Clusters (MQ)	Clusters (MDL)	$MQ/$ DEV	$MDL/$ DEV	$MQ-$ MDL
AEP	162	1.7	2.8	2.3	1.65	1.35	0.5
JGE	159	3.0	6.6	3.1	2.20	1.03	3.5
JUP	135	1.3	2.7	1.8	2.08	1.38	0.9
JMC	132	2.0	3.0	2.1	1.50	1.05	0.9
JUJ	129	1.4	2.4	1.8	1.71	1.29	0.6
JGP	121	1.2	1.7	1.4	1.42	1.17	0.3
JUL	115	1.4	2.6	2.0	1.86	1.43	0.6
JGS	98	1.7	3.0	2.5	1.76	1.47	0.5
JMA	69	1.6	2.3	1.3	1.44	0.81	1.0
JGH	61	1.2	2.7	1.9	2.25	1.58	0.8
Min	-	1.20	1.70	1.30	1.42	0.81	0.30
Mean	-	1.65	2.98	2.02	1.79	1.26	0.96
Median	-	1.50	2.70	1.95	1.74	1.32	0.70
Max	-	3.00	6.60	3.10	2.25	1.58	3.50

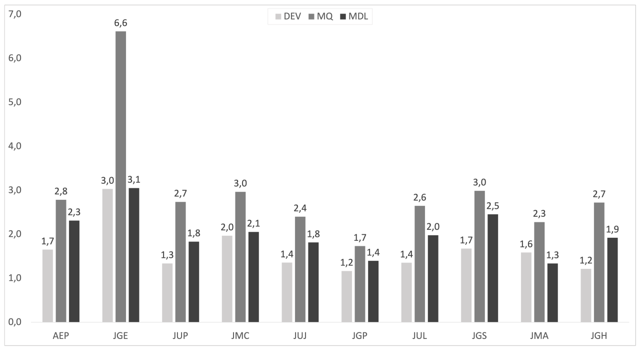

Figure 11: Average of packages per commit (*DEV* x *MQ* x *MDL*)

We can observe a poor performance for the *MQ* distribution of *JGE* with 6.6 packages per commit on average. The remaining projects all have larger values than *DEV* and *MDL*. Once more, we observe a better performance of the *MDL* distribution for *JMA*, with 1.3 against 1.6 and 2.3 from *DEV* and *MQ*, respectively.

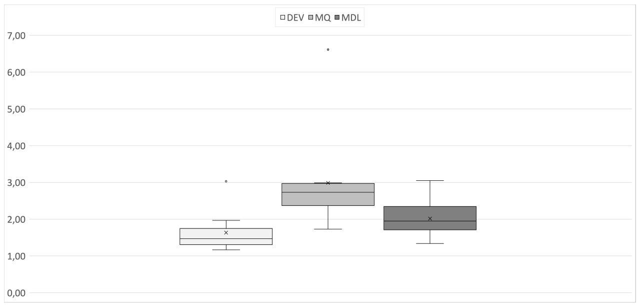

Figure 12: Average of packages per commit (*DEV* x *MQ* x *MDL*) boxplot.

The ratio between *MDL* and *DEV* (column *MDL/DEV*) varies from 0.81 to 1.58, with an average of 1.26 and a median of 1.32. This indicates a tendency to produce results more similar to *DEV* than *MQ*, which can be observed in the *MQ/DEV* column. For *JMA* again, *MDL* actually outperformed *DEV*, with 1.3 against 1.6. The ratio between *MQ* and *DEV* varies from 1.42 to 2.25, with an average of 1.79 and a median of 1.74. This could also be observed from the boxplot in Figure 12.

The correlation (ρ) of the difference between *MDL* and *MQ* ($MDL - MQ$ column) and the number of classes is equal to 0.33, which indicates a medium correlation according to Cohen [12]. In other words, we can say that as the number of classes increases, the *MQ* solutions increase the average number of packages involved in a single commit with their distributions.

4.9 Summary

Next, we summarize the results detailed in the former subsections to draw conclusions from the results of our studies.

MQ generated many more packages than the *MDL* and the *DEV* solutions for all software projects under analysis. Consequently, *MQ* produced solutions with a much smaller average ratio of classes to packages. This confirms the results presented by Barros et al. [6] for the software projects analyzed here. While solutions generated by *MDL* present a larger number of packages than the *DEV* distribution and, subsequently, smaller ratios of classes to packages than the distribution proposed by the developers, we observe a clear improvement toward automated software clustering.

MDL generated solutions with a maximum depth of hierarchy levels compatible with the *DEV* distribution. Furthermore, such solutions have a median *MoJoFM* of 58.10 and are far closer to the *DEV* distribution than solutions generated by *MQ* (median *MoJoFM* of 37.6). Thus, we can conclude that the authoritativeness of solutions produced by an optimization guided by *MDL* is better on average than solutions produced by *MQ*.

Finally, concerning the version control system, we noticed that the *DEV* distribution has the largest percentage of commits involving changes in classes from a single package, with *MDL* coming closer than *MQ* for all software projects analyzed. *MDL*-based solutions also present an average number of packages involved on each commit slightly higher than the *DEV* distribution. We conclude that while *MDL* produces distributions of classes into packages that are still outperformed by the *DEV* distribution on what concerns the concentration of changes, such solutions are far better than those produced by *MQ*.

5 VALIDITY THREATS

According to Wohlin *et al.* [39], four categories of validity threats may endanger the results of experimental studies: construct, internal, external, and conclusion threats.

Construct validity threats are related to generalizing the results to the concepts behind the study, in other words, the relationship between theory and observation. To minimize this kind of threat, the theory and reasoning behind selecting the algorithm, software systems subjected to analysis, and fitness functions were thoroughly presented in section 4. We have also considered open-source projects actively used and maintained in the Software Engineering community to prevent using toy examples.

The **internal validity** of experiments is threatened when the results can be tainted by modeling and measurement errors, or we cannot replicate the behavior of the study. To guarantee the reproducibility of the experiments, all the source code and projects used in the experiments will be made available in a public repository. Furthermore, the steps to reproduce the results will be detailed in the same repository.

The **external validity** threats are related to our ability to generalize the studies' results and whether they can be used in different scopes. Only open-source software systems were explored in the experiments. This could threaten the validity of the results. More centralized development models, such as the systems used in commercial software companies, are developed with distributed and organized collaboration. In open communities, the efforts are distributed among developers around the globe with little or no centralization and control. In this case, the ideal would be to conduct a more comprehensive investigation into different commercial software systems of varying sizes. However, accessing the source code of commercial software systems is more complicated. In addition, the software projects used in the experiments were collected from software written in Java, which can limit generalization to systems written in other programming languages.

The threats related to inaccurate conclusions come under the **conclusion validity** category. In this sense, only ten software systems were tested in the experiments reported in this chapter, which makes it impossible to adopt statistical inference tests. The intention is to increase and diversify the software systems analyzed to guarantee an evaluation using statistical tests.

6 CONCLUSION

This paper presented a set of experiments conducted on hierarchical clustering using *MDL* as a fitness function to improve hierarchical structures. We have combined the *AC* algorithm with *MDL* and compared the results with the ones generated by another driver function, *MQ* – widely used in the literature. The comparisons focused on measuring the authoritativeness of the approaches, comparing the solutions they proposed on an instance basis to those proposed by the developers. Overall, we noticed that *MDL* generated solutions closer to those proposed by the developers than the ones suggested by *MQ*. While we agree that we have few instances, the positive results obtained while examining those justify the need to improve the implementation of the optimizer to cope with larger instances in a reasonable time.

Considering the practical implications of the results, we understand development teams could benefit from using an SMC tool as a refactoring mechanism or semi-automatic assistant to improve the maintainability of the entire solution or its sub-modules. Then, we could evaluate and suggest improvements as the system evolves.

We could also highlight the research results are more related to how hierarchical-based metrics could generate solutions closer to those developers propose. This is mainly because hierarchies make it easier to understand the underlying system. This could motivate other researchers to explore the hierarchical side of decompositions and focus more on techniques that would bring the results closer to those humans would create and maintain.

ARTIFACTS AVAILABILITY

Results dataset available at https://zenodo.org/records/10970792.

REFERENCES

[1] Jesús S. Aguilar-Ruiz, Isabel Ramos, José Cristóbal Riquelme Santos, and Miguel Toro. 2001. An evolutionary approach to estimating software development projects. *Information and Software Technology* 43, 14 (2001), 875–882. https://doi.org/10.1016/S0950-5849(01)00193-8 Publisher: Elsevier.

[2] Giuliano Antoniol, Massimiliano Di Penta, and Mark Harman. 2005. Search-Based Techniques Applied to Optimization of Project Planning for a Massive Maintenance Project. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*. IEEE Computer Society, 240–249. https://doi.org/10.1109/ICSM.2005.79

[3] Andrea Arcuri and Lionel C. Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *Proc. of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*. ACM, 1–10. https://doi.org/10.1145/1985793.1985795

[4] Anthony J. Bagnall, Victor J. Rayward-Smith, and Ian M. Whittley. 2001. The next release problem. *Information and Software Technology* 43, 14 (2001), 883–890. https://doi.org/10.1016/S0950-5849(01)00194-X

[5] Márcio Barros. 2014. An experimental evaluation of the importance of randomness in hill climbing searches applied to software engineering problems. *Empir. Softw. Eng.* 19, 5 (2014), 1423–1465. https://doi.org/10.1007/S10664-013-9294-4

[6] Márcio Barros, Fábio Farzat, and Guilherme Travassos. 2015. Learning from optimization: A case study with Apache Ant. *Inf. Softw. Technol.* 57 (2015), 684–704. https://doi.org/10.1016/J.INFSOF.2014.07.015

[7] Iain Bate and Simon M. Poulding. 2011. Editorial for the special issue on search-based software engineering. *Software: Practice and Experience* 41, 5 (2011), 467–468. https://doi.org/10.1002/SPE.1056

[8] Michael Bowman, Lionel C. Briand, and Yvan Labiche. 2010. Solving the Class Responsibility Assignment Problem in Object-Oriented Analysis with Multi-Objective Genetic Algorithms. *IEEE Transactions on Software Engineering* 36, 6 (2010), 817–837. https://doi.org/10.1109/TSE.2010.70

[9] Lionel Claude Briand, Sandro Morasca, and Victor R. Basili. 1999. Defining and Validating Measures for Object-Based High-Level Design. *IEEE Trans. Software Eng.* 25 (1999), 722–743. https://api.semanticscholar.org/CorpusID:6662624

[10] Colin James Burgess and Martin Lefley. 2001. Can genetic programming improve software effort estimation? A comparative evaluation. *Inf. Softw. Technol.* 43 (2001), 863–873. https://api.semanticscholar.org/CorpusID:206106375

[11] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. 2004. Finding community structure in very large networks. *Physical Review E* 70, 6 (Dec. 2004), 066111. https://doi.org/10.1103/PhysRevE.70.066111

[12] J. Cohen. 1992. A power primer. *Psychological Bulletin* 112, 1 (July 1992), 155–159. https://doi.org/10.1037//0033-2909.112.1.155

[13] R. M. Cormack. 1971. A Review of Classification. *Journal of the Royal Statistical Society. Series A (General)* 134, 3 (1971), 321–367. https://doi.org/10.2307/2344237 Publisher: [Royal Statistical Society, Wiley].

[14] D. Doval, S. Mancoridis, and B.S. Mitchell. 1999. Automatic clustering of software systems using a genetic algorithm. In *STEP '99. Proceedings Ninth International Workshop Software Technology and Engineering Practice*. IEEE Comput. Soc, Pittsburgh, PA, USA, 73–81. https://doi.org/10.1109/STEP.1999.798481

[15] Juan J. Durillo, Yuanyuan Zhang, Enrique Alba, Mark Harman, and Antonio J. Nebro. 2011. A Study of the Bi-Objective next Release Problem. *Empirical Softw. Engg.* 16, 1 (Feb. 2011), 29–60. https://doi.org/10.1007/s10664-010-9147-3 Place: USA Publisher: Kluwer Academic Publishers.

[16] Gordon Fraser and Jerffeson Teixeira de Souza. 2018. Guest editorial: search-based software engineering. *Empirical Software Engineering* (2018), 1–3. https://api.semanticscholar.org/CorpusID:14630566

[17] Simon J. Gibbs, Dennis Tsichritzis, Eduardo Casais, Oscar Nierstrasz, and Xavier Pintado. 1990. Class management for software communities. *Commun. ACM* 33 (1990), 90–103. https://api.semanticscholar.org/CorpusID:16220359

[18] A. D. Gordon. 1987. A Review of Hierarchical Classification. *Journal of the Royal Statistical Society. Series A (General)* 150, 2 (1987), 119–137. https://doi.org/10.2307/2981629 Publisher: [Royal Statistical Society, Wiley].

[19] Mathew Hall and Phil McMinn. 2012. An analysis of the performance of the bunch modularisation algorithm's hierarchy generation approach. In *4 th Symposium on Search Based-Software Engineering*. 19.

[20] Mark Harman, Robert Hierons, and Mark Proctor. 2002. A new representation and crossover operator for search-based optimization of software modularization. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation (GECCO'02)*. Morgan Kaufmann Publishers, San Francisco, CA, USA, 1351–1358.

[21] Mark Harman and Bryan F. Jones. 2001. Search-based software engineering. *Inf. Softw. Technol.* 43, 14 (2001), 833–839. https://doi.org/10.1016/S0950-5849(01)00189-6

[22] Jon Kleinberg and Eva Tardos. 2006. *Algorithm design.* Pearson Addison Wesley.

[23] Craig Larman. 2001. Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process. https://api.semanticscholar.org/CorpusID:119922699

[24] Eugene L. Lawler and D. E. Wood. 1966. Branch-and-Bound Methods: A Survey. *Oper. Res.* 14 (1966), 699–719. https://api.semanticscholar.org/CorpusID:36099120

[25] Ming Li and Paul M.B. Vitányi. 1990. Kolmogorov Complexity and its Applications. In *Algorithms and Complexity*. Elsevier, 187–254. https://doi.org/10.1016/B978-0-444-88071-0.50009-6

[26] Rudi Lutz. 2001. Evolving good hierarchical decompositions of complex systems. *J. Syst. Archit.* 47 (2001), 613–634. https://api.semanticscholar.org/CorpusID:28630106

[27] Spiros Mancoridis, Brian Mitchell, Chris Rorres, Yih-Farn Chen, and Emden Gansner. 1998. Using Automatic Clustering to Produce High-Level System Organizations of Source Code. In *6th International Workshop on Program Comprehension (IWPC '98), June 24-26, 1998, Ischia, Italy*. IEEE Computer Society, Ischia, Italy, 45–52. https://doi.org/10.1109/WPC.1998.693283

[28] Steve McConnell. 2004. *Code Complete, Second Edition.* https://api.semanticscholar.org/CorpusID:60449735

[29] Leandro L. Minku and Xin Yao. 2013. Ensembles and locality: Insight on improving software effort estimation. *Inf. Softw. Technol.* 55 (2013), 1512–1528. https://api.semanticscholar.org/CorpusID:584455

[30] Brian Mitchell and Spiros Mancoridis. 2002. Using heuristic search techniques to extract design abstractions from source code. In *Proceedings of the 4th Annual Conference on Genetic and Evolutionary Computation (GECCO'02)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1375–1382.

[31] Brian Mitchell and Spiros Mancoridis. 2006. On the Automatic Modularization of Software Systems Using the Bunch Tool. *IEEE Transactions on Software Engineering* 32, 3 (March 2006), 193–208. https://doi.org/10.1109/TSE.2006.31

[32] Marlon C. Monçores, Adriana C. F. Alvim, and Márcio O. Barros. 2018. Large Neighborhood Search applied to the Software Module Clustering problem. *Computers & Operations Research* 91 (March 2018), 92–111. https://doi.org/10.1016/j.cor.2017.10.004

[33] Marlon da Costa Monçores. 2015. *Busca em vizinhança grande aplicada ao problema de clusterização de módulos de software.* Master's thesis. http://www.repositorio-bc.unirio.br:8080/xmlui/handle/unirio/11798 Accepted: 2018-06-25T22:06:23Z.

[34] Alexandre Fernandes Pinto. 2014. *Uma heurística baseada em busca local iterada para o problema de clusterização de módulos de software.* Master's thesis. http://www.repositorio-bc.unirio.br:8080/xmlui/handle/unirio/11920 Accepted: 2018-07-10T22:00:52Z.

[35] Kata Praditwong, Mark Harman, and Xin Yao. 2011. Software Module Clustering as a Multi-Objective Search Problem. *IEEE Transactions on Software Engineering* 37, 2 (2011), 264–282. https://doi.org/10.1109/TSE.2010.26

[36] J. Rissanen. 1978. Modeling by shortest data description. *Automatica* 14, 5 (Sept. 1978), 465–471. https://doi.org/10.1016/0005-1098(78)90005-5

[37] C E Shannon. 1948. A Mathematical Theory of Communication. *The Bell System Technical Journal* 27 (Oct. 1948), 623–656.

[38] V. Tzerpos and R.C. Holt. 2000. ACCD: an algorithm for comprehension-driven clustering. In *Proceedings Seventh Working Conference on Reverse Engineering.* 258–267. https://doi.org/10.1109/WCRE.2000.891477 ISSN: 1095-1350.

[39] Claes Wohlin. 2012. *Experimentation in software engineering.* Springer, NY.

[40] Joseph Arthur Wood. 1998. *Improving software designs via the minimum description length principle.* PhD Thesis. https://api.semanticscholar.org/CorpusID:38339071

[41] Jifeng Xuan, He Jiang, Zhilei Ren, and Zhongxuan Luo. 2012. Solving the Large Scale Next Release Problem with a Backbone-Based Multilevel Algorithm. *IEEE Transactions on Software Engineering* 38, 5 (2012), 1195–1212. https://doi.org/10.1109/TSE.2011.92

[42] Edward Yourdon and Larry L. Constantine. 1979. Structured design. Fundamentals of a discipline of computer program and systems design. *Englewood Cliffs: Yourdon Press* (1979). https://ui.adsabs.harvard.edu/abs/1979sdfd.book.....Y/abstract